Quantitative Observables and Averages in Probabilistic Constraint Programming

Alessandra Di Pierro\textsuperscript{1} and Herbert Wiklicky\textsuperscript{2}

\textsuperscript{1} Dipartimento di Informatica, Università di Pisa, Italy
\textsuperscript{2} Department of Computing, Imperial College, London, UK

\textbf{Abstract.} We investigate notions of observable behaviour of programs which include quantitative aspects of computation along with the most commonly assumed qualitative ones. We model these notions by means of a transition system where transitions occur with a given probability and an associated 'cost' expressing some complexity measure (e.g. running time or, in general, resources consumption). The addition of these quantities allows for a natural formulation of the average behaviour of a program, whose specification and analysis is particularly important in the study of system performance and reliability. It also allows for an average-case analysis of programs' complexity, which can be seen as a semantical counterpart of the average-case asymptotic analysis of algorithms. We base our model on the Concurrent Constraint Programming (CCP) paradigm and we argue that it can be an appropriate base for further developments oriented to the analysis and verification of average properties.

1 Introduction

The analysis and verification of a system has a fundamental importance for the system development. The advantages of formal methods for these tasks are well known. Similarly, in the system performance evaluation one can benefit from a formal specification of properties expressing the basic performance measures, such as system throughput and average response time, along with methods for automatically verifying those properties, which are solidly built on some mathematical base. On the other hand, in the practical realisation of systems formal methods are not always the methods of choice, and very often ad hoc methods are preferred. One reason of this relies on the absence in most cases of a viable semantics to be used as a necessary base on which to build a formal deductive system. The most desirable requirements of such a semantics should be rigour and precision on the one hand, and simplicity and clarity on the other hand. While this is in general not a trivial task, it is relatively simple to achieve when the adopted language is a \textit{declarative} one.

Based on these considerations and with possible applications to programs analysis and verification in mind, we adopt in this paper the Concurrent Constraint Programming (CCP) paradigm of \cite{37,36}, whose declarativity descends
from the clear ‘logical reading’ of programs provided by its denotational semantics [11]. We then extend the classical CCP operational semantics [37, 12] so as to include two quantities representing probabilities and ‘cost’ respectively, the latter being some measures for computational complexity. This augmented semantics is formalised in terms of two different notions: One captures the probabilistic I/O behaviour of a program by collecting the results delivered by the program executions together with a probability distribution expressing for each result the likelihood of being computed; the other associates to each result also an index of the complexity of computing that result. The first extension was already introduced by the authors in a previous work [13], where it is called Probabilistic CCP (PCCP). The second notion is a further extension, and turns out to be particularly suitable for defining another kind of observables which captures the average behaviour of a program. This plays a fundamental role in the evaluation and modelling of many relevant performance and reliability indices of probabilistic systems, where one commonly refers to the long-run average time required by the system to perform given tasks. By ‘long-run average’ it is therefore meant an average over an infinite period of time [10, 16]. On the other hand, the average behaviour of a probabilistic program can also be captured by considering the average among all its possible results from a given initial store.

We will show that this correspond to the ‘expected value’ of a suitably defined random variable expressing the particular notion of cost (time, etc.) which the average refers to.

The two kinds of averages mentioned above (i.e. long-run average and expected value) can be naturally formalised in terms of the augmented operational semantics we propose in this paper. The basic idea is that the PCCP mechanism for inferring probabilities can be used for determining the weights in the expected value of a random variable expressing the time, the latter being the quantity measured as the computational cost. This can be generalised to quantities other than the running time. The model we obtain can then be used as a base for the formal analysis and verification of probabilistic systems. We can use it for deterministic programs as well, provided that we assume an appropriate interpretation of the ‘average’, which here has to be meant as an average calculated over the (possibly infinite) different inputs. We also give an example for this latter case, which shows some similarities with the average-case asymptotic analysis of algorithms.

We present some special cases of augmented transition systems, each corresponding to the choice of a particular cost function. Various program properties can be defined in this way, from the simplest I/O behaviour to important ones like termination. We also define a probabilistic version of the latter property, which can be used for a more realistic analysis in all those situations where the classical notion of termination turns out to be ‘too strong’.

Finally, we suggest a possible application of the augmented operational semantics for the analysis of probabilistic programs based on an appropriately defined abstract interpretation methodology, which we plan to develop as a future work.
2 Probabilistic Concurrent Constraint Programming

2.1 Syntax of PCCP

Probabilistic Concurrent Constraint Programming (PCCP) was introduced in its original form in [13]. A prototype interpreter of PCCP to gain practical experiences was later implemented on top of Sistus Prolog [2]. The motivation behind PCCP was the formalisation of randomised algorithms within the CCP framework. These algorithms are characterised by a “coin flipping” device (random choice) which determines the flow of information. Although algorithms incorporating randomness, such as those for calculating \( \pi \) or more generally the integration of complicated functions, have been known in mathematics for a long time (e.g. Monte Carlo algorithms), it is only in the last decade that randomised algorithms have found widespread application in many different areas of computer science, for example as a tool in computational geometry and number theory. Various randomised algorithms and procedures have been investigated since, of which we mention just a few examples: simulated annealing in combinatorial optimisation [1], genetic algorithms [18], probabilistic primality tests in particular for use in crypto-systems [32], and randomised proof procedures (e.g. for linear logic [31]). The benefits of randomisation at the base of the tremendous growth of interest in such algorithms are simplicity and speed. For this reason the best known algorithms for many problems are nowadays randomised ones [28].

In PCCP randomness is expressed in the form of a probabilistic choice, which replaces the CCP nondeterministic choice and allows a program to make stochastic moves during its execution. We also replace the implicit non-deterministic scheduling in the interleaving semantics of the parallel construct by a probabilistic scheduler. This allows us to implement a kind of prioritised parallelism. By these constructs the element of chance is introduced directly at the algorithmic level without the need of any modification at the data level. Thus, the constraint system \( C \) underlying the language doesn’t need to be re-structured according to some probabilistic or fuzzy or belief system (see e.g. [24, 25] or [7, 6]), and we can assume the definition of constraint systems as cylindric algebraic cpo’s given in [37], to which we refer for more details.

\[
A ::= \text{stop} \mid \text{tell}(c) \mid \prod_{i=1}^{n} \text{ask}(c_i) \rightarrow p_i : A_i \mid \prod_{i=1}^{n} q_i : A_i \mid \exists x A \mid p(x)
\]

Table 1. The syntax for PCCP.

