

2

Probabilistic loops: Invariants and variants

2.1	Introduction: loops via recursion	38
2.2	Probabilistic invariants	39
2.3	Probabilistic termination	40
2.4	Invariance and termination together: the loop rule	42
2.5	Three examples of probabilistic loops	44
2.5.1	The martingale	44
2.5.2	Probabilistic amplification	46
2.5.3	Faulty factorial	51
2.6	The Zero-One Law for termination	53
2.7	Probabilistic variant arguments for termination	54
2.8	Termination example: self-stabilisation	56
2.8.1	Variations on the ring	59
2.9	Uncertain termination	61
2.9.1	Example: an inductive termination argument	62
2.10	Proper post-expectations	63
2.10.1	The martingale revisited	64
2.11	Bounded <i>vs.</i> unbounded expectations	68
2.11.1	Unbounded invariants: a counter-example	71
2.12	Informal proof of the loop rule	74
	Chapter notes	77

2.1 Introduction: loops via recursion

We saw in Chap. 1 that iteration is a special case of recursion. But the weakest pre-expectation $wp.prog.postE$ cannot be given a purely syntactic definition for general recursive $prog$ — the definition given earlier (Fig. 1.5.3) is semantic, a least fixed-point over expectation transformers. It does give us an algebraic property of recursive programs, *viz.*

$$(\mathbf{mu} X \cdot C) = C \langle X \mapsto (\mathbf{mu} X \cdot C) \rangle, \quad (2.1)$$

but that is an *equation* rather than a definition.¹ See (2.6) below however for an example of its use in spite of that.

Iteration's special form (1.18) is the case of recursion where the context C is ²

$$(prog; X) \text{ if } pred \text{ else skip},$$

from which with (2.1) we have immediately the property that

$$\begin{aligned} \mathbf{do} \ pred \rightarrow prog \ \mathbf{od} &= (prog; \mathbf{do} \ pred \rightarrow prog \ \mathbf{od}) \\ &\quad \text{if } pred \text{ else} \\ &\quad \text{skip}. \end{aligned} \quad (2.2)$$

Applying wp to both sides, with respect to an arbitrary post-expectation $postE$ we see that

$$\begin{aligned} \text{if} \quad & preE = wp.(\mathbf{do} \ pred \rightarrow prog \ \mathbf{od}).postE \\ \text{then also} \quad & preE = wp.prog.preE \text{ if } pred \text{ else } postE, \end{aligned} \quad (2.3)$$

of which the latter at least is now an equation in expectations (in $preE$) rather than in programs (contrast (2.1)). However, because $preE$ occurs on both sides, we still do not have a definition.³

Because of the issues above, and for historical reasons [Flo67, Hoa69], for standard programs rather than ask for the *weakest* precondition for a loop we are content with a precondition that is sufficiently strong (*i.e.* might be stronger than required, but still implies the weakest one).

¹To turn this into a definition we would add that $(\mathbf{mu} X \cdot C)$ is the LEAST PROGRAM IN THE REFINEMENT ORDER (\sqsubseteq) that has Property (2.1).

²It is TAIL RECURSION, which is what allows the following treatment.

³Recalling Footnote 1 just above, we see that a definition of $preE$ would then be that it is the least such $preE$ in the probabilistic implication order \Rightarrow ; but “taking the least” is not a syntactic operation. Dijkstra finessed this in his presentation of wp for standard loops by using the usual iterative formulation for a least fixed-point [Tar55], the iterates being his predicates H_k [Dij76, p. 35], and then taking the limit with a quantification $(\exists k \cdot H_k)$. But that quantification, being over predicates, is second-order and thus lies outside the first-order logic of wp used here (see however *e.g.* Back and von Wright, and Ward [War89, BvW93]).

We call this “finessing,” rather than *e.g.* “cheating,” because with it Dijkstra successfully confined the second-order reasoning to just the place where it was required.

For probabilistic programs, by analogy, we look not necessarily for the *greatest* pre-expectation but merely for one that is sufficiently small.

Whether standard or probabilistic, the general techniques for loops involve “invariants” and “variants,” and we now consider their probabilistic versions.

In the rest of Part I we use single capital letters A, B, \dots, P, Q, \dots for expressions over the state, that is Boolean-valued predicates, real-valued expectations *etc.* — rather than words $preE, postE$ as earlier — unless the occasion demands extra clarity. As an exception we will write $Inv, Term$ for standard invariants and termination conditions, reserving I, T for their probabilistic counterparts.

2.2 Probabilistic invariants

In a standard loop, the invariant is proved to hold at the beginning of every iteration: as a result it describes a set of states from which the loop body cannot escape and so termination — if it occurs — must also lie within that set. The proof obligation for a standard loop is

$$G \wedge Inv \Rightarrow wp.body.Inv, \quad^4$$

where G is the loop guard and Inv is the predicate describing the invariant set of states.

For a probabilistic loop we have a post-expectation rather than a post-condition; but otherwise the situation is much the same. We therefore say that an expectation I is an invariant of a loop under these conditions:

Definition 2.2.1 PROBABILISTIC INVARIANT Consider a program *loop* defined **do** $G \rightarrow body$ **od**, with predicate G its *guard*. We say that expectation I is an *invariant* of *loop* just when

$$[G] * I \Rightarrow wp.body.I.$$

□

The use of multiplication in the idiom $[G] * I$ is just a convenient way of writing I **if** G **else** 0, with the effect in this case of restricting the probabilistic implication \Rightarrow to require proof only in states satisfying G :

⁴We now dispense with the distinction between Boolean- and $\{0, 1\}$ types for standard predicates, and overload the \Rightarrow -style and wp operators in order to reduce the occurrence of embedding-brackets $[\cdot]$. In particular we use \Rightarrow for “everywhere implies” between predicates as well as “is everywhere no more than” between expectations, to achieve a consistency of notation.

for if G does not hold, then the left-hand side is zero, and so “ \Rightarrow -implies” anything.⁵

As with standard programming, finding invariants is a skill — perhaps one of the main skills, for intricate algorithms — and heuristics for it are important. One such is the following: if the post-expectation of the loop is standard,⁶ some $[post]$ say, then as an aid to the intuition we can look for an expression that gives a lower bound on the probability that we will establish $post$ by (continuing to) execute the loop body.⁷ Often that expression will have the form

$$p * [pred] \tag{2.4}$$

with p a probability and $pred$ a predicate, both expressions over the state.⁸ The standard part $pred$ might be found by conventional techniques, with the probability p estimated by a separate “worst-case” analysis.

From the definition of the embedding $[pred]$ we know that one interpretation of (2.4) is

with probability p if $pred$ holds, and with probability zero otherwise,

and in many cases it will serve directly as an invariant for the loop — whence our interpretation becomes

if $pred$ holds then with probability at least p the loop executed from this state will satisfy $post$ on termination; but

if $pred$ does not hold, so that (2.4) is zero, then we have no information about the probability of achieving $post$, since all probabilities are at least zero anyway.

We see several examples of such invariants in Sec. 2.5.

2.3 Probabilistic termination

The *probability* that a program will terminate generalises the usual definition of termination: recalling that $[true] \equiv 1$ we see that a program’s greatest guaranteed probability of termination is just

$$wp.prog.1 . \tag{2.5}$$

⁵Recall that expectations are non-negative.

⁶We say an expectation is STANDARD if it can be written $[pred]$ for some predicate $pred$, or equivalently if it is $\{0, 1\}$ -valued.

⁷We give examples of this technique in Sec. 2.5, and return to it in Sec. 7.7.8 for loops “**do** $1/2 \rightarrow \dots$ ” with probabilistic guards.

⁸We extend our naming convention to allow lower-case letters like “ p ” for probability-valued expressions over the state.

That raises the issue of whether there is a difference between

certain $wp.(\dots).true$ for a standard program,
termination

and what we could call

“almost-certain” $wp.(\dots).[true]$ for a probabilistic one.
termination

In fact there is a difference: we say that a computation — regarded as a branching tree of steps — terminates *absolutely* if *every* path in the tree leads to termination, *i.e.* is finite; if there are any infinite paths at all, then termination is not absolute.

Even if the computation is probabilistic, the same applies; but we can be more discriminating if we wish. Each step on a path will have an associated probability (often probability one, for standard steps in the computation) and — speaking informally⁹ — the probabilities on those individual steps, multiplied together, determine probabilities for paths and, when added up, for sets of paths as a whole. If the collective probability of the infinite paths in a computation is zero, then we say it terminates *almost-certainly*, and that nontermination is *almost impossible*.

Our semantics, and logic, do not distinguish absolute- and almost-certain termination — in other words, we take the position that the benefit of a simpler logic outweighs any disadvantage of ignoring almost-impossible events.

As a simple example of recursive termination, suppose $prog$ is the program defined¹⁰

$$prog := prog_p \oplus \mathbf{skip} , \quad (2.6)$$

in which we assume that p is some fixed probability not equal to one: on each recursive call, $prog$ has probability \bar{p} of termination, continuing otherwise with further recursion. Elementary probability theory shows that $prog$ terminates with probability one (after an expected p/\bar{p} recursive calls). In the expectation logic we calculate

$$wp.prog.1$$

⁹A rigorous treatment of probability distributions over infinite sets involves σ -algebras [GS92]; but in this discussion we do not need those details. See Footnote 7 on p. 297 for an example.

¹⁰That is, $prog$ is the program $(\mathbf{mu} X \bullet X_p \oplus \mathbf{skip})$; we have merely written out its property (2.1), and have assumed we mean the \sqsubseteq -least such program.

In Sec. 7.7 we see that this kind of program can be written

$$\mathbf{do} p \rightarrow \mathbf{skip} \mathbf{od} .$$

(Compare *e.g.* (7.24) on p. 203.) We explain there how an extended form of variant-argument can show termination of such loops.

$$\begin{aligned}
&\equiv p * wp.prog.1 + \bar{p} * wp.\mathbf{skip}.1 && (2.5); wp \text{ for } p \oplus \\
&\equiv p * wp.prog.1 + \bar{p}, && wp \text{ for } \mathbf{skip}
\end{aligned}$$

so that $\bar{p} * wp.prog.1 \equiv \bar{p}$. Since p is not one, we can divide by \bar{p} to see that indeed $wp.prog.1 \equiv 1$, agreeing with the elementary theory.

However proved, it is true that *prog* terminates with probability one — *i.e.* it terminates almost certainly. But it does not terminate absolutely: regarded as a tree of possible recursive calls, the computation contains an infinite path *recurse, recurse, recurse...* That path’s overall probability, however, is no more than the probability p^n of taking its first n steps, for any n no matter how large — *i.e.* for $p < 1$ it is zero for the whole path. Our logic ignores it.

We return to probabilistic termination in Sec. 2.6.

2.4 Invariance and termination together: the loop rule

The conventional technique for standard loops is to prove “partial correctness” and termination separately, and then to put them together to establish “total correctness.”

Suppose *loop* is standard: if *Inv* is an invariant for it, then we know *Inv* initially is sufficient to achieve $Inv \wedge \bar{G}$ finally, provided the loop terminates. That is *partial correctness*. A separate proof is required to establish the initial states from which termination is guaranteed; from those initial states which additionally satisfy *Inv* the loop *will* establish $Inv \wedge \bar{G}$. That is *total correctness*.

Together, we thus have that if *Inv* is the invariant and *Term* describes states from which termination is guaranteed then

$$Inv \wedge Term \Rightarrow wp.loop.(Inv \wedge \bar{G})$$

for a standard loop.

For probabilistic loops we pursue a similar strategy, of joining partial correctness and termination, although of course we cannot use \wedge for the purpose since we are no longer dealing with Booleans. The general methods are the subject of Chap. 7; here we set out three useful specialisations.¹¹

¹¹We will see that the proof of the general methods is relatively complicated — which is a surprise given the simplicity of Def. 2.2.1 and the fact that it corresponds to a simple recursive argument in elementary probability theory. See Footnote 66 on p. 73 for reassurance that the extra complexity we have deferred until Chap. 7 is probably unavoidable.

