PROBABILISTIC COMPUTATIONS 3

In this chapter, we introduce syntax and semantics of the *probabilistic Guarded Command Language* (pGCL). This model programming language is a syntactic superset of Dijkstra's GCL and enriches it by probabilistic constructs for modeling probabilistic computations.

Semantics of structured probabilistic programs have been first studied by Kozen in the late 70's and early 80's [Koz79; Koz81; Koz83; Koz85]. He did not consider nondeterministic choice as Dijkstra did in his GCL but instead *replaced* it with probabilistic choice. Later, McIver & Morgan (re)introduced nondeterministic choices [MMS96; MM05]. We follow their approach here and present a variant of their pGCL, i.e. a programming language that features both kinds of uncertainties: randomness and nondeterminism.[1] We begin with a note on the difference between these two sources of uncertainty.

3.1 RANDOMNESS VERSUS NONDETERMINISM

Consider a fair (random) coin. If we flip the coin, we do not know upfront what the outcome will be: heads or tails. Since the coin is fair, there is not even a bias towards one of the two outcomes. We could thus think that all we know is that the outcome will be either heads or tails. However, that is actually not quite *all* we know. We do have an additional grain of information in knowing that both outcomes are equally likely to occur, each with probability 1/2.

Suppose a gambler approaches us and proposes the following bet:

> „I bet you a troy ounce of gold that if you flip this fair coin 10 times in a row, it will land on heads each of those times."

Should we bet on it? By knowing that the coin is fair, we actually have a tremendous amount of information about this bet, namely that the chance of winning is

$$1 - \frac{1}{2^{10}} = 99.90234375\% ,$$

and we would thus be well–advised to take the chances, unless we are somewhat overly risk–averse.

We now consider a *nondeterministic coin* that is modeled by the following mechanism: A fair (random) coin is flipped by us and then a blindfolded

1 We will use the terms *random* and *probabilistic* synonymously.

oracle is asked to announce the outcome of the coin flip. The outcome of this nondeterministic coin flip is then either *correct* (in case the oracle's announcement is correct) or *incorrect* (in case the announcement is incorrect) and this is in fact really *all we know* about the outcome of the entire experiment. Does the oracle just randomly (i.e. probabilistically) guess an answer? If yes, is the oracles guess biased in any way? Would that even matter? Or does the oracle even have the superhuman ability to correctly announce the outcome of the coin toss every single time? (Or with high probability?) Is it even possible that we are being tricked in some way and the oracle is somehow reliably informed about the outcome of the coin toss? To make a long story short: There is a myriad of possibilities for the oracle to come to its conclusion and we have no clue about the underlying mechanism.

Suppose that the gambler approaches us and proposes the following bet:

> „I bet you a troy ounce of gold that the oracle will make a correct announcement 10 times in a row.“

Should we bet on it? By knowing basically nothing about the outcomes of the nondeterministic coin, we can only *hope* that the oracle fails to make 10 correct announcements in a row, *but we cannot reasonably associate a probability or even any quantity to this outcome*. We can only state that this unfavorable outcome is *possible*. If we want to play it reasonably safe, we should perhaps not take those chances.

To summarize: Randomness is a kind of uncertainty where we can meaningfully assign a quantity — namely a *probability* — to each of the possible outcomes, whereas nondeterminism is a kind of uncertainty where we can only say for each outcome whether there is a *possibility* for it to occur or not. Random behavior thus allows for quantitative reasoning, whereas nondeterministic behavior only allows for qualitative reasoning. Mixing both behaviors in a single computational process is widely acknowledged to be problematic and is an active area of research (see e.g. [VW06; Var03; TKP09; Mis00; KP17]). We will later present the calculus of McIver & Morgan that attempts to do quantitative reasoning about programs in a language which features both probabilistic and nondeterministic uncertainty by *resolving* nondeterminism in the least favorable way.

3.2 pGCL — A PROBABILISTIC GCL

Access to some source of randomness is a key ingredient for a programming language that models probabilistic computations. There are different concepts for introducing randomness into the computation, e.g. random inputs, internal coin flips, sampling values from predefined probability distributions, etc. We (and many others) choose for our development two sources of randomness:

- ✧ coin flips, biased according to some rational probability, and
- ✧ sampling of values from *discrete* probability distributions.

For incorporating the aforementioned two sources of randomness, we endow Dijkstra's GCL with a *probabilistic choice* and a *random assignment* construct, thereby obtaining a *probabilistic* GCL. Formally, this probabilistic programming language is defined as follows:

DEFINITION 3.1 (A Probabilistic Guarded Command Language):

A. *A function* $\pi\colon \mathsf{Vals} \to [0,1]$ *is called a probability distribution over values if* $\sum_{v \in \mathsf{Vals}} \pi(v) = 1$. *We denote the set of all probability distributions over values by* $\mathcal{D}(\mathsf{Vals})$. *A distribution expression is a function*

$$\mu\colon \Sigma \to \mathcal{D}(\mathsf{Vals})$$

that maps every program state to a probability distribution over values. Recall from Definition 2.1 *that* Vals *is always required to be countable and therefore every probability distribution over values is a* discrete *probability distribution.*

B. *A probability expression is a function*

$$p\colon \Sigma \to [0,1] \cap \mathbb{Q}$$

that maps every program state to a rational *probability.*

C. *The set of programs in probabilistic guarded command language, denoted* pGCL, *is given by the grammar*

$$\begin{array}{rlr}
C \;\longrightarrow & \texttt{skip} & \text{(effectless program)}\\
& \mid\; \texttt{diverge} & \text{(freeze)}\\
& \mid\; x := E & \text{(assignment)}\\
& \mid\; x :\approx \mu & \text{(random assignment)}\\
& \mid\; C \fatsemi C & \text{(sequential composition)}\\
& \mid\; \texttt{if}\,(\varphi)\,\{C\}\,\texttt{else}\,\{C\} & \text{(conditional choice)}\\
& \mid\; \{C\} \,\square\, \{C\} & \text{(nondeterministic choice)}\\
& \mid\; \{C\}\,[p]\,\{C\} & \text{(probabilistic choice)}\\
& \mid\; \texttt{while}(\varphi)\{C\}\,, & \text{(while loop)}
\end{array}$$

where $x \in \mathsf{Vars}$ *is a program variable,* E *is an arithmetic expression over program variables,* μ *is a distribution expression,* φ *is a boolean expression over program variables guarding a choice or a loop, and* p *is a probability expression.*

D. *A* pGCL *program containing no* `diverge` *statements and no while loops is called loop–free.*

E. *A* pGCL *program that contains neither random assignments nor probabilistic choices is called nonprobabilistic. A* pGCL *program that contains no nondeterministic choices is called tame. A* pGCL *program that contains neither constructs of randomness nor constructs of nondeterminism is called deterministic.*

Every language construct from GCL is also contained in pGCL. Their computational effects are exactly the same in pGCL as they are in GCL (see Section 2.1). We thus only go over the probabilistic constructs introduced in pGCL here, starting with the conceptually simpler one.

