Probabilistic imperative programming:
a rigorous approach

Karen Seidel, Carroll Morgan,
and Annabelle McIver∗

17 January 1997

Abstract
Recent work has extended Kozen’s probabilistic semantics [8, 9]
to include demonic nondeterminism both at the operational [5] and
the logical level [13]. That makes it now possible in principle to treat
probabilistic program development with the same standards of rigour
that apply, when appropriate, to imperative programming [3].

In this report we treat several practical aspects of the new models,
not discussed in their more theoretical presentations [5, 13]: a game-
like interpretation of probabilistic and demonic choice acting jointly;
the intuition behind the probabilistic ‘healthiness conditions’ for pred-
icate transformers, linking them to standard probability theory; and
the use of predicate transformers to measure expected efficiency.

1 Introduction

Kozen’s operational [8] and predicate-transformer [9] approaches to proba-
bilistic semantics were restricted to deterministic programs; yet during the
same period and subsequently the trend in standard (non-probabilistic) pro-
gram development was to include demonic nondeterminism as a natural

∗Morgan and McIver are members of the Programming Research Group at the Un-
iversity of Oxford; all authors can be reached at \{carroll,anabel,karen\}@comlab.ox.ac.uk.
McIver is, and Seidel was, supported by the EPSRC.
part of the process [3]. In the decades since then, the importance of demonic nondeterminism as ‘underspecification’ has been increasingly recognised [7, 17, 1], and it is crucial to the notion of imperative program ‘refinement’ [2, 14, 15, 16].

Jifeng He (with CAR Hoare) [5] and we (with JW Sanders) [13] have extended Kozen’s work to include nondeterminism naturally, in principle bringing within reach for probabilistic programs all the well-developed techniques for rigorous specification and development of standard imperative programs.

The contribution of this report is to develop the probabilistic/demonic models’ definitions and properties in a more practical direction, and to show by counter-example that the ‘obvious’ probabilistic logic cannot deal with nondeterminism:

- The operational interaction between demonic and probabilistic choice is given in terms of a simple gambling game involving demons and dice, facilitating informal but rigorous reasoning (Sec. 2.2);

- The link with predicate transformers is explained in terms of the ‘expected winnings’ when gambling as above (Sec. 2.3);

- The probabilistic healthiness conditions governing the new predicate transformer algebra are related directly to elementary probability theory (Secs. 5, 6);

- The flexibility of the framework is illustrated by calculating not only expected correctness but also expected efficiency (‘Mr Bean’s socks’ in Sec. 7); and

- The obvious alternative — and simpler — approach in terms of probabilities and standard postconditions is shown not to be compositional when nondeterminism is present (App. A).

Secc. 3 and 4 summarise the syntax and semantics of the probabilistic language, using the ‘Monty Hall problem’ (Sec. 4.3) as an example.
2 The operational model as a game

2.1 The standard (non-probabilistic) game

We have a board of numbered squares, and a selection of numbered cards laid on it with at most one card per square; winning squares are indicated by coloured markers. (The squares are the program states; the program is the pattern of numbered cards; the winning squares form the postcondition.)

To play the game

An initial square is chosen (according to certain rules which do not concern us); subsequently

• if the square contains a card the card is removed, and play continues from the square whose number appeared on the card, and
• if the square does not contain a card, the game is over.

When the game is over the player has won if his final square contains a marker — otherwise he has lost.

The game so far is deterministic; and because the cards are removed after use it is also guaranteed to terminate. It is easily generalised however to include other features of standard programs:

- **looping** If the cards are *not* removed after use, the game can ‘loop’: in that case the player loses.

- **aborting** If some cards explode when read, the program contains the explicit and catastrophic failure *abort*; in that case the player loses.

- **nondeterminism** If each square can contain several cards, and the rules are modified so that the next state is determined by choosing just one of them without looking first (say the cards are face down), then play is nondeterministic. Taking the demonic (pessimistic) view, the player should expect to lose unless he is guaranteed to reach a winning position.
In the standard game, for each (initial) square one can examine the cards before playing to determine whether a win is guaranteed from there. (The squares from which a win is guaranteed form collectively the weakest precondition.)

2.2 The probabilistic game

Suppose now that each card contains not just one but rather a list of successor squares, and the choice from the list is made by rolling a die. In the deterministic game, play becomes a succession of die rolls, taking the player from square to square; termination (no card) and winning (marker) are defined as before.

When squares can contain several cards face down, each with a separate list of successors to be resolved by die roll, we are dealing with probability and nondeterminism together: first the card is chosen ‘blind’ (nondeterminism); the the card is turned over and a die roll (probability) determines which of its listed alternatives to take.

In the probabilistic game, one can ask for each initial square what is the guaranteed probability of winning. (Because of nondeterminism, as illustrated below, that probability may only be a lower bound.)

In Fig. 1 is an example game illustrating some of the above points. The guaranteed probability of winning from initial state 0 is only 1/2, in spite of the fact that the player can win every time if he is lucky enough to choose the first card; that is because he might be unlucky enough never to choose the first card, and we must assume the worst.

2.3 Expected winnings in the probabilistic game

For standard programs, the operational interpretation of execution supports a ‘logical’ view — given a set of final states (the postcondition) one can examine the program to determine the largest set of initial states (the weakest precondition) from which execution of the program is guaranteed to reach the designated final states. The sets of states are predicates, and the program is being regarded as a predicate transformer.

Regarding sets of states as characteristic functions (from the state space into \{0, 1\}), we generalise to probabilistic predicates by extending the range of those functions to all of \( \mathbb{R} \geq \), the non-negative reals. Probabilistic programs
The die chooses between the alternatives with equal probability.

The winning final positions are the states \{4, 5\}. From initial state 2 a win is guaranteed; from state 0 or 1 the minimum guaranteed probability of winning is 1/2; from state 3 the minimum probability is 0, since the second card might be chosen every time.

Figure 1: A probabilistic and nondeterministic game.

<table>
<thead>
<tr>
<th>square number</th>
<th>one card (nondeterministic choice)</th>
<th>another card (nondeterministic choice)</th>
<th>winning position</th>
<th>probability of winning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Go to 4</td>
<td>Roll die for 5,6</td>
<td>—</td>
<td>1/2</td>
</tr>
<tr>
<td>1</td>
<td>Roll die for 2,3</td>
<td>—</td>
<td>—</td>
<td>1/2</td>
</tr>
<tr>
<td>2</td>
<td>Go to 4</td>
<td>Go to 5</td>
<td>—</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Go to 4</td>
<td>Go to 6</td>
<td>—</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
</tr>
</tbody>
</table>
become functions from probabilistic postconditions to probabilistic weakest preconditions. The corresponding generalisation in the game is as follows.