Lemma 2.4.1 TOTAL CORRECTNESS FOR PROBABILISTIC LOOPS

Let expectation T be the termination probability of *loop*, so that

$$T := wp.(\mathbf{do} \ G \rightarrow \mathbf{prog} \ \mathbf{od}).1 ,$$

and let I be a probabilistic invariant for it (Def. 2.2.1). We consider three cases:

1. If $I \equiv [Inv]$ for some standard Inv , define $preE := T * [Inv]$.
2. If $[Term] \Rightarrow T$ for some standard $Term$, so that $Term$ contains only states where termination is almost certain, define $preE := I * [Term]$. (Thus if T is itself standard, we can again define $preE := I * T$.)
3. If $I \Rightarrow T$, then define $preE := I$.¹²

Then in each case we have that $preE$ is a sufficient pre-expectation for the loop to terminate while maintaining the invariant:¹³

$$preE \Rightarrow wp.loop.([\overline{G}] * I) .$$

Proof Given in Sec. 7.4 of Part II. (For an alternative but “model-based” argument, see also Sec. 2.12 in this chapter.) \square

Notice that Lem. 2.4.1 subsumes the standard loop rule, since if I and T are both standard then both Cases 1 and 2 will apply — whence the pre-expectation is just $[Inv] * [Term] \equiv [Inv \wedge Term]$ either way, as usual.

Case 1 occurs when a loop’s calculations are captured by a standard invariant, but its termination is not guaranteed. Typically “Las-Vegas” algorithms have this property, that they are correct if they terminate but the termination occurs only with some probability [BB96]. (See Sec. 2.5.1.)

¹²See however Sec. 2.6 in this chapter for a generalisation of this case.

¹³A “sufficient precondition” is a predicate which is sufficiently strong to imply the weakest precondition, and the “sufficient” is usually implicit — so that we simply say “a precondition.” Thus by analogy here we mean a (pre-)expectation which is sufficiently low to imply (\Rightarrow) the *greatest* pre-expectation.

As Hehner points out, however, in its normal English sense “a precondition” is *not* implicitly sufficient — rather it is “necessary” (*i.e.* is *implied by* rather than implying the weakest-). We are stuck with the Computer-Science sense which, as we now see, is the opposite.

Case 2 occurs when a loop’s termination is guaranteed, but its calculations are correct only with some probability. This is characteristic of “Monte-Carlo” algorithms [*op. cit.*].¹⁴ (See Sec. 2.5.2.)

Case 3 applies when neither partial correctness nor termination are assured (Sec. 2.5.3).

2.5 Three examples of probabilistic loops

In the examples of this section we illustrate the three cases of Lem. 2.4.1. The formal calculations justifying the invariant or termination conditions themselves are only sketched, so that the interaction of invariance and termination can be highlighted.

2.5.1 *The martingale*

We begin with an example of Case 1, where correctness is certain if the loop terminates, but termination occurs only with some probability.

Imagine a gambler who is placing 50/50 bets: that is he expects to win each bet with probability $1/2$ and receives twice his bet back if he does. He starts with a bet of 1 unit; if he wins, he has increased his capital by 1. If he loses, however, he doubles his bet to 2 and tries again; winning this time also increases his original capital by 1, since he wins 2 but lost 1 before. If he again loses, he continues with 4 units, and the same applies (might win 4, but lost $1 + 2$ before). Eventually he must win, he reasons, in which case he will receive 1 unit more than the total he has already lost; and he will have increased his capital by exactly 1.

Thus by following this strategy, he believes, he is guaranteed eventually to win 1 unit if he continues long enough.

In fact the gambler’s reasoning is only “partially correct,” and the flaw is that he has not assured (successful) termination: there is a chance that he will suffer a losing streak so long that he eventually cannot double his bet any more, because he has exhausted his entire capital. Though that possibility may be remote, his loss if it occurs could be very large.

¹⁴Since we do not distinguish absolute- from almost-certain termination, we must admit that our requirement “termination is guaranteed” does allow nontermination — but only if it is almost impossible. Most descriptions of Monte-Carlo algorithms however assume *absolute* termination, which we would express by using standard *wp* over a version of the program text in which all occurrences of probabilistic choice $_p\oplus$ had been replaced by demonic choice \sqcap . (Since with our quantitative *wp* we have *identified* the two forms of termination, we cannot expect to give a semantic characterisation of absolute- as opposed to almost-certain termination in our model; that is why we refer to the program text. See also Footnote 25 on p. 235.)


```

init   →      c, b := C, 1;
loop   →      do b ≠ 0 →
                if b ≤ c then
                  c := c - b;
                  c, b := c + 2b, 0 1/2 ⊕ b := 2b
                fi
            od

```

The program *prog* is the whole of the above.

The gambler's capital c is initially C , and his intended bet b is initially one.

On each iteration, if his intended bet does not exceed his capital, he is allowed to place it and he has a $1/2$ chance of winning.

If he wins, he receives twice his bet in return and sets his intended bet to 0 to indicate he is finished; if he loses, he receives nothing and doubles his intended bet — hoping to win next time.

If he loses sufficiently often in succession, his intended bet b will eventually be more than he can afford — more than his remaining capital c — and he will then be “trapped” forever within the iteration.

Figure 2.5.1. THE MARTINGALE

We model the martingale as in Fig. 2.5.1. It is a Las-Vegas algorithm because its termination is uncertain — the way the program is written, if the gambler runs out of money he never leaves the casino. If termination does occur, then the correct postcondition is established, that $c = C + 1$, and he goes home with a win of exactly one unit.

It is easy arithmetic to show that $c + b = C + 1$ is a standard invariant of *loop* — treat probabilistic choice $_{1/2 \oplus}$ as demonic choice \sqcap and apply standard reasoning: any invariant of that demonic abstraction will be an invariant of its probabilistic refinement, the loop as given.

In general, the fact that probabilistic choice refines demonic choice is the mathematical relationship that allows us to apply standard reasoning to probabilistic programs, when we need only standard facts about them.

For termination, we consider an initial state in which $1 \leq b \leq c$, meaning that the gambler is still betting (he has not retired by setting b to 0) and that he has sufficient capital to make his next bet (because $c \geq b$). For the loop *not* to terminate, from such a state, the gambler must lose N times in succession from then on, for some N such that the total capital $b + 2b + \dots + 2^N b$ required to reach and make the $(N+1)^{\text{st}}$ bet strictly exceeds his current capital c — we therefore define $N_{b,c}$ to be the least N such that $(2^{N+1} - 1)b > c$.

Thus the probability of nontermination of *loop* is *no more* than $1/2^{N_{b,c}}$ if $1 \leq b \leq c$, and so its converse, the probability of termination under those conditions, must be *no less* than $1 - 1/2^{N_{b,c}}$, which we write $T_{b,c}$; that is, whatever the exact termination probability T might be in general, we have at least that¹⁵

$$T_{b,c} * [1 \leq b \leq c] \quad \Rightarrow \quad T .$$

Thus our *preE* from Case 1 must satisfy

$$\begin{aligned} & \text{preE} \\ \equiv & I * T \\ \Leftarrow & [c + b = C + 1] * T_{b,c} * [1 \leq b \leq c] \quad \text{above} \\ \equiv & T_{b,c} * [c + b = C + 1 \wedge 1 \leq b \leq c] \quad \text{. combine first and third terms} \end{aligned}$$

Referring again to Case 1, we see that the post-expectation of the loop is then

$$[\overline{G}] * I \quad \equiv \quad [b = 0] * [c + b = C + 1] \quad \Rightarrow \quad [c = C + 1] ,$$

indeed establishing that the gambler's capital will increase by exactly one.

We finish by calculating a pre-expectation for the whole program, *viz.*

$$\begin{aligned} & wp.prog.[c = C + 1] \\ \equiv & wp.(init; loop).[c = C + 1] \\ \Leftarrow & wp.init.preE \quad \text{meaning of preE here} \\ \Leftarrow & (T_{b,c} * [c + b = C + 1 \wedge 1 \leq b \leq c]) \langle c, b \mapsto C, 1 \rangle \quad \text{above} \\ \equiv & T_{1,C} * [C + 1 = C + 1 \wedge 1 \leq 1 \leq C] \\ \equiv & T_{1,C} * [C \geq 1] , \end{aligned}$$

so showing that *prog* establishes $c = C + 1$ with probability at least $T_{1,C}$ provided he has some capital to start with.

Thus if we set $P_C := T_{1,C}$, we see that the gambler has a large chance P_C of winning a little (one unit), but a small chance $\overline{P_C}$ of losing a lot — more than half his initial capital.¹⁶

In Sec. 2.10.1 we return to the martingale, to determine his expected winnings either way.

2.5.2 Probabilistic amplification

Our second example relates to Case 2, where termination is certain but correctness is probabilistic.

¹⁵Note again the technique of writing “ $*[pred]$ ” on the left of \Rightarrow to restrict attention to only those states.

¹⁶If he starts with \$1,000, then he has chance $P_{1000} \geq 99.8\%$ of winning his \$1. But with the remaining probability 0.2% he will lose \$511.

```

init  →      a, n := true, N;
loop  →      do n ≠ 0 ∧ a →
               a := Q_p ⊕ true;      ← (2.7)
               n := n - 1
            od

```

The program *prog* is the whole of the above; it attempts to answer in *a* the question posed in *Q*.

Given a natural number *N*, the refutation procedure (2.7) is applied *N* times, with overall refutation occurring if any of those *N* separate applications refutes.

Figure 2.5.2. PROBABILISTIC AMPLIFICATION

Suppose there is a Boolean question *Q* for which there is a probability-*p* refutation procedure

$$a := Q_p \oplus \text{true} , \quad (2.7)$$

which certainly delivers some answer *a*, but does not guarantee that the answer is correct. It is thus a Monte-Carlo algorithm.^{17,18}

For example, if *Q* were the question “is *N* prime,” then the Miller-Rabin test for primality [MR95] is an example of the refutation procedure above with $p := 1/2$: if *N* is prime (so that *Q* is **true**), then the Miller-Rabin test is guaranteed to report “indeed *N* is prime” (setting answer *a* to **true**). If however *N* is not prime, then with probability 1/2 the test will state correctly “*N* is not prime” (setting *a* to **false**)¹⁹ — but with the remaining probability 1 – 1/2 the test fails to refute *N*’s primality (and leaves *a* set “incorrectly” to **true** even though *Q* is **false**).

With a failure probability of 1/2, the Miller-Rabin test used on its own could not be called a “reliable” prime-checker; but if used independently say 10 times in a row, it would fail to give the correct answer with probability less than 1 in 1000 — since for the answer to be wrong overall every one of the individual tests would have had to fail. That technique is called *probabilistic amplification*, and is illustrated in Fig. 2.5.2.

¹⁷A more general description of a Monte-Carlo algorithm would be

$$a := Q_p \oplus (a := \text{true} \sqcap a := \text{false}) ,$$

where the answer can be arbitrarily wrong with some probability \bar{p} . We must use a more specific case, here, for probabilistic amplification.

¹⁸Monte-Carlo algorithms give rise to the complexity class *bounded-error probabilistic polynomial-time* *BPP*, and Las-Vegas algorithms suggest the class *zero-error probabilistic polynomial-time* *ZPP* [BB96, p. 463].

¹⁹In fact in the Miller-Rabin test the probability is *at least* 1/2, which we discuss further below (p. 50).

Let *prog*, *init* and *loop* be as in Fig. 2.5.2: given arbitrary question Q , we are interested in the probability of a correct answer a at the end — that is, our post-expectation is $[a = Q]$. Using our heuristic for invariants (p. 40) we ask “given arbitrary values for a, n, Q , can we estimate the probability of achieving $a = Q$ finally if the loop were executed from there?” We reason informally as follows:

- If a does not hold, then termination is immediate and achieving $a = Q$ depends only on Q ’s current value: the probability is therefore $[Q]$ in this case.²⁰
- If a and Q both hold, then achieving $a = Q$ is certain because the test (2.7) never refutes a true Q .²¹
- If a holds but Q does not, then with probability $1 - \bar{p}^n$ at least one of the n remaining tests will correctly refute Q and set a to **false**.

With some experimentation we formulate the above observations as an expression²²

$$[a] \text{ if } Q \text{ else } (1 - \bar{p}^n [a]) ,$$

which we now show is indeed an invariant for *loop*.