Probabilistic choices. The probabilistic choice construct

$$\{C_1\}\,[p]\,\{C_2\}$$

behaves as follows: It evaluates probability expression p in the current program state σ to obtain a probability $p(\sigma)$. Then, it executes C_1 with probability $p(\sigma)$ and C_2 with probability $1 - p(\sigma)$.

EXAMPLE 3.2 (Probabilistic Choices):

A. The program

$$\{x := x + 1\}\,[2/3]\,\{z := 17\}$$

increments variable x by 1 with probability $2/3$ and it assigns the constant 17 to variable z with probability $1 - 2/3 = 1/3$.

B. If the current program state is σ, then the program

$$\{x := x + 1\}\,[1/|x|+1]\,\{x := x - 1\}$$

increments variable x by 1 with probability $1/|\sigma(x)|+1$ and decrements x by 1 with probability $1 - 1/|\sigma(x)|+1 = |\sigma(x)|/|\sigma(x)|+1$. Therefore, the further away from 0 the value of x is, the less likely it is that x is incremented.

Random Assignments. The random assignment construct

$$x :\approx \mu$$

behaves as follows: It evaluates distribution expression μ in tn the current program state σ to obtain a discrete probability distribution $\pi = \mu(\sigma)$. Then it samples a value from π, thus obtaining a sample value $v \in \mathsf{Vals}$ with probability $\pi(v)$. This value v is then assigned to variable x.

For denoting distribution expressions, we use bra–ket notation [Wikc]. For example, distribution expression

$$\tfrac{1}{2}\cdot|a\rangle + \tfrac{1}{3}\cdot|b\rangle + \tfrac{1}{6}\cdot|c\rangle$$

denotes a distribution where value a is sampled with probability $1/2$, b with probability $1/3$, and c with probability $1/6$.

EXAMPLE 3.3 (Random Assignments):

A. The program

$$x :\approx \tfrac{1}{2}\cdot|x+1\rangle + \tfrac{1}{2}\cdot|x-1\rangle$$

increments or decrements variable x by 1, each with probability $1/2$. It does so by first evaluating the distribution expression $1/2\cdot|x+1\rangle + 1/2\cdot|x-1\rangle$ in the current program state σ. This gives the distribution

$$\pi(v) \;=\; \begin{cases} \frac{1}{2}, & \text{if } v = \sigma(x)+1 \text{ or } v = \sigma(x)-1 \\ 0, & \text{otherwise.} \end{cases}$$

Then the program samples from π. By that, the values $\sigma(x)+1$ and $\sigma(x)-1$ are each sampled with probability $1/2$. The sampled value is then assigned to variable x.

B. Like the program from Example 3.2 B., the program

$$x :\approx \tfrac{1}{|x|+1}\cdot|x+1\rangle + \tfrac{|x|}{|x|+1}\cdot|x-1\rangle$$

too increments variable x by 1 with probability $1/|\sigma(x)|+1$ and decrements x by 1 with probability $|\sigma(x)|/|\sigma(x)|+1$. It does so by first evaluating the distribution expression $1/|x|+1\cdot|x+1\rangle + |x|/|x|+1\cdot|x-1\rangle$ in the current program state σ. This gives the probability distribution

$$\pi(v) \;=\; \begin{cases} \frac{1}{|\sigma(x)|+1}, & \text{if } v = \sigma(x)+1 \\ \frac{|\sigma(x)|}{|\sigma(x)|+1}, & \text{if } v = \sigma(x)-1 \\ 0, & \text{otherwise.} \end{cases}$$

Then the program samples from π. By that, the value $\sigma(x)+1$ is sampled with probability $1/|\sigma(x)|+1$ and the value $\sigma(x)-1$ is sampled with probability $|\sigma(x)|/|\sigma(x)|+1$. The sampled value is then assigned to variable x.

c. The program

$$k :\approx \mathsf{Unif}[1 \ldots 10]$$

assigns to program variable k one of the integers between 1 and 10, each with probability $1/10$.

d. If the current program state is σ, the program

$$k :\approx \mathsf{Unif}[1 \ldots n]$$

assigns to variable k an integer value between 1 and $\sigma(n)$, where n is a program variable,[2] each with probability $1/\sigma(n)$.

We have just provided a more or less formal intuition on what the two probabilistic choice constructs do computationally. In the following sections, we will provide several precise semantics to probabilistic programs.

3.3 SEMANTICS OF pGCL

Semantics of programming languages are precise mathematical descriptions of a program's computational effects. For deterministic (nondeterministic) programs, it often provides a mapping from initial states to (sets of) final states. It thus tells us what the (possible) outcome(s) of excuting a program on a given initial state is (are). Such mappings are qualitative in that a given initial state is either mapped to a given final state or not. In contrast to that, probabilistic programs clearly require *quantitative* information in order to make for a meaningful semantics.

In this section, we describe two different operational semantics of pGCL. They are operational in the sense that they describe a step–by–step, i.e. instruction–by–instruction, execution of a program and the according evolution of program states over the course of the execution. Such semantics are called *small–step semantics* or *structural operational semantics* [Plo04]. We will also show how the two operational semantics are related. Moreover, we will learn how we can think of *the outcome* of a probabilistic computation as a probability distribution over final states. This can be thought of as a *big–step semantics*, that maps input states directly to an outcome.

3.3.1 *Computation Tree Semantics*

Computation trees naturally occur in studies of nondeterministic Turing machines (e.g. [PZ83]) and alternating Turing machines [CKS81], but also in studies of deterministic computations like recursive functions [Gri99]. In

2 We tacitly assume that $\sigma(n)$ is a natural number.

the latter, computation trees model multiple subcomputations that may have to be evaluated. For alternating Turing machines, we can think of computations as games played on trees, where branching represents alternatives for the players to choose from. For nondeterministic Turing machines, branching of their computation tree also represents different alternatives in which a computation may proceed. Computation trees constitute one of the most basic and robust representations of computation and we will thus view them as *the* base layer small–step semantics of probabilistic computations against which all our soundness results will be proved.

