Probabilistic Predicative Programming

Eric C.R. Hehner

Department of Computer Science, University of Toronto
Toronto ON, M5S 2E4, Canada
hehner@cs.utoronto.ca

Abstract. This paper shows how probabilistic reasoning can be applied to the predicative style of programming.

0 Introduction

Probabilistic programming refers to programming in which the probabilities of the values of variables are of interest. For example, if we know the probability distribution from which the inputs are drawn, we may calculate the probability distribution of the outputs. We may introduce a programming notation whose result is known only probabilistically. A formalism for probabilistic programming was introduced by Kozen [3], and further developed by Morgan, McIver, Seidel and Sanders [4]. Their work is based on the predicate transformer semantics of programs; it generalizes the idea of predicate transformer from a function that produces a boolean result to a function that produces a probability result. The work of Morgan et al. is particularly concerned with the interaction between probabilistic choice and nondeterministic choice, which is required for refinement.

The term "predicative programming" [0,2] describes programming according to a first-order semantics, or relational semantics. The purpose of this paper is to show how probabilistic reasoning can be applied to the predicative style of programming.

1 Predicative Programming

Predicative programming is a way of writing programs so that each programming step is proven as it is made. We first decide what quantities are of interest, and introduce a variable for each such quantity. A specification is then a boolean expression whose variables represent the quantities of interest. The term "boolean expression" means an expression of type boolean, and is not meant to restrict the types of variables and subexpressions, nor the operators, within a specification. Quantifiers, functions, terms from the application domain, and terms invented for one particular specification are all welcome.

In a specification, some variables may represent inputs, and some may represent outputs. A specification is implemented on a computer when, for any values of the input variables, the computer generates (computes) values of the output variables to satisfy the specification. In other words, we have an implementation when the specification is true of every computation. (Note that we are specifying computations, not programs.) A specification S is implementable if

$$\forall \sigma \cdot \exists \sigma' \cdot S$$

where $\sigma = x, y, \ldots$ are the inputs and $\sigma' = x', y', \ldots$ are the outputs. In addition, specification S is deterministic if, for each input, the satisfactory output is unique. A program is a specification that has been implemented, so that a computer can execute it.

Suppose we are given specification S. If S is a program, a computer can execute it. If not, we have some programming to do. That means building a program P such that

$S \Leftarrow P$ is a theorem; this is called refinement. Since S is implied by P, all computer behavior satisfying P also satisfies S. We might refine in steps, finding specifications R, Q, ... such that $S \Leftarrow R \Leftarrow Q \Leftarrow ... \Leftarrow P$.

2 Notation

Here are all the notations used in this paper, arranged by precedence level.

0.	$\top\ \bot\ \ 0\ 1\ 2\ \infty\ \ x\ y\ \ (\ )$	booleans, numbers, variables, bracketed expressions
1.	$f\ x$	function application
2.	$x^y\ \ \rightarrow$	exponentiation, function space
3.	$\times\ \ /$	multiplication, division
4.	$+\ \ -\ \ \oplus$	addition, subtraction, modular addition
5.	$,..$	from (including) to (excluding)
6.	$=\ \ \neq\ \ <\ \ >\ \ \leq\ \ \geq\ \ :$	comparisons, inclusion
7.	$\neg$	negation
8.	$\wedge$	conjunction
9.	$\vee$	disjunction
10.	$\Rightarrow\ \ \Leftarrow$	implications
11.	$:=\ \ \textbf{if then else}$	assignment, conditional composition
12.	$\forall\cdot\ \ \exists\cdot\ \ \Sigma\cdot\ \ ;$	quantifiers, sequential composition
13.	$=\ \ \Rightarrow\ \ \Leftarrow\ \ \geq$	equality, implications, comparison

Exponentiation serves to bracket all operations within the exponent. The infix operators / – associate from left to right. The infix operators $\times$ + $\oplus$ $\wedge$ $\vee$; are associative (they associate in both directions). On levels 6, 10, and 13 the operators are continuing; for example, $a=b=c$ neither associates to the left nor associates to the right, but means $a=b \wedge b=c$. On any one of these levels, a mixture of continuing operators can be used. For example, $a \leq b < c$ means $a \leq b \wedge b < c$. The operators $= \Rightarrow \Leftarrow \geq$ are identical to $= \Rightarrow \Leftarrow \geq$ except for precedence.

We use unprimed and primed identifiers (for example, x and x') for the initial and final values of a variable. We use ok to specify that all variables are unchanged.

$$ok \ = \ x'=x \wedge y'=y \wedge ...$$

The assignment notation $x := e$ specifies that x is assigned the value e and that all other variables are unchanged.

$$x := e \ = \ x'=e \wedge y'=y \wedge ...$$

Conditional composition is defined as follows:

$$\textbf{if } b \textbf{ then } P \textbf{ else } Q \ = \ (b \Rightarrow P) \wedge (\neg b \Rightarrow Q)$$
$$= \ b \wedge P \ \vee \ \neg b \wedge Q$$

Sequential composition is defined as follows:

$$P;Q \ = \ \exists \sigma'' \cdot \text{(substitute } \sigma'' \text{ for } \sigma' \text{ in } P) \wedge \text{(substitute } \sigma'' \text{ for } \sigma \text{ in } Q)$$

where $\sigma = x, y, ...$ are the initial values, $\sigma'' = x'', y'', ...$ are the intermediate values, and $\sigma' = x', y', ...$ are the final values of the variables. There are many laws that can be proven from these definitions; one of the most useful is the Substitution Law:

$$x := e; P \ = \ \text{(for } x \text{ substitute } e \text{ in } P)$$

where P is a specification not employing the assignment or sequential composition operators. To account for execution time, we use a time variable; we use t for the time at which execution starts, and t' for the time at which execution ends. In the case of nontermination, $t' = \infty$.

3 Example of Predicative Programming

As an example of predicative programming, we write a program that cubes using only addition, subtraction, and test for zero. Let x and y be natural (non-negative integer valued) variables, and let n be a natural constant. Then $x'=n^3$ specifies that the final value of variable x is n^3. One way to refine (or implement) this specification is as follows:

$x'=n^3 \Leftarrow x:= n;\ x'=x\times n;\ x'=x\times n$

An initial assignment of n to x followed by two multiplications implies $x'=n^3$. Now we need to refine $x'=x\times n$.

$x'=x\times n \Leftarrow y:= x;\ x:= 0;\ x' = x + y\times n$

This one is proven by two applications of the Substitution Law: in the specification at the right end $x' = x + y\times n$, first replace x by 0 and then replace y by x; after simplification, the right side is now identical to the left side, and so the implication is proven. Now we have to refine $x' = x + y\times n$.

$x' = x + y\times n \Leftarrow$ **if** $y=0$ **then** ok **else** $(x:= x+n;\ y:= y-1;\ x' = x + y\times n)$

To prove it, let's start with the right side.

if $y=0$ **then** ok **else** $(x:= x+n;\ y:= y-1;\ x' = x + y\times n)$ Substitution Law twice

$=$ **if** $y=0$ **then** ok **else** $x' = x +n + (y-1)\times n$ now simplify

$=$ **if** $y=0$ **then** ok **else** $x' = x + y\times n$ expand ok and rewrite **if**

$=$ $y=0 \wedge x'=x \wedge y'=y \ \vee \ y\neq 0 \wedge x'=x+y\times n$ In the left disjunct, $y=0$ allows us to add 0 in the form of $y\times n$ to x. We drop $y'=y$.