As a notational convenience in the calculation, we start with the overall post-expectation and reason backwards towards the pre-expectation, indicating between expectations when *wp* is applied to give the lower (the right-hand side of a reasoning step) from the upper (the left-hand side).²³

Working from Fig. 2.5.2, we have

²⁰Note that we do *not* reason here that “ a can be false only if Q is” (and hence that the probability of achieving $a = Q$ in this case is actually one), since using that extra fact would depend on the assumption that the current state has been reached by executing the loop from its initialisation. (See also Sec. 7.7.8 below.)

Making a mistake of that kind in a heuristic is not “dangerous” of course: it just wastes time, since later when we check the putative invariant we will reject it.

²¹One of the reasons this is informal here is that we are ignoring termination for the moment; as remarked in Footnote 20 preceding, rigour comes when we prove, below, that our “guessed” invariant really is one.

²²That is, we try various formulations and see which leads to the simplest calculations to follow.

²³Admittedly this seems strange at first, since it appears we are writing \equiv between expressions that are not equal — but in practice its convenience outweighs any initial uneasiness [vGB98]. As a warning we will include a small dot to the left of the relational symbol (thus “ $\cdot \equiv$ ” but sometimes “ $\cdot \Leftarrow$ ”) to indicate when we are using this notational device. Thus when we write

$$\cdot \Leftarrow \begin{array}{c} \textit{post} \\ \textit{pre} \end{array} \qquad \text{applying } \textit{wp.prog}$$

we mean that we have established “ $\textit{pre} \Rightarrow \textit{wp.prog.post}$ ” by writing down *post*, then *prog* and finally calculating *pre* from those. The advantage will be clear when we chain several such steps together.

$$\begin{aligned}
& \cdot \equiv \begin{array}{l} [a] \text{ if } Q \text{ else } (1 - \bar{p}^n [a]) \\ [a] \text{ if } Q \text{ else } (1 - \bar{p}^{n-1} [a]) \end{array} \quad \text{applying } wp.(n := n - 1) \\
& \cdot \equiv \begin{array}{l} p * [Q] \text{ if } Q \text{ else } (1 - \bar{p}^{n-1} [Q]) \\ \bar{p} * [\text{true}] \text{ if } Q \text{ else } (1 - \bar{p}^{n-1} [\text{true}]) \end{array} \quad \text{applying } wp.(a := Q_p \oplus \text{true}) \\
& \equiv \begin{array}{l} p * 1 \text{ if } Q \text{ else } (1 - \bar{p}^{n-1} * 0) \\ \bar{p} * 1 \text{ if } Q \text{ else } (1 - \bar{p}^{n-1} * 1) \end{array} \quad \text{arithmetic} \\
& \equiv p + \bar{p} * (1 \text{ if } Q \text{ else } (1 - \bar{p}^{n-1})) \quad \text{arithmetic} \\
& \equiv 1 \text{ if } Q \text{ else } (1 - \bar{p}^n) \quad \text{arithmetic} \\
& \Leftarrow \begin{array}{l} [n \neq 0 \wedge a] * [a] \text{ if } Q \text{ else } (1 - \bar{p}^n [a]) \end{array} \quad \text{guard on left allows us to assume } [a] = 1 \text{ on right}
\end{aligned}$$

as required by Def. 2.2.1.

For termination we see by inspection that it is certain whenever $n \geq 0$, so that in Case 2 we can take $Term := 0 \leq n$ and our $preE$ becomes

$$[0 \leq n] * [a] \text{ if } Q \text{ else } (1 - \bar{p}^n [a]) .$$

And for the post-expectation we reason

$$\begin{aligned}
& \begin{array}{l} [n = 0 \vee \bar{a}] \\ * [a] \text{ if } Q \text{ else } (1 - \bar{p}^n [a]) \end{array} \quad \text{negated-guard} * \text{invariant} \\
& \equiv \begin{array}{l} [n = 0 \vee \bar{a}] \\ * [a] \text{ if } Q \text{ else } [\bar{a}] \end{array} \quad \text{both } n = 0 \text{ and } \bar{a} \text{ imply } 1 - \bar{p}^n [a] \equiv [\bar{a}] \\
& \Rightarrow [a] \text{ if } Q \text{ else } [\bar{a}] \quad \text{drop negated guard} \\
& \equiv [a = Q] ,
\end{aligned}$$

as required.

Finally, for the overall pre-expectation we calculate

$$\begin{aligned}
& \Leftarrow \begin{array}{l} wp.prog.[a = Q] \\ wp.init.preE \end{array} \\
& \equiv ([0 \leq n] * [a] \text{ if } Q \text{ else } (1 - \bar{p}^n [a])) \langle a, n \mapsto \text{true}, N \rangle \\
& \equiv [0 \leq N] * 1 \text{ if } Q \text{ else } (1 - \bar{p}^N) , \\
& \Leftarrow [0 \leq N] * (1 - \bar{p}^N) ,
\end{aligned}$$

which is to say that, provided $0 \leq N$, the program has an error probability of no more than \bar{p}^N and moreover (second-last line) is in fact always correct if Q is true.

But what if the refutation probability p is not exact?

Because our overall pre-expectation $1 - \bar{p}^N$ is an increasing function of p , it is tempting to conclude that we can replace $p \oplus$ by $\geq_p \oplus$ in Fig. 2.5.2 without affecting our result. For example, the procedure for the Miller-Rabin primality test refutes with probability *at least* rather than exactly $1/2$, but we still expect its amplified error probability to be no more than $1/2^N$. Although that conclusion does hold for Fig. 2.5.2 (and thus for Miller-Rabin), such reasoning is not valid in general — as we now show by example.

By inspection, the program

$$\begin{aligned} x &:= \text{true}_{3/4} \oplus \text{false}; \\ y &:= \text{true}_p \oplus \text{false} \end{aligned}$$

establishes postcondition $x = y$ with probability at least $p/2 + 1/4$; and we can confirm that with the calculation

$$\begin{aligned} & [x = y] \\ \cdot & \equiv p[x] + \bar{p}[\bar{x}] && \text{applying } wp.(y := \text{true}_p \oplus \text{false}) \\ & \equiv \bar{p} + (p - \bar{p})[x] \\ \cdot & \equiv 3/4 * (\bar{p} + p - \bar{p}) + 1/4 * \bar{p} && \text{applying } wp.(x := \text{true}_{3/4} \oplus \text{false}) \\ & \equiv p/2 + 1/4. \end{aligned}$$

Now $p/2 + 1/4$ is an increasing function of p , but it is *not* true that the related program

$$\begin{aligned} x &:= \text{true}_{3/4} \oplus \text{false}; \\ y &:= \text{true}_{\geq_p} \oplus \text{false} \end{aligned} \tag{2.8}$$

establishes $[x = y]$ with *at least* that probability: repeating the calculation, we find instead

$$\begin{aligned} & [x = y] \\ \cdot & \equiv (p[x] + \bar{p}[\bar{x}]) \min [x] && \text{applying } wp.(y := \text{true}_{\geq_p} \oplus \text{false}) \\ & \equiv p[x] && \text{arithmetic: cases on Boolean } x \\ \cdot & \equiv 3p/4, && \text{applying } wp.(x := \text{true}_{3/4} \oplus \text{false}) \end{aligned}$$

which is not necessarily at least $p/2 + 1/4$.

This effect is due to our interpretation of $\geq_p \oplus$, where the choice of which probability $p \leq p' \leq 1$ to use each time $\geq_p \oplus$ is executed may depend on many factors, and is not determined “just once, at the beginning.” For example, in the Miller-Rabin test the actual probability of a correct answer will depend on internal features of the implementation, which we must abstract as \square since we cannot know what they are: in general, different values of p' might be used on different occasions, though each time still in the required interval $[p, 1]$.

Behaviour like that is an example of demonic choice, modelling the fact that we don’t know which p' will be chosen. In the program above, one possibility — the worst, if we want to achieve $x = y$ finally — is to take $p' := p$ **if** x **else** 1, thus depending explicitly on x . When the first statement

sets x to **true**, the second selects **false** for y with the maximum probability \bar{p} allowed; but when x is set to **false**, the second statement sets y to **true** with probability one.

In fact the program (2.8) written in more primitive terms (recall p. 21) would be

$$\begin{aligned} x &:= \text{true}_{3/4} \oplus \text{false}; \\ y &:= \text{true}_p \oplus \text{false} \quad \sqcap \quad y := \text{true}, \end{aligned} \quad (2.9)$$

where the demonic choice could be implemented (*lhs if x else rhs*) in order to make the chance of establishing $x = y$ as low as possible, as we showed above.

We are seeing an example of the sometimes subtle interaction between probabilistic and demonic choice: in (2.9) the “demon” at \sqcap is “aware” of the outcome of the earlier probabilistic assignment to x , and “uses” that information to avoid our postcondition.

Our earlier, but informal, reasoning — based on $1 - \bar{p}^N$ being increasing in p — delivers a safe conclusion only because in that program the demon’s best strategy (worst, for us) is to choose the lowest probability p allowed, every time, no matter what the values of the state variables. If we altered the program, replacing $_p \oplus$ by $_{\geq p} \oplus$, and restarted the calculation at that point, we would verify that by calculating

$$\begin{aligned} &\equiv \quad \text{applying } wp.(a := Q_{\geq p} \oplus \text{true}) \\ &\quad p * [Q] \text{ if } Q \text{ else } (1 - \bar{p}^{n-1} [Q]) \\ &+ \quad \bar{p} * [\text{true}] \text{ if } Q \text{ else } (1 - \bar{p}^{n-1} [\text{true}]) \\ &\quad \min \quad [Q] \text{ if } Q \text{ else } (1 - \bar{p}^{n-1} [Q]) \\ &\equiv \quad \text{for the above min we always have } lhs \Rightarrow rhs \\ &\quad p * [Q] \text{ if } Q \text{ else } (1 - \bar{p}^{n-1} [Q]) \\ &+ \quad \bar{p} * [\text{true}] \text{ if } Q \text{ else } (1 - \bar{p}^{n-1} [\text{true}]), \end{aligned}$$

after which the reasoning would proceed as before. The justification “*lhs* \Rightarrow *rhs*,” that in every state $lhs \leq rhs$, is the arithmetic that shows the demon — no matter what the state contains — will in this program select the left-hand branch on each occasion.

See also the discussion of Program (B.1) on p. 326.

2.5.3 Faulty factorial

Our final example illustrates Case 3, where both the termination condition and the invariant are probabilistic. Given a natural number N , the program is to (attempt to) set f to $N!$ in spite of a probabilistically faulty subtraction.

The program is shown in Fig. 2.5.3, and is the conventional factorial algorithm except that the decrement of n sometimes increments instead.

```

init   →      n, f := N, 1;
loop   →      do n ≠ 0 →
                f := f * n;
                n := n - 1 p ⊕ n := n + 1
            od

```

The program *prog* is the whole of the above.

The decrementing of *n* fails probabilistically, sometimes incrementing instead.

Figure 2.5.3. FAULTY FACTORIAL

When *p* is one, making the program standard (and decrementing of *n* certain), the invariant $n \geq 0 \wedge N! = f * n!$ suffices in the usual way to show that $wp.prog.[f = N!] \Leftarrow [N \geq 0]$. In general, however, that postcondition is achieved only if the decrement alternative is chosen on each of the *N* executions of the loop body, thus with probability p^N . In the probabilistic case therefore we define invariant $I := p^n * [n \geq 0 \wedge N! = f * n!]$, showing its preservation with this calculation:

$$\begin{aligned}
 & p^n * [n \geq 0 \wedge N! = f * n!] \\
 \cdot & \equiv \text{applying } wp.(n := n - 1 \text{ }_p \oplus n := n + 1) \\
 & p(p^{n-1}) * [n - 1 \geq 0 \wedge N! = f * (n - 1)!] \\
 + & \bar{p}(p^{n+1}) * [n + 1 \geq 0 \wedge N! = f * (n + 1)!] \\
 \Leftarrow & p^n * [n > 0 \wedge N! = f * (n - 1)!] \quad \text{dropping the right additive term} \\
 \cdot & \equiv p^n * [n > 0 \wedge N! = f * n * (n - 1)!] \quad \text{applying } wp.(f := f * n) \\
 \equiv & [n \neq 0] * p^n * [n \geq 0 \wedge N! = f * n!], \quad \text{property of factorial}
 \end{aligned}$$

as required by Def. 2.2.1.