For pGCL we will present a computation tree semantics in the vein of nondeterministic Turing machines, where branching represents alternatives in which the computation may proceed either due to randomness or nondeterminism. The nodes of the tree represent current configurations of the computation and the edges computation steps, i.e. progress in computation. Formally, this computation tree semantics is defined as follows:

DEFINITION 3.4 (Computation Tree Semantics of pGCL):

A. *A configuration $\kappa = \langle C, \sigma, n, \theta, \eta, q\rangle$ comprises of*

- *either a program $C \in$ pGCL that is left to be executed or a symbol $C = \downarrow$ indicating successful termination,*
- *a program state, i.e. a variable valuation, $\sigma \in \Sigma$,*
- *the number $n \in \mathbb{N}$ of computation steps that have been executed in the past over the course of the computation,*
- *the history $\theta \in \mathbb{N}^*$ of all probabilistic choices that were made,*
- *the history $\eta \in \{L, R\}^*$ of all nondeterministic choices that were made, and*
- *the probability $q \in [0, 1] \cap \mathbb{Q}$ of reaching the configuration if nondeterministic choices are resolved according to η.*

We denote the set of all configurations by $\mathbb{K}$. Notice that $\mathbb{K}$ is countable, *since* pGCL, Σ, $\mathbb{N}^*$, $\{L, R\}^*$, *and* $[0, 1] \cap \mathbb{Q}$ *are countable.*

B. *A transition relation $\vdash\, \subseteq \mathbb{K} \times \mathbb{K}$ between configurations is defined as the smallest relation satisfying the rules given in* Figure 3.1. *As usual, we denote by $\vdash^k$ reachability within k applications of $\vdash$ and by $\vdash^*$ the reflexive–transitive closure of $\vdash$.*

C. *The computation tree of executing program $C \in$ pGCL on input $\sigma \in \Sigma$, denoted $\mathcal{T}^{C,\sigma}$, is a tree (κ_0, K, E) where*

- *$\kappa_0 = \langle C, \sigma, 0, \varepsilon, \varepsilon, 1\rangle$ is the root of the tree,*
- *$K = \{\kappa \mid \kappa_0 \vdash^* \kappa\}$ is the set of nodes in the tree, and*
- *$E = (K \times K) \cap \vdash$ is the set of edges of the tree.*

$$\frac{}{\langle \texttt{skip}, \sigma, n, \theta, \eta, q\rangle \vdash \langle \downarrow, \sigma, n+1, \theta, \eta, q\rangle}\ (\text{skip})$$

$$\frac{}{\langle \texttt{diverge}, \sigma, n, \theta, \eta, q\rangle \vdash \langle \texttt{diverge}, \sigma, n+1, \theta, \eta, q\rangle}\ (\text{diverge})$$

$$\frac{v = \sigma(E)}{\langle x := E, \sigma, n, \theta, \eta, q\rangle \vdash \langle \downarrow, \sigma[x \mapsto v], n+1, \theta, \eta, q\rangle}\ (\text{assign})$$

$$\frac{\mu(\sigma)(v) = a > 0 \qquad \aleph^{-1}(v) = i}{\langle x :\approx \mu, \sigma, n, \theta, \eta, q\rangle \vdash \langle \downarrow, \sigma[x \mapsto v], n+1, \theta i, \eta, q \cdot a\rangle}\ (\text{rnd–assign})$$

$$\frac{\langle C_1, \sigma, n, \theta, \eta, q\rangle \vdash \left\langle C_1', \sigma', n+1, \theta', \eta', q'\right\rangle \qquad C_1' \neq \downarrow}{\langle C_1 \fatsemi C_2, \sigma, n, \theta, \eta, q\rangle \vdash \left\langle C_1' \fatsemi C_2, \sigma', n+1, \theta', \eta', q'\right\rangle}\ (\text{seq1})$$

$$\frac{\langle C_1, \sigma, n, \theta, \eta, q\rangle \vdash \langle \downarrow, \sigma', n+1, \theta', \eta', q'\rangle}{\langle C_1 \fatsemi C_2, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_2, \sigma', n+1, \theta', \eta', q'\rangle}\ (\text{seq2})$$

$$\frac{\varphi(\sigma) = \mathsf{true}}{\langle \texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_1, \sigma, n+1, \theta, \eta, q\rangle}\ (\text{if1})$$

$$\frac{\varphi(\sigma) = \mathsf{false}}{\langle \texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_2, \sigma, n+1, \theta, \eta, q\rangle}\ (\text{if2})$$

$$\frac{}{\langle \{C_1\} \mathbin{\square} \{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_1, \sigma, n+1, \theta, \eta L, q\rangle}\ (\text{nondet1})$$

$$\frac{}{\langle \{C_1\} \mathbin{\square} \{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_2, \sigma, n+1, \theta, \eta R, q\rangle}\ (\text{nondet2})$$

$$\frac{p(\sigma) = a}{\langle \{C_1\}\,[p]\,\{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_1, \sigma, n+1, \theta 0, \eta, q \cdot a\rangle}\ (\text{prob1})$$

$$\frac{p(\sigma) = a}{\langle \{C_1\}\,[p]\,\{C_2\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C_2, \sigma, n+1, \theta 1, \eta, q \cdot (1-a)\rangle}\ (\text{prob2})$$

$$\frac{\varphi(\sigma) = \mathsf{true}}{\langle \texttt{while}(\varphi)\{C\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle C \fatsemi \texttt{while}(\varphi)\{C\}, \sigma, n+1, \theta, \eta, q\rangle}\ (\text{while1})$$

$$\frac{\varphi(\sigma) = \mathsf{false}}{\langle \texttt{while}(\varphi)\{C\}, \sigma, n, \theta, \eta, q\rangle \vdash \langle \downarrow, \sigma, n+1, \theta, \eta, q\rangle}\ (\text{while2})$$

Figure 3.1: Inference rules for the transition relation $\vdash$ between configurations. Recall from Definition 2.1 A. that $\aleph$ is a bijective enumeration of Vals.

A configuration represents the current state of a program execution together with a history of probabilistic and nondeterministic choices that have been made in past computation steps and together with the resulting probability with which this configuration is reached. Given a concrete configuration

$$\kappa \;=\; \langle C, \sigma, n, \theta, \eta, q\rangle \;,$$

component C indicates the rest of the program that is left to be executed. C thus plays the role of a *program counter*, except that it does not contain a line number but instead it contains the entire program that is left to be executed. Notice that if C is a while loop $\texttt{while}(\varphi)\{\mathit{body}\}$ that is about to perform one more iteration of its loop body (i.e. $\varphi(\sigma) = \mathsf{true}$), then the (while1)–rule *prepends* $\texttt{while}(\varphi)\{\mathit{body}\}$ with a copy of *body* (thus obtaining $\mathit{body}\fatsemi\texttt{while}(\varphi)\{\mathit{body}\}$) in order to account for the iteration of the loop body. As a consequence, the „program component" of the configurations along consecutive $\vdash$–transitions does not necessarily get shorter and shorter just as the line numbers of a program counter would not grow strictly larger and larger when performing several iterations through a loop.

Component σ is the current program state that contains the variable valuations. We can thus think of σ as the *memory* which the program instructions can read from and write to.