$\Longrightarrow$ $y=0 \wedge x'=x+y\times n \ \vee \ y\neq 0 \wedge x'=x+y\times n$ boolean algebra

$=$ $x'=x+y\times n$

This latest refinement has not raised any new, unrefined specifications, so we now have a complete program. Using identifiers P, Q, and R for the three specifications that are not programming notations, we have

$P \Leftarrow x:= n;\ Q;\ Q$

$Q \Leftarrow y:= x;\ x:= 0;\ R$

$R \Leftarrow$ **if** $y=0$ **then** ok **else** $(x:= x+n;\ y:= y-1;\ R)$

and we can compile it to C as follows:

```
void P (void) {x = n; Q( ); Q( );}
void Q (void) {y = x; x = 0; R( );}
void R (void) {if (y==0) ; else {x = x+n; y = y–1; R( );}}
```

or, to avoid the poor implementation of recursive call supplied by most compilers,

```
void P (void) {x = n; Q( ); Q( );}
void Q (void) {y = x; x = 0; R: if (y==0) ; else {x = x+n; y = y–1; goto R;}}
```

To account for time, we add a time variable t. We can account for real time if we know the computing platform well enough, but let's just count iterations. We augment the specifications to talk about time, and we increase the time variable each iteration.

$x'=n^3 \wedge t'=t+n^2+n \Leftarrow x:= n;\ x'=x\times n \wedge t'=t+x;\ x'=x\times n \wedge t'=t+x$

$x'=x\times n \wedge t'=t+x \Leftarrow y:= x;\ x:= 0;\ x' = x + y\times n \wedge t'=t+y$

$x' = x + y\times n \wedge t'=t+y \Leftarrow$

if $y=0$ **then** ok **else** $(x:= x+n;\ y:= y-1;\ t:= t+1;\ x' = x + y\times n \wedge t'=t+y)$

We leave these proofs for the interested reader.

Here is a linear solution in which n is a natural variable. We can try to find n^3 in terms of $(n-1)^3$ using the identity $n^3 = (n-1)^3 + 3\times n^2 - 3\times n + 1$. The problem is the occurrence of n^2, which we can find using the identity $n^2 = (n-1)^2 + 2\times n - 1$. So we need a variable x for the cubes and a variable y for the squares. We start refining:

$x'=n^3 \Leftarrow x'=n^3 \wedge y'=n^2$

$x'=n^3 \wedge y'=n^2 \Leftarrow$ **if** $n=0$ **then** $(x:= 0;\ y:= 0)$ **else** $(n:= n-1;\ x'=n^3 \wedge y'=n^2;$

We cannot complete that refinement due to a little problem: in order to get the new values of x and y, we need not only the values of x and y just produced by the recursive call, but also the original value of n, which was not saved. So we revise:

$x'=n^3 \Leftarrow x'=n^3 \wedge y'=n^2 \wedge n'=n$

$x'=n^3 \wedge y'=n^2 \wedge n'=n \Leftarrow$

if $n=0$ **then** ($x:= 0$; $y:= 0$)

else ($n:= n-1$; $x'=n^3 \wedge y'=n^2 \wedge n'=n$; $n:= n+1$;

$y:= y + n + n - 1$; $x:= x + y + y + y - n - n - n + 1$)

After we decrease n, the recursive call promises to leave it alone, and then we increase it back to its original value (which fulfills the promise). With time,

$x'=n^3 \wedge t'=t+n \Leftarrow x'=n^3 \wedge y'=n^2 \wedge n'=n \wedge t'=t+n$

$x'=n^3 \wedge y'=n^2 \wedge n'=n \wedge t'=t+n \Leftarrow$

if $n=0$ **then** ($x:= 0$; $y:= 0$)

else ($n:= n-1$; $t:= t+1$; $x'=n^3 \wedge y'=n^2 \wedge n'=n \wedge t'=t+n$; $n:= n+1$;

$y:= y + n + n - 1$; $x:= x + y + y + y - n - n - n + 1$)

Compiling it to C produces

```
void P (void)
{if (n==0) {x = 0; y = 0;}
  else {n = n–1; P( ); n = n+1; y = y+n+n–1; x = x+y+y+y–n–n–n+1;}}
```

Here is linear solution without general recursion. Let z be a natural variable. Let

$Q \;=\; y = 3{\times}x^{2/3} + 3{\times}x^{1/3} + 1 \;\wedge\; z = 6{\times}x^{1/3} + 6 \;\Rightarrow\; x' = (x^{1/3}+n)^3$

or, more convenient for proof,

$Q \;=\; \forall k{:}\ nat \cdot x=k^3 \;\wedge\; y = 3{\times}k^2 + 3{\times}k + 1 \;\wedge\; z = 6{\times}k + 6 \;\Rightarrow\; x' = (k+n)^3$

Then

$x'=n^3 \wedge t'=t+n \Leftarrow x:= 0;\ y:= 1;\ z:= 6;\ Q \wedge t'=t+n$

$Q \wedge t'=t+n \Leftarrow$

if $n=0$ **then** ok

else ($x:= x+y$; $y:= y+z$. $z:= z+6$; $n:= n-1$; $t:= t+1$; $Q \wedge t'=t+n$)

The proofs, which are just substitutions and simplifications, are left to the reader. Compiling to C produces

```
x = 0; y = 1; z = 6;
Q: if (n==0) ; else {x = x+y; y = y+z; z = z+6; goto Q;}
```

4 Exact Precondition

We say that specification S is refined by specification P if $S \Leftarrow P$ is a theorem. That means, quantifying explicitly, that

$\forall \sigma, \sigma' \cdot S \Leftarrow P$

can be simplified to $\top$. For any two specifications S and P, if we quantify over only the output variables σ', we obtain the exact precondition, or necessary and sufficient precondition (called "weakest precondition" by others) for S to be refined by P. For example, in one integer variable x,

$\forall x' \cdot x'>5 \Leftarrow (x:= x+1)$

$= \quad \forall x' \cdot x'>5 \Leftarrow x'=x+1$ One-Point Law

$= \quad x+1 > 5$

$= \quad x > 4$

This means that a computation satisfying $x:= x+1$ will also satisfy $x'>5$ if and only if it starts with $x>4$. (If instead we quantify over the input variables σ, we obtain the exact (necessary and sufficient) postcondition.)

Now suppose P is an implementable and deterministic specification, and R' is a specification that refers only to output (primed) variables. Then the exact (necessary and sufficient) precondition for P to refine R' ("weakest precondition for P to establish postcondition R'") is

$\forall \sigma' \cdot P \Rightarrow R'$ by a generalized one-point law
$=$ $\exists \sigma' \cdot P \wedge R'$
$=$ $P; R$

where R is the same expression as R' except with unprimed variables. For example, the exact precondition for execution of $x := x+1$ to satisfy $x'>5$ is

$x := x+1;\ x>5$ Substitution Law
$=$ $x+1 > 5$
$=$ $x > 4$

5 Probability

A specification tells us whether an observation is acceptable or unacceptable. We now consider how often the various observations occur. For the sake of simplicity, this paper will treat only boolean and integer program variables, although the story is not very different for rational and real variables (summations become integrals).

A distribution is an expression whose value (for all assignments of values to its variables) is a probability, and whose sum (over all assignments of values to its variables) is 1. For example, if $n: nat+1$ (n is a positive natural), then 2^{-n} is a distribution because

$(\forall n: nat+1 \cdot 2^{-n}: prob) \ \wedge \ (\Sigma n: nat+1 \cdot 2^{-n})=1$

where $prob$ is the reals from 0 to 1 inclusive. A distribution is used to tell the frequency of occurrence of values of its variables. For example, 2^{-n} says that n has value 3 one-eighth of the time. If we have two variables $n, m: nat+1$, then 2^{-n-m} is a distribution because

$(\forall n, m: nat+1 \cdot 2^{-n-m}: prob) \ \wedge \ (\Sigma n, m: nat+1 \cdot 2^{-n-m})=1$

Distribution 2^{-n-m} says that the state in which n has value 3 and m has value 1 occurs one-sixteenth of the time.