The exact termination condition depends on *p*. Standard *random walk* results [GW86] show that for $n \geq 0$ the loop terminates certainly when $p \geq 1/2$, but with probability only $(p/\bar{p})^n$ otherwise. In any case, however, we have $T \Leftarrow p^n * [n \geq 0]$, since that is the probability of no subtraction errors occurring at all. Thus

$$I \equiv p^n * [n \geq 0 \wedge N! = f * n!] \Rightarrow p^n * [n \geq 0] \Rightarrow T,$$

so that Case 3 of Lem. 2.4.1 applies and $preE := I$. The post-expectation achieved is $[n = 0] * p^n * [n \geq 0 \wedge N! = f * n!] \Rightarrow [N! = f]$.

We conclude with

$$\begin{aligned}
 & wp.prog.[N! = f] \\
 \Leftarrow & wp.init.preE \\
 \equiv & (p^n * [n \geq 0 \wedge N! = f * n!]) \langle n, f \mapsto N, 1 \rangle \\
 \equiv & p^N * [N \geq 0 \wedge N! = 1 * N!] \\
 \equiv & p^N * [N \geq 0],
 \end{aligned}$$

i.e. that the factorial is correctly calculated with probability at least p^N provided $N \geq 0$.

2.6 The Zero-One Law for termination

In this section we look more carefully at probabilistic termination on its own, showing that Case 3 of Lem. 2.4.1 implies a “Zero-One law” which, in turn, will justify our approach to probabilistic variants in Sec. 2.7.

Let Inv be some standard invariant for our usual (probabilistic) loop, so that

$$G \wedge Inv \Rightarrow wp.body.Inv, \quad (2.10)$$

and suppose we have additionally that

$$\varepsilon [Inv] \Rightarrow T \quad {}^{24} \quad (2.11)$$

for some fixed real $\varepsilon > 0$, where as usual $T := wp.loop.1$. Informally (2.11) says that from every state satisfying Inv the probability of loop termination is at least ε .

We now show that under these conditions the probability of termination from Inv is not just ε — *in fact it is one*.

Define expectation $I' := \varepsilon [Inv]$; we then have

$$\begin{aligned} & [G] * I' \\ \equiv & [G] * \varepsilon [Inv] \\ \equiv & \varepsilon [G \wedge Inv] \\ \Rightarrow & \varepsilon * (wp.body.[Inv]) \quad {}^{25} \quad \text{from (2.10)} \\ \equiv & wp.body.(\varepsilon [Inv]) \quad \text{scaling for } body, \text{ Lem. 1.6.2} \\ \equiv & wp.body.I', \end{aligned}$$

so that I' is also an invariant of $loop$. But from (2.11) we have $I' \Rightarrow T$; and so we continue

$$\begin{aligned} & [Inv] \\ \equiv & I' / \varepsilon \\ \Rightarrow & (wp.loop.([G] * I')) / \varepsilon \quad \text{Lem. 2.4.1 with invariant } I' \\ \equiv & wp.loop.([G] * I' / \varepsilon) \quad \text{scaling for } loop \\ \equiv & wp.loop.[G] \wedge Inv, \end{aligned}$$

²⁴As in normal algebra, we sometimes leave out the explicit multiplication sign...

²⁵... but here we use the multiplication sign explicitly since otherwise $\varepsilon(\dots)$ might look like the traditional function application (for which however we write an explicit dot symbol “.”).

showing that from any state satisfying Inv the loop terminates almost certainly in a state satisfying $\overline{G} \wedge Inv$.

In passing, we note that the above reasoning holds even for properly probabilistic invariants I — *i.e.* not just those of the form $[Inv]$ — and so we have shown that Case 3 of Lem. 2.4.1 can be relaxed to

$$\begin{aligned} 3a. \quad & \text{If } \varepsilon * I \Rightarrow T \text{ for some fixed } \varepsilon > 0, \\ & \text{then define } preE := I. \end{aligned} \tag{2.12}$$

(Recall Footnote 12 on p. 43.) That appears to be a very weak condition, since ε is arbitrary; in fact it requires only that in those states where I is nonzero, the termination probability T cannot be arbitrarily small.

If we continue to concentrate on standard invariants Inv , then the general effect we are seeing can be summarised as a *Zero-One law* for probabilistic processes, stated informally as follows.

Lemma 2.6.1 ZERO-ONE LAW FOR PROBABILISTIC PROCESSES

Let a non-aborting process P act over some state space,²⁶ and suppose that from every state in a subset of states (described by) Inv the probability of P 's eventual escape from Inv is at least ε , for some fixed $\varepsilon > 0$.²⁷

Then P 's escape from Inv is almost certain: it escapes with probability one.

Proof There are many zero-one laws in probability theory, of which ours here is a special case [GS92, p. 189ff]. \square

We can say more succinctly that the infimum over subset Inv of eventual escape probability is either zero or one: it cannot lie properly in between. (That is why Lem. 2.6.1 is called a “Zero-One” law.)

Note that we do not require that for every state in Inv the probability of *immediate* escape is at least ε — that is a stronger condition, from which the certainty of eventual escape is much more obvious (since it is trivially at least $1 - \varepsilon^n$ over n steps).

2.7 Probabilistic variant arguments for almost-certain termination

Termination of standard loops is conventionally shown using “variants” based on the state: they are integer-valued expressions over the state variables, bounded below but still strictly decreased by each iteration of the

²⁶The non-aborting condition is not usually imposed in the literature, where “aborting” processes do not occur. We mention it to ensure that this lemma is not applied inappropriately to program fragments containing **abort**.

²⁷Note that these conditions are formalised exactly by (2.12) for $I := [Inv]$.

loop. That method is complete (up to expressibility) since informally one can always define a variant

the largest number of iterations possible from the current state,

which satisfies the above conditions trivially if the loop indeed terminates.

For probabilistic programs however the standard variant method is not complete (though clearly it remains sound): for example the program

$$\begin{array}{l} \mathbf{do} \ (n \bmod N) \neq 0 \rightarrow \\ \quad n := n + 1 \quad \textstyle\frac{1}{2} \oplus \quad n := n - 1 \\ \mathbf{od} \end{array} \quad (2.13)$$

is almost certain to terminate, yet from the fact that its body can both increment and decrement n it is clear there can be no strictly decreasing variant.

With the results of the previous section however we are able to justify the following variant-based rule for probabilistic termination, sufficient for many practical cases including (2.13).²⁸

Lemma 2.7.1 VARIANT RULE FOR LOOPS Let V be an integer-valued expression in the program variables, defined at least over all states satisfying some predicate Inv . Suppose further for our usual *loop* that

1. if the set of states satisfying $G \wedge Inv$ is not finite, then there are fixed integer constants L (low) and H (high) such that

$$G \wedge Inv \quad \Rightarrow \quad L \leq V < H, \quad ^{29}$$

and

2. the predicate Inv is a (standard) invariant for *loop* and
3. for some fixed probability $0 < \varepsilon \leq 1$ and for all integers N we have

$$\varepsilon * [G \wedge Inv \wedge (V=N)] \quad \Rightarrow \quad wp.body.[V < N].$$

Then termination is certain from any state in which Inv holds: we have $[Inv] \Rightarrow T$, where T is the termination condition of *loop*.

Proof Informally we argue as follows.

The probability of V 's eventual escape from any point in the interval $[L..H)$ cannot be less than ε^{H-L} , since that is the (possibly remote, but still nonzero) probability of the $H - L$ consecutive decrements that would

²⁸Later, in Chap. 7, we show it complete over finite state spaces: any finite-state loop which terminates with probability one must have a probabilistic variant that shows it does.

²⁹Note that either way the existence of such L, H is guaranteed — but in the finite case there is no need to say explicitly what they are. The condition amounts to requiring that V take only finitely many values over $G \wedge Inv$.

cause eventual escape.³⁰ We then appeal to Sec. 2.6, using ε^{H-L} here for ε there.

See Chap. 7 for a full proof. \square

Lem. 2.7.1 shows termination given an integer-valued variant bounded *above and below* such that on each iteration a strict decrease is guaranteed *with at least fixed probability* $\varepsilon > 0$.³¹ Note that the probabilistic variant *is allowed to increase* — but not above H . (We have emphasised the parts that differ from the standard variant rule.)

The termination of Program (2.13) now follows immediately from Lem. 2.7.1 with variant $n \bmod N$ itself, and $L, H := 0, N$ — trivially, every iteration strictly decreases $n \bmod N$ with probability at least $1/2$.³²

In some circumstances it is convenient to use other forms of variant argument, related to Lem. 2.7.1; one easily proved from it is the more conventional rule in which

the variant is bounded below (but not necessarily above); it must decrease with fixed probability $\varepsilon > 0$; and it cannot increase.

That rule follows (informally) from Lem. 2.7.1 by noting that since the variant cannot increase its initial value determines the upper bound H required by the lemma, and it shows termination for example of the loop

$$\begin{array}{l} \text{do } n > 0 \rightarrow \\ \quad n := n - 1 \quad 1/2 \oplus \text{ skip} \\ \text{od} , \end{array}$$

for which variant n suffices with $L := 0$.

2.8 Termination example: self-stabilisation

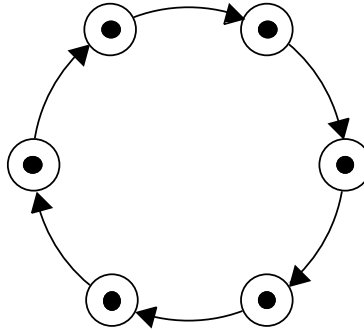
As an example of the variant technique, we apply Lem. 2.7.1 to Herman's *probabilistic self-stabilisation* [Her90], a distributed probabilistic algorithm that can be used for leadership election in a ring of synchronously executing processors.³³

³⁰For integers a, b , we write $[a..b)$ for the closed-open interval of integers i satisfying $a \leq i < b$.

³¹Since the variant is bounded above as well as below, we might just as well guarantee a strict *increase* with some probability; and in some problems that is more convenient. Clearly the two alternatives are of equal power.

³²When $n \bmod N = N - 1$, in fact decrease is certain, *i.e.* has probability greater than $1/2$; but “at least” is sufficient. Note that in Program (2.13) the state space is not finite, which is why we must give L, H explicitly.

³³In Herman's original presentation each processor contains a single bit, and the aim is via pairwise-local and symmetric processing to reach a stable global configuration of those bits (modulo rotation around the ring). ...



In this example we have $N = 6$ and, initially, one token in each processor.

Figure 2.8.1. HERMAN'S RING

We consider N identical processors connected clockwise in a ring, as illustrated in Fig. 2.8.1. A single processor — a *leader* — is chosen from them in the following way. Initially each processor is given exactly one token; the leader is the first processor to obtain all N of them.

Fix some probability p with $0 < p < 1$. On each step (synchronously) every processor performs the following actions:

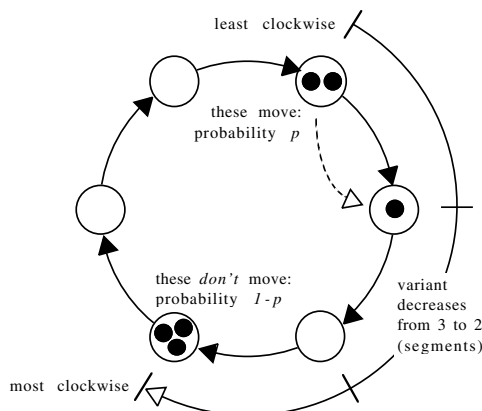
- Make a local probabilistic decision either to *pass* (probability p) or to *keep* (probability \bar{p}) all its tokens.
- If *passing*, then send *all* its tokens to the next (clockwise) processor; if *keeping*, do nothing.
- Receive tokens passed (if any) from the previous (anticlockwise) processor, adding them to the tokens currently held (if any).

We show easily that with probability one eventually a single processor will have all N tokens. We define

- the invariant *Inv* to be that the total number of tokens is constant (at N),
- the guard *G* (true while the computation continues) to be that more than one processor holds tokens and
- the variant *V* to be the shortest length of any ring segment (contiguous sequence of arcs) containing all tokens. (See Fig. 2.8.2.)