The component n of configuration κ is the number of computation steps that have been executed in the past. We can think of n as a *runtime tracker*.

Components θ and η are the *histories of choices* that have been made in the past. Whenever a probabilistic or nondeterministic choice is made, this choice is recorded by appending it in either to θ or η, respectively. The history of nondeterministic choices η is a sequence of letters L and R for Left and Right. For the history of probabilistic choices θ, we need infinitely many symbols because for the random assignment it is conceivable that we can choose a value v from arbitrarily (or even countably infinitely) many values.[3] We thus make use of the canonical enumeration $\aleph$ of Vals introduced in Definition 2.1 A. and record $\aleph^{-1}(v)$ in the history of probabilistic choices whenever we sample value v at a random assignment.

Lastly, component q is the probability with which configuration κ was reached, i.e. the multiplied probabilities with which all probabilistic choices on the path from the root of a computation tree to κ were made.

The transition relation $\vdash$ represents single computation steps. Thus, if $\kappa \vdash \kappa'$, then executing a single atomic instruction, checking a single guard (of a conditional choice or a while loop), or flipping a single coin (probabilistic or nondeterministic) takes the computation from configuration κ to configuration κ'.

The computation tree $\mathcal{T}^{C,\sigma}$ as a whole is a tree–representation of all computations that can emanate from executing program C on input σ. The root

3 This is for instance the case for the random assignment $x :\approx \mathsf{Unif}[0 \ldots n]$.

$\kappa_0 = \langle C, \sigma, 0, \varepsilon, \varepsilon, 1\rangle$ of the tree is the initial configuration. It is trivially reached with probability 1 and no (i.e. 0) execution steps have been executed in the past. Also, neither probabilistic nor nondeterministic choices have been made so far. Both histories are thus the empty word ε. The set of nodes $K = \{\kappa \mid \kappa_0 \vdash^* \kappa\}$ is the set of all configurations reachable with non–zero probability under some strategy for resolving nondeterministic choices. The set of edges $E = (K \times K) \cap \vdash$ connects the reachable configurations according to a small–step execution semantics.

Notice that in case of a deterministic program C, the computation „tree“ degenerates to a sequential list because branching only occurs when probabilistic or nondeterministic coins are flipped. In particular notice that conditional choices and while loops do *not* cause branching in the computation tree, since the current program state either satisfies the guard or not and the subsequent configuration is therefore completely determined by the current configuration (in particular the program state). Let us now look at an example of a computation tree:

Example 3.5 (Computation Trees of Probabilistic Programs):
Consider the program C given by

```
while(c = 1){
    {c := 0} [1/2] {skip}
}
```

and some initial state σ with $\sigma(c) = 1$. Then the computation tree $\mathcal{T}^{C,\sigma}$ of executing C on σ is shown in Figure 3.2.

We can observe that the entire computation tree $\mathcal{T}^{C,\sigma}$ from Example 3.5 is isomorphic to its own subtree emanating from $\langle \texttt{while}(\cdots\}, \sigma, 3, 1, \varepsilon, 1/2\rangle$ and also to the subtree emanating from $\langle \texttt{while}(\cdots\}, \sigma, 6, 1, \varepsilon, 1/4\rangle$. This is due to the fact that future progress in computation in general does not depend on the (histories of) probabilistic and nondeterministic choices that were made in the past but solely on the current program C that is left to be executed and the current program state σ. This independence from the histories can be captured formally by an *equivalence of $\vdash$–transitions* as follows:

Proposition 3.6 (Equivalence of $\vdash$–Transitions):
The following equivalence holds:

$$\begin{aligned} &\langle C, \sigma, n, \theta, \eta, q\rangle \;\vdash\; \langle C', \sigma', n+1, \theta w, \eta u, q'\rangle \\ &\quad\text{iff}\quad \langle C, \sigma, 0, \varepsilon, \varepsilon, 1\rangle \;\vdash\; \langle C', \sigma', 1, w, u, q'/q\rangle \end{aligned}$$

$$\begin{tikzcd}[column sep=tiny]
& \langle \texttt{while}(c=1)\{\{c:=0\}\,[1/2]\,\{\texttt{skip}\}\}, \sigma, 0, \varepsilon, \varepsilon, 1\rangle \arrow[d, "(\text{while1})"] & & \\
& \left\langle \begin{array}{l}\{c:=0\}\,[1/2]\,\{\texttt{skip}\}\,; \\ \texttt{while}(\cdots\}\end{array}, \sigma, 1, \varepsilon, \varepsilon, 1\right\rangle \arrow[dl, "{(\text{seq1}), (\text{prob1})}"'] \arrow[dr, "{(\text{seq1}), (\text{prob2})}"] & & \\
\left\langle \begin{array}{l}c:=0\,; \\ \texttt{while}(\cdots\}\end{array}, \sigma, 2, 0, \varepsilon, 1/2\right\rangle \arrow[d, "{(\text{seq1}), (\text{assign})}"] & & \left\langle \begin{array}{l}\texttt{skip}\,; \\ \texttt{while}(\cdots\}\end{array}, \sigma, 2, 1, \varepsilon, 1/2\right\rangle \arrow[d, "{(\text{seq1}), (\text{skip})}"] & \\
\langle \texttt{while}(\cdots\}, \sigma[c\mapsto 0], 3, 0, \varepsilon, 1/2\rangle \arrow[d, "(\text{while2})"] & & \langle \texttt{while}(\cdots\}, \sigma, 3, 1, \varepsilon, 1/2\rangle \arrow[d, "(\text{while1})"] & \\
\langle \downarrow, \sigma[c\mapsto 0], 4, 0, \varepsilon, 1/2\rangle & & \left\langle \begin{array}{l}\{c:=0\}\,[1/2]\,\{\texttt{skip}\}\,; \\ \texttt{while}(\cdots\}\end{array}, \sigma, 4, 1, \varepsilon, 1/2\right\rangle \arrow[dl, "{(\text{seq1}), (\text{prob1})}"'] \arrow[dr, "{(\text{seq1}), (\text{prob2})}"] & \\
& \left\langle \begin{array}{l}c:=0\,; \\ \texttt{while}(\cdots\}\end{array}, \sigma, 5, 10, \varepsilon, 1/4\right\rangle \arrow[d, "{(\text{seq1}), (\text{assign})}"] & & \left\langle \begin{array}{l}\texttt{skip}\,; \\ \texttt{while}(\cdots\}\end{array}, \sigma, 5, 11, \varepsilon, 1/4\right\rangle \arrow[d, "{(\text{seq1}), (\text{skip})}"] \\
& \langle \texttt{while}(\cdots\}, \sigma[c\mapsto 0], 6, 10, \varepsilon, 1/4\rangle \arrow[d, "(\text{while2})"] & & \langle \texttt{while}(\cdots\}, \sigma, 6, 11, \varepsilon, 1/4\rangle \arrow[d, "(\text{while1})"] \\
& \langle \downarrow, \sigma[c\mapsto 0], 7, 10, \varepsilon, 1/4\rangle & & \vdots
\end{tikzcd}$$

Figure 3.2: Computation tree $\mathcal{T}^{C,\sigma}$ of executing the probabilistic program C = $\texttt{while}(c=1)\{\{c:=0\}\,[1/2]\,\{\texttt{skip}\}\}$ on a state σ with $\sigma(c)=1$. Edge labels indicate which $\vdash$–rules are used to derive this transition. Notice that $\mathcal{T}^{C,\sigma}$ is *infinite* since all configurations of the form $\langle \texttt{while}(\cdots\}, \sigma, 3n, 1^n, \varepsilon, 1/2^n\rangle$ for $n \in \mathbb{N}$ are reachable along a rightmost path.