Rather than placing winning markers on the board, we place money — rather than strictly winning or losing, the player simply keeps whatever money he finds in his final square. In Fig. 2 we show our original game translated into this new context. Note that the probability of winning (Fig. 1) translates into the equivalent expected winning (Fig. 2) as the corresponding fraction of £1, illustrating this important fact:

\[
\text{The expected value of a characteristic function over a distribution is the same as the probability assigned to the set that function describes.}
\]

Thus using expectations is at least as general as using probabilities explicitly, since we can always restrict ourselves to \(\{0, 1\}\)-valued functions from which probabilities are then recovered.

For probabilistic programs, the operational interpretation of execution thus supports a ‘logical’ view also — given a function from final states to \(\mathbb{R}_\geq\) (the probabilistic postcondition) one can examine the program beforehand to determine for each initial state the minimum expected (or ‘average’) win when the game is played repeatedly from there (the weakest probabilistic precondition) — also therefore a function from states to \(\mathbb{R}_\geq\). The functions are

<table>
<thead>
<tr>
<th>square number</th>
<th>one card (nondeterministic choice)</th>
<th>another card (nondeterministic choice)</th>
<th>winning value</th>
<th>expected win</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Go to 4</td>
<td>Roll die for 5,6</td>
<td>£0</td>
<td>50p</td>
</tr>
<tr>
<td>1</td>
<td>Roll die for 2,3</td>
<td>—</td>
<td>£0</td>
<td>50p</td>
</tr>
<tr>
<td>2</td>
<td>Go to 4</td>
<td>Go to 5</td>
<td>£0</td>
<td>£1</td>
</tr>
<tr>
<td>3</td>
<td>Go to 4</td>
<td>Go to 6</td>
<td>£0</td>
<td>£0</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>£1</td>
<td>£1</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>£1</td>
<td>£1</td>
</tr>
<tr>
<td>6</td>
<td>—</td>
<td>—</td>
<td>£0</td>
<td>£0</td>
</tr>
</tbody>
</table>
probabilistic predicates, and the program is being regarded as a probabilistic predicate transformer.

We are not limited to £1 coins for indicating postconditions — that is only an artefact of embedding standard postconditions into the probabilistic world. In general any amount of money can be placed in a square, and that is the key to allowing a smooth sequential composition of programs at the logical level — for if the program game (say) of Fig. 2 were executed after some other program prog, the precondition of the two together with respect to the postcondition \( \{4, 5\} \), say, would be calculated by applying \( wp.prog \) to the expected win column of the game table. That is because sequential composition of programs becomes, as usual, functional composition of the corresponding predicate transformers: we have

\[
wp.(prog; game).\{4, 5\} := wp.prog.(wp.game.\{4, 5\}),
\]

and that column contains non-integer values (for example 50p).

Another reason for allowing arbitrary values in \( \mathbb{R}_\geq \) is that using only standard postconditions (\( \{0, 1\} \)-valued) — equivalently, using explicit probabilities (recall the important fact above) — is not discriminating enough when nondeterminism is present: certain programs are identified that should be distinguished. (See App. A.)

3 Programming language syntax

Let prog range over programs, B over Boolean expressions, and P over real number expressions between 0 and 1 inclusive; assume that x stands for a list of distinct variables, and E for a list of expressions; and let the program scheme \( C \) be a program in which the program name X can appear. The syntax of our probabilistic programming language is as follows:

\[
prog := \text{abort} | \text{skip} | x := E | prog; prog |
if B then prog else prog fi | prog \oplus prog | prog \sqcap prog |
(\mu X \cdot C)
\]

The first five constructs, namely abort, skip, assignment, sequential composition and the conditional statement, are conventional [3].
The remaining constructs are for probabilistic choice, nondeterministic choice and recursion: given $P$ in the closed interval $[0, 1]$ we write $\text{prog} \ P \oplus \text{prog}'$ for the probabilistic choice between programs prog and prog'; they have probability $P$ and $1 - P$ respectively of being selected. In many cases $P$ will be a constant $p$ with $0 \leq p \leq 1$, but it can be more general; in fact with $P$ a $\{0, 1\}$-valued expression over state variables, the probabilistic choice $p \oplus$ can replace the conditional by making ‘certain’ choices in one direction or the other.

A probabilistic choice of three or more alternatives can be expressed using nested binary choices with appropriately weighted branches, and we define the shorthand

$$(\text{prog}_0 \ @ \ P_0, \ \text{prog}_1 \ @ \ P_1, \ \cdots) \ := \ \text{prog}_0 \ P_0 \oplus (\text{prog}_1 \ P_1/(1 - P_0) \oplus \cdots)$$

for such cases, in which we require the $P_i$ to be non-zero and to sum to 1. When all the programs $\text{prog}_i$ are assignments with the same left-hand side, say $x := E_i$, we write even more briefly

$$x := (E_0 \ @ \ P_0, \ E_1 \ @ \ P_1, \ \cdots).$$

The expression $\text{prog} \ \cap \text{prog}'$ denotes an explicit nondeterministic choice between prog and prog'. When convenient we write nondeterministic choice between assignments to the same variable $x$ as

$$x: \in \ \{E_0, E_1, \cdots\},$$

in that case abbreviating $x := E_0 \ \cap \ x := E_1 \ \cap \cdots$. More generally we write $x: \in S$ or $x: \notin S$ for set-valued expression $S$.\footnote{None of our examples requires a choice from the empty set.}

The construct $(\mu X \cdot C)$ behaves as prescribed by the program context $C$ except that it invokes itself recursively whenever it reaches a point where the program name $X$ appears. In the usual way, iteration is is a special case of recursion:

$$\text{do } B \rightarrow \text{prog od} \ := \ (\mu X \cdot \text{if } B \text{ then } (\text{prog}; X) \text{ else skip fi}).$$

We illustrate the syntax of our language with the example program of Fig. 3. There are three curtains labelled A, B, C, and a prize is hidden.
\( \text{pc} \in \{A, B, C\}; \) 

Prize hidden behind curtain.

\( \text{cc} := \{A @ \frac{1}{3}, B @ \frac{1}{3}, C @ \frac{1}{3}\}; \) 

Contestant chooses randomly.