If we have a distribution of several variables and we sum over some of them, we get a distribution describing the frequency of occurrence of the values of the other variables. If $n, m: nat+1$ are distributed as 2^{-n-m}, then $\Sigma m: nat+1 \cdot 2^{-n-m}$, which is 2^{-n}, tells us the frequency of occurrence of values of n.

If a distribution of several variables can be written as a product of distributions whose factors partition the variables, then each of the factors is a distribution describing the variables in its part, and the parts are said to be independent. For example, we can write 2^{-n-m} as $2^{-n} \times 2^{-m}$, so n and m are independent.

The average value of number expression e as variables v vary over their domains according to distribution p is

$\Sigma v \cdot e \times p$

For example, the average value of n^2 as n varies over $nat+1$ according to distribution 2^{-n} is $\Sigma n: nat+1 \cdot n^2 \times 2^{-n}$, which is 6. The average value of $n-m$ as n and m vary over $nat+1$ according to distribution 2^{-n-m} is $\Sigma n, m: nat+1 \cdot (n-m) \times 2^{-n-m}$, which is 0.

6 Probabilistic Specifications

To facilitate the combination of specifications and probabilities, add axioms

$\top = 1$

$\bot = 0$

equating booleans with numbers.

Let S be an implementable deterministic specification. Let p be the distribution describing the initial state σ. Then the distribution describing the final state σ' is

$\Sigma\sigma\cdot S \times p$

which is a generalization of the formula for average. Here is an example in two integer variables x and y. Suppose x starts with value 7 one-third of the time, and starts with value 8 two-thirds of the time. Then the distribution of x is

$(x{=}7) \times 1/3 \;+\; (x{=}8) \times 2/3$

The probability that x has value 7 is therefore

$(7{=}7) \times 1/3 \;+\; (7{=}8) \times 2/3$

$= \quad \top \times 1/3 \;+\; \bot \times 2/3$

$= \quad 1 \times 1/3 \;+\; 0 \times 2/3$

$= \quad 1/3$

Similarly, the probability that x has value 8 is 2/3, and the probability that x has value 9 is 0. Let X be the preceding distribution of x. Suppose that y also starts with value 7 one-third of the time, and starts with value 8 two-thirds of the time, independently of x. Then its distribution Y is given by

$Y \;=\; (y{=}7) / 3 \;+\; (y{=}8) \times 2/3$

and the distribution of initial states is $X \times Y$. Let S be

if $x{=}y$ **then** ($x{:=}\ 0$; $y{:=}\ 0$) **else** ($x{:=}\ abs(x{-}y)$; $y{:=}\ 1$)

Then the distribution of final states is

$\Sigma x, y\cdot S \times X \times Y$

$= \quad \Sigma x, y\cdot \quad (x{=}y \wedge x'{=}y'{=}0 \;\vee\; x{\neq}y \wedge x'{=}abs(x{-}y) \wedge y'{=}1)$

$\qquad \times \;((x{=}7) / 3 \;+\; (x{=}8) \times 2/3)$

$\qquad \times \;((y{=}7) / 3 \;+\; (y{=}8) \times 2/3)$

$= \quad (x'{=}y'{=}0) \times 5/9 \;+\; (x'{=}y'{=}1) \times 4/9$

We should see $x'{=}y'{=}0$ five-ninths of the time, and $x'{=}y'{=}1$ four-ninths of the time.

A probability distribution such as $(x'{=}y'{=}0) \times 5/9 \;+\; (x'{=}y'{=}1) \times 4/9$ describes what we expect to see. It can equally well be used as a probabilistic specification of what we want to see. A boolean specification is just a special case of probabilistic specification. We now generalize conditional composition and sequential composition to apply to probabilistic specifications as follows. If b is a probability, and P and Q are distributions of final states, then

if b **then** P **else** $Q \;=\; b \times P \;+\; (1{-}b) \times Q$

$P;Q \;=\; \Sigma\sigma''\cdot$ (substitute σ'' for σ' in P) $\times$ (substitute σ'' for σ in Q)

are distributions of final states. For example, in one integer variable x, suppose we start by assigning 0 with probability 1/3 or 1 with probability 2/3; that's

if 1/3 **then** $x{:=}\ 0$ **else** $x{:=}\ 1$

Subsequently, if $x{=}0$ then we add 2 with probability 1/2 or 3 with probability 1/2, otherwise we add 4 with probability 1/4 or 5 with probability 3/4; that's

if $x{=}0$ **then if** 1/2 **then** $x{:=}\ x{+}2$ **else** $x{:=}\ x{+}3$

else if 1/4 **then** $x{:=}\ x{+}4$ **else** $x{:=}\ x{+}5$

Notice that the programmer's **if** gives us conditional probability. Our calculation

if 1/3 **then** x:= 0 **else** x:= 1;
if x=0 **then if** 1/2 **then** x:= x+2 **else** x:= x+3
else if 1/4 **then** x:= x+4 **else** x:= x+5
$=$ $\Sigma x''\cdot$ $((x''=0)/3 + (x''=1)\times 2/3)$
$\times$ $((x''=0) \times ((x'=x''+2)/2 + (x'=x''+3)/2)$
$+ (x''\neq 0) \times ((x'=x''+4)/4 + (x'=x''+5)\times 3/4))$
$=$ $(x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2$

says that the result is 2 with probability 1/6 , 3 with probability 1/6 , 5 with probability 1/6 , and 6 with probability 1/2 .

We earlier used the formula $\Sigma\sigma\cdot S \times p$ to calculate the distribution of final states from the distribution p of initial states and an operation specified by S . We can now restate this formula as $(p'; S)$ where p' is the same as p but with primes on the variables. And the formula $(S; p)$ giving the exact precondition for implementable deterministic S to refine p' also works when S is a distribution.

Various distribution laws are provable from probabilistic sequential composition. Let n be a number, and let P , Q , and R be probabilistic specifications. Then

$n\times P; Q = n\times(P; Q) = P; n\times Q$
$P+Q; R = (P; R) + (Q; R)$
$P; Q+R = (P; Q) + (P; R)$

Best of all, the Substitution Law still works. (We postpone disjunction to Section 10.)

7 Random Number Generators

Many programming languages provide a random number generator (sometimes called a "pseudo-random number generator"). The usual notation is functional, and the usual result is a value whose distribution is uniform (constant) over a nonempty finite range. If n: *nat*+1 , we use the notation *rand n* for a generator that produces natural numbers uniformly distributed over the range 0,..n (from (including) 0 to (excluding) n). So *rand n* has value r with probability $(r: 0,..n) / n$. (Recall: r: 0,..n is $\top$ or 1 if r is one of 0, 1, 2, ..., n–1 , and $\bot$ or 0 otherwise.)