The syntax of a PCCP agent is given in Table 1, where \( c \) and \( c_i \) are finite constraints in \( C \). We will assume that \( C \) is countable. A PCCP program \( P \) is then an object of the form \( D.A \), where \( D \) is a set of procedure declarations of the form \( p(x) : \neg A \) and \( A \) is an agent.
In the following we will sometimes use a shorthand notation for the guards in the choice by writing \( c \rightarrow p : A \) instead of \( \text{ask}(c) \rightarrow p : A \).

### 2.2 Informal Semantics

PCCP agents are the same as CCP agents but for the non-deterministic choice construct, which is replaced by the probabilistic choice

\[
\begin{align*}
\prod_{i=1}^{n} \text{ask}(c_i) & \rightarrow \tilde{p}_i : A_i, \\
\| \|_{i=1}^{n} q_i & : A_i.
\end{align*}
\]

In the probabilistic choice, the probability associated with each alternative expresses how likely it is that, by repeating the same computation "sufficiently" often, the computation will continue by actually performing that alternative. This can be seen as restricting the original non-determinism by imposing some requirements on the frequency of choices. The operational meaning of the probabilistic choice construct is as follows: First, check whether constraints \( c_i \) are entailed by the store. This is expressed in Table 2 by \( d \vdash c_j \), where \( \vdash \) is the partial ordering (entailment) in the underlying constraint system, and \( d \) represents the current store. Then we have to normalise the probability distribution by considering only the enabled agents, i.e. the agents such that \( c_i \) is entailed. This means that we have to re-define the probability distribution so as only enabled agents have non-zero probabilities and the sum of these probabilities is one. In general, this can be done by considering for enabled agents the normalised transition probability,

\[
\tilde{p}_i = \frac{p_i}{\sum_{j \in \mathcal{E}} p_j},
\]

where the sum \( \sum_{j \in \mathcal{E}} p_j \) is over all enabled agents. When there are no enabled agents normalisation is not necessary. There might occur the situation where \( \sum_{j \in \mathcal{E}} p_j = 0 \) (all enabled agents have probability zero), in this case the normalisation will consist in the assignment of a uniform distribution to the enabled agents. Finally, one of the enabled agents is chosen according to the new probability distribution \( \tilde{p}_i \).

The intuitive semantics of the prioritised interleaving is very similar. Again we replace the (implicit) non-determinism of the scheduler, which has to decide in the interleaving semantics which agent has to be executed first, by a probabilistic version. This selection has to be made among all those agents \( A_i \) which are active, i.e. can make a transition. Then we have to normalise the priorities \( p_i \) of the active agents. This normalisation is done in the same way as described above for the probabilistic choice. Finally, the scheduler chooses one of the active agents according to the new probability distribution \( \tilde{p}_i \).

Note that in the definition of the transition relation/system (see Section 2.3) each transition (i.e. single computational step from one agent to its continuation)
will have a probability associated to it. This ‘forces’ a probabilistic treatment not only of the choice but also of the interleaving (parallel construct). For this reason, we opted in the design of the language for an explicit probabilistic definition of such a construct, which makes the language more flexible than the one we would obtain by keeping the classical (non-probabilistic) syntax of parallelism. This would result in our semantics in a scheduler which always chooses among all active agents according to a uniform probability distribution.

2.3 Transition System

\[
\begin{align*}
\text{R1} & \quad \{\text{tell}(c), d\} \rightarrow_1 \{\text{stop}, c \sqcup d\} \\
\text{R2} & \quad \left[\prod_{i=1}^{n} \text{ask}(c_i) \rightarrow p : A_i, d\right] \rightarrow_{\tilde{p}} \{A_j, d\} \quad j \in [1, n] \text{ and } d \vdash c_j \\
\text{R3} & \quad \left\langle A_j, c \right\rangle \rightarrow_{p} \left\langle A'_j, c' \right\rangle \\
& \quad \left\langle \prod_{i=1}^{n} p_i : A_i, c \right\rangle \rightarrow_{\tilde{p}} \left\langle \prod_{i \neq j} p_i : A_i \parallel p_j : A_j, c' \right\rangle \quad j \in [1, n] \\
\text{R4} & \quad \left\langle A, d \sqcup \exists x, e \right\rangle \rightarrow_{p} \left\langle A', d' \right\rangle \\
& \quad \exists x. A, c \rightarrow_{p} \exists x. A', c \sqcup \exists x, d' \\
\text{R5} & \quad \{p(y), c\} \rightarrow_1 \{A, c\} \quad p(x) : -A \in P
\end{align*}
\]

\textbf{Table 2.} The transition system for PCCP.

The operational semantics of PCCP has a simple definition in terms of a transition system, \((\text{Con} f, \rightarrow_p)\), where \(\text{Con} f\) is the set of configurations \(\langle A, d \rangle\) representing the state of the system at a certain moment, namely the agent \(A\) which has still to be executed and the common store \(d\); and the transition relation \(\rightarrow_p\) is defined in Table 2. The rules given are closely related to the ones for (monotonic) CCP, and we refer to \([11]\) for a detailed description.

Rule \textbf{R1} describes the effect of \text{tell}(c): this agent always terminates successfully with probability one, and the new store is the least upper bound of the constraint \(c\) and the (current) store \(d\), i.e. \(c \sqcup d\). Note that we use the agent \text{stop} to mark successful termination in contrast to other agents which might get \text{stuck} (e.g. because no guard is enabled).