\( \text{ac} : \not\in \{\text{pc}, \text{cc}\}; \) 

Another curtain opened; empty.

if clever then \( \text{cc} : \not\in \{\text{cc}, \text{ac}\} \) else skip fi

Changes his mind — or not?

The three ‘curtain’ variables \( \text{ac}, \text{cc}, \text{pc} \) are of type \( \{A, B, C\} \).

Written in full, the first three statements would be

\[
\begin{align*}
\text{pc} & := A \cap \text{pc} := B \cap \text{pc} := C \\
\text{cc} & := A \frac{1}{3} \oplus (\text{cc} := B \frac{1}{3} \oplus \text{cc} := C) \\
\text{ac} & : \in \{A, B, C\} - \{\text{pc}, \text{cc}\}.
\end{align*}
\]

The last statement is written using \( \not\in \) just for convenience — in fact it executes deterministically, since \( \text{cc} \) and \( \text{ac} \) are guaranteed to differ at that point.

Figure 3: An example program

nondeterministically behind one of them, say \( \text{pc} \). A contestant hopes to win the prize by guessing where it is hidden: he chooses randomly to point to curtain \( \text{cc} \). The host then tries to get the contestant to change his choice, showing that the prize is not behind some other curtain \( \text{ac} \) — either the contestant has chosen it already or it is behind the other closed curtain.

Should the contestant change his mind?

4 Programming language semantics

4.1 Standard \( \text{wp} \)

Let \( St \) be the set of states over which the program operates.

Standard predicates are sets of such states, and are partially ordered by inclusion: one predicate implies another if it is contained in the other. A standard program is a predicate transformer, a function which maps a postcondition (a predicate over final states) to a weakest precondition (a predicate
\[
\begin{align*}
wp.\text{abort}.Q & := false \\
wp.\text{skip}.Q & := Q \\
wp.(x := e).Q & := Q[x := E] \\
wp.(\text{prog}; \text{prog}').Q & := wp.\text{prog}.(wp.\text{prog}'.Q) \\
wp.(\text{if } B \text{ then prog else prog}' \text{ fi}).Q & := (B \land wp.\text{prog}.Q) \lor (\neg B \land wp.\text{prog}'.Q) \\
wp.(\text{prog} \sqcap \text{prog}').Q & := wp.\text{prog}.Q \land wp.\text{prog}'.Q \\
(\mu X \cdot C) & := \text{least fixed point of } \text{cntx}: \mathcal{D}St \rightarrow \mathcal{D}St, \text{ defined so that } wp.C = \text{cntx}.(wp.X).
\end{align*}
\]

Figure 4: Standard \( wp \)-semantics

over initial states): the weakest precondition identifies all the initial states starting from which the program is guaranteed to terminate in a state satisfying the postcondition. Thus the standard predicate-transformer semantics is given by the function space \( \mathcal{D}St \) defined

\[
\mathcal{D}St := \mathcal{P}St \rightarrow \mathcal{P}St,
\]

with a refinement ordering \( \sqsubseteq \) derived pointwise from the ordering on predicates. Given postcondition \( Q \) and program \( \text{prog} \), we write \( wp.\text{prog}.Q \) for the weakest precondition of \( \text{prog} \) with respect to \( Q \).

Fig. 4 shows the weakest precondition semantics of the standard constructs of our language. For example, we have

\[
wp.(n := 1).(n = 1) \equiv true
\]

because \( n := 1 \) is guaranteed to terminate in a final state satisfying \( n = 1 \) whatever the initial state. On the other hand

\[
wp.(n := 1 \sqcap n := 2).(n = 1) \equiv false
\]

because nondeterministic choice is demonic in the sense that it can always choose the alternative which does not lead to the desired postcondition; therefore no initial state for that program can be guaranteed to lead to \( n = 1 \) finally.
The entailment relation $\Rightarrow$ between standard predicates becomes pointwise $\leq$ over their probabilistic counterparts.

Figure 5: Equivalent operations on predicates

### 4.2 Probabilistic $wp$

To introduce probability, we generalise our predicates in the way suggested in Sec. 2. We first recall that standard predicates can equivalently be defined as characteristic functions from states to $\{0, 1\}$ with the pointwise $\leq$-ordering. We use brackets $[.]$ to indicate when we take that view, so that $[true]$ and $[false]$ are the constant functions 1 and 0 over the program variables. The standard logical operations on predicates can then be translated in a variety of ways into arithmetic operations that agree over the standard values, as summarised in Fig. 5; all are applied pointwise.

Our more general predicates are then functions from states to the non-negative reals $\mathbb{R}_{\geq}$ rather than to the doubleton set $\{0, 1\}$. That allows us to write for example

$$wp.(n := 1 \oplus n := 2).[n = 2] \equiv 1/2$$
where \([n = 2]\) is the expression that is 1 in states where \(n\) has the value 2 and is 0 otherwise, and \(1/2\) is the constant expression which is \(1/2\) in every state. Like standard programs, probabilistic programs are predicate transformers — the only difference is the type of the predicate transformed.

**Definition 4.1** The space of probabilistic predicates over \(St\) is defined

\[
P St := (St \to \mathbb{R}_{\geq}, \Rightarrow),
\]

where \(\mathbb{R}_{\geq}\) is the non-negative reals, and the entailment relation \(\Rightarrow\) is inherited pointwise from the normal \(\leq\) ordering in \(\mathbb{R}_{\geq}\). The probabilistic predicate transformer model for programs is

\[
T St := (P St \to P St, \sqsubseteq),
\]

where the refinement order \(\sqsubseteq\) is derived pointwise from entailment \(\Rightarrow\) on \(P St\).

When the state space \(St\) is finite, it is straightforward to show that both \(P St\) and \(T St\) are complete partial orders.\(^2\)

Let \(Q: St \to \mathbb{R}_{\geq}\) now be a probabilistic predicate. In Fig. 6 we give a probabilistic semantics to the constructs of our language. It has the important feature that the standard programming constructs behave as usual, and are described just as concisely.

The worst program \(\text{abort}\) cannot be guaranteed to terminate in any proper state and therefore maps every postcondition to 0. The immediately terminating program \(\text{skip}\) does not change anything, therefore a postcondition \(Q\) holds after execution of \(\text{skip}\) only if it held before. The precondition of the assignment \(x := E\) is the postcondition with the expression \(E\) substituted for \(x\). Sequential composition is functional composition. The semantics of conditional and nondeterministic choice are also the same as usual, except that arithmetic operators replace the logic operators.