Functional notation for a random number generator is inconsistent. Since $x=x$ is a law, we should be able to simplify *rand n* = *rand n* to $\top$, but we cannot because the two occurrences of *rand n* might generate different numbers. Since $x+x = 2\times x$ is a law, we should be able to simplify *rand n* + *rand n* to 2 × *rand n* , but we cannot. To restore consistency, we replace each use of *rand n* with a fresh integer variable r whose value has probability $(r: 0,..n) / n$ before we do anything else. Or, if you prefer, we replace each use of *rand n* with a fresh variable r: 0,..n whose value has probability $1/n$. (This is a mathematical variable, not a state variable; in other words, there is no r' .) For example, in one state variable x ,

x:= *rand* 2; x:= x + *rand* 3 replace the two *rand*s with r and s
$=$ Σr: 0,..2$\cdot$ Σs: 0,..3$\cdot$ (x:= r; x:= x + s) × 1/2 × 1/3 Substitution Law
$=$ Σr: 0,..2$\cdot$ Σs: 0,..3$\cdot$ $(x' = r+s) / 6$ sum
$=$ $((x' = 0+0) + (x' = 0+1) + (x' = 0+2) + (x' = 1+0) + (x' = 1+1) + (x' = 1+2)) / 6$
$=$ $(x'=0) / 6 + (x'=1) / 3 + (x'=2) / 3 + (x'=3) / 6$

which says that x' is 0 one-sixth of the time, 1 one-third of the time, 2 one-third of the time, and 3 one-sixth of the time.

Whenever *rand* occurs in the context of a simple equation, such as r = *rand n* , we don't need to introduce a variable for it, since one is supplied. We just replace the deceptive equation with $(r: 0,..n) / n$. For example, in one variable x ,

$x := rand\ 2;\ x := x + rand\ 3$ replace assignments
$=$ $(x' : 0,..2)/2;\ (x' : x,..x+3)/3$ sequential composition
$=$ $\Sigma x'' \cdot (x'' : 0,..2)/2 \times (x' : x'',..x''+3)/3$ sum
$=$ $1/2 \times (x' : 0,..3)/3 \ + \ 1/2 \times (x' : 1,..4)/3$
$=$ $(x'=0) / 6 \ + \ (x'=1) / 3 + (x'=2) / 3 \ + \ (x'=3) / 6$

as before.

Although *rand* produces uniformly distributed natural numbers, it can be transformed into many different distributions. We just saw that $rand\ 2 + rand\ 3$ has value n with distribution $(n=0 \vee n=3) / 6 \ + \ (n=1 \vee n=2) / 3$. As another example, $rand\ 8 < 3$ has boolean value b with distribution

$\Sigma r : 0,..8 \cdot (b = (r<3)) / 8$
$=$ $(b=\top) \times 3/8 \ + \ (b=\bot) \times 5/8$
$=$ $5/8 - b/4$

which says that b is $\top$ three-eighths of the time, and $\bot$ five-eighths of the time.

8 Blackjack

This example is a simplified version of the card game known as blackjack. You are dealt a card from a deck; its value is in the range 1 through 13 inclusive. You may stop with just one card, or have a second card if you want. Your object is to get a total as near as possible to 14 , but not over 14 . Your strategy is to take a second card if the first is under 7 . Assuming each card value has equal probability (actually, the second card drawn has a diminished probability of having the same value as the first card drawn, but let's ignore that complication), we represent a card as $(rand\ 13) + 1$. In one variable x , the game is

$x := (rand\ 13) + 1;$ **if** $x<7$ **then** $x := x + (rand\ 13) + 1$ **else** ok

First we introduce variables $c, d : 0,..13$ for the two uses of *rand* , each with probability 1/13 . The program becomes

$x := c+1;$ **if** $x<7$ **then** $x := x+d+1$ **else** ok Substitution Law
$=$ **if** $c+1 < 7$ **then** $x' = c+d+2$ **else** $x' = c+1$

Then x' has distribution

$\Sigma c, d : 0,..13 \cdot$ (**if** $c+1 < 7$ **then** $x' = c+d+2$ **else** $x' = c+1$) $\times$ 1/13 $\times$ 1/13

by several omitted steps

$=$ $((2 \le x' < 7) \times (x' - 1) + (7 \le x' < 14) \times 19 + (14 \le x' < 20) \times (20 - x')) / 169$

Alternatively, we can use the variable provided rather than introduce new ones, as follows.

$x := (rand\ 13) + 1;$ **if** $x<7$ **then** $x := x + (rand\ 13) + 1$ **else** ok

replace assignments and ok

$=$ $(x' : 1,..14)/13;$ **if** $x<7$ **then** $(x' : x+1,..x+14)/13$ **else** $x'=x$ replace ; and **if**
$=$ $\Sigma x'' \cdot (x'' : 1,..14)/13 \times ((x''<7) \times (x' : x''+1,..x''+14)/13 + (x'' \ge 7) \times (x'=x''))$

by several omitted steps

$=$ $((2 \le x' < 7) \times (x' - 1) + (7 \le x' < 14) \times 19 + (14 \le x' < 20) \times (20 - x')) / 169$

That is the distribution of x' if we use the "under 7 " strategy. We can similarly find the distribution of x' if we use the "under 8 " strategy, or any other strategy. But which strategy is best? To compare two strategies, we play both of them at once. Player x will play "under n " and player y will play "under $n+1$ " using exactly the same cards (the result would be no different if they used different cards, but it would require more variables). Here is the new game:

if $c{+}1 < n$ **then** $x{:=}\ c{+}d{+}2$ **else** $x{:=}\ c{+}1$;
if $c{+}1 < n{+}1$ **then** $y{:=}\ c{+}d{+}2$ **else** $y{:=}\ c{+}1$;
$y{<}x{\le}14 \ \vee\ x{\le}14{<}y$ This line is the condition that x wins. We want to know the probability that it is true. Factor out $x{:=}$ and $y{:=}$.
$=$ $x{:=}$ **if** $c{+}1 < n$ **then** $c{+}d{+}2$ **else** $c{+}1$;
$y{:=}$ **if** $c{+}1 < n{+}1$ **then** $c{+}d{+}2$ **else** $c{+}1$;
$y{<}x{\le}14 \ \vee\ x{\le}14{<}y$ Use the substitution law twice.
$=$ (**if** $c{+}1{<}n{+}1$ **then** $c{+}d{+}2$ **else** $c{+}1$) $<$ (**if** $c{+}1{<}n$ **then** $c{+}d{+}2$ **else** $c{+}1$) ≤ 14
$\vee$ (**if** $c{+}1{<}n$ **then** $c{+}d{+}2$ **else** $c{+}1$) $\le 14 <$ (**if** $c{+}1{<}n{+}1$ **then** $c{+}d{+}2$ **else** $c{+}1$)
$=$ $c = n{-}1 \ \wedge\ d > 13{-}n$

Now the probability that x wins is

$\Sigma c, d{:}\ 0,..13{\cdot}\ (c = n{-}1 \ \wedge\ d > 13{-}n) \times 1/13 \times 1/13$
$=$ $(n{-}1)\ /\ 169$

By similar calculations we can find that the probability that y wins is $(14{-}n)\ /\ 169$, and the probability of a tie is $12/13$. For $n{<}8$, "under $n{+}1$" beats "under n". For $n{\ge}\ 8$, "under n" beats "under $n{+}1$". So "under 8" beats both "under 7" and "under 9".

9 Dice

If you repeatedly throw a pair of six-sided dice until they are equal, how long does it take? The program is

$R \ \Leftarrow\ u{:=}\ (rand\ 6) + 1;\ v{:=}\ (rand\ 6) + 1;$ **if** $u{=}v$ **then** ok **else** $(t{:=}\ t{+}1;\ R)$

for an appropriate definition of R. First, introduce variables $r, s{:}\ 0,..6$, each having probability $1/6$, for the two uses of $rand$, and simplify by eliminating variables u and v.

