1

Introduction to pGCL:
Its logic and its model

1.1 Sequential program logic

Since the mid-1970’s, any serious student of rigorous program development
will have encountered “assertions about programs” — they are predicates
which, when inserted into program code, are supposed to be “true at that
point of the program.” Formalised — i.e., made into a logic — they look
like either

\[
\begin{align*}
\{\text{pre}\} \text{ prog} \{\text{post}\} & , \quad \text{Hoare-style} \\
\text{pre} \Rightarrow \text{wp.prog.post} , & \quad \text{Dijkstra-style}
\end{align*}
\]

(1.1)
in each case meaning “from any state satisfying precondition pre, the
sequential program prog is guaranteed to terminate in a state satisfying
postcondition post.” Formulae pre and post are written in first-order
predicate logic over the program variables, and prog is written in a sequen-
tial programming language. Often Dijkstra’s Guarded Command Language
[Dij76], called GCL, is used in simple expositions like this one, since it
contains just the essential features, and no clutter.

A conspicuous feature of Dijkstra’s original presentation of guarded com-
mands was the novel “demonic” choice. He explained that it arose naturally
if one developed programs hand-in-hand with their proofs of correctness:
if a single specification admitted say two implementations, then a third
possibility was program code that seemed to choose unpredictably between
the two. Yet in its pure form, where for example

\[
\text{prog} \ominus \text{prog}'
\]

(1.2)
is a program that can unpredictably behave either as prog or as prog’,
this “demonic” nondeterminism seemed at first — to some — to be an
unnecessary and in fact gratuitously confusing complication. Why would
anyone ever want to introduce unpredictability deliberately? Programs are
unpredictable enough already.

If one really wanted programs to behave in some kind of “random” way,
then more useful surely would be a construction like the

\[
\text{prog} \ominus\oplus \text{prog}'
\]

(1.3)
that behaves as prog on half of its runs, and as prog’ on the other half. Of
course on any particular run the behaviour is unpredictable, and even over
many runs the proportions will not necessarily be exactly “50/50” — but
over a long enough period one will find approximately equal evidence of
each behaviour.

A logic and a model for programs like (1.3) was in fact provided in the
early 1980’s [Koz81, Koz85], where in the “Kozen style” the pre- and post-
formulae became real- rather than Boolean functions of the state, and \(\ominus\)
was replaced by \(\oplus\) in the programming language. Those logical statements

\[\text{We will use the Dijkstra-style.}\]**
1.1. Sequential program logic

(1.1) now took on a more general meaning, that “if program \( prog \) is run many times from the same initial state, the average value of \( post \) in the resulting final states is at least the actual value that \( pre \) had in the initial state.” Naturally we are relying on the expressions’ \( pre \) and \( post \) having real- rather than Boolean type when we speak of their average, or expected value.

The original — standard, we call it — Boolean logic was still available of course via the embedding \( false \rightarrow 0, 1 \).

Dijkstra’s demonic \( \sqcap \) was not so easily discarded, however. Far from being “an unnecessary and confusing complication,” it is the very basis of what is now known as refinement and abstraction of programs. (The terms are complementary: an implementation refines its specification; a specification abstracts from its implementation.) To specify “set \( r \) to a square-root of \( s \)” one could write directly in the programming language GCL

\[
\begin{align*}
\text{\texttt{r := } -\sqrt{s} \quad \text{\texttt{r := } } \sqrt{s}},
\end{align*}
\]

something that had never been possible before. This explicit, if accidental, “programming feature” caught the tide that had begun to flow in that decade and the following; the idea that specifications and code were merely different ways of describing the same thing (as advocated by Abrial, Hoare and others; making an early appearance in Back’s work [Bac78] on what became the Refinement Calculus [Mor88b, Bac88, Mor87, Mor94b, BvW98]; and as found at the heart of specification and development methods such as \( Z \) [Spi88] and \( VDM \) [Jon86]).

Unfortunately, probabilistic formalisms were left behind, and did not embrace the new idea: replacing \( \sqcap \) by \( \oplus \), they lost demonic choice; without demonic choice, they lost abstraction and refinement; and without those, they had no nontrivial path from specification to implementation, and no development calculus or method.

\(^2\)Admittedly this is a rather clumsy notation when compared with those designed especially for specification, e.g.

\[
\begin{align*}
\text{\texttt{r := } } \sqrt{s} & & \text{a specification statement (Back, Morgan, Morris)} \\
(\text{\texttt{r := } } \sqrt{s})^2 & = s & \text{(the body of) a \( Z \) schema (Abrial, Oxford)} \\
\text{any } r' \text{ with } (r')^2 & = s & \text{then } r := r' \end{align*}
\]

But the point is that the specification could be written in a “programming language” at all: it was beginning to be realised that there was no reason to distinguish the meanings of specifications and of programs (a point finally crystallised in the subtitle Assigning Programs to Meanings of Abrial’s book [Abr96a], itself a reference 30 years further back to Floyd’s paper [Flo67] where it all began).

6 1. Introduction to \( pGCL \)

To have a probabilistic development method, we need both \( \sqcap \) and \( \oplus \) — we cannot abandon one for the other. Using them together, we can for example describe “flip a nearly fair coin” as

\[
c := \text{heads} \oplus 0.49 \sqcap \text{tails} \quad \text{c := heads} \oplus 0.51 \sqcap \text{tails}.
\]

What we are doing here is specifying a coin which is within 1% of being fair — just as well, since perfect coins do not exist in nature, and so we could never implement a specification that required one.\(^3\) This program abstracts, slightly, from the precise probability of heads or tails.

In this introduction we will see how the seminal ideas of Floyd, Hoare, Dijkstra, Abrial and others can be brought together and replayed in the probabilistic context suggested by Kozen, and how the milestones of sequential program development and refinement — the concepts of

- program assertions;
- loop invariants;
- loop variants;
- program algebra (e.g. monotonicity and conjunctivity)

— can be generalised to include probability. Our simple programming language will be Dijkstra’s, but with \( \oplus \) added and — crucially — demonic choice \( \sqcap \) retained: we call it \( pGCL \).

Section 1.2 gives a brief overview of \( pGCL \) and its use of so-called expectations rather than predicates in its accompanying logic; Section 1.3 then supplies operational intuition by relating \( pGCL \) operationally to a form of gambling game. (The rigorous operational semantics is given in Chap. 5, and a deeper connection with games is given in Chap. 11.) Section 1.4 completes the background by reviewing elementary probability theory.

Section 1.5 gives the precise syntax and expectation-transformer semantics of \( pGCL \), using the infamous “Monty Hall” game as an example. Finally, in Sec. 1.6 we make our first acquaintance with the algebraic properties of \( pGCL \) programs.

Throughout we write \( f.x \) instead of \( f(x) \) for function application of \( f \) to argument \( x \), with left association so that \( f.g.x \) is \( (f(g))(x) \); and we use “:\=” for is defined to be. For syntactic substitution we write \( \text{expr} \left( \text{var} \mapsto \text{term} \right) \).

\(^3\)That means that probabilistic formalisms without abstraction in their specifications must introduce probability into their refinement operator if they are to be of any practical use: writing for example \( prog \sqsubseteq 0.99 \) \( prog' \) can be given a sensible meaning even if the probability in \( prog \) is exact [DGJP02, vBMOW03, Yin03]. But we do not follow that path here.
to indicate replacing \textit{var} by \textit{term} in \textit{expr}. We use “overbar” to indicate complement both for Booleans and probabilities: thus \texttt{true} is \texttt{false}, and \texttt{p} is \texttt{1} \texttt{- p}.

1.2 The programming language \textit{pGCL}

We’ll use \textit{square brackets} [·] to convert Boolean-valued predicates to arithmetic formulae which, for reasons explained below, we call \textit{expectations}. Stipulating that \texttt{[false]} is zero and \texttt{[true]} is one makes \texttt{[P]} in a trivial sense the probability that a given predicate \texttt{P} holds: if false, it holds with probability zero; if true, it holds with probability one.\footnote{Note that this nicely complements our “overbar” convention, because for any predicate \texttt{P} the two expressions \texttt{[P]} and \texttt{[\neg P]} are therefore the same.}

For our first example, consider the simple program

\begin{equation}
\begin{array}{l}
x := -y \\
\hat{x} := y
\end{array}
\end{equation}

(1.5)

over integer variables \texttt{x, y}, using the new construct \texttt{\hat{\oplus}} which we interpret as “choose the left branch \texttt{x := \neg y} with probability \texttt{1/3}, and choose the right branch with probability \texttt{1 - 1/3}.”