Proposition 3.6 can be interpreted as a form of reconditioning or rescaling: If we arrive at some configuration $\langle C, \sigma, n, \theta, \eta, q \rangle$ with probability q and transition from there to a configuration $\langle C', \sigma', n+1, \theta w, \eta u, q' \rangle$ then this happens with probability q'/q. We can then renormalize q to 1, reset the runtime tracker, and delete all histories: *Given* that we have somehow reached a configuration with program C and state σ we can restart from there and transit with probability q'/q to configuration $\langle C', \sigma', 1, w, u, q'/q \rangle$. This rescaling can be interpreted as a *Markov property* showing that the probability of transiting to a next configuration depends solely on the current state of the execution (comprising of program and program state) and not on any historic events. In Section 3.3.3, we thus „quotient out" the runtime tracker, the histories, and the probability q and by that obtain a Markov decision process semantics for pGCL.

3.3.2 *Distributions over Final States*

In the previous section we saw that execution of a probabilistic program can result in multiple (even infinitely many) possible computation paths since — in contrast to deterministic programs — the behavior of the program depends not only on the initial state but also on nondeterminism and randomness. This goes so far that a program may terminate only with a certain probability strictly between 0 and 1.

So how can we now conceive of *<u>the</u> outcome* of a probabilistic computation? Let us leave nondeterminism out of the picture for the moment, i.e. let us consider only tame programs. Despite the fact that executing such programs does not necessarily yield a unique final state, tame probabilistic programs do yield a unique *probability distribution* over final states. To be more precise, we are dealing with *subdistributions*, i.e. distributions with a total mass that may be less than 1. The „mass defect" of the resulting subdistribution then represents the probability of nontermination. To see that this is not just a convenient way for modeling, consider the program

$$\{\texttt{diverge}\}\,[1/3]\,\{x := 5\}\ .$$

Starting this program in some initial state σ will give a proper subdistribution over final states μ given by

$$\mu(\tau) \;=\; \begin{cases} \frac{2}{3}, & \text{if } \tau = \sigma[x \mapsto 5] \\ 0, & \text{otherwise.} \end{cases}$$

For each potential final state τ, this subdistribution μ gives us by $\mu(\tau)$ the precise, *unnormalized* probability that the execution of the above program will terminate in final state τ. The total mass of μ is $2/3$ and the missing

$$\frac{\begin{array}{c}\langle C,\sigma,n,\theta,\eta,q\rangle \vdash \langle C',\sigma,n+1,\theta,\eta L,q\rangle \\ \langle C,\sigma,n,\theta,\eta,q\rangle \vdash \langle C'',\sigma,n+1,\theta,\eta R,q\rangle \\ \mathfrak{s}(C,\sigma) = L\end{array}}{\langle C,\sigma,n,\theta,\eta,q\rangle \vdash_{\mathfrak{s}} \langle C',\sigma,n+1,\theta,\eta L,q\rangle}\ (\mathfrak{s}\text{–sched1})$$

$$\frac{\begin{array}{c}\langle C,\sigma,n,\theta,\eta,q\rangle \vdash \langle C',\sigma,n+1,\theta,\eta L,q\rangle \\ \langle C,\sigma,n,\theta,\eta,q\rangle \vdash \langle C'',\sigma,n+1,\theta,\eta R,q\rangle \\ \mathfrak{s}(C,\sigma) = R\end{array}}{\langle C,\sigma,n,\theta,\eta,q\rangle \vdash_{\mathfrak{s}} \langle C'',\sigma,n+1,\theta,\eta R,q\rangle}\ (\mathfrak{s}\text{–sched2})$$

$$\frac{\langle C,\sigma,n,\theta,\eta,q\rangle \vdash \langle C',\sigma',n+1,\theta',\eta,q'\rangle}{\langle C,\sigma,n,\theta,\eta,q\rangle \vdash_{\mathfrak{s}} \langle C',\sigma',n+1,\theta',\eta,q'\rangle}\ (\mathfrak{s}\text{–sched3})$$

Figure 3.3: Inference rules for defining the $\mathfrak{s}$–scheduled transition relation $\vdash_{\mathfrak{s}}$ between configurations, where $\mathfrak{s} \in \mathsf{Scheds}$ is a scheduler.

probability mass of 1/3 is the probability that no final state is reached, i.e. the probability of nontermination.

Let us now bring nondeterminism back into the picture and see how we can describe the outcomes of executing general pGCL programs on given initial states in a systematic way, namely by extracting probability distributions over final program states from computation trees. For that, we first need a way to remove nondeterminism from the computation tree so that it becomes a purely probabilistic transition system, since otherwise there need not exist a unique probability distribution. We do this by means of restricting the number of successors in the $\vdash$ transition relation.

Definition 3.7 (Scheduled $\vdash$–Transitions):
A scheduler $\mathfrak{s}$ is a function

$$\mathfrak{s}\colon \mathsf{pGCL} \times \Sigma \to \{L, R\}$$

mapping pairs of programs and program states to either letter L or R. The set of all schedulers is denoted by Scheds.

For $\mathfrak{s} \in \mathsf{Scheds}$, the $\mathfrak{s}$–scheduled transition relation $\vdash_{\mathfrak{s}}$ is given by the inference rules in Figure 3.3.

Schedulers are a means of resolving nondeterminism in systems that feature both probabilistic and nondeterministic uncertainty [Var85; Put05]. Scheduled transitions behave just like unscheduled ones (see rule ($\mathfrak{s}$–sched3)) with a single exception (see rules ($\mathfrak{s}$–sched1) and ($\mathfrak{s}$–sched2)): When program C executes a nondeterministic choice, the scheduled relation $\vdash_{\mathfrak{s}}$ selects a *single* successor configuration according to scheduler $\mathfrak{s}$, whereas the unscheduled relation $\vdash$ has to two successors in pari passu.