$R \ \Leftarrow$ **if** $r{=}s$ **then** $t'{=}t$ **else** $(t{:=}\ t{+}1;\ R)$

But there's a problem. As it stands, we could define $R \ =$ **if** $r{=}s$ **then** $t'{=}t$ **else** $t'{=}\infty$, which says that the execution time is either 0 or ∞. The problem is that variable r stands for a single use of $rand$, and similarly for s. In the previous example, we had no loops, so a single appearance of $rand$ was a single use of $rand$. Now we have a loop, and r has the same value each iteration, and so has s. The solution to this problem is to parameterize r and s by iteration or by time. We introduce $r, s{:}\ time{\to}(0,..6)$, with $r\ t$ and $s\ t$ each having probability $1/6$. The program is

$R \ \Leftarrow$ **if** $r\ t = s\ t$ **then** $t'{=}t$ **else** $(t{:=}\ t{+}1;\ R)$

Now we can define R to tell us the execution time.

$(\forall i{:}\ t,..t'{\cdot}\ r\ i \neq s\ i) \ \wedge\ r\ t' = s\ t'$

says that t' is the first time (from t onward) that the two dice are equal. The refinement is proved as follows:

if $rt{=}st$ **then** $t'{=}t$ **else** $(t{:=}\ t{+}1;\ (\forall i{:}\ t,..t'{\cdot}\ ri{\neq}si) \wedge rt'{=}st')$ case and substitution
$=$ $rt{=}st \wedge t'{=}t \ \vee\ rt{\neq}st \wedge (\forall i{:}\ t{+}1,..t'{\cdot}\ ri{\neq}si) \wedge rt'{=}st'$ axioms of $\forall$
$=$ $(\forall i{:}\ t,..t'{\cdot}\ ri{\neq}si) \wedge rt'{=}st'$

Treating ri and si as though they were simple variables, $ri{\neq}si$ with probability

$\Sigma ri, si{:}\ 0,..6{\cdot}\ (ri{\neq}si) \times 1/6 \times 1/6 \ =\ 5/6$

and $ri{=}si$ with probability

$\Sigma ri, si{:}\ 0,..6{\cdot}\ (ri{=}si) \times 1/6 \times 1/6 \ =\ 1/6$

So we offer the hypothesis that the final time t' has the distribution

$(t'{\ge}t) \times (5/6)^{t'-t} \times 1/6$

We can verify this distribution as follows. The distribution of the implementation (right side) is

$\Sigma rt, st \cdot$ (**if** $rt{=}st$ **then** $t'{=}t$ **else** $(t{:=}\ t{+}1;\ (t'{\geq}t) \times (5/6)^{t'-t} \times 1/6)) \times 1/6 \times 1/6$ sum

$=$ $(6 \times (t'{=}t)\ +\ 30 \times (t{:=}\ t{+}1;\ (t'{\geq}t) \times (5/6)^{t'-t} \times 1/6)) \times 1/6 \times 1/6$ substitution

$=$ $(6 \times (t'{=}t)\ +\ 30 \times (t'{\geq}t{+}1) \times (5/6)^{t'-t-1} \times 1/6) \times 1/6 \times 1/6$ arithmetic

$=$ $(t'{=}t) \times 1/6\ +\ (t'{\geq}t{+}1) \times (5/6)^{t'-t} \times 1/6$

$=$ $(t'{\geq}t) \times (5/6)^{t'-t} \times 1/6$

The last line is the distribution of the specification, which concludes the proof.

The alternative to introducing new variables r and s is as follows. Starting with the implementation,

$u{:=}\ (rand\ 6) + 1;\ v{:=}\ (rand\ 6) + 1;$ replace *rand* and

if $u{=}v$ **then** $t'{=}t$ **else** $(t{:=}\ t{+}1;\ (t'{\geq}t) \times (5/6)^{t'-t} \times 1/6)$ Substitution Law

$=$ $((u'{:}\ 1,..7) \wedge v'{=}v \wedge t'{=}t)/6;\ (u'{=}u \wedge (v'{:}\ 1,..7) \wedge t'{=}t)/6;$ replace first ;

if $u{=}v$ **then** $t'{=}t$ **else** $(t'{\geq}t{+}1) \times (5/6)^{t'-t-1} / 6$ and simplify

$=$ $((u'{:}\ 1,..7) \wedge (v'{:}\ 1,..7) \wedge t'{=}t)/36;$ replace remaining ;

if $u{=}v$ **then** $t'{=}t$ **else** $(t'{\geq}t{+}1) \times (5/6)^{t'-t-1} / 6$ and replace **if**

$=$ $\Sigma u'', v''{:}\ 1,..7 \cdot \Sigma t'' \cdot \quad (t''{=}t)/36 \times ((u''{=}v'') \times (t'{=}t'')$

$+\ (u''{\neq}v'') \times (t'{\geq}t''{+}1) \times (5/6)^{t'-t''-1} / 6)$ sum

$=$ $1/36 \times (6 \times (t'{=}t)\ +\ 30 \times (t'{\geq}t{+}1) \times (5/6)^{t'-t-1} / 6)$ combine

$=$ $(t'{\geq}t) \times (5/6)^{t'-t} \times 1/6$

which is the probabilistic specification.

The average value of t' is

$\Sigma t' \cdot t' \times (t'{\geq}t) \times (5/6)^{t'-t} \times 1/6\ =\ t{+}5$

so on average it takes 5 additional throws of the dice to get an equal pair.

10 Nondeterminism

According to some authors, nondeterminism comes in several varieties: angelic, demonic, oblivious, and prescient. To illustrate the differences, consider

$x{:=}\ rand\ 2;\ y{:=}\ 0$ **or** $y{:=}\ 1$

and we want the result $x'{=}y'$. If **or** is angelic nondeterminism, it chooses between its operands $y{:=}\ 0$ and $y{:=}\ 1$ in such a way that the desired result $x'{=}y'$ is always achieved. If **or** is demonic nondeterminism, it chooses between its operands in such a way that the desired result is never achieved. Both angelic and demonic nondeterminism require knowledge of the value of variable x when choosing between assignments to y. Oblivious nondeterminism is restricted to making a choice without looking at the current (or past) state. It achieves $x'{=}y'$ half the time. Now consider

$x{:=}\ 0$ **or** $x{:=}\ 1;\ y{:=}\ rand\ 2$

and we want $x'{=}y'$. If **or** is angelically prescient, x will be chosen to match the future value of y, always achieving $x'{=}y'$. If **or** is demonically prescient, x will be chosen to avoid the future value of y, never achieving $x'{=}y'$. If **or** is not prescient, then $x'{=}y'$ is achieved half the time.

In predicative programming, nondeterminism is disjunction. Angelic, demonic, oblivious, and prescient are not kinds of nondeterminism, but ways of refining nondeterminism. In the example

$x{:=}\ rand\ 2;\ (y{:=}\ 0) \vee (y{:=}\ 1)$

with desired result $x'{=}y'$, we can refine the nondeterminism angelically as $y{:=}\ x$, or demonically as $y{:=}\ 1{-}x$, or obliviously as either $y{:=}\ 0$ or $y{:=}\ 1$. In the example

$(x{:=}\ 0) \vee (x{:=}\ 1);\ y{:=}\ rand\ 2$

with desired result $x'{=}y'$, we first have to replace $rand\ 2$ by boolean variable r having probability $1/2$. Then we can refine the nondeterminism with angelic prescience as $x{:=}\ r$,

or with demonic prescience as $x := 1-r$, or without prescience as either $x := 0$ or $x := 1$.