Recall [Dij76] that for any predicate \texttt{post over final states}, and a standard command \texttt{prog}, the “weakest precondition” predicate \texttt{wp prog. post} acts over initial states: it holds just in those initial states from which \texttt{prog} is guaranteed to reach \texttt{post}. Now suppose \texttt{prog} is probabilistic, as Program (1.5) is: what can we say about the \textit{probability} that \texttt{wp prog. post} holds in some initial state?\footnote{Throughout we use \texttt{standard} to mean “non-probabilistic.”}

It turns out that the answer is just \texttt{wp prog. [post]}, once we generalise \texttt{wp prog} to expectations instead of predicates. For that, we begin with the two definitions\footnote{Here we are defining the language as we go along, but all the definitions are collected together in Fig. 1.5.3 (p. 26).}

\begin{align}
\text{wp}(x := E). postE & := \text{“postE with } x \text{ replaced everywhere by } E” \quad \text{(1.6)} \\
\text{wp}(\text{prog} \oplus \text{prog'}). postE & := \text{p * wp prog. postE} \quad \text{(1.7)} \\
& \quad \text{+ } \text{p * wp prog'. postE}
\end{align}

in which \texttt{postE} is an expectation, and for our example program we ask what is the \textit{probability} that the predicate “the final state will satisfy \texttt{x \geq 0}” holds in some given initial state of the program (1.5)?

To find out, we calculate \texttt{wp prog. [post]} using the definitions above; that is

\begin{align*}
\text{wp}(x := -y \hat{\oplus} x := +y). [x \geq 0]
\end{align*}

\begin{align*}
\equiv^8 & \quad (1/3) \ast \text{wp}(x := -y). [x \geq 0] \\
& \quad \text{+ (2/3) \ast \text{wp}(x := +y). [x \geq 0]}
\end{align*}

\begin{align*}
\equiv & \quad (1/3) \ast [y \geq 0] \text{ + (2/3) \ast } [y \geq 0] \ast [y < 0]/3 + [y = 0] + 2 [y > 0]/3. \quad \text{(using arithmetic)}
\end{align*}

Thus our answer is the last arithmetic formula above, which we call a “pre-expectation” — and the probability we seek is found by reading off the formula’s value for various initial values of \texttt{y}, getting

\begin{align*}
\text{when } y < 0, & \quad 1/3 + 0 + 2(0)/3 = 1/3 \\
\text{when } y = 0, & \quad 0/3 + 1 + 2(0)/3 = 1 \\
\text{when } y > 0, & \quad 0/3 + 0 + 2(1)/3 = 2/3.
\end{align*}

Those results indeed correspond with our operational intuition about the effect of \texttt{\hat{\oplus}}.

For our second example we illustrate abstraction from probabilities: a demonic version of Program (1.5) is much more realistic in that we set its probabilistic parameters only within some tolerance. We say informally (but still precisely) that

\begin{align}
\bullet \ x := \neg y & \text{ is to be executed with probability at least } 1/3, \\
\bullet \ x := +y & \text{ is to be executed with probability at least } 1/4 \text{ and} \\
\bullet \ & \text{it is certain that one or the other will be executed.} \quad \text{(1.8)}
\end{align}

Equivalently we could say that alternative \texttt{x := \neg y} is executed with probability between \texttt{1/3} and \texttt{3/4}, and that otherwise \texttt{x := +y} is executed (therefore with probability between \texttt{1/4} and \texttt{2/3}).

With demonic choice we can write Specification (1.8) as

\begin{equation}
x := \neg y \hat{\oplus} x := +y \quad \sqcap \quad x := \neg y \hat{\oplus} x := +y. \quad \text{(1.9)}
\end{equation}

because we do not know or care whether the left or right alternative of \texttt{\sqcap} is taken — and it may even vary from run to run of the program, resulting in an “effective” \texttt{\hat{\oplus}} with \texttt{p} somewhere between the two extremes.\footnote{Later we explain the use of “\texttt{\sqcap}” rather than “\texttt{\sqor}.”}

\footnote{We will see later that a convenient notation for (1.9) uses the abbreviation \texttt{prog p\hat{\sqor} prog’ := prog p\hat{\sqor} prog’ \sqcap prog’ p\hat{\sqor} prog’;} we would then write it \texttt{x := \neg y \hat{\oplus} x := +y}, or even \texttt{x := \neg y \hat{\oplus} +y}. }
To treat Program (1.9) we need a third definition, 
\[ wp.( \text{prog} \sqcap \text{prog'}).\text{postE} \;:=\; wp.\text{prog}.\text{postE} \;\min\; wp.\text{prog'}.\text{postE}, \]  
(1.10)
using \( \min \) because we regard demonic behaviour as attempting to make the achieving of post as improbable as it can. Repeating our earlier calculation (but more briefly) gives this time
\[ wp.\text{Program (1.9)}.[x \geq 0] \]
\[ \equiv \frac{[y \leq 0]}{3} + \frac{2[y \geq 0]}{3} \]  
using (1.6), (1.7), (1.10)
\[ \equiv \frac{[y < 0]}{3} + \frac{[y = 0]}{4} + \frac{[y > 0]}{4}. \]  
using arithmetic

Our interpretation has become

\begin{itemize}
  \item When \( y \) is initially negative, a demon chooses the left branch of \( \sqcap \) because that branch is more likely (2/3 vs. 1/4) to execute \( x := +y \) — the best we can say then is that \( x \geq 0 \) will hold with probability at least 1/3.
  \item When \( y \) is initially zero, a demon cannot avoid \( x \geq 0 \) — either way the probability of \( x \geq 0 \) finally is one.
  \item When \( y \) is initially positive, a demon chooses the right branch because that branch is more likely to execute \( x := -y \) — the best we can say then is that \( x \geq 0 \) finally with probability at least 1/4.
\end{itemize}

The same interpretation holds if we regard \( \sqcap \) as abstraction instead of as run-time demonic choice. Suppose Program (1.9) represents some mass-produced physical device and, by examining the production method, we have determined the tolerance (1.8) we can expect from a particular factory. If we were to buy one from the warehouse, all we could conclude about its probability of establishing \( x \geq 0 \) is just as calculated above.

Refinement is the converse of abstraction: we have

**Definition 1.2.1 Probabilistic refinement** For two programs \( \text{prog}, \text{prog'} \) we say that \( \text{prog'} \) is a refinement of \( \text{prog} \), written \( \text{prog} \sqsubseteq \text{prog'} \), whenever for all post-expectations \( \text{postE} \) we have
\[ \text{wp.}\text{prog}.\text{postE} \;\Rightarrow\; \text{wp.}\text{prog'}.\text{postE} \]  
(1.11)

We use the symbol \( \Rightarrow \) for \( \leq \) (extended pointwise) between expectations, which emphasises the similarity between probabilistic- and standard refinement.\(^{10} \)

\[ x := -y \;\dagger\dagger \; x := +y \]  
(1.12)

From (1.11) we see that in the special case when expectation \( \text{postE} \) is an embedded predicate \( \text{[post]} \), the meaning of \( \Rightarrow \) ensures that a refinement \( \text{prog'} \) of \( \text{prog} \) is at least as likely to establish \( \text{post} \) as \( \text{prog} \) is.\(^{11} \) That accords with the usual definition of refinement for standard programs — for then we know \( \text{wp.}\text{prog}.\text{[post]} \) is either zero or one, and whenever prog is certain to establish post (whenever \( \text{wp.}\text{prog}.\text{[post]} \equiv 1 \)) we know that \( \text{prog'} \) also is certain to do so (because then \( 1 \Rightarrow \text{wp.}\text{prog'}.\text{[post]} \)).

For our third example we prove a refinement: consider the program

\[ x := -y \;\dagger\dagger \; x := +y \]  
(1.12)

which clearly satisfies Specification (1.8); thus it should refine Program (1.9), which is just that specification written in pGCL. With Definition (1.11), we find for any \( \text{postE} \) that
\[ \text{wp.}\text{Program (1.12)}.\text{postE} \]
\[ \equiv \text{wp}.(x := -y).\text{postE}/2 \quad \text{definition } \dagger\dagger, \text{at (1.7)} \]
\[ + \text{wp}.(x := +y).\text{postE}/2 \]
\[ \equiv \text{postE}^-/2 + \text{postE}^+/2 \quad \text{introduce abbreviations} \]
\[ \equiv (3/5)(\text{postE}^-/3 + 2\text{postE}^+/3) \quad \text{arithmetic} \]
\[ + (2/5)(3\text{postE}^-/4 + \text{postE}^+/4) \]
\[ \leq \text{postE}^-/3 + 2\text{postE}^+/3 \quad \text{any linear combination exceeds min} \]
\[ \min \text{3postE}^-/4 + \text{postE}^+/4 \]
\[ \equiv \text{wp.}\text{Program (1.9)}.\text{postE}. \]

The refinement relation (1.11) is indeed established for the two programs.