...³³We transform (data-refine) the algorithm, first by introducing tokens to represent the equality or difference of adjacent processors' bits, and then by allowing the tokens to accumulate rather than requiring them to annihilate each other when they meet.

Our termination argument first appeared in the article on which this chapter is partly based [Mor96].



The variant is the length of a shortest segment containing all tokens. It decreases with nonzero probability at least $p * \bar{p}$ — but it might also go up.

Figure 2.8.2. HERMAN'S-RING VARIANT

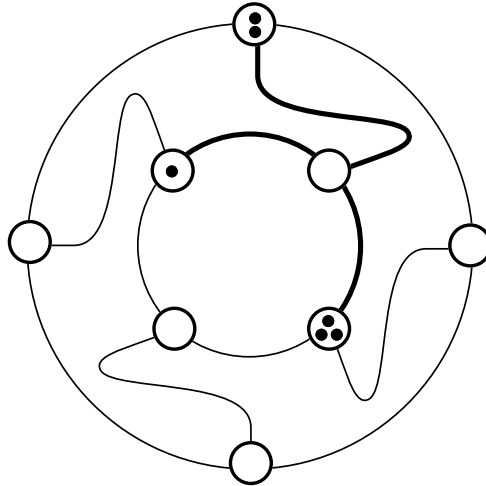
With those definitions, for proof of termination we simply refer to Lem. 2.7.1:

- the state space is finite (so that we need not bound our variant),
- the invariant Inv is trivially maintained and
- the variant V decreases strictly with probability at least $p * \bar{p}$, which is nonzero since p itself is neither zero nor one — let the trailing (least-clockwise) processor in the shortest segment decide to *pass* while the leading (most-clockwise) processor decides to *keep*.

Thus Lem. 2.7.1 has given us almost-certain termination: that with probability one eventually only one processor contains tokens (\bar{G}), and that it has all N of them (Inv).³⁴

³⁴In Herman's original algorithm each processor has at most one token — and when two tokens collide, both disappear. The argument for termination in a state where exactly one processor has (now only) a single token is as above, but with invariant "the number of tokens is odd, and each process has at most one." (Thus the number of tokens must be odd initially, to establish the invariant.)

See Footnote 67 on p.74 where we give the exact expected (quadratic) number of steps to termination for the case where there are initially three tokens. Elsewhere, we prove an expected worst-case steps-to-termination result of $O(N^2)$ [MM04a]; together with the quadratic result for three tokens, that gives expected complexity of $\Theta(N^2)$.



For each node, the probability that it passes all of its tokens to any directly connected neighbour (including itself) is at least some constant $p > 0$.

The variant is the smallest number of edges in any connected sub-graph that contains all tokens. It decreases with probability at least $p^N = p^{N-1} * p$, a lower bound on the probability that all nodes keep their tokens except one, which instead passes all its tokens to its unique neighbour in the graph.³⁵

With probability one, all tokens collect in a single processor.

Figure 2.8.3. HERMAN'S-GRAPH VARIANT

2.8.1 Variations on the ring

The variant technique makes it easy to see that the use of a ring above is not essential for correctness: there are many other possibilities. In Fig. 2.8.3, for example, we see that in an arbitrary *undirected* connected graph, the variant is simply the smallest number of edges of any connected sub-graph containing all tokens.³⁵

For a *directed* and strongly connected graph,³⁶ a suitable variant would be the length of a shortest directed path containing all tokens. (Our original ring is a special case of this; in general, however, there could be repeated edges.) Call a *min-straggler* any processor at the trailing end of some directed all-token-covering path of minimum length: with nonzero prob-

³⁵That sub-graph must be acyclic, since otherwise its size could be decreased, without losing any tokens, by breaking a cycle; if it is acyclic then it must contain a univalent node; any univalent node must contain tokens, else it and its edge could be removed.

³⁶That is, there is a directed path from any processor to every other.

ability, on any synchronous step it decreases the variant by passing all its tokens to the processor next along, while the other processors retain theirs.

If we consider *asynchronous* networks, then we must include an assumption of fairness. Unfortunately, *almost fairness* — in which the probability of the scheduler ignoring any processor forever is zero — is not enough in this case.³⁷ For example, in our original but now asynchronous directed ring, with just two tokens the almost-fair “schedule each processor until it passes its tokens one step clockwise” could avoid termination by circulating the tokens at a constant separation forever.³⁸

But there are many variations of fairness, and one can simply choose one — or even invent one — “strong enough to do the job.” In this case we propose so-called “*k*-fairness,” requiring that there is some fixed integer *k* (no matter how large) such that no processor is overlooked more than *k* times in a row.³⁹ For termination we then argue intuitively that any execution sequence can be divided into blocks of length *k* + 1 steps, and within each block the argument as for the original synchronous case applies: there is a nonzero probability that during the block the size of some separation measure will decrease.

More precisely, let the graph be directed and strongly connected. Say that the *priority* of a processor is the number of steps since it was last scheduled; and define

$$\begin{aligned} V_0 &:= \text{the smallest length of any directed path containing} \\ &\quad \text{all tokens,} \\ \text{and } V_1 &:= \text{the maximum priority among all min-stragglers.} \end{aligned}$$

Now V_0 can take only finitely many values; and V_1 , bounded below by zero, is bounded above by *k* due to our *k*-fairness scheduling policy. Thus construct the overall variant $V := (k + 1)V_0 - V_1$,⁴⁰ and observe that on each step the processor scheduled is either

- a min-straggler, in which case with nonzero probability it passes its tokens along the min-path, and the expression $(k + 1)V_0$ decreases by *k* + 1 (and $-V_1$ can increase by no more than *k*); or the processor is
- not a min-straggler, in which case with nonzero probability it keeps its tokens so that $-V_1$ decreases and V_0 is unchanged.⁴¹

³⁷It is however sufficient for the *Dining Philosophers* of Sec. 3.2.

³⁸See however Footnote 17 on p.93 concerning the dependence of the scheduler’s almost-fairness on the algorithm it is scheduling.

³⁹Unlike most forms of fairness, this *k*-fairness is a safety property.

⁴⁰Using two or more variants like this, “layered” one within the other, is discussed more generally in Sec. 3.1.4.

⁴¹Since in this case the processor keeps its tokens, the overall configuration is unchanged and — in particular — whether or not some other processor is a min-straggler is unchanged also. Thus since no min-straggler was scheduled, V_1 must increase (by one).

Thus in an arbitrary finite, directed and strongly connected token-graph, if each processor when scheduled passes all its tokens to some directed neighbour (including possibly itself), with some nonzero probability for each neighbour, then even under demonic asynchronous scheduling the tokens are almost certain to end up all together in one processor — provided that the scheduling is k -fair.⁴²

Finally, we mention *probabilistic* fairness, where on each asynchronous step there is a constant nonzero lower-bound probability that any given processor will be scheduled. This gives a very direct and simple argument: the nonzero probability of variant decrease is just the probability that the min-straggler will be selected *times* the probability that it passes its tokens upstream.

Although probabilistic fairness might be expensive to implement, for “real-world” scheduling caused by natural phenomena it might be a reasonable assumption.

2.9 Uncertain termination

In the arguments of Sec. 2.5, the cases where termination was “uncertain,” *i.e.* occurred with guaranteed probability strictly less than one, were made by inspection. We are not aware of a general “variant-style” approach in such circumstances, and in many cases informal arguments suffice anyway. For completeness, however, we set out here the details of a straightforward and rigorous inductive argument based on the fact that wp for loops is defined as a least fixed-point.

We know for our usual $G, body$ loop that we have

$$wp.loop.1 \equiv (\sqcup k: \mathbb{N} \bullet T_k) \quad ^{43} \tag{2.14}$$

$$\begin{aligned} \text{where } T_0 &:= [\overline{G}] \\ T_{k+1} &:= 1 \text{ if } \overline{G} \text{ else } wp.body.T_k, \end{aligned}$$

⁴²In Sec. 3.2.1 we call demonic scheduling “adversarial.”

⁴³We write \mathbb{N} for the NATURAL NUMBERS, the non-negative integers including zero.

The comprehension $(\sqcup k: \mathbb{N} \bullet T_k)$ gives the quantifier \sqcup , then the bound variable k and the set \mathbb{N} from which it is drawn, then — following the “ \bullet ” separator — the expression T_k , evaluated over all values of the bound variable within the given set, to which the quantifier is applied. The parentheses are always used: they give the scope of the bound variable.

a fact derived directly from the expression of least fixed-points as \mathbb{N} -suprema, which in turn follows from the fact that programs denote continuous expectation-transformers.⁴⁴ Each T_k term is a lower bound for the probability of termination in no more than k executions of the loop body; and, in practice, using the above amounts to a formalisation of the inductive argument that would have been applied by inspection — as we now illustrate.

2.9.1 Example: an inductive termination argument

We consider the program

$$\mathbf{do} \ n \neq 0 \rightarrow \ n := 0 \ \oplus_{1/n^2} \ n := n + 1 \ \mathbf{od} \ ,$$

whose probability of termination lies strictly between zero and one. By inspection we guess that the loop *fails* to terminate from initial state $n = N$ with probability given by the infinite product

$$\left(\prod_{n=N}^{\infty} 1 - 1/n^2 \right) ,$$

and in Case $N = 2$ we find we are looking at

$$3/4 * 8/9 * 15/16 * 24/25 * \dots \ , \quad (2.15)$$

whence we notice that the partial products

$$\begin{array}{ccccccc} & 3/4 & , & 2/3 & , & 5/8 & , & 3/5 & \dots \\ & \uparrow & & \uparrow & & \uparrow & & \uparrow & \\ \text{term} & 2 & & 3 & & 4 & & 5 & , \end{array}$$

suggest $1/2 + 1/2n$ for the n^{th} term — a fact easily verified by induction over n and giving just $1/2$ in the limit as n tends to infinity.

In Case $N = 3$ the limit would be $4/3$ of that (to remove the first multiplicative term $3/4$ in the product (2.15) above), *i.e.* $2/3$ and which suggests $1 - 1/N$ in general; again this is easily verified by induction (but this time on N).

Thus we guess that the probability of termination from the initial state in which $n = N$ is just $1/N$, and that therefore for any $n > 0$ the probability p of termination within k steps satisfies the equation

$$1/n = p + \bar{p}/(n+k) \ ,$$

expressing that termination overall ($1/n$) is either termination within k steps (p) or termination at step k or after ($\bar{p}/(n+k)$) — which gives $k/(n * (n+k-1))$ for p .

⁴⁴From (1.19) we have $wp.loop.1 \equiv (\mu Q \bullet wp.body.Q \ \mathbf{if} \ G \ \mathbf{else} \ 1)$, whose right-hand side as a function of Q generates the inductive definition of T_k above when applied repeatedly starting from zero.

So far, the reasoning is informal (though convincing).

If we need absolute formality, we would use the above to hazard that for $k \geq 1$ we have

$$T_k \equiv k/(n * (n + k - 1)) \text{ if } n \neq 0 \text{ else } 1, \quad (2.16)$$

which — rigorously — we can prove by induction over k using (2.14). The equality $T_0 \equiv [n = 0]$ comes from (2.14) directly; note that for $k \geq 1$ the conditional $n \neq 0$ avoids our having to consider the indeterminate $1/0$ in (2.16).

For $k + 1$ we now reason

$$\begin{aligned} & T_{k+1} \\ \equiv & 1 \text{ if } n = 0 \text{ else } wp.body.T_k & (2.14) \\ \equiv & \text{inductive hypothesis (2.16)} \\ & 1 \text{ if } n = 0 \text{ else } wp.body.(k/(n * (n + k - 1)) \text{ if } n \neq 0 \text{ else } 1) \\ \equiv & \text{definition } wp.body; \text{ arithmetic}^{45} \\ & 1 \\ & \text{if } n = 0 \text{ else} \\ & 1 \text{ }_{1/n^2 \oplus} (k/((n + 1) * (n + k)) \text{ if } n + 1 \neq 0 \text{ else } 1) \\ \equiv & 1 \text{ if } n = 0 \text{ else } 1 \text{ }_{1/n^2 \oplus} k/((n + 1) * (n + k)) \quad n \geq 0 \text{ is invariant} \\ \equiv & 1 \text{ if } n = 0 \text{ else } (k + 1)/(n * (n + k)), \quad \text{expanding } 1/n^2 \oplus \end{aligned}$$

as required to prove (2.16) for all k . Thus — from (2.14) — we have

$$(\sqcup k: \mathbb{N} \bullet k/(n * (n + k - 1)) \text{ if } n \neq 0 \text{ else } 1),$$

that is

$$1/n \text{ if } n \neq 0 \text{ else } 1$$

as a lower bound for the overall termination probability $wp.loop.1$.