Suppose we have one natural variable n whose initial value is 5. After executing the nondeterministic specification $ok \vee (n := n+1)$, we can say that the final value of n is either 5 or 6. Now suppose this specification is executed many times, and the distribution of initial states is $n=5$ (n always starts with value 5). What is the distribution of final states? Nondeterminism is a freedom for the implementer, who may refine the specification as ok, which always gives the answer $n'=5$, or as $n := n+1$, which always gives the answer $n'=6$, or as

$$\textbf{if } even\ t \textbf{ then } ok \textbf{ else } n := n+1$$

which gives $n'=5$ or $n'=6$ unpredictably. In general, we cannot say the distribution of final states after a nondeterministic specification. If we apply the formula $\Sigma\sigma\cdot S \times p$ to a specification S that is nondeterministic, the result may not be a distribution. For example,

$$\Sigma n\cdot\ (ok \vee (n := n+1)) \times (n=5) \ =\ n'=5 \vee n'=6$$

which is not a distribution because

$$\Sigma n'\cdot\ n'=5 \vee n'=6 \ =\ 2$$

Although $n'=5 \vee n'=6$ is not a distribution, it does accurately describe the final state.

Suppose the initial value of n is described by the distribution $(n=5)/2 + (n=6)/2$. Application of the formula $\Sigma\sigma\cdot S \times p$ to our nondeterministic specification yields

$$\begin{array}{ll} & \Sigma n\cdot\ (ok \vee (n := n+1)) \times ((n=5)/2 + (n=6)/2) \\ = & (n'=5 \vee n'=6)/2 + (n'=6 \vee n'=7)/2 \end{array}$$

Again, this is not a distribution, summing to 2 (the degree of nondeterminism). Interpretation of nondistributions is problematic, but this might be interpreted as saying that half of the time we will see either $n'=5$ or $n'=6$, and the other half of the time we will see either $n'=6$ or $n'=7$.

Nondeterministic choice ($P \vee Q$), probabilistic choice ($\textbf{if } rand\ 2 \textbf{ then } P \textbf{ else } Q$), and deterministic choice ($\textbf{if } b \textbf{ then } P \textbf{ else } Q$) are not three different, competing ways of forming a choice. Rather, they are three different degrees of information about a choice. In fact, nondeterministic choice is equivalent to an unnormalized random choice. In one variable x,

$$\begin{array}{lll} & (x := 0) \vee (x := 1) & \\ = & x' :\ 0,..2 & \\ = & 2 \times (x' :\ 0,..2)/2 & \text{introduce } rand \text{ the same way we eliminate it} \\ = & 2 \times (x' = rand\ 2) & \\ = & 2 \times (x := rand\ 2) & \\ \geq & x := rand\ 2 & \end{array}$$

Thus we prove

$$(x := 0) \vee (x := 1) \ \geq\ x := rand\ 2$$

which is the generalization of refinement to probabilistic specifications. Nondeterministic choice can be refined by probabilistic choice.

It is a well known boolean law that nondeterministic choice can be refined by deterministic choice.

$$P \vee Q \ \Leftarrow\ \textbf{if } b \textbf{ then } P \textbf{ else } Q$$

In fact, nondeterministic choice is equivalent to deterministic choice in which the determining expression is a variable of unknown value.

$$P \vee Q \ =\ \exists b\colon bool\cdot\ \textbf{if } b \textbf{ then } P \textbf{ else } Q$$

(The variable introduced is a mathematical variable, not a state variable; there is no b'.)

This is what we will do: we replace each nondeterministic choice with an equivalent existentially quantified deterministic choice, choosing a fresh variable each time. Then we move the quantifier outward as far as possible. If we move it outside a loop, we must then index the variable by iteration or by time, exactly as we did with the variable that replaces

occurrences of *rand* . All programming notations distribute over disjunction, so in any programming context, existential quantifiers (over a boolean domain) can be moved to the front. Before we prove that specification R is refined by a program containing a nondeterministic choice, we make the following sequence of transformations. (The dots are the context, or uninteresting parts, which remain unchanged from line to line.)

$R \Leftarrow$ ·····$(P \vee Q)$·····

$R \Leftarrow$ ·····$(\exists b\cdot$ **if** b **then** P **else** $Q)$·····

$R \Leftarrow (\exists b\cdot$ ·····(**if** b **then** P **else** $Q)$·····)

$\forall b\cdot\ (R \Leftarrow$ ·····(**if** b **then** P **else** $Q)$·····)

A refinement is proved for all values of all variables anyway, even without explicit universal quantification, so effectively the quantifier disappears.

With this transformation, let us look again at the example $ok \vee (n{:=}\ n{+}1)$. With input distribution $n{=}5$ we get

$\Sigma n\cdot$ (**if** b **then** ok **else** $n{:=}\ n{+}1) \times (n{=}5)$

$=$ **if** b **then** $n'{=}5$ **else** $n'{=}6$

which is a distribution of n' because

$\Sigma n'\cdot$ **if** b **then** $n'{=}5$ **else** $n'{=}6$

$=$ **if** b **then** $(\Sigma n'\cdot\ n'{=}5)$ **else** $(\Sigma n'\cdot\ n'{=}6)$

$=$ **if** b **then** 1 **else** 1

$=$ 1

With input distribution $(n{=}5)/2 + (n{=}6)/2$ we get

$\Sigma n\cdot$ (**if** b **then** ok **else** $n{:=}\ n{+}1) \times ((n{=}5)/2 + (n{=}6)/2)$

$=$ **if** b **then** $(n'{=}5)/2 + (n'{=}6)/2$ **else** $(n'{=}6)/2 + (n'{=}7)/2$

which is again a distribution of n' . These answers retain the nondeterminism in the form of variable b , which was not part of the question, and whose value is unknown.

11 Monty Hall's Problem

To illustrate the combination of nondeterminism and probability, we look at Monty Hall's problem, which was the subject of an internet discussion group; various probabilities were hypothesized and argued. We will not engage in any argument; we just calculate. The problem is also treated in [4].

Monty Hall is a game show host, and in this game there are three doors. A prize is hidden behind one of the doors. The contestant chooses a door. Monty then opens one of the doors, but not the door with the prize behind it, and not the door the contestant has chosen. Monty asks the contestant whether they (the contestant) would like to change their choice of door, or stay with their original choice. What should the contestant do?

Let p be the door where the prize is. Let c be the contestant's choice. Let m be the door Monty opens. If the contestant does not change their choice of door, the program is

$(p{:=}\ 0) \vee (p{:=}\ 1) \vee (p{:=}\ 2)$;
$c{:=}$ *rand* 3;
if $c{=}p$ **then** $(m{:=}\ c{\oplus}1) \vee (m{:=}\ c{\oplus}2)$ **else** $m{:=}\ 3{-}c{-}p$;
ok

The first line $(p{:=}\ 0) \vee (p{:=}\ 1) \vee (p{:=}\ 2)$ says that the prize is placed behind one of the doors; the contestant knows nothing about the criteria used for placement of the prize, so from their point of view it is a nondeterministic choice. The second line $c{:=}$ *rand* 3 is the contestant's random choice of door. In the next line, $\oplus$ is addition modulo 3 ; if the contestant happened to choose the door with the prize, then Monty can choose either of the other two (nondeterministically); otherwise Monty must choose the one door that differs

from both c and p. This line can be written more briefly and more clearly as $c' = c \neq m' \neq p = p'$. The final line ok is the contestant's decision not to change door.