The introduction of 3/5 and 2/5 in the third step can be understood by noting that demonic choice \( \sqcap \) can be implemented by any probabilistic choice whatever: in this case we used \( \dagger\dagger \). Thus a proof of refinement using program algebra might read

\[ \text{Program (1.12)} \]
\[ x := -y \;\dagger\dagger \; x := +y \]

---

\(^{10}\)We are aware that "\( \dagger\dagger \)" looks more like "\( \geq \)" than it does "\( \leq \)"; but for us its resemblance to "\( \Rightarrow \)" is the important thing.

\(^{11}\)Similar conflicts of interest arise when logicians use "\( \supset \)" for implies although, interpreted set-theoretically, implies is in fact "\( \subseteq \)". And then there is "\( \sqsubseteq \)" for refinement, which corresponds to "\( \supseteq \)" of behaviours.
1.3. An informal computational model for \( pGCL \)

We now use a simple card-and-dice game as an informal introduction to the computational model for \( pGCL \), to support the intuition for probabilistic choice, demonic choice and their interaction. To start with, we consider the simplest case: non-looping programs without \( \sqcap \) or \( \sqcup \).

### 1.3.1 The standard game

Imagine 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 coloured markers indicate 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.

This simple game is deterministic: any initial state always leads to the same final state. And because the cards are removed after use it is also guaranteed to terminate, if the board is finite. It is easily generalised however to include other features of standard programs:

\[
\begin{align*}
= & \quad (x := -y \quad \sqcup\quad x := +y) \\
\sqcap\quad (x := -y \quad \sqcup\quad x := +y) \\
\sqcap & \quad x := -y \quad \sqcup\quad x := +y \quad (\forall) \subseteq (\forall \sqcup) \text{ for any } p^{12} \\
= & \quad \text{Program (1.9)} .
\end{align*}
\]

### 1.3.2 The probabilistic game

Suppose now that each card contains not just one but, rather, a list of successor squares, face-down, and the rules are modified so that the next state is determined by choosing just one of them “blind,” then play is nondeterministic.

Taking the demonic (pessimistic) view, the player should expect to lose unless he is guaranteed to reach a winning position no matter which blind choices he makes.

In the standard game, for each (initial) square one can examine the cards before playing to determine whether a win is guaranteed from there. But once the game has started, the cards are turned face-down.

The set of squares from which a win is guaranteed is the weakest precondition.\(^{13}\)

\( \sqcup \)

looping If the cards are not removed after use, the game can “loop.” A looping-forever player loses.

aborting If a card reads “go to jail,” the program is said to “abort” and the player can be sent to any square whatever, including a special supplementary “jail” square from which there is no escape. A jailed player loses.

deemonic nondeterminism If each square can contain several cards, face-down, and the rules are modified so that the next state is determined by choosing just one of them “blind,” then play is nondeterministic. Taking the demonic (pessimistic) view, the player should expect to lose unless he is guaranteed to reach a winning position no matter which blind choices he makes.

In the probabilistic game one can ask for the greatest guaranteed probability of winning; as in the standard case, the prediction will vary depending on the initial square. (It’s because of demonic nondeterminism, as illustrated below, that the probability might be only a lower bound.)

---

\(^{12}\)By \( (\forall) \subseteq (\forall \sqcup) \) we mean that for all \( \text{prog} \) we have \( \text{prog} \sqcap \text{prog}' \subseteq \text{prog} \sqcup \text{prog}' \), which is an instance of our Law 7 given on p. 323, in Sec. B.1 on program algebra.

\(^{13}\)A glance at Fig. 6.7.1 (p. 173) will show where we are headed in the visualisation of probabilistic preconditions!

\(^{14}\)Note that we still call this game “deterministic,” in spite of the probabilistic choices, and there are good mathematical reasons for doing so. (In Chap. 5, for example, we see that such programs are maximal in the refinement order.) An informal justification is that deterministic programs are those with repeatable behaviours and, even for probabilistic programs, the output distribution is repeatable (to within statistical confidence measures) provided the program contains no demonic choice; see e.g. p. 135.
1.3. An informal computational model for pGCL

In Fig. 1.3.1 is an example game illustrating some of the above points. The greatest 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 in the pile; but he might be unlucky enough never to choose the first card, and we must assume the worst.

1.3.3 Expected winnings in the probabilistic game

For standard programs, the computational model of execution supports a complementary, “logical” view — given a set of final states (the postcondition) we 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.\(^{15}\)

Probabilistic programs become functions from probabilistic preconditions to probabilistic weakest preconditions — we call them post-expectations and greatest pre-expectations. 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. 1.3.2 we show the effect of translating our original game. In fact, not much changes: the probability of winning (in Fig. 1.3.1) translates into the equivalent expected payoff (Fig. 1.3.2) as the corresponding fraction of £1, illustrating this important fact:

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 post-expectation) 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 greatest pre-expectation) — also therefore a function from states to \(\mathbb{R}_\geq\).

\(^{15}\)In later chapters we will be more precise about the range of expectations, requiring them in particular to be bounded above.

To play from a square, you first pick one of the face-down cards. (In the diagram, we are seeing what’s on the cards with our X-ray vision.) Then you roll a dice to choose one of the alternatives on the card. (In this case the dice is two-sided, i.e. it is a coin.)

As special cases, a standard step (non-probabilistic) has only one alternative per card, but possibly many cards; and a deterministic step has only one card, but possibly many alternatives on it. A standard and deterministic step has one card, and only one alternative.

The winning final positions — the postcondition — are the states \{4,5\}, marked with a £1 coin. 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 zero, since the second card might be chosen every time.

The probabilities are summarised in Fig. 1.3.2.

Figure 1.3.1. Card-and-dice game operational semantics for pGCL.
1.3. An informal computational model for $pGCL$

The post-expectation:

Final state

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Payoff awarded if this state reached

The probability of winning (ending on a £1) (from Fig. 1.3.1):

Initial state

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/0</td>
<td>1/1</td>
<td>1/1</td>
<td></td>
</tr>
</tbody>
</table>

Greatest guaranteed probability of winning

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>50p</td>
<td>50p</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1.3.2. A probabilistic and nondeterministic gambling game

Since the functions are expectations, the program is being regarded as an expectation transformer.\(^{16}\)

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 of Fig. 1.3.2 were executed after some other program prog, the precondition of the two together with respect to the postcondition \(\{4, 5\}\) would be calculated by applying \(wp. prog\) to the greatest pre-expectation table for game. That is because sequential composition of programs becomes, as usual, functional composition of the corresponding transformers: we have

\[
\text{expected win table}\]

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

and that table 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, and the semantics becomes non-compositional.

(See Sec. A.1 for why this happens.)

16For deterministic (yet probabilistic) programs, the card-game model and the associated transformers are essentially Kozen’s original construction [Koz81, Koz85]. We have added demonic (and later angelic) nondeterminism.

1.4 Behind the scenes: elementary probability theory

In probability theory, an event is a subset of some given sample space \(S\), so that the event is said to have occurred if the sampled value is in that set; a probability distribution \(Pr\) 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 the sample space are events.\(^{17}\)

In our case we consider countable sample spaces, and take every (sub-)set of \(S\) to be an event — and so we can regard a probability distribution more simply as a function from \(S\) directly to probabilities (rather than from its subsets). Thus \(Pr: S \to [0,1]\), and the probability of a more general event is now just the sum of the probabilities of its elements: we are using discrete distributions.\(^{18}\)

A random variable \(X\) is a function from the sample space to the non-negative reals,\(^{19}\) and the expected value \(\text{Exp}. X\) of that random variable is defined in terms of the (discrete) probability distribution \(Pr\); we have the summation

\[
\text{Exp}. X := \left( \sum_{s \in S} Pr.s \times X.s \right).
\]

It represents the “average” value of \(X.s\) over many repeated samplings of \(s\) according to the distribution \(Pr\).\(^{20}\)

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

If a function \(\text{Exp}\) is of type \((S \to \mathbb{R}_\geq) \to \mathbb{R}_\geq\), and it is

- **non-negative** so that \(\text{Exp}.X \geq 0\) for all \(X: S \to \mathbb{R}_\geq\),
- **linear** so that for \(X, Y: S \to \mathbb{R}_\geq\) and \(c, d: \mathbb{R}_\geq\) we have

\[
\text{Exp}.(c \times X + d \times Y) = c \times \text{Exp}.X + d \times \text{Exp}.Y
\]

\(^{17}\)This may occur if the sample space is uncountable, for example; the general technique for such cases involves \(\sigma\)-algebras [GSS92]. See Footnote 7 on p. 297 for an example.