The single purpose of a scheduler is thus to *resolve nondeterministic choices*. The benefit is that this allows for defining a probability distribution over final states that is established by executing a probabilistic program on some initial state under a scheduler that resolves nondeterminism.

Definition 3.8 (Probability Distribution Semantics of pGCL): *Let C be a* pGCL *program, $\sigma \in \Sigma$ be an initial program state and $\mathfrak{s} \in$* Scheds *be a scheduler. Then the distribution over final states established by executing C on input σ under scheduler $\mathfrak{s}$, denoted $\llbracket C \rrbracket_{\sigma}^{\mathfrak{s}}$, is a (sub)probability distribution over program states*[4] *given by*

$$\llbracket C \rrbracket_{\sigma}^{\mathfrak{s}}(\tau) \;=\; \sum_{\langle\downarrow, \tau, n, \theta, \eta, q\rangle \in K} q\,, \qquad \text{where}$$

$$K \;=\; \Big\{ \langle\downarrow,\, \tau,\, n,\, \theta,\, \eta,\, q\rangle \;\Big|\; \langle C,\, \sigma,\, 0,\, \varepsilon,\, \varepsilon,\, 1\rangle \;\vdash_{\mathfrak{s}}^{*}\; \langle\downarrow,\, \tau,\, n,\, \theta,\, \eta,\, q\rangle \Big\}\,.$$

Notice that in case of tame programs, the scheduler is entirely irrelevant and the programs naturally produce probability (sub)distributions. We thus omit the scheduler for tame programs and simply write $\llbracket C \rrbracket_{\sigma}$. We conclude this section with some examples of both nondeterministic and tame programs.

Example 3.9 (Probability Distributions of pGCL Programs):

A. Consider the program

$$\{x :\!= 1 - x\} \,\Box\, \{\mathtt{skip}\}\,⨾$$
$$\{x :\!= 0\}\,[1/2]\,\{x :\!= 1\}\,,$$

some initial state σ, and *any* scheduler $\mathfrak{s}$. Then the probability distribution $\llbracket C \rrbracket_{\sigma}^{\mathfrak{s}}$ is given by

$$\llbracket C \rrbracket_{\sigma}^{\mathfrak{s}}(\tau) \;=\; \begin{cases} \frac{1}{2}, & \text{if } \tau = \sigma[x \mapsto 0] \text{ or } \tau = \sigma[x \mapsto 1] \\ 0, & \text{otherwise.} \end{cases}$$

Note that the probability distribution is unique, no matter what scheduler is imposed.

4 I.e. $\llbracket C \rrbracket_{\sigma}^{\mathfrak{s}} \colon \Sigma \to [0, 1]$, such that $\sum_{\tau \in \Sigma} \llbracket C \rrbracket_{\sigma}^{\mathfrak{s}}(\tau) \;\leq\; 1$.

B. Consider the program

$$\{x := 0\}\,[1/2]\,\{x := 1\}\,\mathbin{;} \\ \{x := 1 - x\} \mathbin{\Box} \{\mathtt{skip}\}\ ,$$

some initial state σ, and scheduler $\mathfrak{s}_{min}$ given by

$$\mathfrak{s}_{min}(C, \sigma') \;=\; \begin{cases} L, & \text{if } \sigma'(x) = 1 \\ R, & \text{otherwise.} \end{cases}$$

Then the probability distribution $[\![C]\!]_\sigma^{\mathfrak{s}_{min}}$ is given by

$$[\![C]\!]_\sigma^{\mathfrak{s}}(\tau) \;=\; \begin{cases} 1, & \text{if } \tau = \sigma'[x \mapsto 0] \\ 0, & \text{otherwise.} \end{cases}$$

C. Consider the program C_{geo} given by

```
while(c = 1){
    {c := 0} [1/2] {x := x + 1}
}
```

and some initial state σ with $\sigma(c) = 1$ and $\sigma(x) = 0$. Then the computation tree $\mathcal{T}^{C_{geo},\sigma}$ of executing C_{geo} on σ is shown in Figure 3.4. The resulting probability distribution $[\![C_{geo}]\!]_\sigma$ is given by

$$[\![C]\!]_\sigma(\tau) \;=\; \begin{cases} \frac{1}{2^{n+1}}, & \text{if } \tau = \sigma[c, x \mapsto 0, n], \text{ for } n \in \mathbb{N} \\ 0, & \text{otherwise.} \end{cases}$$

The program C thus establishes a geometric distribution on x whenever it is ran on an initial state σ with $\sigma(c) = 1$ and $\sigma(x) = 0$. Notice that schedulers are irrelevant and thus omitted since C_{geo} is tame.

3.3.3 *Markov Decision Process Semantics*

In Section 3.3.1, we presented a very basic computation tree semantics of pGCL programs in which configurations contained histories of choices that were made in the past. We also noticed that computation steps are independent from those histories (see Proposition 3.6). In this section, we present a semantics for pGCL in which those histories are omitted. The semantics we are about to present will be based on Markov decision processes (MDPs). We do not introduce them here but instead refer to Appendix B for basic