We replace $rand\ 3$ with variable r. We introduce variable P of type $0, 1, 2$ in order to replace the nondeterministic assignment to p with

if $P=0$ **then** $p := 0$ **else if** $P=1$ **then** $p := 1$ **else** $p := 2$

or more simply $p := P$. And since we never reassign p, we really don't need it as a variable at all. We introduce variable M to express the nondeterminism in Monty's choice. Our program is now deterministic (in terms of unknown P and M) and so we can append to it the condition for winning, which is $c=P$. We have

$c := r$;
$m :=$ **if** $c=P$ **then if** M **then** $c \oplus 1$ **else** $c \oplus 2$ **else** $3-c-P$;
$c = P$ substitution law twice
$=$ $r = P$

Not surprisingly, the condition for winning is that the random choice made by the contestant is the door where the prize is. Also not surprisingly, its probability is

$\Sigma r \cdot (r=P) \times 1/3$
$=$ $1/3$

If the contestant takes the opportunity offered by Monty of switching their choice of door, then the program, followed by the condition for winning, becomes

$c := r$;
$m :=$ **if** $c=P$ **then if** M **then** $c \oplus 1$ **else** $c \oplus 2$ **else** $3-c-P$;
$c := 3-c-m$;
$c = P$

In the first line, the contestant chooses door c at random. In the second line, Monty opens door m, which differs from both c and P. In the next line, the contestant changes the value of c but not to m; thanks to the second line, this is deterministic; this could be written more briefly and more clearly as $c \neq c' \neq m = m'$. The final line is the condition for winning. After a small calculation (c starts at r and then changes; the rest is irrelevant), the above four lines simplify to

$r \neq P$

which says that the contestant wins if the random choice they made originally was not the door where the prize is. Its probability is

$\Sigma r \cdot (r \neq P) \times 1/3$
$=$ $2/3$

Perhaps surprisingly, the probability of winning is now $2/3$, so the contestant should switch.

12 Mr.Bean's Socks

Our next example originates in [4]; unlike Monty Hall's problem, it includes a loop. Mr.Bean is trying to get a matching pair of socks from a drawer containing an inexhaustible supply of red and blue socks (in the original problem the supply of socks is finite). He begins by withdrawing two socks from the drawer. If they match, he is done. Otherwise, he throws away one of them at random, withdraws another sock, and repeats. The choice of sock to throw away is probabilistic, with probability $1/2$ for each color. As for the choice of sock to withdraw from the drawer, we are not told anything about how this choice is made, so it is nondeterministic. How long will it take him to get a matching pair?

Here is Mr.Bean's program (omitting the initialization). Variables L and R represent the color of socks held in Mr.Bean's left and right hands.

$L'=R' \Leftarrow$
  if $L=R$ **then** ok
  else (**if** $rand\ 2$ **then** ($L:=red$) $\vee$ ($L:=blue$) **else** ($R:=blue$) $\vee$ ($R:=red$);
    $t:=t+1$; $L'=R'$)

As always, we begin by replacing the use of $rand$ by a variable h (for hand), and we introduce variable d to express the nondeterministic choices. Due to the loop we index these variables with time. The refinement

$L'=R' \Leftarrow$ **if** $L=R$ **then** ok
  else (**if** $h\ t$ **then if** $d\ t$ **then** $L:=red$ **else** $L:=blue$
    else if $d\ t$ **then** $R:=blue$ **else** $R:=red$;
    $t:=t+1$; $L'=R'$)

is easily proven. Now we need a hypothesis concerning the probability of execution times.

Suppose the nondeterministic choices are made such that Mr.Bean always gets from the drawer a sock of the same color as he throws away. This means that the nondeterministic choices become

if $d\ t$ **then** $L:=red$ **else** $L:=blue$ $=$ ok
if $d\ t$ **then** $R:=blue$ **else** $R:=red$ $=$ ok

(which means that $d\ t$ just happens to have the same value as $L=red \wedge R=blue$ each time). If I were watching Mr.Bean repeatedly retrieving the same color sock that he has just thrown away, I would soon suspect him of doing so on purpose, or perhaps a malicious mechanism that puts the wrong sock in his hand. But the mathematics says nothing about purpose or mechanism; it may be just a fantastic coincidence. In any case, we can prove that execution takes either no time or forever

if $L=R$ **then** $t'=t$ **else** $t'=\infty$ $\Leftarrow$
  if $L=R$ **then** ok **else** ($t:=t+1$; **if** $L=R$ **then** $t'=t$ **else** $t'=\infty$)

but we cannot prove anything about the probability of those two possibilities.

At the other extreme, suppose Mr.Bean gets from the drawer a sock of the opposite color as he throws away. Then the nondeterministic choices become

if $d\ t$ **then** $L:=red$ **else** $L:=blue$ $=$ $L:=R$
if $d\ t$ **then** $R:=blue$ **else** $R:=red$ $=$ $R:=L$

(which means that $d\ t$ just happens to have the same value as $L=blue \wedge R=red$ each time). Again, if I observed Mr.Bean doing that each time the experiment is rerun, I would suspect a mechanism or purpose, but the mathematics is silent about that. Now we can prove

if $L=R$ **then** $t'=t$ **else** $t'=t+1$ $\Leftarrow$
  if $L=R$ **then** ok
  else (**if** $h\ t$ **then** $L:=R$ **else** $R:=L$;
    $t:=t+1$; **if** $L=R$ **then** $t'=t$ **else** $t'=t+1$)

which says that execution takes time 0 or 1, but we cannot attach probabilities to those two possibilities. If we make no assumption at all about dt, leaving the nondeterministic choices unrefined, then the most we can prove about the execution time is

if $L=R$ **then** $t'=t$ **else** $t'>t$

Another way to refine the nondeterministic choice is with a probabilistic choice. If we attach probability $1/2$ to each of the values of dt, then the distribution of execution times is **if** $L=R$ **then** $t'=t$ **else** $(t'>t) \times 2^{t-t'}$. To prove it, we start with the right side of the refinement, weakening ok to $t'=t$.

$\Sigma ht, dt\cdot$ (**if** $L{=}R$ **then** $t'{=}t$
else (**if** ht **then if** dt **then** $L{:=}\ red$ **else** $L{:=}\ blue$
else if dt **then** $R{:=}\ blue$ **else** $R{:=}\ red$;
$t{:=}\ t{+}1$; **if** $L{=}R$ **then** $t'{=}t$ **else** $(t'{>}t) \times 2^{t-t'}$))
$\times\ 1/2 \times 1/2$ factor and sum

$=$ **if** $L{=}R$ **then** $t'{=}t$
else ($(L{:=}\ red;\ t{:=}\ t{+}1;$ **if** $L{=}R$ **then** $t'{=}t$ **else** $(t'{>}t) \times 2^{t-t'}$)
$+\ (L{:=}\ blue;\ t{:=}\ t{+}1;$ **if** $L{=}R$ **then** $t'{=}t$ **else** $(t'{>}t) \times 2^{t-t'}$)
$+\ (R{:=}\ blue;\ t{:=}\ t{+}1;$ **if** $L{=}R$ **then** $t'{=}t$ **else** $(t'{>}t) \times 2^{t-t'}$)
$+\ (R{:=}\ red;\ t{:=}\ t{+}1;$ **if** $L{=}R$ **then** $t'{=}t$ **else** $(t'{>}t) \times 2^{t-t'}$)) / 4
Substitution Law

$=$ **if** $L{=}R$ **then** $t'{=}t$
else ((**if** $red{=}R$ **then** $t'{=}t{+}1$ **else** $(t'{>}t{+}1) \times 2^{t+1-t'}$)
$+$ (**if** $blue{=}R$ **then** $t'{=}t{+}1$ **else** $(t'{>}t{+}1) \times 2^{t+1-t'}$)
$+$ (**if** $L{=}blue$ **then** $t'{=}t{+}1$ **else** $(t'{>}t{+}1) \times 2^{t+1-t'}$)
$+$ (**if** $L{=}red$ **then** $t'{=}t{+}1$ **else** $(t'{>}t{+}1) \times 2^{t+1-t'}$)) / 4
R is either red or $blue$, and similarly L

$=$ **if** $L{=}R$ **then** $t'{=}t$ **else** $(t'{=}t{+}1)\ /\ 2\ +\ (t'{>}t{+}1) \times 2^{t+1-t'}\ /\ 2$

$=$ **if** $L{=}R$ **then** $t'{=}t$ **else** $(t'{>}t) \times 2^{t-t'}$

which is the probability specification. That concludes the proof. The average value of t' is

$\Sigma t'\cdot\ t' \times$ **if** $L{=}R$ **then** $t'{=}t$ **else** $(t'{>}t) \times 2^{t-t'}$

$=$ **if** $L{=}R$ **then** t **else** $\Sigma t'\cdot\ t' \times (t'{>}t) \times 2^{t-t'}$

$=$ $t\ +$ **if** $L{=}R$ **then** 0 **else** $\Sigma n{:}\ nat{+}1\cdot\ n\ /\ 2^n$

$=$ $t\ +$ **if** $L{=}R$ **then** 0 **else** 2

so, if the initial socks don't match, Mr.Bean draws an average of two more socks from the drawer.