2.10 Proper post-expectations

Thus far we have concentrated mainly on probabilistic correctness, that is the (worst-case) probability that a given program will establish some postcondition. It is clear, though, that our reasoning tools can be applied to what we will call *proper* post-expectations, the more general case where

⁴⁵For convenience — and by analogy with programs — we write $exp_1 \text{ }_p \oplus \text{ }_p exp_2$ for $p * exp_1 + \bar{p} * exp_2$.

they are not of the form $[post]$ — and that in turn allows us to ask more general questions about programs’ expected behaviour.

For example, if we were interested in a program’s running time, in terms of the number of loop iterations, we could include an extra variable n in the program text and arrange for it to be incremented on each iteration.⁴⁶ Its final value then gives the number of iterations required, calculated “in parallel” with whatever calculation the program was designed to perform.⁴⁷ To prove an upper-bound N for that complexity, we would write $n \leq N$ for the postcondition.

In the probabilistic setting the same technique applies — but we have extra opportunities as well. A post-expectation of $[n \leq N]$ delivers as a pre-expectation the probability that no more than N steps are taken — but a post-expectation of just n itself will deliver a lower bound for the *expected* number of steps, which is something we could not do before.⁴⁸

As our first example of proper post-expectations, we return to the martingale program of Sec. 2.5.1.

2.10.1 The martingale revisited

We modify our earlier program (Fig. 2.5.1) so that termination is certain, occurring either when the gambler finally wins a bet or when he runs out of money; the new version is shown in Fig. 2.10.1.

Note informally that on each iteration the gambler’s expected win is exactly zero: with probability $1/2$ he gains b ; and with probability $1/2$ he loses b . Our demonic/pessimistic wp establishes easily that indeed his expected win is no less than zero:

$$\begin{aligned} & c \\ \cdot \equiv & c + 2b \quad \text{ }_{1/2} \oplus \quad c \quad \text{applying } wp.(c, b := c + 2b, 0 \quad \text{ }_{1/2} \oplus \quad b := 2b) \\ \cdot \equiv & c + b \quad \text{ }_{1/2} \oplus \quad c - b \quad \text{applying } wp.(c := c - b) \\ \equiv & c . \end{aligned}$$

That is, by showing $c \equiv wp.body.c$ we verify that his expected capital c after a bet is at least its value before the bet.

The “at least” in the previous sentence comes not from an inequality in the calculation — for all the relations were \equiv — but rather from the

⁴⁶This is the approach advocated for example by Hehner [Heh89] for treatment of program termination in general.

⁴⁷In fact this is an example of the well-known technique of program SUPERPOSITION, where an extra “monitoring” layer is superposed on the original computation without disturbing it. Superposition is, in turn, an example of *data refinement*, which we discuss in Chap. 4.

⁴⁸One could argue that an upper bound is more likely to be useful, but lower bounds are what wp delivers and so — at least for the moment — we are stuck with that. But at (2.21) further below we show how to adapt our techniques to determine upper and exact bounds as well.

$$\begin{array}{ll}
\textit{init} & \rightarrow \quad c, b := C, 1; \\
\textit{loop} & \rightarrow \quad \mathbf{do} \ 0 < b \leq c \rightarrow \\
\textit{body} & \rightarrow \quad \left\{ \begin{array}{l} c := c - b; \\ c, b := c + 2b, 0 \end{array} \right. \text{ }_{1/2} \oplus \text{ } b := 2b \\
& \mathbf{od}
\end{array}$$

The program *prog* is the whole of the above.

The gambler's capital c is initially C , and his intended bet b is initially one.

On each iteration, if his intended bet does not exceed his capital, he is allowed to place it and has $1/2$ chance of winning.

If he wins, he receives twice his bet in return, and sets his intended bet to 0 to indicate he is finished; if he loses, he receives nothing and doubles his intended bet — hoping to win next time.

The gambler stops either because his bet has been set to zero (because he won) or because his intended bet has become more than he can afford.

Figure 2.10.1. THE MARTINGALE, REVISITED

inherent pessimism built-in to *wp*, which resolves demonic choice towards lower pre-expectations. In this *deterministic* program, however, there is no demonic choice, and so the “at least” is overly cautious: knowing that we have calculated the exact greatest pre-expectation, and that the program is deterministic, we can take the result as exact. We formalise that as follows.

To verify — with less hand-waving — that his expected winnings are indeed no *more-* (as well as no less) than zero, as a coding trick we take post-expectation $C + 1 - c$, that is how far he has to go towards his goal of an overall increase of one unit. Almost the same arithmetic as above shows

$$C + 1 - c \equiv wp.\textit{body}.(C + 1 - c), \quad (2.17)$$

which as before establishes a lower bound of zero, this time on the expected change in $C + 1 - c$. But that becomes an *upper* bound of zero on the expected change in c itself, since $C + 1$ is constant and c occurs in a negative sense — and so we have that his expected winnings in one iteration are *exactly* zero.

Now there are two issues raised by the technique illustrated at (2.17): the first $(\dagger)^{49}$ is that tricky expectations like the $C + 1 - c$ above must still

⁴⁹We employ symbols \dagger, \ddagger for “local” cross-references; the target is indicated with a \dagger in the margin, and occurs within a few paragraphs of the point (\dagger) of reference.

be non-negative and bounded above;⁵⁰ and the second (§) is that we don't want to repeat trivial calculations as at (2.17) if we have effectively done them already.

† For the first issue, in fact we “know” that $C + 1 - c$ is non-negative because of the standard loop-invariants $c + b = C + 1$ (recall p. 45) and $b, c \geq 0$ (easily verified here). This appealing style of “reasoning within a standard invariant” is an extension of the modular reasoning we saw earlier in Sec. 1.7, but now applied to the multiple iterations of a loop rather than to a single program fragment. Both standard predicates are now the single *Inv*, which we call an *auxiliary* invariant. Its use is justified by the following lemma, whose proof is similar to the proof of Lem. 1.7.1:

Lemma 2.10.2 REASONING WITHIN A LOOP INVARIANT For our usual loop G, body let both $[Inv]$ and I be invariants, separately. That is, suppose we have

$$\begin{array}{lll} G \wedge Inv & \Rightarrow & wp.\text{body}.Inv \\ \text{and } [G \wedge Inv] * I & \Rightarrow & wp.\text{body}.I \end{array} \quad 51$$

Then in fact $[Inv] * I$ is an invariant of the loop as well. \square

The importance of Lem. 2.10.2 is that it justifies our using the two invariants separately: we use invariant $C+1-c$ to investigate the post-expectation of primary interest; and separately we prove invariance of an auxiliary, “well-formedness” condition for it, in this case the predicate

$$c + b = C + 1 \quad \wedge \quad b, c \geq 0$$

which guarantees $0 \leq (C + 1 - c) \leq C + 1$. (See p. 70 however for further comments on this calculation.)

‡ For the second issue, we note that *deterministic* programs satisfy the stronger property of *linearity* (as opposed to sublinearity alone): if *prog* is (pre-)deterministic, then we have

$$\begin{aligned} & wp.\text{prog}.(c_1 * \text{post}E_1 + c_2 * \text{post}E_2) \\ \equiv & c_1 * wp.\text{prog}.\text{post}E_1 + c_2 * wp.\text{prog}.\text{post}E_2, \end{aligned} \quad (2.18)$$

for non-negative reals c_1, c_2 and expectations $\text{post}E_1, \text{post}E_2$.⁵² A sufficient — but not necessary⁵³ — condition for a program to be deterministic is of course that it contains no occurrence of \square .

⁵⁰We have not emphasised “bounded above” so far, though it is one of our assumptions about expectations generally. Below we will see that it is sometimes crucial.

⁵¹Note how we use the auxiliary *Inv* to help prove the invariance of *I*.

⁵²We return to this, strengthening it, at Def. 8.3.2.

⁵³Program $x := 1 \square x := 1$ is deterministic, for example.

For deterministic *and terminating* programs $prog$ we can therefore avoid repeating calculations like (2.17); for if

$$preE \equiv wp.prog.postE \quad ^{54} \quad (2.19)$$

and if — as above — we can appeal to a standard invariant Inv assuring $postE \leq H$ for some constant H , then we reason for any state satisfying Inv that

$$\begin{aligned} & wp.prog.(H - postE) \\ =^{55} & wp.prog.H - wp.prog.postE && \text{refer (2.18)} \\ = & H * wp.prog.1 - wp.prog.postE && \text{scaling} \\ = & H - wp.prog.postE && \text{prog is terminating} \\ = & H - preE. && \begin{array}{l} \text{refer (2.19):} \\ \text{note that } \equiv \text{ there is necessary} \\ \text{to avoid inappropriate } \leq \text{ here} \end{array} \end{aligned}$$

Thus from (2.19) we know directly that the expected final value of $postE$ is *at least* $preE$; and from the subsequent calculation we know that the expected final value of $H - postE$ is at least $H - preE$, *i.e.* that the expected final value of $postE$ is *at most* $preE$ (because H is constant). Thus the expected final value of $postE$ is exactly $preE$.

To summarise, we can say that (2.19) means that $preE$ is the exact expected final value of $postE$ whenever $prog$ is deterministic and terminating, and $postE$ is bounded above (possibly within some auxiliary invariant).

For loops as a whole, unfortunately, we do not usually have an exact pre-expectation like (2.19) to work from — even when the loop is deterministic. That is because the conclusion of our loop rule Lem. 2.4.1 is intrinsically an inequality $preE \Rightarrow \dots$. We can however reach similar conclusions if our invariant is *exact* in this sense, that whenever the guard G holds (and possibly some auxiliary invariant as well) we have

⁵⁴Note that the equality \equiv does *not* mean “the expected final value of $postE$ after $prog$ is exactly $preE$ ” — that is, after all, precisely what we are trying to establish. Rather it means “the GREATEST LOWER BOUND of the possible expected final values of $postE$ is exactly $preE$ ”. The significance of the \equiv is that it is necessary for the conclusions we are about to draw.

$$I = wp.body.I . \quad ^{55} \quad (2.20)$$

For an exact and bounded invariant I , provided the loop body is deterministic and terminating, we therefore have that

$$\begin{aligned} &\text{in any initial state from which loop termination is guaran-} \\ &\text{teed, an exact invariant } I \text{ gives the exact expected value of} \quad (2.21) \\ &\text{any expression } postE \text{ that agrees with } I \text{ everywhere on } \bar{G}. \quad ^{56} \end{aligned}$$

Thus finally we can return to Fig. 2.10.1. We have shown that c is an exact invariant, and that it is bounded implicitly within the accompanying standard invariant; and it is easy to see that termination of the loop is certain (because variant b is set to zero with probability $1/2$ on each iteration). Thus we conclude that the gambler's expected final capital c is in fact C exactly, which is just what he started with.

What he had hoped for, of course, was that $C + 1 \Rightarrow wp.prog.c$ — but that is not what he got.

2.11 Bounded *vs.* unbounded expectations

We conclude the chapter by looking more closely at the importance — or not — of expectations' being non-negative and bounded, constraints which we have so far accepted because they assure well-definedness of wp in a straightforward way. At the core of our approach is the operation of *taking the expected value*, for us of a so-called post-expectation over a distribution of final states realised by executing a program. Even if the program reaches a potentially infinite set of final states, that expected value is well defined for all non-negative and bounded post-expectations — which is exactly why we impose those conditions.

⁵⁵As noted in Footnote 51 on p.33, ordinary “=” is appropriate in these two places because the equality is contingent on the preceding text, “whenever Inv holds” or “whenever G holds.”