Both rules \textbf{R2} and \textbf{R3} refer to normalised quantities \(\tilde{p}_i\). These are the result of the normalisation process as described informally in the previous section. More precisely, this process can be defined for a generic set of real numbers \(X = \{x_i\}\), as the process of replacing each \(x_i\) by \(\tilde{x}_i\) as follows:
Definition 1. Let \( x = \sum x_i \), and \( n \) the cardinality of \( X = \{x_i\} \). Then

\[
\tilde{x}_i = \begin{cases} \\
\frac{x_i}{x} & \text{if } x \neq 0 \\
\frac{1}{n} & \text{otherwise}
\end{cases}
\]

Rule R2 describes probabilistic choice. An agent \( A_i \) is called enabled iff its guard \( c_i \) is entailed by the store, i.e. \( d \vdash c_i \). The normalisation process described above is applied only to the \( p_i \)'s associated to enabled agents, and the choice is then among the enabled agents.

The following is an example of the effect of the normalisation process in the context of a probabilistic choice:

\[
\begin{align*}
\text{ask}(c) & \rightarrow 2 : A \\
\text{ask}(d) & \rightarrow 1 : B \\
\text{true} & \rightarrow 0 : C
\end{align*}
\]

In case both \( c \) and \( d \) are entailed the normalisation will assign the probability \( \frac{2}{3} \) to \( A \) and \( \frac{1}{3} \) to \( B \) and 0 to \( C \). If either \( c \) or \( d \) is enabled then the corresponding agent is executed with probability 1, while \( C \) still has probability 0. If neither \( c \) nor \( d \) is entailed by the store, then \( C \) will be executed with probability 1.

Rule R3 describes a prioritised interleaving: Each time the scheduler has to select an agent to be executed, it will choose according to the probabilities \( p_i \) among active agents. An agent \( A \) is called active if it can make a transition, i.e. there exists \( \langle A', d' \rangle \in Conf \) such that \( \langle A, d \rangle \xrightarrow{p} \langle A', d' \rangle \). Note that there is no rule for a transition from the stop agent to any other agent, i.e. stop is never active. The normalisation process described above is applied to the \( p_i \)'s associated to active agents.

The following example illustrates the normalisation in the context of the parallel construct:

\[
\frac{2}{3} : A \parallel \frac{1}{3} : B
\]

The scheduler will select agent \( A \) with probability \( \frac{2}{3} \) each time it has to make a selection; while the probability for \( B \) being selected is only \( \frac{1}{3} \). That means in practice that during the interleaving process \( A \) will get the CPU slot twice as often than \( B \). We can rephrase this by saying that \( A \) will run twice as fast as \( B \). In this sense the probabilities \( p_i \) actually represent priorities.

We can use this prioritised interleaving to implement a kind of sequentiality by assigning priority 1 to the agent to be executed first and priority 0 to the agent to be executed last. Thus, for example, the agent

\[
1 : A \parallel 0 : B
\]

represents the sequential agent \( A ; B \), which executes \( B \) after \( A \) has been completed. A sequential composition of more than two agents can be implemented in an analogous way by using the appropriate bracketing.
Example 2. The following example can be useful to clarify the operational semantics for the probabilistic choice and the prioritised parallel operator. Consider the two agents $A$ and $B$ defined as

$$A \equiv \text{ask}(c) \rightarrow \frac{3}{4} : \text{tell}(e) \text{[true]} \rightarrow \frac{1}{4} : \text{tell}(d)$$

$$B \equiv \text{tell}(c),$$

and their parallel execution $\frac{1}{4} : A || \frac{3}{4} : B$ starting with the “empty” store $\text{true}$.

The execution of the agent $A$ depends on the current store: if $c$ is entailed, then $A$ has a choice between executing $\text{tell}(e)$ and $\text{tell}(d)$, otherwise only $\text{tell}(d)$ is enabled. Therefore, if $A$ is executed with store $\text{true}$ its behaviour is deterministic: only one branch is enabled and the associated transition probability becomes $\tilde{p} = 1$ after renormalisation, i.e. we get

$$\langle A, \text{true} \rangle \rightarrow_1 \langle \text{stop}, d \rangle.$$ 

If $A$ is executed with store $c$, both branches are enabled and therefore we have two possible transitions (the probabilities stay unchanged as they are normalised already):

$$\langle A, c \rangle \rightarrow_2 \langle \text{stop}, c \sqcup e \rangle,$$

and

$$\langle A, c \rangle \rightarrow_2 \langle \text{stop}, c \sqcup d \rangle.$$ 

The behaviour of $B$ is much simpler: it will always (deterministically) add $c$ to the current store. So, in the store $\text{true}$ the execution of $B$ will result in the transition:

$$\langle B, \text{true} \rangle \rightarrow_1 \langle \text{stop}, c \rangle.$$ 

and in any other store $d$, it will simply add $c$ to it:

$$\langle B, d \rangle \rightarrow_1 \langle \text{stop}, d \sqcup c \rangle.$$ 

Note that in general any parallel combination of an agent $C$ with $\text{stop}$ is equivalent to just $C$ because (as mentioned above) $\text{stop}$ is never active. In particular, a parallel composition of several $\text{stop}$ agents corresponds to (successful) termination.

These observations give us the derivation tree of $\langle \frac{1}{4} : A || \frac{3}{4} : B, \text{true} \rangle$ as depicted in Figure 1. In this derivation there are two different types of branchings: The one at the top level reflects the selection of the scheduler for the execution of either $A$ or $B$ first; thus the left hand side of the tree corresponds to the interleaving $A; B$ whereas the right hand side depicts the interleaving $B; A$. The lower level branching on the right hand side corresponds to the probabilistic choice of the agent $A$. The representation of both the two possible interleavings in the same derivation tree allows for a more compact description of the execution of the agent in question. It should nevertheless be clear that each interleaving originates a different derivation tree.
Rule **R4** in Table 2 deals with the introduction of local variables; we use the notation $\exists_x^d A$ for the agent $A$ with local store $d$ containing information on $x$ which is hidden in the external store (see [36, 37] for further details). Obviously, the transition probability $p$ is not changed by hiding.

In the recursion rule **R5** for the procedure call $p(y)$, we assume that before $p(y)$ is replaced by the body of its definition in the program $P$, the link between the actual parameter $y$ and the formal parameter $x$ has been correctly established. In [37], this link is elegantly expressed by using the hiding operation $\exists_x$ and one only fresh variable. As this is a deterministic operation the transition probability in this rule is one.