\(^{18}\)The price paid for using discrete distributions is that there are some “everyday” situations we cannot describe, such as the uniform “continuous” 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 \(pGCL\) — at least, not at this stage.

\(^{19}\)Footnote 12 on p. 134 gives a more generous definition.

\(^{20}\)Although the parentheses may look odd around \(\sum\) — we write \((\sum \cdots)\) rather than \(\sum (\cdots)\) — we always indicate the scope of bound variables (like \(s\)) with explicit delimiters, since it helps to avoid errors when doing calculations.

\(^{21}\)Our “important fact” (p. 13) is now stated “if \(X\) is the characteristic function of some event \(P\), then \(\text{Exp}.X\) is the probability that event \(P\) will occur.”
and normalised so that it satisfies $\text{Exp}_\varnothing = 1$, where $\varnothing$ is the constant function returning 1 for all arguments in $S$,

then it is an expectation over some probability distribution: it can be shown that it is expressible uniquely in the form (1.13) for some $\mathbf{Pr}$.

The relevance of the above is that our real-valued expressions over the state — what we are calling “expectations” — are random variables, and that the expression

$$\wp_{\text{prog}} \cdot \text{postE},$$  \hspace{1cm} (1.14)

as a function of initial values for the state variables, is a random variable as well. As a function of state variables, it is the expected value of the random variable postE (also a function of state variables, but those taken after execution) over the distribution of final states produced by executions of prog, and so

$$\text{preE} \Leftrightarrow \wp_{\text{prog}} \cdot \text{postE}$$ \hspace{1cm} (1.15)

says that preE gives in any initial state a lower bound for the expected value of postE in the final distribution reached via execution of prog begun in that initial state.

In general, we call random variables post-expectations when they are to be evaluated in a final state, and we call them pre-expectations when they are calculated as at (1.14). And, like pre- and postconditions in standard programs, if placed “between” two programs a single random variable is a post-expectation for the first and a pre-expectation for the second.

But how do prog and an initial state determine a distribution? In fact the underlying distributions are found on the cards of the game from Sec. 1.3 — 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.\footnote{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 [F71, p. 135].} For any postcondition postE written, say, as an expression over names $N$ of squares, and initial square $N_0$, the expression $\wp_{\text{move}} \cdot \text{postE} (N \mapsto N_0)$ is the expectation of postE over the distribution of square names given on the card found at $N_0$.

\footnote{For nondeterministic programs we are thus considering a function from state to sets of distributions, from a square to the set of cards there; again we see the general computational model underlying the expectation-transformer semantics.}

For example, in Figs. 1.3.1 and 1.3.2 we see the above features: program move is given by the layout of the cards (Fig. 1.3.1); and the resulting pre- and post-expectations are tabulated in Fig. 1.3.2. All three tables there are random variables over the state space $\{0, \ldots, 6\}$.

When we move to more general programs, we must relax the conditions that characterise expectations. If prog is possibly nonterminating — if it is recursive or contains abort — then $\wp_{\text{prog}} \cdot \text{postE}$ may violate the normalisation condition $\text{Exp}_{\varnothing} = 1$. 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 [Koz81] and later Jones [Jon90], who defined expectations with respect to “probability distributions” which may sum to less than one. Those are in fact a special case of Jones’s evaluations,\footnote{She was working in a much more general context.} and she gave conditions similar to the above for their existence [Jon90, p. 117].

Finally, if program prog is not deterministic then we move further away from elementary theory, because $\wp_{\text{prog}} \cdot \text{postE}$ is no longer an expectation even in the weak sense: it is not linear. It is still however the minimum of the two expectations $\wp_{\text{prog}} \cdot \text{postE}$ and $\wp_{\text{prog}}' \cdot \text{postE}$. This definition is one of the main features of this approach.

Thus although linearity is lost, it is not gone altogether: we retain so-called sub-linearity,\footnote{The actual property is slightly more general than we give here; see Sec. 1.6.} which implies that for any $c_1, c_2 \in \mathbb{R}_2$ and any program prog we still have

$$\wp_{\text{prog}} (c_1 \cdot \text{postE}_1 + c_2 \cdot \text{postE}_2) \leq c_1 \cdot \wp_{\text{prog}} \cdot \text{postE}_1 + c_2 \cdot \wp_{\text{prog}} \cdot \text{postE}_2.$$

And clearly non-negativity continues to hold.

The characterisations of expectations given above for the simpler cases might suggest that non-negative and sublinear functionals uniquely determine a set of probability distributions — and, in Chap. 5, that is indeed shown to be the case: sublinearity is the key “healthiness condition” for expectation transformers.\footnote{Halpern and Pucella [HP02] have recently studied similar properties.}

1.5 Basic syntax and semantics of $pGCL$

1.5.1 Syntax

Let prog range over programs and $p$ over real number expressions taking values between zero and one inclusive; assume that $x$ stands for a list of distinct variables, and $\text{expr}$ for a list of expressions (of the same length as $x$
where appropriate); and let the program \textit{scheme} \( \mathcal{C} \) be a program in which program \textit{names} like \( xxx \) can appear. The syntax of \( pGCL \) is as follows:

\[
\begin{align*}
\text{prog} & := \text{abort} \mid \text{skip} \mid x := E \mid \text{prog} \cdot \text{prog} \\
\text{prog} & = \text{prog} \oplus \text{prog} \mid \text{prog} \cap \text{prog} \\
\text{mu} & \ (xxx \cdot \mathcal{C})
\end{align*}
\]

The first four constructs, namely \texttt{abort, skip, assignment and sequential composition}, are just the conventional ones [Dij76]. The remaining constructs are for probabilistic choice, nondeterministic choice and recursion: given \( p \) in the closed interval \([0, 1] \) we write \( \text{prog} \oplus \text{prog} \) for the probabilistic choice between programs \( \text{prog} \) and \( \text{prog}' \); they have probability \( p \) and \( 1-p \) respectively of being selected. In many cases \( p \) will be a constant, but in general it can be an expression over the state variables.

### 1.5.2 Shortcuts and “syntactic sugar”

For convenience we extend our logic and language with the following notations.

**Boolean embedding** — For predicate \( \text{pred} \) we write \([\text{pred}]\) for the expectation “1 if \( \text{pred} \) else 0”.27

**Conditional** — The conditional

\[
\begin{align*}
\text{prog if } \text{pred} \text{ else } \text{prog}' \mid \text{or if } \text{pred} \text{ then } \text{prog} \text{ else } \text{prog}' \text{ fi}
\end{align*}
\]

chooses program \( \text{prog} \) (resp. \( \text{prog}' \)) if Boolean \( \text{pred} \) is true (resp. false). It is defined \( \text{prog} \ [\text{pred}] \oplus \text{prog}' \).

If \( \text{else} \) is omitted then \texttt{else skip} is assumed. (See also the “hybrid” conditional of Sec. 3.1.2.)

**Implication-like relations** — For expectations \( \exp, \exp' \) we write

\[
\exp \Rightarrow \exp' \text{ for } \exp \text{ is everywhere less than or equal to } \exp'
\]

\[
\exp \equiv \exp' \text{ for } \exp \text{ and } \exp' \text{ are everywhere equal}
\]

\[
\exp \subseteq \exp' \text{ for } \exp \text{ is everywhere greater than or equal to } \exp'
\]

We distinguish \( \exp \Rightarrow \exp' \) from \( \exp \leq \exp' \) — the former is a statement \textit{about} \( \exp \) and \( \exp' \), thus true or false as a whole; the latter is itself a Boolean-valued expression over the state, possibly true in some states and false in others.28 Similarly we regard \( \exp = \exp' \) as

true in just those states where \( \exp \) and \( \exp' \) are equal, and false in the rest.

The closest standard equivalent of \( \Rightarrow \) is the entailment relation \( |\) between predicates\( ^{29} \) — and in fact \( \text{post} |\) \( \text{post}' \) exactly when \( \text{[post]} \Rightarrow \text{[post]}' \)”, meaning that the “embedding” of \( \Rightarrow \) is \( \Rightarrow \).