In the latter case, we must avoid the stronger $[G] * I \equiv wp.body.I$ that first comes to mind, since it would not usually be true: equality outside G is irrelevant, because at the beginning of every iteration G holds unconditionally. In fact what we are using is

$$[G] * I \equiv [G] * wp.body.I .$$

⁵⁶Saying the loop body must be deterministic is strictly more demanding than saying that the loop as a whole must be deterministic, since demonic choice “along the way” can be cancelled out by the time the loop terminates. Consider

$$\begin{aligned} \text{do } x > 0 \rightarrow & \quad = \quad x := x \min 0 . \\ & x := x - 1 \sqcap x - 2; \\ & x := x \max 0 \\ \text{od} \end{aligned}$$

$$\begin{array}{ll}
\textit{init} & \rightarrow \quad b, n := \text{true}, 0; \\
\textit{loop} & \rightarrow \quad \mathbf{do} \, b \rightarrow \\
\textit{body} & \rightarrow \quad \left\{ \begin{array}{l} b := \text{true}_{1/2} \oplus \text{false}; \\ n := n + 1 \end{array} \right. \\
& \quad \mathbf{od}
\end{array}$$

The program *prog* is the whole of the above; its termination with probability one is shown by variant $[b]$, which termination is used in both (\dagger , \ddagger) of the following analyses.⁵⁷

\dagger To establish the final distribution achieved by *prog*, we use post-expectation $[n = N]$; the pre-expectation will give us the probability that it is achieved, for any N .

The probability of establishing $n = N$ is (we guess) $[n < N] / 2^{N-n}$ if b is true, and just $[n = N]$ itself if b is false. That suggests the invariant

$$[n < N] / 2^{N-n} \quad \mathbf{if} \, b \, \mathbf{else} \quad [n = N] ,$$

which is readily verified and gives overall pre-expectation (using *wp.init*) of $[0 < N] / 2^N \, \mathbf{if} \, \text{true} \, \mathbf{else} \, [0 = N]$, that is just $1/2^N$ provided $N \geq 1$.

\ddagger For the expected value of n after running *prog* we use instead the post-expectation n itself, and we guess that its expected final value is $n + K$ for some constant K , provided b is true; if b is false, its expected final value is just n . That suggests invariant $(n + K) \, \mathbf{if} \, b \, \mathbf{else} \, n$, and to determine K we apply *wp.body*:

$$\begin{array}{ll}
& n + K \, \mathbf{if} \, b \, \mathbf{else} \, n \\
\cdot \equiv & n + 1 + K \, \mathbf{if} \, b \, \mathbf{else} \, n + 1 \quad \text{applying } wp.(n := n + 1) \\
\cdot \equiv & n + 1 + K_{1/2} \oplus n + 1 \quad \text{applying } wp.(b := \text{true}_{1/2} \oplus \text{false}) \\
\equiv & n + 1 + K/2 .
\end{array}$$

For exact invariance we must have $(n + K \, \mathbf{if} \, b \, \mathbf{else} \, n) = n + 1 + K/2$ whenever the guard b holds, for which $K = 1 + K/2$ is sufficient. That gives $K = 2$, and the expected final value of n is thus $0 + 2 \, \mathbf{if} \, \text{true} \, \mathbf{else} \, 0$ — that is, the loop runs for an expected two iterations.

Figure 2.11.1. A GEOMETRIC DISTRIBUTION

For mixed-sign or unbounded post-expectations, however, well-definedness is not assured: such cases must be treated individually. Consider the program of Fig. 2.11.1, for example, which has infinitely many potential final states — albeit with sharply diminishing probability as n increases.

⁵⁷We could write the program more simply as

$$n := 1; \, \mathbf{do} \, 1/2 \rightarrow n := n + 1 \, \mathbf{od} ,$$

but our current invariant/variant techniques do not apply to “probabilistic guards.” Section 7.7 shows how to extend them.

The program is well defined, terminating in state $n = N$ with probability $1/2^N$ for $N \geq 1$: it generates an instance of the *geometric distribution* over the final value of n . (See Fig. 7.7.9 however for a more concise way of writing the program.)

Now if we choose the mixed-sign post-expectation $(-2)^n$, its expected value over that distribution would be

$$\left(\sum_{N:=1}^{\infty} (-2)^N / 2^N \right) = -1 + 1 + -1 + 1 \cdots \quad (2.22)$$

That is, although *prog* itself may be well defined, the greatest pre-expectation $wp.prog.(-2)^n$ is not — and that is a good reason for avoiding mixed signs in general.

Stepping back from the brink, we can consider unbounded but same-sign, say non-negative expectations — and then the indeterminate sum (2.22) cannot occur. In fact, if we adjoin a formal ∞ to the non-negative reals, the operation of taking expected value is well defined even for unbounded expressions.⁵⁸ Thus for example in Fig. 2.11.1 we have

$$wp.prog.(2^n) \equiv \left(\sum_{N:=1}^{\infty} 2^N / 2^N \right) \equiv 1 + 1 + \cdots \equiv \infty.$$

Because of that well-definedness, we can discuss examples such as the martingale (Sec. 2.10.1) with post-expectation c which, though unbounded, still has a well-defined expected value if we allow ∞ . As explained in Lem. 2.10.2, we actually use the bounded invariant

$$[c + b = C + 1 \wedge b, c \geq 0] * c \quad (2.23)$$

in the loop rule Lem. 2.4.1, reflecting the fact that although c itself is unbounded, in practice it is effectively kept within bounds by other constraints built into the program. Since (2.23) is no more than (the unbounded) c , the pre-expectation calculated for it will also be a pre-expectation for c — which is what we are interested in.

In the case of the geometric distribution program of Fig. 2.11.1, however, we cannot use the same argument for the second analysis (\dagger) — for in that case n actually does increase potentially without bound, though with exponentially-low probability.

In the next section we look more closely at unbounded invariants and, in particular, whether and when we can use them safely.

⁵⁸Unbounded non-*positive* expectations can be similarly treated (using $-\infty$) and we will occasionally make use of them. The advantage of non-negative expectations, however, is that we know more about their algebra — for example that they satisfy sublinearity Def. 1.6.1 and its consequences (Sec. 1.6).

$$\begin{array}{ll}
\textit{init} & \rightarrow \quad n := 1; \\
\textit{loop} & \rightarrow \quad \mathbf{do} \ n \neq 0 \rightarrow \\
& \quad n := n + 1 \quad \textstyle\frac{1}{2} \oplus \ n - 1 \\
& \quad \mathbf{od}
\end{array}$$

The program *prog* is the whole of the above; it is an example of a *random walk* over the integers. It can be shown (Sections 3.3 and 10.4) that this program terminates with probability one.

Expectation n is trivially an invariant of *loop*; thus we might reason

$$\begin{array}{ll}
& wp.\textit{prog}.n \\
\Leftarrow & wp.\textit{prog}.([n = 0] * n) \quad \text{monotonicity of } wp.\textit{prog} \\
\Leftarrow & wp.\textit{init}.n \quad \text{loop rule Lem. 2.4.1 applied inappropriately } (\dagger) \\
\equiv & 1,
\end{array}$$

concluding that the expected value of n on termination is at least one, obviously incorrect since on termination n is exactly zero.

† The use of the loop rule is inappropriate because the invariant n is not bounded.

Figure 2.11.2. THE SYMMETRIC RANDOM WALK AS A COUNTER-EXAMPLE

2.11.1 Unbounded invariants: a counter-example

An important limitation remains, in spite of our ability to use unbounded post-expectations as at (2.23) above: it is that the loop rule Lem. 2.4.1 is sound only for bounded non-negative invariants, as the counter-example of Fig. 2.11.2 shows.⁵⁹

Unfortunately, for Fig. 2.11.2 there is no standard invariant of the random walk that we can use in the style of Lem. 2.10.2: we say that such invariants are *intrinsically* unbounded.

Yet we do wish to have access to unbounded invariants for performance measures, as in Fig. 2.11.1 where invariant $(n+K)$ **if** b **else** n is intrinsically unbounded as well — as indeed would be the case for any loop containing a “loop counter” variable for determining the expected number of a possibly unbounded number of iterations.⁶⁰

⁵⁹In fact it is only Case 2 that requires boundedness explicitly, because Cases 1 and 3 imply it.

⁶⁰Such “unboundedly iterating” loops do not occur in elementary treatments of standard sequential programs, where it is common to observe that from their terminating initial states all programs are “image-finite,” or have “only bounded nondeterminism” if everyday programming constructs are used [Dij76, Chap. 9]; semantically, that amounts to saying that such programs are *continuous* (here our Lem. 5.6.6, and corresponding to a similar property for standard programs [Dij76, Property 5 p. 72]). That is the reason, for example, that on p. 21 in Sec. 1.5.2 we insisted that the abbreviation for demonic choice require finiteness of the choice. ...

It can be shown that

a sufficient condition for an unbounded invariant to be used soundly in the loop rule (Case 2) is that the greatest expected value of $[G] * I$, that is of the invariant evaluated only “within the loop” as the body is about to execute, is guaranteed to tend to zero as the loop continues to iterate.⁶¹ (2.24)

For the geometric distribution we observe informally that the probability of Fig. 2.11.1’s exceeding N iterations is no more than $1/2^N$; that in N iterations the value of n cannot exceed N ; and thus that the expected value of $[G] * I$ in this case is no more than $N/2^N$, which indeed tends to zero as N increases.

A more interesting example is given in Fig. 2.11.3. Because $a + b + c$ is invariant in that program, the “original” state space is finite,⁶² a subset of the possible triples $\{a, b, c \mid 0 \leq a, b, c \leq A + B + C\}$.⁶³ Only the “superposed” counter n is unbounded.

It is thus a form of bounded two-dimensional random walk, and its termination is almost certain.⁶⁴ Indeed, since $A + B + C$ consecutive losses for any player is guaranteed to cause termination, the probability of continuing to play decreases exponentially.⁶⁵

...⁶⁰A significant and distinguishing feature of probabilistic programs is that unbounded *probabilistic* choice is common, and does not lose continuity.

The geometric-distribution program Fig. 2.11.1 is a simple example of that, since it can set n finally to any positive integer. Attempting however to replace its $1/2 \oplus$ with the closest standard equivalent \square , to produce an unboundedly nondeterministic *standard* program, simply results in **abort** instead.

⁶¹The “is guaranteed to” turns out to be important: if the loop body is demonic, then the limit of zero must apply for all its deterministic refinements. (If it’s deterministic itself, then the “for all refinements” is trivial of course.)

A sketch proof of (2.24) is given in Footnote 9 on p. 330.

⁶²The same applies to the geometric-distribution program.

⁶³We use the same syntax for set comprehensions as we do for comprehensions in general, except that the brackets $\{\cdot\cdot\}$ do “double duty,” both giving the scope of the bound variables and indicating that we are constructing a set. In this case the bound variables a, b, c vary over their implicit type \mathbb{N} but are further constrained by the predicate $0 \leq a, b, c \leq A + B + C$. Since no “ \bullet ” appears, the default expression is used, the triple (a, b, c) of bound variables as given.

⁶⁴The invariant $a + b + c$ gives it only two degrees of freedom (rather than three): the walker moves randomly over a triangle, until eventually — with probability one — he falls off one of the edges.

Interestingly, $(a - b) \bmod 3$ etc. are also invariant, which makes it clear that he can reach only a subset of the triangle.

⁶⁵The probability that a fixed player will not lose every one of a block of $A+B+C$ consecutive rounds is $p := 1 - (2/3)^{A+B+C}$; the probability that the player survives N such blocks is no more than p^N .

```

a, b, c: = A, B, C;
n: = 0;
do a ≠ 0 ∧ b ≠ 0 ∧ c ≠ 0 →
  a, b, c: = a-1, b-1, c-1;
   $\left\{ \begin{array}{l} a: = a+3 \quad @ \ 1/3 \\ b: = b+3 \quad @ \ 1/3 \\ c: = c+3 \quad @ \ 1/3; \end{array} \right.$ 
  n: = n+1
od

```

This program is based on the following game (but simplifies its presentation). Three gamblers start with A, B, C coins respectively, and on each round:

- each player commits one coin;
- the coins are flipped repeatedly until
- one player is “the odd man out,”
- and he collects all three coins.

The game is over when one player is broke; variable n counts the number of rounds until that happens.