Given two configurations $C$ and $C'$ we define a computational path leading from $C$ to $C'$ and its associated probability as follows.

**Definition 3.** A *path* $\pi$ of length $n$ between two configurations $C$ and $C'$ is a sequence of configurations $C = C_1, C_2, \ldots, C_n = C'$ such that for all $i, C_i \rightarrow p_i C_{i+1}$. We define the *probability* associated to a path $\pi$ as $\text{prob}(\pi) = \prod_{i=1}^{n} p_i$.

We will denote by $\text{Path}(C, C')$ the set of all paths between two configurations $C$ and $C'$.

We now define the transitive closure $\rightarrow^*_p$ of the transition relation $\rightarrow_p$ in Table 2 as follows.

**Definition 4.** Let $C, C' \in \text{Conf}$. Then $C \rightarrow^*_p C'$ iff $\text{Path}(C, C') \neq \emptyset$ and $p = \sum_{\pi \in \text{Path}(C, C')} \text{prob}(\pi)$.

### 2.4 Results and Observables

The basic notion of *observables*, as in [13], captures the *exact* results of both finite and infinite computations together with their associated probabilities.
Given a PCCP program, we define the result $\mathcal{R}_P$ of an agent $A$ and an initial store $d$ as the (multi-)set of all pairs $\langle c, p \rangle$, where $c$ is the least upper bound of the partial constraints accumulated during a computation starting from $d$, and $p$ is the probability of reaching that result.

$$\mathcal{R}_P(A, d) = \{ \langle c, p \rangle \mid \langle A, d \rangle \rightarrow^* \langle B, c \rangle \} \cup \{ \langle \prod_i d_i, \prod_i p_i \rangle \mid \langle A, d_0 \rangle \rightarrow_p \ldots \}. $$

The first term describes the results of finite computations, where the least upper bound of the partial stores corresponds to the final store. Note that this includes both successful termination — i.e. when $B = \text{stop}$ — and suspended computations — i.e. the agent in the final configuration is not the stop agent and is unable to make a transition. The second term covers the infinite results.

Because of non-determinism, there might be different computational paths leading to the same result. Thus, we need to `compactify’ the results so as to identify all those pairs with the same constraint as a first component. This operation is formally defined as follows.

**Definition 5.** Let $S = \{ \langle c_{ij}, p_{ij} \rangle \}_{i,j}$ be a (multi-)set of results, where $c_{ij}$ denote the $j$th occurrence of the constraint $c_i$, and let $P_{c_i}$ be the sum of all probabilities occurring in the set associated with $c_i$. The compactification of $S$ is defined as follows, where the notation $P_{c_i} = \sum_{c_i} p_{c_i}$ denotes the sum of all the probabilities associated with $c_i$ (in $S$):

$$\mathcal{K}(S) = \{ \langle c_i, P_{c_i} \rangle \mid P_{c_i} = \sum_{c_i} p_{c_i} = \sum_{j} p_{ij} \}_{i}. $$

We observe that this operation may not always result in a probability distribution when infinite computations are involved. In particular, this may happen when the derivation tree has infinitely many infinite branches. This case needs a more complicated, measure-theoretical treatment which we will not develop here for lack of space.

Now we can define the observables associated to an agent $A$ and an initial store $d$ as:

$$\mathcal{O}_P(A, d) = \mathcal{K}(\mathcal{R}_P(A, d)).$$

Note that this notion of observables differs from the classical notion of input/output behaviour in CCP. In the classical case a constraint $c$ belongs to the input/output observables of a given agent $A$ if at least one path leads from the initial store $d$ to the final result $c$. In the probabilistic case we have to consider all possible paths leading to the same result $c$ and combine the associated probabilities.

**Example 6.** In the example given above, we can immediately see from the derivation in Figure 1 that the probabilistic result is given by:

$$\mathcal{R}_P \left( \frac{1}{3} : A \parallel \frac{2}{3} : B, \text{true} \right) = \{ \langle c \cup d, \frac{1}{3} \rangle, \langle c \cup e, \frac{1}{2} \rangle, \langle d \cup c, \frac{1}{6} \rangle \}. $$
Compactification and the simple fact that $d \cup c = c \cup d$ finally give us the probabilistic observables as:

$$O_p \left( \frac{1}{3}: A \parallel \frac{2}{3}: B, \text{true} \right) = \left\{ \left\langle c \cup d, \frac{1}{2} \right\rangle, \left\langle c \cup e, \frac{1}{2} \right\rangle \right\}.$$

3 Quantitative Observables

3.1 Cost Function and Properties

In the context of this paper we are interested in modelling properties which are related to the cost of a computation. By "cost" of a computation we mean a measure for any kind of resource which can be used and consumed during the computation, such as time, memory, etc.

In general, we will represent such a cost by means of a random variable:

**Definition 7.** A cost function $Q$ is a random variable on the constraint system $C$, i.e. a map $Q : C \rightarrow \mathbb{R}$ from the set of constraints into the reals.

The function $Q$ can be static, i.e. it is fixed throughout the computation, or dynamic, i.e. it depends on the computational history. A dynamic cost function can then be represented as the limit of a sequence of cost functions, each of which is associated to a particular stage of the computation.

Such cost functions allow us to calculate the resource consumption associated to each individual result of the execution of a program, as well as to calculate the average properties of the program. We distinguish between two types of average properties: One captures the average cost of the final results, and is modelled by the expected value, $E(Q)$, of the random variable $Q$ expressing that cost; the other one is related to the long-run behaviour of a program and captures the average cost, $A(Q)$, of a single computation in the limit. We will refer to the first type as average properties and to the second one as long-run average properties.

3.2 Augmented Semantics

For dealing with dynamical cost functions we have to provide additional information about how the cost evolves during the computation. For this purpose we extend the transition system for PCCP given in Table 2 by associating to each transition step its computational cost defined by an appropriate cost function $Q$. The transition relation $\rightarrow^s_q$ expresses the fact that a transition between two configurations takes place with probability $p$ and has a computational cost $q$, and is defined in Table 3, which gives a general definition scheme for such augmented transition systems.