The precondition of probabilistic choice is the weighted average of the preconditions of its branches. Since the average is greater than or equal to

\(^2\)When \(St\) is infinite we must make various small adjustments for closure under limits (completeness), one of which is to require that each probabilistic predicate be bounded above by some real number.
Our use of $\times$ and $+$ for the conditional is just one of several equivalent possibilities: since the predicates $[B]$ and $[\neg B]$ are standard, we could have for example used $\sqcap$, $\sqcup$ instead.

Figure 6: Probabilistic $wp$-semantics

the minimum it follows immediately that probabilistic choice refines non-deterministic choice, which corresponds to our intuition. In fact we consider probabilistic choice to be a deterministic programming construct; that is we say that a program is deterministic if it is free of demonic nondeterminism unless it aborts.\(^3\)

Finally, recursive programs have least-fixed-point semantics as usual.

4.3 Example: Expected correctness

We illustrate the semantics by returning to the program of Fig. 3. Consider the postcondition $[\text{pc} = \text{cc}]$, which takes value 1 in those final states in which the candidate has correctly chosen the prize. Working backwards through the program’s four statements, we have first (by standard $wp$ calculations) that

$$wp. (\text{if clever then (cc: } \notin \{\text{cc, ac}\} \text{ else skip fi}).[\text{pc} = \text{cc}]$$

\(^3\)Some writers call that pre-determinism: ‘deterministic if terminating’.
\[
\begin{align*}
&\equiv [\text{clever}] \times \{(\text{ac, cc, pc}) = \{A, B, C\} \} + [-\text{clever}] \times [\text{pc} = \text{cc}],
\end{align*}
\]
because (in case \text{clever}) the nondeterministic choice is guaranteed to pick \text{pc} only when it cannot avoid doing so.

Standard reasoning suffices for our next step also:
\[
\begin{align*}
wp&(\text{ac} \not\in \{\text{pc, cc}\}) \\
&\cdot ([\text{clever}] \times \{(\text{ac, cc, pc}) = \{A, B, C\} \} + [-\text{clever}] \times [\text{pc} = \text{cc}]) \\
&\equiv [\text{clever}] \times [\text{pc} \neq \text{cc}] + [-\text{clever}] \times [\text{pc} = \text{cc}].
\end{align*}
\]
For the \text{clever} case note that \{(\text{ac, cc, pc}) = \{A, B, C\} \} holds (in the postcondition) iff all three elements differ, and that the statement itself establishes only two of the required three inequalities — that \text{ac} \neq \text{pc} and \text{ac} \neq \text{cc}. The weakest precondition supplies the third.

For the \text{¬clever} case note that neither \text{pc} nor \text{cc} is assigned to by \text{ac} \not\in \{\text{pc, cc}\}, so that \text{pc} = \text{cc} holds afterwards iff it held before.

The next statement is probabilistic, and so produces a probabilistic precondition involving the factors 1/3 given explicitly in the program; we have
\[
\begin{align*}
wp&(\text{cc} := (A @ \frac{1}{3}, B @ \frac{1}{3}, C @ \frac{1}{3})) \\
&\cdot ([\text{clever}] \times [\text{pc} \neq \text{cc}] + [-\text{clever}] \times [\text{pc} = \text{cc}]) \\
&\equiv [\text{clever}] / 3 \times ( [\text{pc} \neq A] + [\text{pc} \neq B] + [\text{pc} \neq C]) \\
&\quad + [-\text{clever}] / 3 \times ( [\text{pc} = A] + [\text{pc} = B] + [\text{pc} = C]) \\
&\equiv [\text{clever}] / 3 \times 2 + [-\text{clever}] / 3 \times 1 \\
&\equiv 2[\text{clever}] / 3 + [-\text{clever}] / 3.
\end{align*}
\]
Then for the first statement \text{pc} :\in \{A, B, C\} we only note that \text{pc} does not appear in the final condition above, thus leaving it unchanged under \text{wp}: with simplification it becomes
\[
(1 + [\text{clever}]) / 3,
\]
which is thus the precondition for the whole program.

Since the postcondition \{\text{pc} = \text{cc}\} is standard (is the characteristic function of the set of states in which \text{pc} = \text{cc}), we are able to interpret the precondition directly as the probability that \text{pc} = \text{cc} will be satisfied on termination: we conclude that the contestant has 2/3 probability of choosing the prize if the Boolean \text{clever} is \text{true} initially, and only 1/3 if it is not.
5 Elementary probability theory

In probability theory, an event is a subset of some given sample space \( St \), so that the event is said to have occurred if the sampled value is in that set; a probability distribution \( \text{Pr} \) over the sample space is a function from its events into the closed interval \([0, 1]\), giving for each event the probability of its occurrence. In the general case, for technical reasons, not necessarily all subsets of \( St \) are events.

In our case we do take every (sub-)set of states to be an event, and so can regard a probability distribution more simply as a function from \( St \) directly (rather than from its subsets) to probabilities: thus \( \text{Pr}: St \rightarrow [0, 1] \), and the probability of a more general event is now just the sum of the probabilities of its elements.\(^4\)

A random variable \( X \) is a function from the sample space to the non-negative reals; and the expectation \( E.X \) of that random variable is defined in terms of the (discrete) probability distribution \( \text{Pr} \); we have

\[
E.X := \sum_{s \in St} (\text{Pr}.s \times X.s).
\]  

It represents the average value of \( X.s \) over many repeated samplings of \( s \).\(^5\)

In fact expectations can also be characterised without referring directly to an underlying probability distribution:

If a function \( E \) is of type \( (St \rightarrow \mathbb{R}_\geq) \rightarrow \mathbb{R}_\geq \), and

1. it is non-negative, so that \( E.X \geq 0 \) for all \( X: St \rightarrow \mathbb{R}_\geq \),
2. it is linear, so that for \( X, Y: St \rightarrow \mathbb{R}_\geq \) and \( c, d: \mathbb{R}_\geq \) we have

\[
E.(cX + dY) = c(E.X) + d(E.Y)
\]

\(^4\)The price paid for using simple discrete distributions is that there are some ‘everyday’ situations we cannot describe, such as the uniform distribution over the real interval \([0, 1]\) that might be the result of the program ‘choose a real number \( x \) randomly so that \( 0 \leq x \leq 1 \).’ We get away with it because no such program can be written in the language of Sec. 3.