**Multi-way probabilistic choices** — A probabilistic choice over \( N \) alternatives can be written horizontally

\[
(\text{prog}_1 \oplus p_1 \mid \cdots \mid \text{prog}_N \oplus p_N)
\]

or vertically

\[
| \begin{align*}
\text{prog}_1 @ p_1 \\
\text{prog}_2 @ p_2 \\
\vdots \\
\text{prog}_N @ p_N
\end{align*}|
\]

in which the probabilities are enumerated and sum to no more than one.30 We can also write a “probabilistic comprehension” \( [\ i : I \cdot \text{prog}_i @ p_i \ ] \) over some countable index set \( I \). In general, we have

\[
\text{wp.} (\text{prog}_1 \oplus p_1 | \cdots | \text{prog}_N \oplus p_N \mid \text{post}_E) := \text{wp.} (\text{prog}_1 @ p_1 | \cdots | \text{prog}_N @ p_N \mid \text{post}_{E}) + \cdots + \text{wp.} (\text{prog}_N @ p_N \mid \text{post}_{E}) .
\]

It means “execute \( \text{prog}_1 \) with probability at least \( p_1 \), and \( \text{prog}_2 \) with probability at least \( p_2 \) ... “31

If the probabilities sum to 1 exactly, then it is a simple \( N \)-way probabilistic branch; if there is a deficit \( 1-\Sigma p_i \), it gives the probability of aborting.

When all the programs \( \text{prog}_i \) are assignments with the same left-hand side, say \( x := \text{expr}_i \), we write even more briefly

\[
x := (\text{expr}_1 @ p_1 | \cdots | \text{expr}_N @ p_N) .
\]

**Variations on \( \oplus \)** — By \( \text{prog} \oplus \text{prog}' \) we mean \( \text{prog} \oplus \text{prog}' \) and in general we write \( \text{prog} \oplus \text{prog}' \) for

\[
| \begin{align*}
\text{prog} @ p \\
\text{prog}' @ p' \\
\text{prog} \cap \text{prog}' @ 1 - (p+p')
\end{align*}|
\]

the program that executes \( \text{prog} \) with probability at least \( p \) and \( \text{prog}' \)

\[ ^{29} \text{One predicate ENTAILS another, written } \models , \text{just when it implies the other in all states.} \]

\[ ^{30} \text{See Sec. 4.3 for an example of the vertical notation.} \]

\[ ^{31} \text{It is ”at least } p_i , \text{because if the probabilities sum to less than one there will be an ”aborting” component, which might behave like } \text{prog}_i . \]
with probability at least \( p' \); we assume \( p + p' \leq 1 \). By \( \geq p \otimes \) we mean \( p \otimes 0 \), and so on. (See also (B.3) on p. 328.)

**Demonic choice** — We write demonic choice between assignments to the same variable \( x \) as

\[
x \in \{ \text{expr}_1, \text{expr}_2, \cdots \},
\]

or

\[
x = \text{expr}_1 \land \text{expr}_2 \land \cdots ,
\]

in each case abbreviating \( x = \text{expr}_1 \lor \text{expr}_2 \lor \cdots \). More generally we can write \( x \in \text{expr} \) or \( x \notin \text{expr} \) if \( \text{expr} \) is set-valued, provided the implied choice is finite.\(^{32}\)

**Iteration** — The construct \((\mu x \text{xxx} \cdot C)\) behaves as prescribed by the program context \( C \) except that it invokes itself recursively whenever it reaches a point where the program name \( \text{xxx} \) appears in \( C \). Then, in the usual way, iteration is a special case of recursion:

\[
\begin{align*}
\text{do } & \text{pred } \rightarrow \text{body } \od \ \\
& (\mu \text{xxx} \cdot (\text{body}; \text{xxx}) \text{ if } \text{pred } \text{else } \text{skip}) .
\end{align*}
\]

1.5.3 Example of syntax: the “Monty Hall” game

We illustrate the syntax of our language with the example program of Fig. 1.5.1. There are three curtains, labelled \( A, B \) and \( C \), and a prize is hidden nondeterministically behind one of them, say \( p_c \). A contestant hopes to win the prize by guessing where it is hidden: he chooses randomly to

\[\text{point to curtain } cc. \text{ The host then tries to get the contestant to change his mind, showing that the prize is not behind some other curtain } ac — \text{ which means that either the contestant has chosen it already or it is behind the other closed curtain. Should the contestant change his mind?}\]

1.5.4 Intuitive interpretation of \(\mu\)GCL expectations

In its full generality, an expectation is a function describing how much each program state is “worth.”

The special case of an embedded predicate \([\text{pred}]\) assigns to each state a value of zero or one; states satisfying \(\text{pred}\) are worth one, and states not satisfying \(\text{pred}\) are worth zero. The more general expectations arise when one estimates, in the initial state of a probabilistic program, what the worth of its final state will be. That estimate, the “expected worth” of the final state, is obtained by summing over all final states

the worth of the final state multiplied by the probability the program “will go there” from the initial state.

Naturally the “will go there” probabilities depend on “from where,” and so that expected worth is a function of the initial state.

When the worth of final states is given by \([\text{post}]\), the expected worth of the initial state turns out to be just the probability that the program will reach \(\text{post} \). That is because
expected worth of initial state

\[ p \equiv (\text{probability } \text{prog} \text{ reaches } \text{post}) \ast (\text{worth of states satisfying } \text{post}) \]

\[ + (\text{probability } \text{prog} \text{ does not reach } \text{post}) \ast (\text{worth of states not satisfying } \text{post}) \]

\[ \equiv (\text{probability } \text{prog} \text{ reaches } \text{post}) \ast 1 \]

\[ + (\text{probability } \text{prog} \text{ does not reach } \text{post}) \ast 0 \]

\[ \equiv \text{probability } \text{prog} \text{ reaches } \text{post} ; \]

note we have relied on the fact that all states satisfying \text{post} have worth one.

More general analyses of programs \text{prog} in practice lead to conclusions of the form

\[ p \equiv \wp . \text{prog}. [\text{post}] \]

for some \( p \) and \text{post} which, given the above, we can interpret in two equivalent ways:

- the expected worth \( [\text{post}] \) of the final state is at least the value of \( p \) in the initial state; or
- the probability that \text{prog} will establish \text{post} is at least \( p \).

Each interpretation is useful, and in the following example we can see them acting together: we ask for the probability that two fair coins when flipped will show the same face, and calculate

\[ \wp . \left( x \equiv H \uplus x \equiv T ; y \equiv H \uplus y \equiv T \right), \quad [x = y] \]

\[ \equiv \wp . (x \equiv H \uplus x \equiv T) \cdot ([x = H] / 2 + [x = T] / 2) \]

\[ \equiv (1/2) \cdot ([x = H] / 2 + [x = T] / 2) \]

We can then use the second interpretation above to conclude that the faces are the same with probability \( 1/2 \).

But part of the above calculation involves the more general expression

\[ \wp . (x \equiv H \uplus x \equiv T) \cdot ([x = H] / 2 + [x = T] / 2) \]

and what does that mean on its own? It must be given the first interpretation, that is as an expected worth, since “will establish \( [x = H] / 2 + [x = T] / 2 \)” makes no sense. Thus it means

the expected value of the expression \( [x = H] / 2 + [x = T] / 2 \)

after executing the program \( x \equiv H \uplus x \equiv T \),

which the calculation goes on to show is in fact \( 1/2 \). But for our overall conclusions we do not need to think about the intermediate expressions — they are only the “glue” that holds the overall reasoning together.

### 1.5.5 Semantics

The probabilistic semantics is derived from generalising the standard semantics in the way suggested in Sec. 1.3. Let the state space be \( S \).

**Definition 1.5.2** EXPPECTATION SPACE The space of expectations over \( S \) is defined

\[ \mathcal{E}S := (S \rightarrow \mathbb{R}_{\geq}, \supseteq) \]

where the entailment relation \( \supseteq \), as we have seen, is inherited pointwise from the normal \( \leq \) ordering in \( \mathbb{R}_{\geq} \). The expectation-transformer model for programs is

\[ \mathcal{T}S := (\mathcal{E}S \rightarrow \mathcal{E}S, \subseteq) \]

where we write the functional arrow backward just to emphasise that such transformers map final post-expectations to initial pre-expectations, and where the refinement order \( \subseteq \) is derived pointwise from entailment \( \Rightarrow \) on \( \mathcal{E}S \).

\[ \square \]

---

34 We must say “at least” in general, because possible demonic choice in \text{prog} means that the pre-expectation is only a lower bound for the actual expected value the program could deliver; and some analyses give only the weaker \( p \nRightarrow \wp . \text{prog}. [\text{post}] \) in any case. See also Footnote 14 on p. 89.

35 See Fig. 1.5.3 for this definition.

36 (Recall Footnote 34.) If we do know, by other means say, that the program is deterministic (though still probabilistic), then we can say the pre-expectation is exact.