This leads to a new notion of results which extends $R_P$ by giving for each computed constraint the information about its computational cost. In order to define this notion of quantitative results, it will be necessary — as we did for the probabilities — to appropriately collect the partial costs along a computation.
\begin{align*}
\text{R.1} & \quad \langle \text{tell}(c), d \rangle \rightarrow \langle \text{stop}, c \cup d \rangle \\
\text{R.2} & \quad \bigoplus_{i=1}^{n} \langle \text{ask}(c_i) \rightarrow \pi_i : A_i, d \rangle \rightarrow_{\mathcal{R}} \langle A_j, d \rangle \quad j \in [1, n] \text{ and } d \vdash c_j \\
\text{R.3} & \quad \langle A_i, c \rangle \rightarrow_{\mathcal{R}} \langle A'_i, c' \rangle \\
\text{R.4} & \quad \langle A, d \cup \exists, c \rangle \rightarrow_{\mathcal{R}} \langle A', d' \rangle \\
\text{R.5} & \quad \langle p(y), c \rangle \rightarrow_{\mathcal{R}} \langle A, c \rangle \quad p(x) : -A \in P
\end{align*}

Table 3. A generic augmented transition system for PCCP.

Typically, we will have to sum up the costs for each transition along a computational path (e.g., execution time), but in some cases it might be appropriate to use a different method (e.g., memory usage might be dealt best by looking at the maximum instead). Therefore, in the following we will use a generic \textit{collection} method \( \Gamma \).

Due to non-determinism the same configuration may be reached via different paths. We therefore need a second method to combine the costs associated to the different paths in an appropriate way. A very common choice is to sum up all the costs. However, depending on the interpretation of the quantity in question, other choices might be equally reasonable. Thus, we leave the \textit{compactification} method \( \Omega \) unspecified as it depends on the specific application.

Given a PCCP program, we define the quantitative results \( \mathcal{R}(A, d) \) of an agent \( A \) and an initial store \( d \) as the set of all pairs \( \{\langle c, p, q \rangle\} \), where \( c \) is the least upper bound of the partial costs associated during the computation starting from \( d \); \( p \) is the probability of reaching that result; \( q \) is the associated cost.

\[
\mathcal{R}(A, d) = \left\{ \langle c, p, q \rangle \mid \langle A, d \rangle \xrightarrow{\mathcal{R}} \langle B, c \rangle \not\xrightarrow{\mathcal{R}} \right\} \cup \\
\left\{ \left( \bigoplus_{i=1}^{n} \pi_i : \prod_i \pi_i : A_i, d_i \right) \longrightarrow_{\mathcal{R}} \langle A, d_c \rangle \right\}.
\]

The symbol \( \xrightarrow{\mathcal{R}} \) indicates the transitive closure of the transition relation \( \longrightarrow_{\mathcal{R}} \), and is defined in a similar way as the transitive closure of \( \longrightarrow_p \) in Definition 4 by applying the appropriate methods \( \Gamma \) and \( \Omega \). A computational path in the new transition relation and its associated cost and probability are the obvious extension of Definition 3.

\textbf{Definition 8.} Let \( C, C' \in \text{Conf} \). Then \( C \xrightarrow{\mathcal{R}} C' \) iff \( \text{APath}(C, C') \neq \emptyset \) and \( p = \sum_{\pi \in \text{APath}(C, C')} \text{prob}(\pi) \) and \( q = \sum_{\pi \in \text{APath}(C, C')} \text{cost}(\pi) \).
Note that in the set $\mathcal{R}(A, d)$ tuples might occur which contain the same constraint, i.e. the same qualitative information. We can identify all such tuples by combining the associated costs in some appropriate way.

The *quantitative observables* based on the quantitative results $\mathcal{R}(A, d)$ are then defined by:

$$\mathcal{O}(A, d) = \left\{ \langle c, p, q \rangle \mid p = \sum_c p_c, q = \bigwedge_i q_i \right\}$$

where $\Sigma$ and $\Omega$ is over all tuples in $\mathcal{R}(A, d)$ with the same constraint $c$.

4 Observing the Average

In this section we use the notion of quantitative observables defined in Section 3 in order to formalise various properties which reflect some “average behaviour” of a program. This formalisation can be used as a base for performance modelling and evaluation purposes, as well as for an average case analysis of PCCP programs.

In the following we will define and investigate two different types of average properties. One of them is based on the expectation value of the cost function, while the other refers to the time average. We call the first type of properties *average properties*, and the second one *long-run average properties*.

4.1 Expectation

An obvious way to analyse the average behaviour of a program is to consider the average cost of its (compactified) final results.

$$E = \sum_{\langle c, p, q \rangle \in \mathcal{O}(A, d)} p \cdot q.$$ 

Note that in the above expression, $q$ is the value of the discrete random variable $Q$ on the constraint system defined via the augmented transition system by

$$Q(c) = q \text{ iff } \langle c, p, q \rangle \in \mathcal{O}(A, d).$$

On the other hand, the probability information contained in the quantitative observables can be seen as a discrete random distribution on the constraint system $P : C \rightarrow [0, 1]$, i.e. a map into the real interval $[0, 1]$, defined by

$$P(c) = p \text{ iff } \langle c, p, q \rangle \in \mathcal{O}(A, d),$$

with $\sum_{c \in C} P(c) = 1$.

Therefore, the quantity $E$ defined above corresponds exactly to the *expectation value* or *mean* of $Q$ with respect to $P$:

$$E = E(Q, P) = \sum_{c \in C} q \cdot \text{Prob}(Q(c) = q),$$

where $\text{Prob}(Q(c) = q)$ is the probability distribution given by $P$, and so the probability $p$ associated to the constraint $c$ in the observables $\mathcal{O}(A, d)$. 
4.2 Time Average

An alternative notion of average behaviour is obtained by considering the average cost along a single computation of an agent $A$ and an initial store $d$,

$$\langle A, d_0 \rangle \rightarrow_{p_0} \langle A_1, d_1 \rangle \rightarrow_{p_1} \ldots$$

This leads to the definition of a long-run average property as:

$$A = A(Q, d) = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} E(Q_i, P_i),$$

where $P_i$ and $Q_i$ denote the probability distribution on the partial results and the cost function at the $i$th computational step, respectively. In the case of a static cost function $Q$, where for all $i$, $Q_i = Q$, the above expression becomes simply:

$$A = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} E(Q, P_i).$$

Note that the relevance of this notion is only to infinite computations; for terminating computations this average would always correspond to the probability distribution of the final configuration.