\(^5\)Our ‘important fact’ is now stated ‘if \( X \) is the characteristic function of some event \( A \), then \( E.X \) is the probability that event \( A \) will occur.’
3. it satisfies $E.\mathbf{1} = 1$, where $\mathbf{1}$ is the constant function returning 1 for all arguments in $St$,

then it is an expectation over some probability distribution: it can be shown\(^6\) that it is expressible uniquely in the form (2) for some $Pr$.

The relevance of the above to our semantics is first that probabilistic predicates have the right type to be random variables, and second that for fixed program $\text{prog}$ and initial state $s$ the function

$$(\lambda Q: PSt \cdot wp.\text{prog}.Q.s)$$

has the right type to be an expectation. In fact it is easily checked that $wp.\text{skip}.Q.s$ and $wp.(x := E).Q.s$, if regarded as functions of the postcondition $Q$ for a fixed initial state $s$, satisfy the conditions above; similarly we can see that conditional, sequential composition and probabilistic choice preserve them. Thus when program $\text{prog}$ is deterministic and terminating we have that $wp.\text{prog}.Q.s$ is the expectation of the random variable $Q$ with regard to some probability distribution determined by $\text{prog}$ and $s$.

But how do $\text{prog}$ and $s$ determine a distribution? In fact the underlying distributions are found on the cards of the game from Sec. 2 — the sample space is the set of squares, and each card gives an explicit distribution over that space. If we consider the deterministic game, and regard 'make one move in the game' as a program in its own right, then we have a function from initial state to final distribution — the function taking a square to the card that square contains.\(^7\) For any postcondition $Q$ and square $s$ the expression $wp.\text{move}.Q.s$ gives the expectation — in the usual probabilistic sense — of

\(^6\)It is a special case of the Riesz representation theorem which states, loosely speaking, that knowledge of the expectation (assumed to be given directly) of every random variable uniquely determines an underlying probability distribution. See for instance Feller [4, p. 135].

\(^7\)For nondeterministic programs we are thus considering a function from state to sets of distributions, from a square to the set of cards there; that is the general operational model underlying the predicate transformer semantics.
Q, a random variable, over the probability distribution found on the card in that square.

When we move to more general programs, we must relax the conditions that characterise expectations. If prog possibly non-terminating — if it is recursive or contains abort — then \( wp.prog.Q.s \) may violate the third condition \( E.1 = 1 \). However as a function which satisfies the first two conditions it can still be regarded as an expectation in a weak sense. That was shown by Kozen [8] and later Jones [6], who defined expectations with regard to ‘probability distributions’ which may sum to less than 1. Jones called such distributions evaluations, and she gave conditions similar to the above for their existence [6, p. 117].

Finally, if program prog is not deterministic then we move further away from elementary theory, because \( wp.prog.Q.s \) is no longer an expectation even in the weak sense: it not linear. It is still however the minimum of a set of expectations: if prog and prog’ are deterministic programs then \( wp.(prog \sqcap prog’).Q.s \) is the pointwise minimum of the two expectations \( wp.prog.Q.s \) and \( wp.prog’.Q.s \).

Thus although the second condition is lost, it is not gone altogether: we retain sub-linearity,\(^8\) implying that for any \( c, d : \mathbb{R}_{\geq} \) and any program prog we still have

\[
wp.prog.(cQ + dR).s \geq c(wp.prog.Q.s) + d(wp.prog.R.s) .
\]

And clearly the first condition \( E.X \geq 0 \) continues to hold.

The characterisations of expectations given above for the simpler cases might lead one to conjecture that non-negative and sublinear functionals uniquely determine a set of probability distributions — and in [13] that is shown to be the case: sublinearity is the key ‘healthiness condition’ for probabilistic predicate transformers.

6 Probabilistic healthiness conditions

In [3] Dijkstra imposes ‘healthiness’ (well-formedness) conditions on the standard predicate transformers: they are conjunctivity, feasibility\(^9\), monotonicit-
ity and continuity. The conditions are important because they characterise exactly those programs which can be given an alternative semantics as relations between initial states and finite states, and they can be used to prove general laws for reasoning about programs. In this section we consider their probabilistic analogues.

All standard programs $\text{prog}$ are positively conjunctive: they satisfy

$$ wp.\text{prog}.(Q \land Q') \equiv wp.\text{prog}.Q \land wp.\text{prog}.Q' \quad (3) $$

for all standard postconditions $Q, Q'$. One of the probabilistic generalisations of conjunction is minimum (recall Fig. 5), and the above suggests we investigate $\sqcap$-distributivity.

Return for example to the program of Fig. 3, and consider its second statement

$$ \text{cc} := (A @ \frac{1}{3}, B @ \frac{1}{3}, C @ \frac{1}{3}) . $$

Writing $\text{stmt}$ for the above, with postcondition $[\text{cc} \neq C] \sqcap [\text{cc} \neq A]$ we find

$$ wp.\text{stmt}.( [\text{cc} \neq C] \sqcap [\text{cc} \neq A]) $$
$$ \equiv wp.\text{stmt}.[\text{cc} \neq C \land \text{cc} \neq A] $$
$$ \equiv wp.\text{stmt}.[\text{cc} = B] $$
$$ \equiv 1/3 $$
$$ \not\equiv 2/3 \sqcap 2/3 $$
$$ \equiv wp.\text{stmt}.[\text{cc} \neq C] \sqcap wp.\text{stmt}.[\text{cc} \neq A] . $$

Thus probabilistic programs do not distribute $\sqcap$ in general, and we must find something else.

We begin by recalling that for any events $X, Y$ and any probability distribution $\Pr$ we have\footnote{The first step is the modularity law for probabilities.}

$$ \Pr.(X \cap Y) = \Pr.X + \Pr.Y - \Pr.(X \cup Y) $$
$$ \geq (\Pr.X + \Pr.B - 1) \sqcup 0 . \quad \Pr.(X \cup Y) \leq 1 \text{ and } \Pr.(X \cap Y) \geq 0 $$

We are not dealing with exact probabilities however: when nondeterminism is present we have only lower bounds. Thus we address the question

Given only $\Pr.X \geq p$ and $\Pr.Y \geq q$, what is the most precise lower bound for $\Pr.(X \cap Y)$ in terms of $p$ and $q$?
From the reasoning above we obtain
\[(p + q - 1) \sqcup 0\] (4)
immediately as a lower bound. But to see that it is the lower bound we must show that for any \(X, Y, p, q\) there is a probability distribution \(Pr\) such that the bound is attained; and that is illustrated in Fig. 7, where an explicit distribution is given in which \(Pr.X = p\), \(Pr.Y = q\) and \(Pr.(X \cap Y)\) is as low as possible, reaching \((p + q - 1) \sqcup 0\) exactly.