Figure 2.11.3. THE “THREE-UP” GAME

Remarkably, the expression $n + abc/(a+b+c-2)$ is an exact invariant for the loop.⁶⁶ We have

$$\begin{aligned}
 & \cdot \equiv \frac{n + abc/(a+b+c-2)}{n+1 + abc/(a+b+c-2)} && \text{applying } wp.(n: = n+1) \\
 & \cdot \equiv n+1 + \frac{(a+3)bc + a(b+3)c + ab(c+3)}{3(a+b+c+1)} && \text{applying } wp. \text{ “three-way choice”} \\
 & \cdot \equiv n+1 + \frac{(a+2)(b-1)(c-1) + (a-1)(b+2)(c-1) + (a-1)(b-1)(c+2)}{3(a+b+c-2)} && \text{applying } wp.(a, b, c: = a-1, b-1, c-1) \\
 & \equiv n+1 + \frac{abc - (a+b+c) + 2}{a+b+c-2} && \text{arithmetic}
 \end{aligned}$$

⁶⁶This problem went unsolved for twenty-five years, until the correct invariant was discovered in 1966 [Hon03, p103]. It is presented as *a* solution to a recursion, including the comment “Since recursions can have more than one solution... we need to show that our result is the only [one]... by a standard argument in the theory of absorbing chains [*op. cit.*, p. 107].” Feller is cited (*e.g.* [Fel71]) for the more advanced techniques.

Our theory combining invariance and termination has this extra reasoning “built-in,” which may account for some of the intricacy of the proof of Lem. 2.4.1. We take advantage of it here by using *exact* invariants (2.21).

$$\equiv n + abc/(a+b+c-2) . \quad \text{arithmetic}$$

Now because the invariant increases only linearly (remember a, b, c are bounded above, so only n 's increase is important in the long run), while the probability of continuing decreases exponentially, we can appeal to (2.24) above to justify our use of the loop rule to conclude that the expected value of n on termination is given by the calculation

$$\begin{aligned} & n + abc/(a+b+c-2) && \text{invariant} \\ \cdot \equiv & abc/(a+b+c-2) && \text{applying } wp.(n := 0) \\ \cdot \equiv & ABC/(A+B+C-2) , && \text{applying } wp.(a, b, c := A, B, C) \end{aligned}$$

since the negated loop guard (a prerequisite of termination) zeroes the right summand of the invariant, leaving just n for the overall post-expectation. A game starting with *e.g.* 10 coins each would thus take $10^3/28 \simeq 35.7$ rounds on average.⁶⁷

For the *unbounded* random walk, however, we cannot argue as above to show that the expected value of $[n \neq 0] * n$ tends to zero as the number of iterations increases. Indeed, since n is invariant we know that the expected value of just n itself remains exactly its initial value one, no matter how many iterations have occurred; we know also, from simple arithmetic, that

$$n \equiv [n = 0] * n + [n \neq 0] * n ;$$

and trivially the left-hand summand $[n = 0] * n$ is identically zero. So the expected value of $[G] * I$, in this case $[n \neq 0] * n$, remains at its initial value of one, and does *not* tend to zero as the loop continues to iterate. Figure 2.11.4 illustrates that for the first few steps.

We return to the random walk in Sec. 3.3, where in a more general setting again we see that bounding the invariant is necessary.

2.12 Informal proof of the loop rule

Finally, the above discussion provides an intuitive explanation for the soundness of the most general Case 3a (at (2.12) on p. 54) of our loop rule Lem. 2.4.1 for bounded invariants, an argument which we now summarise for deterministic programs.

⁶⁷This result can be adapted to give an *exact* result for the expected time-to-termination of Herman's Ring (Sec. 2.8) in the case where the initial number of tokens is three and their separations are A, B, C places: it is $4ABC/(A+B+C)$ [MM04a].

Variable n (columns) is the random walker's distance from his goal, back at the origin where $n = 0$; his steps (rows) however can move away from as well as towards his goal.

He starts in Step 0, located at $n = 1$ with probability one.

Step	$n = 0$	1	2	3	4	5	6	7
0		1						
1	.5		.5					
2	.5	.25		.25				
3	.625		.25		.125			
4	.625	.125		.1875		.0625		
5	.6875		.1563		.125		.0313	
6	.6875	.0781		.1406		.0781		.0156
For $n \neq 0$, the expected value of n remains one...		* 1 ↓ .0781	+	* 3 ↓ .4219	+	* 5 ↓ .3906	+	* 7 ↓ .1094
								= 1.0

...even though the total probability
 of being in this region where $n \neq 0$
 tends to zero as the iterations continue...

...because the probability of termination tends to one.

After 6 steps, the walker has reached his goal $n = 0$ with probability .6875 and, as the number of steps increases, that probability will tend to one — so that the probability he is still in the region $n \neq 0$ tends to zero.

In spite of that, the expected value of n over the region $n \neq 0$ remains one no matter how many steps are taken; above for example we calculate the expected value $.0781 + .4219 + .3906 + .1094 = 1$ at Step 6.

This “paradoxical” fact — that the expected value of an expression can remain nonzero over a region whose distribution tends to zero in the limit — can occur only if the expression is unbounded. That is what accounts for the potential failure of the loop rule for unbounded invariants.

Figure 2.11.4. THE RANDOM WALK, TABULATED

1. The general condition (2.12) relating invariant I and termination probability T is that we should have $\varepsilon * I \Rightarrow T$ for some $\varepsilon > 0$.⁶⁸ Assume that holds.
2. Choose arbitrary $\varepsilon_1, \varepsilon_2 > 0$.
3. Since trivially $T \geq \varepsilon_1$ everywhere in the region $G \wedge (T \geq \varepsilon_1)$, by the Zero-One Law the probability of eventual escape from there is one.⁶⁹
4. Thus by iterating sufficiently often we can make the probability that we have not yet terminated, *but* that there remains a probability ε_1 that we eventually will, as small as we like. That is, from (3) we can choose N sufficiently large so that

$$\text{Pr}_n.(G \wedge (T \geq \varepsilon_1)) \leq \varepsilon_2$$

for all $n \geq N$, where by Pr_n we mean the probability determined by the distribution of states after n steps.

5. We now do some arithmetic: after those $n \geq N$ steps we have

$$\begin{aligned}
& \text{Exp}_n.([G] * I) \\
= & \text{Exp}_n.([G \wedge (T < \varepsilon_1)] * I) + \text{Exp}_n.([G \wedge (T \geq \varepsilon_1)] * I) \\
& \text{Exp}_n \text{ distributes +} \\
\leq & \text{Exp}_n.([G \wedge (T < \varepsilon_1)] * \varepsilon_1/\varepsilon) + \text{Exp}_n.([G \wedge (T \geq \varepsilon_1)] * I) \\
& I \Rightarrow T/\varepsilon \text{ from (1); } [T < \varepsilon_1] \text{ multiplier} \\
\leq & \varepsilon_1/\varepsilon + \text{Exp}_n.([G \wedge (T \geq \varepsilon_1)] * I) \quad [G \wedge (T < \varepsilon_1)] \Rightarrow 1 \\
\leq & \varepsilon_1/\varepsilon + \text{Exp}_n.[G \wedge (T \geq \varepsilon_1)] * (1/\varepsilon) \\
& I \Rightarrow T/\varepsilon \Rightarrow 1/\varepsilon \text{ from (1); Exp}_n \text{ scales} \\
\leq & (\varepsilon_1 + \varepsilon_2)/\varepsilon, \quad \text{expected value of characteristic function} \\
& \text{is probability, and (4)}
\end{aligned}$$

where by Exp_n we mean the function taking the expected value of its argument over the distribution given by Pr_n .

6. Since ε is fixed, and the positive $\varepsilon_1, \varepsilon_2$ are otherwise unconstrained, we can make $(\varepsilon_1 + \varepsilon_2)/\varepsilon$ arbitrarily close to zero: thus from (5) we

⁶⁸Remember that this implies boundedness of I , since $T \Rightarrow 1$; and boundedness is of course exactly what we do not have in the random walk tabulated in Fig. 2.11.4, where n can be arbitrarily large.

⁶⁹This is not a circular argument: we are merely presenting the same result in two different ways. One way is to prove Lem. 2.4.1 in our expectation logic (Chap. 7), and to observe that the Zero-One Law is a consequence of it; the other way is to give an argument in the semantics directly, as we do here, appealing to the Zero-One Law from the probability literature [GS92].

have that

$$(\lim_{n \rightarrow \infty} \text{Exp}_n.([G] * I)) = 0.$$

7. Because the invariant's expected value does not decrease, after n steps $\text{Exp}_n.I$ is no less than I 's initial value.
8. But in the limit that expected value of I must come entirely from the region \overline{G} ; we reason

$$\begin{aligned}
& \text{“initial value of } I \text{”} \\
\leq & \text{“expected value of } I \text{ as } n \rightarrow \infty \text{”} && \text{from (7)} \\
= & (\lim_{n \rightarrow \infty} \text{Exp}_n.I) \\
= & && \text{distribution of } + \text{ through lim} \\
& (\lim_{n \rightarrow \infty} \text{Exp}_n.(\overline{[G]} * I)) + (\lim_{n \rightarrow \infty} \text{Exp}_n.([G] * I)) \\
= & (\lim_{n \rightarrow \infty} \text{Exp}_n.(\overline{[G]} * I)) + 0 && \text{from (6)} \\
= & (\lim_{n \rightarrow \infty} \text{Exp}_n.(\overline{[G]} * I)).
\end{aligned}$$

So we have shown that if $\varepsilon * I \Rightarrow T$ for some $\varepsilon > 0$ then the expected value of $\overline{[G]} * I$ is in the limit no less than I 's initial value — which is essentially Case 3a of the loop rule.

Chapter notes

The invariant technique — first proposed by Floyd [Flo67] and incorporated by Hoare into his programming logic [Hoa69] in the early 1970's — remains the principal intellectual tool for developing and reasoning about loops in program code.⁷⁰

Although many proof frameworks for verification of probabilistic *systems* are not based explicitly on Hoare-logic (exceptions include Harel and Feldman [FH84] and den Hartog and de Vink [dHdV02], as mentioned on p. 36), nevertheless many of them do make use of *fixed points*, which is the fundamental mathematical idea underlying the invariant proof method. The basic temporal operators of *pCTL*, for example, have a fixed-point semantics (see Chapters 9 and 11) — similarly

⁷⁰The idea is much used elsewhere as well, for example in transition-style reasoning for state-based concurrency. That suggests a line of research into probabilistic concurrent systems based on expectations as here, perhaps in the style of Back and Kurki-Suonio's *action systems* [BKS83], Misra and Chandy's *UNITY* [CM88] (subsequently extended to probability-one reasoning by Rao [Rao94]) and Abrial's *Event-B* [Abr96b].

de Alfaro and Henzinger [dAH00] use explicit fixed-point expressions to express probabilistic “reachability” and “safety” properties, the latter being similar to a *global invariant* of the whole system (in the sense used here).

Similarly, consequences of Zero-One laws turn up elsewhere, especially in model-checking frameworks where the finiteness of the state space makes them particularly appropriate. Early examples include Vardi’s work on model checking probability-one temporal properties [Var85] and Hart, Sharir and Pnueli’s results on termination [HSP83] — all of whom use models which contain both probability and demonic nondeterminism. Hart *et al.* formulated the probabilistic variant rule Lem. 2.7.1 (and the related Zero-One Law) and showed it to be sound and finitarily complete [*op. cit.*]; our contribution has been to do the same at the level of program logic.

De Alfaro and Henzinger [*op. cit.*] also treat *angelic* interpretations of nondeterminism within a general theory of concurrent games. Consequently their formulations of “reachability” properties need to be more intricate than the similar “eventually” property defined here.

In fact Zero-One laws in logic appear still more generally — Halpern and Kapron [HK94] for example consider all structures over a given state space: given a formula ϕ they ask in what fraction of those structures is ϕ valid. Zero-One laws are relevant here because it turns out that for almost all of them the formula is true, or for almost all of them the formula is false.

Abstraction, Refinement and Proof for Probabilistic
Systems

McIver, A.; Morgan, C.C.

2005, XX, 388 p. 63 illus., Hardcover

ISBN: 978-0-387-40115-7