Example 9. The concept of long-run average can be illustrated by a simple example inspired by the following legend:

Once there was a man who had two sons and a precious diamond ring. On his deathbed he wanted to be as fair as possible with their sons, and so decided that the ring had to be owned by the two brothers for one year each. The argument of the father then was that each of the brothers would own “half the ring” without having to destroy it.

The above scenery can be described by the following (deterministic) PCCP program:

```
father(x) : − sonA(x)
sonA(x) : − 1 : tell(y = [a|x]) || 0 : sonB(y)
sonB(x) : − 1 : tell(y = [b|x]) || 0 : sonA(y)
father([])
```

First, the ring is given to son A, who marks his ownership by adding an ‘a’ to the history list and (after one year) passes the ring to son B who proceeds in a similar way.

1 Arguments like this are not just children’s stories but were seriously discussed by mathematicians of the 18th century when the problem of defining limits and results of infinite series was still unsettled [21].
We can now ask for the average ownership of the two brothers. In order to do this we introduce two static cost functions:

\[
Q_{\text{sonA}}(c) = \begin{cases} 
1 & \text{if } c + y = [a|z] \\
0 & \text{otherwise}
\end{cases}
\]

\[
Q_{\text{sonB}}(c) = \begin{cases} 
1 & \text{if } c + y = [b|z] \\
0 & \text{otherwise}
\end{cases}
\]

i.e. both brothers own the ring respectively, when the store implies that the local history list \( y \) starts with their marking.

It is easy to see that the time average for both cost functions is \( \frac{1}{2} \), i.e. with

\[
A_n(Q_{\text{sonX}}, c_0) = \frac{1}{n} \sum_{i=1}^{n} Q_{\text{sonX}}(c_n)
\]

we get

\[
A_n(Q_{\text{sonA}}, \text{true}) = \begin{cases} 
\frac{1}{n} & \text{if } n = 2k \\
\frac{1}{2k+1} & \text{if } n = 2k + 1
\end{cases}
\]

and similarly for son B. Therefore, we get for both sons the long-run average:

\[
A(Q_{\text{sonX}}, \text{true}) = \lim_{n \to \infty} A_n(Q_{\text{sonX}}, \text{true}) = \frac{1}{2}
\]

Some questions arise naturally, which are related to the types of averages discussed above: Does the time average exist? What is the relation between expectation value and long-run average? How do averages depend on the initial configuration or store? A detailed analysis of these questions can be found in [15], assuming a denotational semantics based framework.

5 Examples

In order to illustrate our methodology we present in this section some examples showing how properties can be defined by means of a suitably defined cost function and corresponding augmented transition system. The first property we consider is the simplest one, namely the input/output behaviour. We show how both the classical and the probabilistic I/O observables can be formulated by means of an augmented transition system. We then consider an important property which is at the basis of many programs’ analyses, namely the property of termination. Finally, we propose two different ways to measure the complexity (computational length) of a program and we discuss each of them after having defined the corresponding augmented transition systems.

5.1 I/O Observables

**Classical I/O** The notion of classical observables describing the input/output behaviour of non-deterministic CCP programs can be reconstructed from the
notion of quantitative observables ignoring the information on the probability and cost of a result. So, from the quantitative results \( R \) we can obtain a (multi-)set of constraints by replacing \( \langle c, p, q \rangle \) by \( c \).

Another notion of I/O observables can be defined by ignoring the probabilities of the results and keeping only the information about their costs. This is obtained by replacing in \( R \) each tuple \( \langle c, p, q \rangle \) by the pair \( \langle c, q \rangle \), where \( q \) is the cost constructed along each computational path by using some collection operation. Of course, this property is not the most appropriate one to look at, if we want to compute any kind of (probability) weighted sum of costs, (i.e. the expectation value), as we would need it for an average case analysis. However, we can still use such observables for a worst-case analysis by looking at the smallest or largest cost of a result \( c \). In this case the compactification operator to be considered would be \( \Omega = \max \) or \( \Omega = \min \).

**Probabilistic I/O** The standard PCCP observables defined in Section 2 can be obtained by simply ignoring the costs in the quantitative observables, that is by replacing \( \langle c, p, q \rangle \) by \( \langle c, p \rangle \).

These observables can still be interpreted as quantitative observables by considering the static cost function \( Q(c) = 0 \), for all \( c \in \mathcal{C} \), together with \( \Gamma = \Sigma \) and \( \Omega = \Sigma \).

Another way to reconstruct the standard observables of a PCCP program as some kind of quantitative observable is the following: Take an augmented transition system where the labels \( p \) and \( q \) are the same. If we then use for the collection \( \Gamma = \Pi \) and for the compactification \( \Omega = \Sigma \) the tuples in the resulting quantitative observables are all of the form \( \langle c, p, p \rangle \), i.e. cost and probability are identical.

### 5.2 Termination

Program termination analysis is a fundamental one in the systems’ development. The common notion of termination considers a program as terminating if and only if all its computational paths are finite. However, in many situation this notion might be ‘too strong’, and a probabilistic formulation of that property could allow for a more realistic analysis of the system. Such a probabilistic formulation would tolerate the occurrence of an infinite computation provided that its probability of actually taking place is zero.

In this section we define a notion of quantitative observables by means of which the property of termination can be formalised in both its classical and probabilistic version. To this purpose we introduce in Table 4 an augmented transition system where the cost function \( Q \) associates to each constraint the number of transition steps required to compute that constraint.