In the previous paragraph, we chose to leave the initial drawing nondeterministic, and to assign probabilities to the drawing of subsequent socks. Clearly we could attach probabilities to the initial state too. Or we could attach probabilities to the initial state and leave the subsequent drawings nondeterministic. The theory is quite general. But in this problem, if we leave both the initial and subsequent drawings nondeterministic, attaching probabilities only to the choice of hand, we can say nothing about the probability of execution times or average execution time.

13 Partial Probabilistic Specifications

Suppose we want x to be 0 one-third of the time. We don't care how often x is 1 or 2 or anything else, as long as x is 0 one-third of the time. To express the distribution of x would be overspecification. The first two lines below specify just what we want, and the last two lines are one way to refine the specification as a distribution.

if $1/3$ **then** $x{=}0$ **else** $x{\neq}0$

$=$ $(x{=}0)/3 + (x{\neq}0){\times}2/3$

$\geq$ $(x{=}0)/3 + (x{=}1){\times}2/3$

$=$ **if** $1/3$ **then** $x{=}0$ **else** $x{=}1$

In general, a superdistribution is a partial probabilistic specification, which can be refined to a distribution. In general, a subdistribution is unimplementable.

Now suppose we want x to be 0 or 1 one-third of the time, and to be 1 or 2 one-third of the time. Two distributions that satisfy this informally stated specification are

$(x{=}0)/3 + (x{=}2)/3 + (x{=}3)/3$
$(x{=}1)/3 + (x{=}3){\times}2/3$

The smallest expression that is greater than or equal to both these expressions (the most refined expression that is refined by both these expressions) is

$(x{=}0)/3 + (x{=}1)/3 + (x{=}2)/3 + (x{=}3){\times}2/3$

Unfortunately, this new expression is also refined by

$(x{=}2)/3 + (x{=}3){\times}2/3$

which does not satisfy the informally stated specification. The problem is known as convex closure, and it prevents us from formalizing the specification as a superdistribution. We must return to the standard form of specification, a boolean expression, this time about the partially known distribution. Let $p\,x$ be the probability distribution of x. Then what we want to say is

$(\forall x{\cdot}\ 0{\le}px{\le}1)\ \wedge\ (\Sigma x{\cdot}\ px){=}1\ \wedge\ p0{+}p1 = p1{+}p2 = 1/3$

This specification can be refined in the normal way: by reverse implication. For example,

$(\forall x{\cdot}\ 0{\le}px{\le}1)\ \wedge\ (\Sigma x{\cdot}\ px){=}1\ \wedge\ p0{+}p1 = p1{+}p2 = 1/3$
$\Leftarrow\quad p0 = p2 = p3 = 1/3\ \wedge\ \forall x{:}\ x{\neq}0 \wedge x{\neq}2 \wedge x{\neq}3{\cdot}\ px{=}0$
$=\quad \forall x{\cdot}\ px = ((x{=}0)/3 + (x{=}2)/3 + (x{=}3)/3)$

14 Conclusion

Our first approach to probabilistic programming was to reinterpret the types of variables as probability distributions expressed as functions. In that approach, if x was a variable of type T, it becomes a variable of type $T{\to}prob$ such that $\Sigma x = \Sigma x' = 1$. All operators then need to be extended to distributions expressed as functions. Although this approach works, it was too low-level; a distribution expressed as a function tells us about the probability of its variables by their positions in an argument list, rather than by their names. So we opened the probability expressions, leaving free the variables whose probabilities are being described.

By considering specifications and programs to be boolean expressions, and by considering boolean to be a subtype of numbers, we can make probabilistic calculations directly on programs and specifications. Without any new mechanism, we include probabilistic timing. From the distribution of execution times we can calculate the average execution time; this is often of more interest than the worst case execution time, which is the usual concern in computational complexity.

We include an **if then else** notation (as is standard), and we have generalized booleans to probabilities (as in [4]), so we already have a probabilistic choice notation (for example, **if** 1/3 **then** P **else** Q); there is no need to invent another. We have used the *rand* "function", not because we advocate it (we don't), but because it is found in many programming languages; we cope with it by replacing it with something that obeys the usual laws of mathematical calculation.

Informal reasoning to arrive at a probability distribution, as is standard in studies of probability, is essential to forming a reasonable hypothesis. But probability problems are notorious for misleading even professional mathematicians; hypotheses are sometimes wrong. Sometimes the misunderstanding can be traced to a different understanding of the problem. Our first step, formalization as a program, makes one's understanding clear. After that step, we offer a way to prove a hypothesis about probability distributions.

Nondeterministic choice is handled by introducing a variable to represent the nondeterminacy. In [4], instead of calculating probabilities, they calculate a lower bound on probabilities: they find the precondition that ensures that the probability of outcome σ' is

at least p. In contrast to that, from the distribution of prestates we calculate the entire range of possible distributions of poststates. With less mechanism we obtain more information. We did not treat nondeterministic choice and probabilistic choice as different kinds of choice; nondeterminism can be refined, and one way to refine it, is probabilistically; the "at least" inequality is the generalization of refinement.

The convex closure problem, which prevents partial probabilistic specification, is a serious disappointment. It limits not only the work described in this paper, but any attempt to generalize specifications to probabilities, such as [4] where it is discussed at length. The only way around it seems to be to abandon probabilistic specification, and to write boolean specifications about distribution-valued variables.

Probabilistic specifications can also be interpreted as "fuzzy" specifications. For example, $(x'=0)/3 + (x'=1)\times 2/3$ could mean that we will be one-third satisfied if the result x' is 0, two-thirds satisfied if it is 1, and completely unsatisfied if it is anything else.

15 Acknowledgements

I thank Carroll Morgan for getting me interested in probabilistic programming, and for consultation concerning nondeterminism. I thank Yannis Kassios for a suggestion concerning the sequential composition of probabilistic specifications.

16 References

[0] E.C.R.Hehner: "Predicative Programming", *Communications ACM*, volume 27, number 2, pages 134-151, 1984 February

[1] E.C.R.Hehner: *a Practical Theory of Programming*, Springer, New York, 1993; second edition 2004 available free at www.cs.utoronto.ca/~hehner/aPToP

[2] C.A.R.Hoare: "Programs are Predicates", in C.A.R.Hoare, J.C.Shepherdson (editors): *Mathematical Logic and Programming Languages*, Prentice-Hall Intenational, pages 141-154, 1985

[3] D.Kozen: Semantics of Probabilistic Programs, *Journal of Computer and System Sciences*, volume 22, pages 328-350, 1981

[4] C.C.Morgan, A.K.McIver, K.Seidel, J.W.Sanders: "Probabilistic Predicate Transformers", *ACM Transactions on Programming Languages and Systems*, volume 18, number 3, pages 325-353, 1996 May