Having established the importance of the expression (4) we define for general use
\[p & q := (p + q - 1) \sqcup 0,\]
and so we have shown that \(Pr.X & Pr.Y \leq Pr.(X \cap Y)\) is the best we can do in general.

Returning to probabilistic predicates, the events \(X, Y\) are expressed as characteristic functions \([X], [Y]\) and the distribution \(Pr\) (over final states) is — as we have seen — supplied by execution of a program. Once we notice that \(Q & R \equiv Q \sqcap R\) whenever \(Q, R\) are standard, we are led to conjecture that
\[wp.prog.(Q & R) \Leftrightarrow wp.prog.Q & wp.prog.R\] (5)
is the probabilistic analogue of (3). Indeed in our example we now have equality:

\[
\begin{align*}
wp.\text{stmt}.(\{\text{cc} \neq \text{C}\} \& \{\text{cc} \neq \text{A}\}) \\
\equiv \\
wp.\text{stmt}.[\text{cc} = \text{B}] \\
\equiv \\
1/3 \\
\equiv \\
2/3 \& 2/3 \\
\equiv \\
wp.\text{stmt}.[\text{cc} \neq \text{C}] \& wp.\text{stmt}.[\text{cc} \neq \text{A}].
\end{align*}
\]

In fact the exact probabilistic healthiness condition is slightly more general still — it is ‘sublinearity’:

**Definition 6.1** Define \(x \ominus y := (x - y) \sqcup 0\), and say that a predicate transformer \(j: \mathcal{T} \text{St}\) is sublinear if and only if for all \(Q, R: \mathcal{P}\text{St}\) and \(c, d, e: \mathbb{R}_\geq\) we have

\[
c(j.Q) + d(j.R) \ominus e \Rightarrow j(cQ + dR \ominus e),
\]

where we write \(cQ\) for the pointwise multiplication of the predicate \(Q\) by the scalar \(c\).\(^{11}\)

We say that sublinearity is ‘the’ healthiness condition because it can be shown \([13]\) that a probabilistic predicate transformer corresponds to a probabilistic game (Sec. 2.2) if and only if it is sublinear. (It is straightforward to check by structural induction, for example, that \(wp.\text{prog}\) is sublinear for all programs \(\text{prog}\) we can write in the language of (1) — that is ‘only if’.)

The property (5), which we reached by informal reasoning, is the instance of sublinearity in which \(c = d = e = 1\).

For ease of use (and comprehension), sublinearity can be broken up into a collection of simpler consequences of it: monotonicity and continuity are as in the standard case, but over the probabilistic entailment \(\Rightarrow\), and can be proved directly from Def. 6.1 \([13]\) (assuming a finite state-space in the case of continuity).

Another consequence of sublinearity is ‘feasibility’: recall that any standard program \(\text{prog}\) satisfies \(wp.\text{prog}.\neg\equiv \neg\), the Law of the excluded miracle. In the probabilistic semantics, the postcondition \(Q\) is a random variable and the precondition at a given initial state \(s\) is the expectation of

---

\(^{11}\)Because \(\ominus\) does not associate with +, we give it a (lower) syntactic precedence explicitly: by \(x + y \ominus z\) we mean \((x + y) \ominus z\).
Q for some probability distribution. That is feasible only if the precondition at s is no more than the maximum value of Q, as expressed by the following definition:

**Definition 6.2**  A probabilistic predicate transformer \( j: T St \) is feasible if for all \( Q: P St \) we have \( j.Q \xrightarrow{} \sqcup Q \), where \( \sqcup Q \) is the constant expression taking the maximum value of \( Q \) in all states.

Note that the usual \( j.0 \equiv 0 \) results when \( Q \) is 0.

There is an additional consequence of sublinearity which does not seem to have a standard analogue, namely scaling in the sense that \( j.(cQ) \equiv c(j.Q) \) for any \( c: \mathbb{R} \geq \).

Having discovered a probabilistic analogue of positive conjunctivity, it is natural to ask for an analogue of disjunctivity. That turns out to be linearity:

**Definition 6.3**  A probabilistic predicate transformer \( j \) is said to be linear iff for all predicates \( Q, R: P St \) and scalars \( c, d: \mathbb{R} \geq \)

\[
j.(cQ + dR) \equiv c(j.Q) + d(j.R) \]

In can be shown shown that, in the probabilistic model, linearity characterises the deterministic programs, just as in the standard model they are characterised by positive disjunctivity [10].

### 7  Example: Mr Bean’s socks

In this more substantial example, we show how expectations specialise both to probabilistic correctness and to probabilistic efficiency.

Mr Bean is trying to get a matching pair of socks from a drawer of his bedside table: there are \( N \) red and \( N \) blue socks inside, all jumbled up. Without looking into the drawer he takes one sock after another, never holding more than two: if they’re different he throws one away and takes another; if they’re the same, he stops.
Suppose he always throws away the blue sock; then he is guaranteed to end up with a matching pair, but if he is unlucky it could take him \( N + 2 \) steps. We show that if instead he throws away either sock at random, with probability \( \frac{1}{2} \), then the expected number of steps is only (just less than) 4. The disadvantage however is that he now has a \( \frac{1}{2}^{2N-2} \) chance of not getting a matching pair at all — and in that case he explodes.

To describe the algorithm let variable \( ss \) be a sequence of \( N \) red and \( N \) blue values, representing the order in which the socks will be drawn, and let \( l, r \) represent the sock held at any moment in Mr Bean’s left, right hand respectively. The program is then

\[
\text{socks} := l, r, n := ss_0, ss_1, 2; \\
do \ l \neq r \rightarrow \\
\quad l := ss_{n \frac{1}{2}} \oplus r := ss_n; \\
\quad n := n + 1 \\
\od
\]

We consider probabilistic correctness first: the precondition \( wp\text{.socks.}1 \) gives the probability of successful termination — from the (negated) loop guard we have then \( l = r \) so that the socks will be matching. By the semantics of sequential composition we have

\[
wp\text{.socks.}1 \equiv wp\text{.}(l, r, n := ss_0, ss_1, 2).(wp\text{.}(\text{do } B \rightarrow \text{body od}).1)
\]

where \( B \) is \( l \neq r \) and \( \text{body} \) is \( (l := ss_{n \frac{1}{2}} \oplus r := ss_n); n := n + 1 \). Since \( \text{body} \) is deterministic we can use the following rule, proved in App. B, to obtain the exact precondition of the loop:

**Theorem 7.1** If \( \text{body} \) is deterministic then

\[
wp\text{.}(\text{do } B \rightarrow \text{body od}).Q \equiv \sum_{i=0}^{\infty} h_i.([\neg B] \times Q),
\]

where \( h_{\gamma} := [B] \times wp\text{.body.} \gamma \).