The quantitative observables, \( \mathcal{O}_f \), we are interested in for analysing termination are then defined by using summation for the cost collection along a computational path, i.e. \( \Gamma = \Sigma \); as for the compactification of the costs in the final results we take the maximum of the costs associated to each computational
path leading to the same constraint, i.e. $\Omega = \text{max}$. Alternatively, we could use a weighted sum for $\Omega$. However, as our interest is only in distinguishing if a computation is finite or infinite, $\Omega = \text{max}$ suffices for our purposes.

Based on the resulting observables $\mathcal{O}_T(A, d)$ we now define two notions of termination: One, which we call strong termination, corresponds to the commonly assumed idea of a program with no infinite computations; the other one, which we call probabilistic termination, considers a program as terminating if the probability associated to its infinite computations is zero.

**Definition 10.** An agent $A$ is called strongly terminating if all its computational paths starting from any given initial store $d$ are finite, i.e. for all $\langle c, p, q \rangle \in \mathcal{O}_T(A, d)$ we have that $q < \infty$.

According to this definition programs with only extremely unlikely infinite paths are classified as “non-terminating”. It would be more realistic to ignore possible infinite paths if their probability vanishes. This justifies the following definition.

**Definition 11.** An agent $A$ (executed from the initial store $d$) is called probabilistically terminating if the probability of an infinite path is zero, i.e. for all $\langle c, p, \infty \rangle \in \mathcal{O}_T(A, d)$ the probability $p = 0$.

The following example can be useful to clarify the difference between the two notions of termination defined above.

**Example 12 (Gambler’s Ruin).** Consider the following PCCP procedure which implements a so called “Random Walk in one Dimension” illustrating what is also known as “Gambler’s Ruin”:

\[
\begin{align*}
\text{walk}(x, y) : & \quad \neg \text{ask}(x \neq y) \rightarrow \text{1 : walk}(x + 1, y) \\
& \quad \text{ask}(x \neq y) \rightarrow \text{1 : walk}(x, y + 1) \\
& \quad \text{ask}(x = y) \rightarrow \text{1 : stop}
\end{align*}
\]
Let \( x \) be the number of won games (or number of pounds won) and let \( y \) be the number of lost games (or number of pounds lost). Then we can interpret \( walk(1, 0) \) as meaning that the game starts with a one pound stake and is over when all money is lost.

Elementary results from probability theory show that the game will terminate with a ruined gambler with probability 1, despite the fact that there exists the possibility of (infinitely many) infinite paths, (i.e. enormously rich gamblers).

Although the observables \( O_T(walk(1, 0), true) \) include infinite computations (corresponding to infinite random walks), the sum of the probabilities associated to all finite paths (i.e. random walks which “return” to store \( x = y \)) is one, e.g. [22, 23]. Thus, the probability of (all) infinite paths must be zero. As a consequence, this program, which classically does not terminate, does terminate in a probabilistic sense: If one continues playing, most certainly he will ultimately lose everything.

As another example we consider a simple program generating all natural numbers and discuss its properties related to I/O behaviour and termination.

**Example 13 (Randomised Counting).** Consider the following PCCP program:

\[
\text{nat}(x) : - \text{true} \rightarrow \frac{1}{2} : \text{tell}(x = 0) \\
\quad \text{true} \rightarrow \frac{1}{2} : \exists_y (1 : \text{tell}(x = s(y)) \parallel 0 : \text{nat}(y))
\]

The standard CCP I/O behaviour of this program abstracts from any consideration about the probabilities and the costs associated to each result computed by \( \text{nat}(x) \) (i.e. the natural numbers), so the classical observables corresponds to the set [11]:

\[
O_C(\text{nat}, \text{true}) = \{x = 0, x = s(0), x = s(s(0)), \ldots, x = s^n(0), \ldots\}.
\]

We can now be a little bit ‘more concrete’ and consider the information on the transition probability provided by the augmented semantics. We then get the probabilistic observables:

\[
O_P(\text{nat}, \text{true}) = \{\langle x = 0, 1/2 \rangle, \langle x = s(0), 1/4 \rangle, \ldots, \langle x = s^{-1}(0), 1/2^n \rangle, \ldots\}.
\]

from which we can also see the likelihood that a given number is actually generated.

A further step will lead us to observe the number of calls to the procedure \( \text{nat}(x) \) that are necessary to generate a given number. We have then the ‘most informative’ semantics:

\[
O_T(\text{nat}, \text{true}) = \{\langle x = 0, 1/2, 0 \rangle, \langle x = s(0), 1/4, 1 \rangle, \ldots, \langle x = s^{-1}(0), 1/2^n, n - 1 \rangle, \ldots\},
\]

where, according to our intuition, we can see that when \( n \) tends to infinity the probability vanishes, whereas the cost grows bigger and bigger.

Although this program does not terminate in the classical sense (there exists an infinite computational path), it is probabilistically terminating as the probability of a path with infinite length is zero.
5.3 Recursion Depth

A crude measure for the running time of an agent is the recursion depth. The corresponding cost function can be defined by associating a non-zero cost to the procedure call as in Table 5.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>( (\text{tell}(c), d) \rightarrow_1 \langle \text{stop}, c \cup d \rangle )</td>
</tr>
<tr>
<td>R2</td>
<td>( \langle \bigwedge_{i=1}^n \text{ask}(c_i) \rightarrow p_k : A_i, d \rangle \rightarrow_0 \langle A_j, d \rangle \quad j \in [1, n] ) and ( d \vdash c_j )</td>
</tr>
<tr>
<td>R3</td>
<td>( \langle A_j, c \rangle \rightarrow_0 \langle A'_j, c' \rangle \rightarrow_0^{\pi} \langle A'_j, c' \rangle \quad j \in [1, n] )</td>
</tr>
<tr>
<td>R4</td>
<td>( \langle A, d \cup \exists_c c \rangle \rightarrow_0^{\pi} \langle A', d' \rangle \rightarrow_0^{\pi} \exists_c A', c \cup \exists_c d' )</td>
</tr>
<tr>
<td>R5</td>
<td>( (p(y), c) \rightarrow_0^{\pi} \langle A, c \rangle \quad p(x) : -A \in P )</td>
</tr>
</tbody>
</table>

**Table 5.** Augmented transition system for recursion depth.

The notion of observables, \( O_R(A, d) \), we obtain are defined by means of a collection method \( \Gamma \), which is again just summing up, while \( \Omega \) is the weighted or average sum:

\[
\sum_{c_i} q_i = \frac{\sum_{c_i} p_i \cdot q_i}{\sum_{c_i} p_i},
\]

i.e. the *partial cost* of a constraint \( c_i \) is the weighted sum of the cost collected over all possible paths leading to the same result or store containing \( c_i \) divided by the sum of the probabilities to reach that result. This is necessary in order to renormalise the probabilities appropriately. We can then use the resulting observables \( O_R(A, d) \) to compute the average recursion depth.