37 See p. 271 for an example of this same analogy, but in the context of temporal logic.
Although both \( E \) and \( T \) are lattices, neither is a complete partial order, because \( \mathbb{R}_\geq \) itself is not. (It lacks an adjoined \( \infty \) element.) In addition, when \( S \) is infinite (see e.g. Sec. 8.2 of Part II) we must impose the condition on elements of \( E \) that each of them be bounded above by some non-negative real.\(^{38}\)

In Fig. 1.5.3 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.

Note that our semantics states how \( wp \cdot \text{prog} \) in each case transforms an expression in the program variables; that is, we give a procedure for calculating the greatest pre-expectation by purely syntactic manipulation. An alternative view is to see the post-expectations as mathematical functions of type \( E \), and the expressions \( wp \cdot \text{prog} \) are then of type \( TS \).

The expression-based view is more convenient in an introduction, and for the treatment of specific programs; the function-based view is more convenient (and, for recursion, necessary) for general properties of expectation transformers. In this chapter and the rest of Part I we retain the expression-based view as far as possible; but in Part II we use the more mathematical notation. (See for example Sec. 5.3.)

The worst program \texttt{abort} cannot be guaranteed to terminate in any proper state and therefore maps every post-expectation to 0. The immediately terminating program \texttt{skip} does not change anything, therefore the expected value of post-expectation \( \text{postE} \) after execution of \texttt{skip} is just its actual value before. The pre-expectation of the assignment \( x := \text{expr} \) is expressed by \( \text{expl} \cdot \text{postE} \) substituted for \( x \). Sequential composition is functional composition. The semantics of demonic choice \( \sqcap \) reflects the dual metaphors for it: abstraction, we must take the minimum because we are giving a guarantee over all possible implementations; as a demon’s behaviour, we assume he acts to make our expected winnings as small as possible.

The pre-expectation of probabilistic choice is the weighted average of the pre-expectations of its branches. Since any such average is no less than the minimum it follows immediately that probabilistic choice refines demonic choice to the weakest common upper bound.

\(^{38}\)A partial order differs from the familiar “total” orders like “\( \leq \)” in that two elements can be “incomparable”; the most common example is subset \( \subseteq \) between sets, which satisfies reflexivity (a set is a subset of itself), anti-symmetry (two sets cannot be subsets of each other without being the same set) and transitivity (one set within a second within a third is a subset of the third directly as well). But it is not true that for any two sets one is necessarily a subset of the other.

A lattice is a non-empty partially ordered set where for all \( x, y \) in the set there is a GREATEST LOWER BOUND \( x \sqcap y \) and a LEAST UPPER BOUND \( x \sqcup y \). This holds e.g. for the lattice of sets, as above; but the collection of non-empty sets is not a lattice, because \( x \sqcap y \) (which is how \( x \sqcap y \) is written for sets) is not necessarily non-empty even if \( x \) and \( y \) are.

A partial order \( \sqsubseteq \) is CHAIN- or DIRECTED COMPLETE — then called a cpo — when it contains all chains of sets or directed sets respectively, where a chain is a set totally ordered by \( \sqsubseteq \) and a set is DIRECTED if for any \( x, y \) in the set there is a \( z \) also in the set such that \( x, y \sqsubseteq z \). (Since a chain is directed, directed completeness implies chain completeness; in fact with the Axiom of Choice, chain- and directed completeness are equivalent.)

All of these details can be found in standard texts [DP90].

\(^{39}\)There is a difference between requiring that there be an upper bound for all expectations (we do not) and requiring that each expectation separately have an upper bound (we do).

In the first case, we would be saying that there is some \( M \) such that every expectation \( \alpha \) in \( E \) satisfied \( \alpha \sqsubseteq M \). That would be convenient because it would make both \( E \) and \( TS \) complete partial orders, trivially; and that would e.g. allow us to use a standard treatment of fixed points.

But we adopt the second case where, for each expectation \( \alpha \) separately, there is some \( M_\alpha \) such that \( \alpha \sqsubseteq M_\alpha \); and, as \( \alpha \) varies, these \( M_\alpha \)’s can increase without bound. That is why \( E \) is not complete and is, therefore, why we will need a slightly special argument when dealing with fixed points.

--

Figure 1.5.3. Probabilistic wp-semantics of \( pGCL \)

\begin{verbatim}
wp.abort.postE := 0
wp.skip.postE := postE
wp.(x := expr).postE := postE (x := expr)
wp.(prog; prog').postE := wp.prog(wp.prog'.postE)
wp.(prog \sqcap prog').postE := wp.prog.postE \min wp.prog'.postE
wp.(prog \oplus prog').postE := p * wp.prog.postE + \neg p * wp.prog'.postE
\end{verbatim}

Recall that \( \neg p \) is the complement of \( p \).

The expression on the right gives the greatest pre-\text{expectation} of \( \text{postE} \) with respect to each \( pGCL \) construct, where \( \text{postE} \) is an expression of type \( E \) over the variables in state space \( S \). (For historical reasons we continue to write \( wp \) instead of \( up \).)

In the case of recursion, however, we cannot give a purely syntactic definition. Instead we say that

\[ (\mu \text{xxx} . C) := \text{least fixed-point of the function } cntx : TS \rightarrow TS \]

defined so that \( cntx.(\text{wp.xxx}) = \text{wp.C} \).\(^{40}\)

\(^{40}\)Because \( TS \) is not complete, to ensure existence of the fixed point we insist that the transformer-to-transformer function \( cntx \) be “feasibility-preserving,” i.e. that if applied to a feasible transformer it returns a feasible transformer again. “Feasibility” of transformers is one of the “healthiness conditions” we will encounter in Sec. 1.6. For convenience, we usually assume that \( cntx \) is continuous as well.

See Lem. 5.6.8 on p. 148.
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.\footnote{Some writers call that \textsc{pre-determinism}: “deterministic if terminating.”}

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

\subsection*{Example of semantics: Monty Hall again}

We illustrate the semantics by returning to the program of Fig. 1.5.1. Consider the post-expectation $[pc \leftarrow cc]$, which takes value one just 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

\[ wp.\ (cc \notin \{cc, ac\}) \text{ if clever else skip} \quad [pc \leftarrow cc] \equiv [\text{clever}] \ast [\{ac, cc, pc\} = \{\text{A, B, C}\}] + [\neg \text{clever}] \ast [pc = cc] , \]

because (in case clever) the nondeterministic choice is guaranteed to pick pc only when it cannot avoid doing so.\footnote{In Fig. 1.5.1 we said that this fourth statement “executes deterministically”; yet here we have called it nondeterministic. On its own, it is nondeterministic; but in the context of the program its nondeterminism is limited to making a choice from a singleton set, as our subsequent calculations will show.}

Standard reasoning suffices for our next step also:

\begin{align*}
wp.\ (ac \notin \{pc, cc\}) & , (\{\text{clever}\} \ast [\{ac, cc, pc\} = \{\text{A, B, C}\}] + [\neg \text{clever}] \ast [pc = cc]) \\
& \equiv [\text{clever}] \ast [pc \neq cc] + [\neg \text{clever}] \ast [pc = cc] .
\end{align*}

For the clever case note that $\{ac, cc, pc\} = \{\text{A, B, C}\}$ holds (in the post-expectation) iff all three elements differ, and that the statement itself establishes only two of the required three inequalities — that $ac \neq pc$ and $ac \neq cc$. The weakest precondition supplies the third.

For the $\neg$clever case note that neither $pc$ nor $cc$ is assigned to by $ac \notin \{pc, cc\}$, so that $pc = cc$ holds afterwards iff it held before.

The next statement is probabilistic, and so produces a probabilistic pre-expectation involving the factors $1/3$ given explicitly in the program; we have

\begin{align*}
wp.\ (cc = (A \@ B \@ C \@ \frac{1}{3})). \\
([\text{clever}] \ast [pc \neq cc] + [\neg \text{clever}] \ast [pc = cc]) & \equiv [\text{clever}] \ast [pc \neq a] + [pc \neq B] + [pc \neq C]) \\
& + [\neg \text{clever}] \ast [pc = a] + [pc = B] + [pc = C]) .
\end{align*}

That “healthiness condition” [Dij76] is used to prove many general properties of programs.