The theorem states roughly that for deterministic loops it is sufficient to consider the \( i \)-fold iterates separately, adding up their results. (In standard
semantics, we would exploit determinism by disjoining the results rather than adding them.)

Taking $Q := 1$ we get

$$h^0([\neg B] \times Q) \equiv h^0[\neg B] \equiv [l = r]$$

$$h^1[\neg B] \equiv [l \neq r]/2 \times [n + 1 \leq 2N]$$

Note that the conditions relating $n$ and $N$ come from our assuming that $ss_n$ acts as abort for $n \geq 2N$, since the sequence is indexed from 0 up to but not including $2N$ — that is why we say that Mr Bean explodes if he finds the drawer is empty.

Our calculation continues

$$wp.(l, r, n := ss_0, ss_1, 2).(wp.(do G \rightarrow body od).1)$$

$$\equiv wp.(l, r, n := ss_0, ss_1, 2)$$

$$\cdot ([l = r] + \sum_{i=1}^{\infty} [l \neq r]/2^i \times [n + i \leq 2N])$$

$$\equiv wp.(l, r, n := ss_0, ss_1, 2)$$

$$\cdot ([l = r] + [l \neq r] \times \sum_{i=1}^{2N-n} 1/2^i)$$

$$\equiv [ss_0 = ss_1] + [ss_0 \neq ss_1] \times (1 - 1/2^{2N-2})$$

$$\equiv 1 - [ss_0 \neq ss_1]/2^{2N-2}.$$
≡
loop rule
\[
wp.(l, r, n := ss_0, ss_1, 2) \cdot (n \times [l = r] + \sum_{i=1}^{\infty} (n + i)/2^i \times [l \neq r] \times [n + i \leq 2N])
\]
≡
\[
wp.(l, r, n := ss_0, ss_1, 2) \cdot (n \times [l = r] + [l \neq r] \times \sum_{i=1}^{2N-n} (n + i)/2^i)
\]
⇒
\[
wp.(l, r, n := ss_0, ss_1, 2) \cdot (n \times [l = r] + [l \neq r] \times (n + 2))
\]
≡
\[
2[ss_0 = ss_1] + 4[ss_0 \neq ss_1]
\]
assignment
≡
\[
2 + 2[ss_0 \neq ss_1]
\]

Thus at least two steps are needed and, if the first two socks do not match, no more than two more.

Finally, the *average-case* analysis of a program is based on the assumption that its initial values are uniformly distributed. We can analyse the behaviour above under that assumption by considering the larger program

\[
ss := ?; \; \text{socks },
\]

where we take \(ss := ?\) to be a uniform selection over the possible sequences of sock colours. The average number of steps to termination then turns out to be just more than 3, because

\[
wp.(ss := ?; \; \text{socks}).n
\]
⇒
\[
wp.(ss := ?).(2 + 2[ss_0 \neq ss_1])
\]
≡
\[
2 + 2(wp.(ss := ?).[ss_0 \neq ss_1]) \quad \text{above}
\]
≡
\[
2 + 2(N/(2N - 1))
\]
≡
\[
3 + 1/(2N - 1)
\].

The deferred justification is that since \([ss_0 \neq ss_1]\) is standard the expression

\[
wp.(ss := ?).[ss_0 \neq ss_1]
\]
is just the probability that the randomly chosen \(ss\) will satisfy \(ss_0 \neq ss_1\); and note that whatever \(ss_0\) is, exactly \(N\) of the remaining \(2N - 1\) socks are different from it.
8 Summary and conclusions

Regarding standard predicates as characteristic functions into \( \{0, 1\} \) allows their easy generalisation to functions into \( \mathbb{R} \geq \), giving probabilistic predicates; and probabilistic predicate transformers then extend the standard semantics to include an operator for probabilistic choice.

Probabilistic predicates are random variables, and the probabilistic weakest precondition is the minimum expected value of the postcondition over all possible resolutions of demonic nondeterminism in the execution of the program. For a standard postcondition that is the same as the minimum probability the program’s establishing it, as shown in the prize choosing and sock selection example; if the program itself is standard that probability is either 1 or 0, thus returning the characteristic function of the standard precondition.

The sock selection example also showed that properly probabilistic predicates are useful for reasoning about efficiency: by calculating the expected value of a counting variable we determined the expected number of steps to its termination. (If the program is nondeterministic, however, the weakest precondition gives only a lower bound for the efficiency — not what is usually required. That is circumvented by coding the desired counting variable ‘negatively’: if the sock example had been nondeterministic, we could have used postcondition \( N - n \).)

The theory of probabilistic predicate transformers with nondeterminism is given in [13], where in particular the role of sublinearity (Def. 6.1) is identified: it characterises a the subspace of the predicate transformers that has an equivalent operational semantics of relations between initial and final probabilistic distributions over the state space [5]. All the programming constructs of the probabilistic language of guarded commands belong to that subspace, which means that the programmer who uses the language can elect to reason about it either axiomatically or operationally.

Further investigation of the theory of probabilistic predicate transformers is reported in [10].