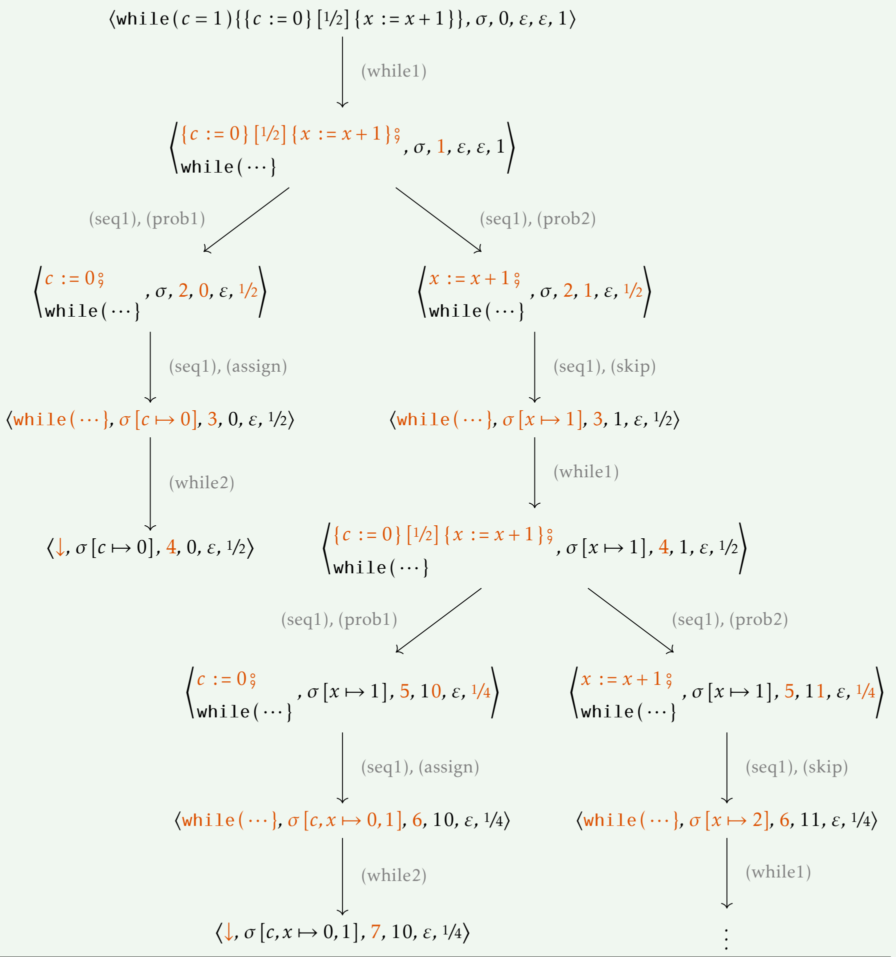
⟨while(c = 1){{c := 0} [1/2] {x := x + 1}}, σ, 0, ε, ε, 1⟩
(while1)
⟨{c := 0} [1/2] {x := x + 1} ⨾ while(⋯), σ, 1, ε, ε, 1⟩
(seq1), (prob1)
(seq1), (prob2)
⟨c := 0 ⨾ while(⋯), σ, 2, 0, ε, 1/2⟩
⟨x := x + 1 ⨾ while(⋯), σ, 2, 1, ε, 1/2⟩
(seq1), (assign)
(seq1), (skip)
⟨while(⋯}, σ[c ↦ 0], 3, 0, ε, 1/2⟩
⟨while(⋯}, σ[x ↦ 1], 3, 1, ε, 1/2⟩
(while2)
(while1)
⟨↓, σ[c ↦ 0], 4, 0, ε, 1/2⟩
⟨{c := 0} [1/2] {x := x + 1} ⨾ while(⋯}, σ[x ↦ 1], 4, 1, ε, 1/2⟩
(seq1), (prob1)
(seq1), (prob2)
⟨c := 0 ⨾ while(⋯}, σ[x ↦ 1], 5, 10, ε, 1/4⟩
⟨x := x + 1 ⨾ while(⋯}, σ[x ↦ 1], 5, 11, ε, 1/4⟩
(seq1), (assign)
(seq1), (skip)
⟨while(⋯}, σ[c, x ↦ 0, 1], 6, 10, ε, 1/4⟩
⟨while(⋯}, σ[x ↦ 2], 6, 11, ε, 1/4⟩
(while2)
(while1)
⟨↓, σ[c, x ↦ 0, 1], 7, 10, ε, 1/4⟩
⋮

Figure 3.4: The computation tree $\mathcal{T}^{C_{geo},\sigma}$ yielded by executing program C_{geo} = while(c=1){{c := 0} [1/2] {x := x + 1}} on an initial state σ with $\sigma(c) = 1$ and $\sigma(x) = 0$. Edge labels indicate which $\vdash$–rules are used to derive this transition. Notice that $\mathcal{T}^{C_{geo},\sigma}$ is *infinite* since all configurations of the form $\langle$while($\cdots$}$, \sigma[x \mapsto n], 3n, 1^n, \varepsilon, 1/2^n\rangle$ for $n \in \mathbb{N}$ are reachable along a rightmost path.

s	α	s'	$P(s,\alpha)(s')$
$\langle C', \sigma' \rangle$	N	$\langle C'', \sigma'' \rangle$	$\begin{cases} q, & \text{if } \langle C', \sigma', 0, \varepsilon, \varepsilon, 1 \rangle \vdash \langle C'', \sigma'', 1, \theta, \varepsilon, q \rangle \\ 0, & \text{otherwise.} \end{cases}$
$\langle C', \sigma' \rangle$	L	$\langle C'', \sigma'' \rangle$	$\begin{cases} 1, & \text{if } \langle C', \sigma', 0, \varepsilon, \varepsilon, 1 \rangle \vdash \langle C'', \sigma', 1, \varepsilon, L, 1 \rangle \\ 0, & \text{otherwise.} \end{cases}$
$\langle C', \sigma' \rangle$	R	$\langle C'', \sigma'' \rangle$	$\begin{cases} 1, & \text{if } \langle C', \sigma', 0, \varepsilon, \varepsilon, 1 \rangle \vdash \langle C'', \sigma', 1, \varepsilon, R, 1 \rangle \\ 0, & \text{otherwise.} \end{cases}$
$\langle \downarrow, \sigma' \rangle$	N	$\langle sink \rangle$	1
$\langle sink \rangle$	N	$\langle sink \rangle$	1
— all other cases —			0

Table 3.1: Definition of the transition probability function of operational MDPs.

definitions, to Baier & Katoen for more details [BK08, Chapter 10], and to Puterman for a dedicated in–depth treatment [Put05].

A Markov decision process semantics for pGCL was presented by Gretz, Katoen, and McIver [GKM12; GKM14]. Earlier work on viewing programs as Markov decision processes was presented by Monniaux [Mon05]. The semantics we present in the following is basically the one of Gretz *et al.* and differs only in minor technical details (e.g. in that Gretz *et al.* do not consider random assignments).

DEFINITION 3.10 (Operational Markov Decision Processes):
The operational MDP of executing program $C \in$ pGCL on input $\sigma \in \Sigma$ is the MDP (cf. Definition B.1*) $\mathcal{M}^{C,\sigma} = (S, \langle C, \sigma \rangle, \{L, R, N\}, P)$, where*

- $S = \{\langle C', \sigma' \rangle \mid \langle C, \sigma, 0, \varepsilon, \varepsilon, 1 \rangle \vdash^* \langle C', \sigma', n, \theta', \eta', q' \rangle\} \cup \{\langle sink \rangle\}$ *is a set of states,*
- $\langle C, \sigma \rangle$ *is the initial state,*
- $\{L, R, N\}$ *is the set of actions, and*
- P *is the transition probability function defined according to the rules in* Table 3.1.

A state of an operational MDP represents a *collection* of all computation tree configurations that share the same program and program state. A designated

$\langle sink \rangle$ state acts as a sink after reaching a state of the form $\langle \downarrow, \ldots \rangle$ indicating termination of the computation. This sink is needed since in MDPs every state needs at least one successor state. The actions are given by the letters L, R, and N which stand for *L*eft, *R*ight, and *N*one, respectively. L and R indicate which branch is chosen when performing a nondeterministic choice whereas N is the default action when no nondeterministic choice is to be executed.