In pGCL the healthiness condition becomes “sublinearity,” a generalisation of conjunctivity: \footnote{Having discovered a probabilistic analogue of conjunctivity, we naturally ask for an analogue of disjunctivity. That turns out to be “super-linearity” — which when combined with sublinearity gives (just) linearity, and is characteristic of deterministic probabilistic programs, just as disjunctivity (with conjunctivity) characterises deterministic standard programs. See Sec. 8.3.}

\subsection{Healthiness and algebra for pGCL}

Recall that all standard GCL constructs satisfy the important property of conjunctivity\footnote{Some writers call that \textsc{conjunctivity}: “deterministic if terminating.”} — that is, for any GCL command prog and post-conditions $post, post'$ we have

\[ wp.\ prog.(post \land post') = wp.\ prog.post \land wp.\ prog.post' . \]

That “healthiness condition” [Dij76] is used to prove many general properties of programs.

In pGCL the healthiness condition becomes “sublinearity,” a generalisation of conjunctivity:

\begin{definition}[Sublinearity of pGCL]
Let $c_0, c_1, c_2$ be non-negative reals, and $postE_1, postE_2$ expectations; then all pGCL constructs prog satisfy

\[ wp.\ prog.(c_1 \ast postE_1 + c_2 \ast postE_2 \ominus c_0) \iff c_1 \ast wp.\ prog.postE_1 + c_2 \ast wp.\ prog.postE_2 \ominus c_0 , \]

which property of prog is called sublinearity. Truncated subtraction $\ominus$ is defined

\[ x \ominus y := (x - y) \max 0 , \]

\end{definition}

\[ \equiv (\{\text{clever}\} / 3) \ast 2 + (\neg \text{clever}) / 3 \ast 1 \quad \text{type of } pc \in \{\text{A, B, C}\} \]

\[ \equiv 2 \{\text{clever}\} / 3 + (\neg \text{clever}) / 3 . \]

Then for the first statement $pc \in \{\text{A, B, C}\}$ we only note that $pc$ does not appear in the final condition above, thus leaving it unchanged under wp; with simplification it becomes

\[ (1 + [\text{clever}]) / 3 , \]

which is thus the pre-expectation for the whole program.

Since the post-expectation $[pc = cc]$ is standard (it is the characteristic function of the set of states in which $pc = cc$), we are able to interpret the pre-expectation directly as the probability that $pc = cc$ will be satisfied on termination; we conclude that the contestant has $2/3$ probability of finding the prize if he is clever, and only $1/3$ if he is not.

\section{Introduction to pGCL}

\[ wp.\ prog.\ (post \land post') = wp.\ prog.post \land wp.\ prog.post' . \]

Footnote 50 on p. 33 explains how typing might be propagated this way.

\[ wp.\ prog.\ (post \land post') = wp.\ prog.post \land wp.\ prog.post' . \]

Footnote 50 on p. 33 explains how typing might be propagated this way.
the maximum of the normal difference and zero. It has syntactic precedence
lower than +. □

Although it has a strange appearance, from sublinearity we can extract
a number of very useful consequences, as we now show. We begin with
monotonicity, feasibility and scaling.\footnote{These properties are collected together in Sec. 5.6, and restated in Part II as
Defs. 5.6.3–5.6.5.}

\section*{Definition 1.6.2 Healthiness conditions}
\begin{itemize}
  \item \textbf{monotonicity}: increasing a post-expectation can only increase the
pre-expectation. Suppose \( \text{post}E \Rightarrow \text{post}E' \) for two expectations
\( \text{post}E, \text{post}E' \); then
\[
\text{wp} \text{. prog} \cdot \text{post}E' \\
\equiv \text{wp} \text{. prog} \cdot (\text{post}E' + (\text{post}E' - \text{post}E))
\]
\end{itemize}
\begin{itemize}
  \item \textbf{feasibility}: pre-expectations cannot be “too large.” First note that
\[
\text{wp} \text{. prog} \cdot 0 \\
\equiv \text{wp} \text{. prog} \cdot (2 \cdot 0)
\]
\end{itemize}
\begin{itemize}
  \item \textbf{scaling}: multiplication by a non-negative constant distributes through
commands. Note first that \( \text{wp} \text{. prog} \cdot (c \cdot \text{post}E) \equiv c \cdot \text{wp} \text{. prog} \cdot \text{post}E \)
directly from sublinearity.
\end{itemize}

\subsection*{1. Introduction to pGCL}

For \( \Rightarrow \) we have two cases: when \( c \) is zero, trivially from feasibility
\[
\text{wp} \text{. prog} \cdot (0 \cdot \text{post}E) \equiv \text{wp} \text{. prog} \cdot 0 \equiv 0 \equiv 0 \cdot \text{wp} \text{. prog} \cdot \text{post}E ;
\]
and for the other case \( c \neq 0 \) we reason
\[
\text{wp} \text{. prog} \cdot (c \cdot \text{post}E) \\
\equiv c(1/c) \cdot \text{wp} \text{. prog} \cdot (c \cdot \text{post}E) \quad c \neq 0 \]
\[
\equiv c \cdot \text{wp} \text{. prog} \cdot ((1/c) \cdot \text{post}E) \quad \text{sublinearity using } 1/c
\]
\[
\equiv c \cdot \text{wp} \text{. prog} \cdot \text{post}E ,
\]
thus establishing \( \text{wp} \text{. prog} \cdot (c \cdot \text{post}E) \equiv c \cdot \text{wp} \text{. prog} \cdot \text{post}E \)
generally. (See p. 53 for an example of scaling’s use.) □

The remaining property we examine is so-called “probabilistic conjunc-
tivity.” Since standard conjunction “\&” is not defined over numbers, we
have many choices for a probabilistic analogue “\&” of it, requiring only
\[
0 \& 0 = 0
\]
\[
0 \& 1 = 0
\]
\[
1 \& 0 = 0
\]
\[
1 \& 1 = 1
\]
for consistency with embedded Booleans.

Obvious possibilities for \& are multiplication \( \ast \) and minimum \( \ominus \), and
each of those has its uses; but neither satisfies anything like a generalisation
of conjunctivity. Return for example to the program of Fig. 1.5.1, and
consider its second statement
\[
\text{wp} \text{. prog} \cdot \text{post}E \ominus \text{max} \text{. post}E \\
\]
\[
\text{wp} \text{. prog} \cdot \text{post}E \ominus \text{max} \text{. post}E \\
\]
\[
\equiv 0 \Rightarrow \text{wp} \text{. prog} \cdot (\text{post}E' - \text{post}E)
\]
\end{itemize}
\begin{itemize}
  \item \textbf{feasibility}: pre-expectations cannot be “too large.” First note that
\[
\text{wp} \text{. prog} \cdot 0 \\
\equiv \text{wp} \text{. prog} \cdot (2 \cdot 0)
\]
\end{itemize}
\begin{itemize}
  \item \textbf{scaling}: multiplication by a non-negative constant distributes through
commands. Note first that \( \text{wp} \text{. prog} \cdot (c \cdot \text{post}E) \equiv c \cdot \text{wp} \text{. prog} \cdot \text{post}E \)
directly from sublinearity.
\end{itemize}

\footnote{Note how the general (1.21) implies the strictness condition \( \text{wp} \text{. prog} \cdot 0 \equiv 0 \), a direct numeric embedding of Dijkstra’s Law of the Excluded Miracle.}
whose right-hand side is inspired by sublinearity when \(c_0, c_1, c_2 : = 1, 1, 1\). The operator is commutative; and if we restrict expectations to \([0, 1]\) it is associative as well. Note however that it is not idempotent.\(^{48}\)

We now state a (sub-)distribution property for \&\(^\alpha\), a direct consequence of sublinearity.

**sub-conjunctivity:** the operator \& sub-distributes through expectation transformers. From sublinearity with \(c_0, c_1, c_2 : = 1, 1, 1\) we have

\[
wp.\prog.(postE \& postE') \subseteq wp.\prog.postE \& wp.\prog.postE'
\]

for all \(\prog\).

(Unfortunately there does not seem to be a full (\(\equiv\)) conjunctivity property for expectation transformers.)

Beyond sub-conjunctivity, we say that \& generalises conjunction for several other reasons as well. The first is of course that it satisfies the standard properties (1.22).

The second reason is that sub-conjunctivity (a consequence of sublinearity) implies “full” conjunctivity for standard programs. Standard programs, containing no probabilistic choices, take standard \([post]\)-style post-expectations to standard pre-expectations; they are the embedding of \(GCL\) in \(pGCL\), and for standard \(\prog\) we now show that

\[
wp.\prog.(\{post\} \& \{post'\}) \equiv wp.\prog.[post] \& wp.\prog.[post'],
\]

(1.24)

First note that “\(\subseteq\)” comes directly from sub-conjunctivity above, taking \(postE, postE'\) to be \([post], [post']\).