Probabilistic reasoning in practice is further addressed in [12], where the probabilistic healthiness conditions are used to formulate and prove general rules for reasoning about (even nondeterministic) loops: probabilistic invariants and variants. Their use is thoroughly demonstrated by a number of examples.
References


A Predicates are inadequate

Could it be simpler? Consider restricting ourselves to standard predicates, and to probabilistic judgements of the form

\[ p \vdash \{Q\} \text{prog} \{R\}, \]  

meaning ‘from any initial state in \(Q\) the program \(\text{prog}\) will with probability at least \(p\) reach a final state in \(R\)’. A typical rule in the resulting system would be

\[
\frac{p \vdash \{Q\} \text{prog}_0 \{R\} \quad q \vdash \{R\} \text{prog}_1 \{S\}}{pq \vdash \{Q\} \text{prog}_0; \text{prog}_1 \{S\}}
\]
for any constant probabilities $p, q$, programs $\text{prog}_0, \text{prog}_1$ and standard predicates $Q, R, S$: the probabilistic choices in $\text{prog}_0$ and $\text{prog}_1$ are independent, and multiplication is monotonic. Indeed by defining

$$p \vdash \{Q\} \text{prog} \{R\} : = pQ \Rightarrow wp.\text{prog}.R$$

such statements become special cases within our current system, and the above sequential composition rule is easily proved from sublinearity.\(^{13}\)

But the judgements (6) are not enough. Consider the two programs

$$\begin{align*}
\text{prog}_0 & : = n := 4 \land (n := 5 \uplus n := 6) \\
\text{prog}_1 & : = (n := 4 \land n := 5) \uplus (n := 4 \land n := 6).
\end{align*}$$

(They correspond to executing the game of Fig. 1 from initial squares 0 and 1 respectively.) In Fig. 8 we set out all eight possible judgements, showing that in this simpler system $\text{prog}_0$ and $\text{prog}_1$ would be identified.

But now define a further program

$$\text{prog} : = \text{if } n = 4 \text{ then } (n := 5 \uplus n := 6) \text{ else skip fi ,}$$

\(^{13}\)Recall Def. 6.1: use its consequences scaling and monotonicity.
and consider the sequential compositions \(( \text{prog}_0; \text{prog} )\) and \(( \text{prog}_1; \text{prog} )\) with respect to the postcondition \( n = 5 \): we have

\[
\frac{1}{2} \vdash \{ \text{true} \} \text{prog}_0; \text{prog} \{ n = 5 \}
\]

but

\[
\frac{1}{2} \not\vdash \{ \text{true} \} \text{prog}_1; \text{prog} \{ n = 5 \},
\]

and in fact the best we can do for \( \text{prog}_1 \) is

\[
\frac{1}{4} \vdash \{ \text{true} \} \text{prog}_1; \text{prog} \{ n = 5 \}.
\]

Thus standard postconditions are not enough: if the programs \(( \text{prog}_0; \text{prog} )\) and \(( \text{prog}_1; \text{prog} )\) are different, then \( \text{prog}_0 \) and \( \text{prog}_1 \) cannot be the same if this simpler semantics is to be compositional.\(^\text{15}\)

Fig. 9 shows that \( \text{prog}_0 \) and \( \text{prog}_1 \) are indeed distinguished by properly probabilistic postconditions.

Nondeterminism is to blame for the above effects. If we recall from Def. 6.3 that deterministic programs are linear, it is clear that standard

\[\begin{array}{|c|c|c|}
\hline
Q & \text{wp.prog}_0.Q & \text{wp.prog}_1.Q \\
\hline
[ n = 4 ] + 2[ n = 5 ] & 1 & 1/2 \\
[ n = 4 ] + 2[ n = 6 ] & 1/2 & 1/2 \\
\hline
\end{array}\]

\(^{14}\)For \(( \text{prog}_0; \text{prog} )\) note that it doesn’t matter how the initial nondeterministic choice is resolved, since the result is \( 1/2 \) either way. For \(( \text{prog}_1; \text{prog} )\) however the probability of establishing \( n = 5 \) is \( 1/2 \cap 1 = 1/2 \) for the left branch of the initial choice \( 1 \oplus \), but \( 1/2 \cap 0 = 0 \) for the right branch; thus overall it is only \( (1/2 + 0)/2 = 1/4 \).

\(^{15}\)A further (but only informal) argument that \( \text{prog}_0 \) and \( \text{prog}_1 \) should be distinguished is the observation that \( \text{prog}_0 \) should terminate in states \( 5, 6 \) ‘with equal frequency’, however low or high that might be — but \( \text{prog}_1 \) does not have that property.
postconditions are enough for those: over a finite state space at least, linearity determines general weakest preconditions from the weakest preconditions with respect to the standard ‘point’ postconditions that correspond to single states.

Thus it seems that any semantics for the probabilistic language of guarded commands — with its demonic nondeterminism — must be at least as powerful as the system we have proposed. For a more extensive discussion of alternative models see [5].

\section*{B The deterministic loop theorem}

The proof of Thm. 7.1 is as follows. From the definition of the loop construct, we have

\[ \text{do } B \rightarrow \text{body } \text{od } = \mu . \text{cntx} , \]

where \text{cntx} is defined so that

\[ \text{cntx}.(wp . \text{prog}) = wp .(\text{if } B \text{ then } (\text{body}; \text{prog}) \text{ else skip fi}) . \]

Assuming continuity, we take the fixed point directly from the limit, so that

\[ \mu . \text{cntx} = \sqcup_{n=0}^{\infty} \text{cntx}^n.(wp . \text{abort}) , \]

and when \text{body} is deterministic we show by induction that

\[ \text{cntx}^n.(wp . \text{abort}) . Q \equiv \sum_{i=0}^{n-1} h^i ([\neg B] \times Q) , \]

with \text{h} as defined in Thm. 7.1. The base case is trivial; then for any postcondition \( Q \) we have

\[ \text{cntx}^{n+1}.(wp . \text{abort}) . Q \]

\[ \equiv \]

\[ wp .(\text{if } B \text{ then } (\text{body}; \text{cntx}^n.(wp . \text{abort})) \text{ else skip fi}) . Q \]

\[ \equiv \]

\[ [B] \times wp .(\text{body}; \text{cntx}^n.(wp . \text{abort})) . Q \] + \([\neg B] \times Q \]

\[ \equiv \]

\[ [B] \times wp . \text{body} .(\sum_{i=0}^{n-1} h^i ([\neg B] \times Q)) + [\neg B] \times Q \]
$$\equiv \sum_{i=0}^{n-1}([B] \times wp.body.h^i.([-B] \times Q)) + [-B] \times Q \quad \text{linearity of body}$$
$$\equiv \sum_{i=0}^{n-1}(h^{i+1}.([-B] \times Q)) + [-B] \times Q \quad \text{definition h}$$
$$\equiv \sum_{i=0}^{n} h^i.([-B] \times Q) \quad \text{arithmetic}$$

Thus in the limit

$$wp.(\text{do } B \rightarrow \text{body od)}.Q \equiv \sum_{i=0}^{\infty} h^i.([-B] \times Q),$$

as claimed. The fact that \textbf{body} is deterministic is used in the step referring to its linearity (Def. 6.3).