The probability $P(s, \alpha)(s')$ determined by the transition probability function P is the probability of making a transition from state s to state s' with action α. Let us very briefly go over the definition of the transition probability function in Table 3.1: The first rule deals with deterministic and probabilistic instructions (i.e. guard evaluations, probabilistic choices, deterministic assignments, and random assignments) in the self–evident way. The associated action is N since no nondeterministic choice is performed.

The next two rules cover nondeterministic choices. The transition probabilities are either 1 or 0 since no randomness is involved; the action must be either L or R in order to determine which branch is chosen.

The rule $P(\langle \downarrow, \sigma' \rangle, N)(\langle sink \rangle) = 1$ leads terminated executions into the designated sink state. Recall that a sink state is necessary since every MDP state has to have a successor state. Likewise, the sink state has to have a successor state which is again the sink state. This is captured by the rule $P(\langle sink \rangle, N)(\langle sink \rangle) = 1$.

The operational MDP $\mathcal{M}^{C,\sigma}$ is a potentially more compact representation of the computation tree $\mathcal{T}^{C,\sigma}$. An example where the representation is more compact (in fact: *finite instead of infinite*) is given by reconsidering Example 3.5 from an MDP point of view:

Example 3.11 (Operational MDPs of Probabilistic Programs):
Reconsider the program C from Example 3.5 given by

```
while(c = 1){
    {c := 0} [1/2] {skip}
}
```

and consider some initial state σ with $\sigma(c) = 1$. Then the operational MDP of executing C on σ is shown in Figure 3.5. Notice that this MDP can be translated into a *Markov chain*, since the program C is tame, i.e. the program contains no nondeterministic choices.

We notice that the MDP of Figure 3.5 captures nicely the automorphism on the computation tree of Figure 3.2, namely by the back edge from state $\langle \texttt{skip} ; \texttt{while}(\cdots\}, \sigma \rangle$ to the initial state of the MDP. This „folding“ of the computation tree into itself is what makes the MDP finite whereas the com-

$$\begin{tikzcd}
 & {} \arrow[d] & \\
 & \langle \texttt{while}(c=1)\{\{c:=0\}\,[1/2]\,\{\texttt{skip}\}\}, \sigma\rangle \arrow[d] & \\
 & \left\langle \begin{array}{l}\{c:=0\}\,[1/2]\,\{\texttt{skip}\}\fatsemi\\ \texttt{while}(\cdots\}\end{array}, \sigma\right\rangle \arrow[dl, "\frac{1}{2}"'] \arrow[dr, "\frac{1}{2}"] & \\
\left\langle \begin{array}{l}c:=0\fatsemi\\ \texttt{while}(\cdots\}\end{array}, \sigma\right\rangle \arrow[d] & & \left\langle \begin{array}{l}\texttt{skip}\fatsemi\\ \texttt{while}(\cdots\}\end{array}, \sigma\right\rangle \arrow[uul, bend right] \\
\langle \texttt{while}(\cdots\}, \sigma[c\mapsto 0]\rangle \arrow[d] & & \\
\langle \downarrow, \sigma[c\mapsto 0]\rangle \arrow[dr] & & \\
 & \langle \mathit{sink}\rangle \arrow[loop right] &
\end{tikzcd}$$

Figure 3.5: Operational MDP $\mathcal{M}^{C,\sigma}$ of executing the probabilistic program $C = \texttt{while}(c=1)\{\{c:=0\}\,[1/2]\,\{\texttt{skip}\}\}$ on some state σ with $\sigma(c) = 1$. Unlabeled edges are transitions with probability 1. All transitions in this MDP are associated with action N.

putation tree was infinite. An advantage of having an MDP representation of a pGCL program's computation readily available is that it makes probabilistic programs amenable to *fully automated probabilistic model checking tools*, at least in case of finite operational MDPs. In case of infinite operational MDPs, *bounded model checking* is an appropriate technique. A first approach for bounded model checking of probabilistic programs which builds upon the Storm probabilistic model checker [Deh+17] was presented in [Jan+16].

Let us finally show how the MDP semantics is related to the distributions over final states presented in the previous section. We do so by constructing from an operational MDP an operational *Markov chain* (MC) that is induced by a given scheduler.

DEFINITION 3.12 (Operational Markov Chains Induced by Schedulers):

A. *Let $\mathcal{M}^{C,\sigma} = (S, \langle C, \sigma \rangle, \{L, R, N\}, P')$ be the operational MDP of executing C on σ and let $\mathfrak{s} \in \mathsf{Scheds}$ be a scheduler. Then the operational MC of executing program $C \in \mathsf{pGCL}$ on input $\sigma \in \Sigma$ under scheduler $\mathfrak{s}$ is the MC (cf.* Definition B.3*) $\mathcal{M}_{\mathfrak{s}}^{C,\sigma} = (S, \langle C, \sigma \rangle, P)$, where for all $s, s' \in S$ we have*[5]

$$P(s)(s') \;=\; P'\big(s, \mathfrak{s}(s)\big)(s')\,.$$

B. *We denote by $\Pr_{\mathcal{M}_{\mathfrak{s}}^{C,\sigma}}(\Diamond s)$ the probability of eventually reaching state s in the operational MC $\mathcal{M}_{\mathfrak{s}}^{C,\sigma}$ (cf.* Definition B.4*).*

Schedulers for MDPs thus play the same role as schedulers in computation trees: their purpose is to resolve the nondeterminism in order to obtain a fully probabilistic transition system. Having means of resolving nondeterminism in MDPs as well as computation trees, we can now relate reachability probabilities in operational MCs to the distributions over final states presented in the previous section.

PROPOSITION 3.13:
Let $\mathfrak{s} \in \mathsf{Scheds}$ be a scheduler, let $[\![C]\!]_\sigma^{\mathfrak{s}}$ be the distribution over final states established by executing C on input σ under scheduler $\mathfrak{s}$, and let $\mathcal{M}_{\mathfrak{s}}^{C,\sigma}$ be the operational MC of executing C on σ under scheduler $\mathfrak{s}$. Then for all final states $\tau \in \Sigma$,

$$[\![C]\!]_\sigma^{\mathfrak{s}}(\tau) \;=\; \Pr_{\mathcal{M}_{\mathfrak{s}}^{C,\sigma}}\big(\Diamond \langle \downarrow, \tau \rangle\big)\,.$$

Intuitively the above proposition states that the probability of reaching a certain final program state τ in a computation tree under scheduler $\mathfrak{s}$ is the

5 We are being a little bit sloppy on notation here: a scheduler $\mathfrak{s}$ is a function of type $\mathsf{pGCL} \times \Sigma \to \{L, R\}$ and thus takes as argument a *pair*. However, since an MDP state s for us is a pair from the set $\mathsf{pGCL} \times \Sigma$, we can safely write $\mathfrak{s}(s)$.