For “\(\geq\)” we appeal to monotonicity, because \([post] \& [post'] \Rightarrow [post]\) whence \(wp.\prog.(\{post\} \& \{post'\}) \Rightarrow wp.\prog.[post]\), and similarly for \(post'\). Putting those together gives

\[
wp.\prog.(\{post\} \& \{post'\}) \Rightarrow wp.\prog.[post] \min wp.\prog.[post'],
\]

by elementary arithmetic properties of \(\Rightarrow\). But on standard expectations — which \(wp.\prog.[post]\) and \(wp.\prog.[post']\) are, because \(\prog\) is standard — the operators \(\min\) and \& agree.

A last attribute linking \& to \& comes straight from elementary probability theory. Let \(X\) and \(Y\) be two events, not necessarily independent: then

if the probability of \(X\) is at least \(p\), and the probability of \(Y\) is at least \(q\), the most that can be said in general about the joint event \(X \cap Y\) is that it has probability at least \(p \& q\).

\(^{48}\)A binary operator \(\odot\) is idempotent just when \(x \odot x = x\) for all \(x\).

---

1.7 Healthiness example: modular reasoning

As an example of the use of healthiness conditions, we formulate and prove a simple but very powerful property of \(pGCL\) programs, important for “modular” reasoning about them.

By modular reasoning in this case we mean determining, first, that a program \(\prog\) of interest has some standard property; then for subsequent (possibly probabilistic) reasoning we assume that property. This makes

\(^{49}\)The first step is the modularity law for probabilities.
We use the healthiness conditions of the previous section, and are bounded above by some nonzero $M$. Given that the current state satisfies $\mathit{pre}$, we then have

\[ wp.\mathit{prog}.([\mathit{post}] \ast \mathit{post}E) = M \ast wp.\mathit{prog}.([\mathit{post}] \ast \mathit{post}E/M) \]

scaling

\[ \geq M \ast wp.\mathit{prog}.((\mathit{post} \& \mathit{post}E)/M) \]

\[ \geq M \ast wp.\mathit{prog}.(\mathit{post}E/M) \]

Assumption (1.26)

\[ = M \ast wp.\mathit{prog}.(\mathit{post}E/M) \]

arithmetic

\[ = wp.\mathit{prog}.\mathit{post}E \]

scaling

The opposite inequality is immediate (in all states) from the monotonicity healthiness property, since $\mathit{post}$ is standard; $\mathit{post}E/M \Rightarrow 1$. Thus, still assuming $\mathit{pre}$ holds in the current state, we conclude with

\[ wp.\mathit{prog}.\mathit{post}E = wp.\mathit{prog}.([\mathit{post}] \ast \mathit{post}E) \]

\[ = wp.\mathit{prog}.([\mathit{post}] \ast \mathit{post}E') \quad \text{above} \]

\[ = wp.\mathit{prog}.([\mathit{post}] \ast \mathit{post}E') \quad \text{as above, but for } \mathit{post}E' \]

\[ \square \]

This kind of reasoning is nothing new for standard programs, and indeed is usually taken for granted (although its formal justification appeals to conjunctivity). It is important that it is available in $\mathit{pGCL}$ as well.\textsuperscript{52}

1.8 Interaction between probabilistic- and demonic choice

We conclude with some illustrations of the interaction of demonic and probabilistic choice. Consider two variables $x, y$, one chosen demonically and the other probabilistically. Suppose first that $x$ is chosen demonically and $y$ probabilistically, and take post-expectation $[x = y]$. Then

\[ \begin{align*}
\text{wp.\mathit{prog}} & = \text{wp.\mathit{prog}}.([\mathit{post}] \ast \mathit{post}E) \\
& = \text{wp.\mathit{prog}}.([\mathit{post}] \ast \mathit{post}E') \\
& = \text{wp.\mathit{prog}}.\mathit{post}E' \quad \text{as above, but for } \mathit{post}E' 
\end{align*} \]

\textsuperscript{52}Lemma 1.7.1 holds even when $\mathit{post}E, \mathit{post}E'$ are unbounded, provided of course that $\text{wp.\mathit{prog}}$ is defined for them; the proof of that can be given by direct reference to the definition of $\text{wp}$ over the model, as set out in Chap. 5.

We will need that extension for our occasional excursions beyond the “safe” bounded world we have formally dealt with in the logic (e.g. Sections 2.11 and 3.3).
\begin{align*}
\text{wp.(} & (x = 1 \land x = 2); (y = 1 \uplus y = 2). [x = y] \\
\equiv & \text{wp.(} x = 1 \land x = 2).([x = 1]/2 + [x = 2]/2) \\
\equiv & ([1 = 1]/2 + [1 = 2]/2) \min ([2 = 1]/2 + [2 = 2]/2) \\
\equiv & (1/2 + 0/2) \min (0/2 + 1/2) \\
\equiv & 1/2
\end{align*}

from which we see that program establishes \( x = y \) with probability at least \( 1/2 \): no matter which value is assigned to \( x \), with probability \( 1/2 \) the second command will assign the same to \( y \).

Now suppose instead that it is the second choice that is demonic. Then we have

\begin{align*}
\text{wp.(} & (x = 1 \uplus x = 2); (y = 1 \land y = 2). [x = y] \\
\equiv & \text{wp.(} x = 1 \uplus x = 2).([x = 1] \min [x = 2]) \\
\equiv & ([1 = 1] \min [1 = 2])/2 + ([2 = 1] \min [2 = 2])/2 \\
\equiv & (1 \min 0)/2 + (0 \min 1)/2 \\
\equiv & 0
\end{align*}

reflecting that no matter what value is assigned probabilistically to \( x \), the demon could choose subsequently to assign a different value to \( y \).

Thus it is clear that the execution order of occurrence of the two choices plays a critical role in their interaction, and in particular that the demon in the first case cannot make the assignment “clairvoyantly” to \( x \) in order to avoid the value that later will be assigned to \( y \).

1.9 Summary

Being able to reason formally about probabilistic programs does not of course remove per se the complexity of the mathematics on which they rely: we do not now expect to find astonishingly simple correctness proofs for all the large collection of randomised algorithms that have been developed over the decades [MR95]. However it should be possible in principle to locate and determine reliably what are the probabilistic/mathematical facts the construction of a randomised algorithm needs to exploit... which is of course just what standard predicate transformers do for conventional algorithms.

In the remainder of Part I we concentrate on proof rules that can be derived for pGCL — principally for loops — and on examples.

The theory of expectation transformers with nondeterminism is given in Part II, where in particular the role of sublinearity is identified and proved: it characterises a subspace of the predicate transformers that has an equivalent operational semantics of relations between initial and final probabilistic distributions over the state space — a formalisation of the gambling game of Sec. 1.3. 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.

Chapter notes

In the mid 1970’s, Rabin demonstrated how randomisation could be used to solve a variety of programming problems [Rab76]; since then, the range of applications has increased considerably [MR95], and indeed we analyse several of them as case studies in later chapters. In the meantime — fuelled by randomisation’s impressive applicability — the search for an effective logic of probabilistic programs became an important research topic around the beginning of the 1980’s, and remained so until the mid 1990’s. Ironically, the major technical difficulty was due, in the main, to one of standard programming’s major successes: demonic nondeterminism, the basis for abstraction. It was a challenging problem to decide what to do about it, and how it should interact with the new probabilistic nondeterminism.

The first probabilistic logics did not treat demonic nondeterminism at all — Feldman and Harel [FH84] for instance proved soundness and completeness for a probabilistic PDL, which was (in our terms) purely deterministic. The logical language allowed statements about programs to be made at the level of probability distributions and, as we discuss in Sec. A.2, that proves to be an impediment to the natural introduction of a demon. A Hoare-style logic based on similar principles has also been explored by den Hartog and de Vink [dBdV02].

The crucial step of a quantitative logic of expectations was taken by Kozen [Koz85]. Subsequently Jones [Jon90], with Plotkin and using the evaluations from earlier work of Saheb-Djahromi [SD80] that were based directly on topologies rather than on \( \sigma \)- or Borel algebras, worked on more general probabilistic powerdomains; as an example of her technique she specialised it to the Kozen-style logic for deterministic programs, resulting in the sub-probability measures that provide a neat way to quantify nontermination.\footnote{The notion of sub-probability measures to characterise termination was present much earlier, for example in the work of Feldman and Harel [FH84].}

In 1997 He et al. [HSM97] finally proposed the operational model containing all the ingredients for a full treatment of abstraction and program refinement in the context of probability — and that model paved the way for the “demonic/probabilistic” program logic based on expectation transformers. Subsequently Ying [Yin03] has worked towards a probabilistic refinement calculus in the style of Back [BoW98].