Note that this average recursion depth, could be computed perhaps in an easier way if we used the quantitative result \( R_R(A, d) \) instead of the quantitative observables \( O_R(A, d) \). The average length of all paths, taking into account that there might be several ways leading to the same result is obviously:

\[
\sum_{\langle c_i, p_i, q_i \rangle \in R_R(A, d)} p_i \cdot q_i
\]

which, if we combine the partial costs leading to the same store \( c_i \):

\[
\sum_{c_i} p_i \cdot q_i = \sum_{c_i} \frac{p_i}{\sum_{c_i} p_i} \cdot \frac{p_i}{\sum_{c_i} p_i} \cdot q_i
\]
allows us to compute the average based on the observables as:

\[ \sum_{c_i \in \mathcal{C}} \sum_{c_i} p_i \cdot \Omega_{q_i} q_i = \sum_{(c_i, p_i, q_i) \in \mathcal{O}_R(A, d)} p_i \cdot q_i \]

**Example 14 (Randomised Counting, revisited).** Consider again the PCCP program from Example 13:

\[
\text{nat}(x) : = \text{true} \rightarrow \frac{1}{2} : \text{tell}(x = 0) \\
\underline{\text{true} \rightarrow \frac{1}{2} : \exists y \text{tell}(x = s(y)) \parallel \text{true} : \text{tell}(y = y))}
\]

The observables \(\mathcal{O}_R(\text{nat}, \text{true})\) coincidently coincides with \(\mathcal{O}_T(\text{nat}, \text{true})\):

\[ \mathcal{O}_R(\text{nat}, \text{true}) = \{ (x = 0, 1/2, 0), (x = s(0), 1/4, 1), \ldots \]
\[ \ldots, (x = s^n 1(0), 1/2^n, n-1), \ldots \}. \]

Based on \(\mathcal{O}_R(\text{nat}, \text{true})\) we can estimate the number of calls we can expect on average by executing the program \(\text{nat}(x)\) starting with store \text{true}:

\[ E(\text{nat}, \text{true}) = \sum_{q = 0}^{\infty} \frac{q}{2^q} \]

From elementary discrete mathematics theory we know that this sum is finite and equal to 2 [20]. That is, this program, though classically not even terminating, has an average running time of just two iterations.

**Example 15 (Binary List Search).** The following is a program which searches a given list for an element in the list. In order to simplify the presentation we assume a list of length \(2^k\) for some \(k\).

\[
\text{search}(x, \text{list}, n) : = \text{ask}(x = \text{list}[n/2]) \rightarrow 1 : \text{tell}(i = n/2) \\
\underline{\text{ask}(x < \text{list}[n/2]) \rightarrow 1 : \text{search}(x, \text{list}[1 : n/2], n/2)} \\
\underline{\text{ask}(x > \text{list}[n/2]) \rightarrow 1 : \text{search}(x, \text{list}[n/2 + 1 : n], n/2)}
\]

As the guards of the three alternatives are mutually exclusive, this program is deterministic, despite its formulation as a (probabilistic) nondeterministic choice. Therefore, the notions of classical and probabilistic observables are quite trivial in this case. In fact, any call to \text{search} will always terminate after a finite number of steps, once \(x\) is found by establishing \(i\) as the “index” of \(x\) in the \text{list} with probability 1.

A slightly more interesting observable is \(\mathcal{O}_R(\text{search})\), which for a given input \(x\) gives the number of recursive calls needed to find \(x\). This is fixed once \(x\) is fixed. Thus the average behaviour of such a program can be estimated by comparing runs of the program on different inputs.

As an example, consider the case \(n = 4\). Then we have four possible calls \(\text{search}(x, [1, 2, 3, 4], 4)\) with \(x = 1, x = 2, x = 3\) and \(x = 4\). It is easy to verify that they take respectively 2, 1, 2 and 3 steps (or recursive calls).
Assuming a uniform distribution on the input values \( x \), we can now compute the average running time of \( \text{search} \) by

\[
E(\text{search}(x, [1, 2, 3, 4], \text{true})) = \frac{2 + 1 + 2 + 3}{4} = 2.
\]

We can obtain the same result by considering the corresponding probabilistic program

\[
\text{searchall}(\text{list}, n) : \quad - \text{true} \rightarrow \frac{1}{4} : \text{search}(1, \text{list}, n) \\
\text{true} \rightarrow \frac{1}{4} : \text{search}(2, \text{list}, n) \\
\text{true} \rightarrow \frac{1}{4} : \text{search}(3, \text{list}, n) \\
\text{true} \rightarrow \frac{1}{4} : \text{search}(4, \text{list}, n)
\]

It is easy to check that the observables of \( \text{searchall} \),

\[
\mathcal{O}_R(\text{searchall}([1, 2, 3, 4], \text{true})) = \{ \langle i = 1, 1/4, 2 \rangle, \langle i = 2, 1/4, 1 \rangle, \langle i = 3, 1/4, 2 \rangle, \langle i = 4, 1/4, 3 \rangle \},
\]

can be obtained from the observables of each call to \( \text{search} \) as:

\[
\mathcal{O}_R(\text{searchall}([1, 2, 3, 4], \text{true})) = \frac{1}{4} \bigcup_{i=1}^{4} \mathcal{O}_R(\text{search}(i, [1, 2, 3, 4], \text{true})),
\]

and its average running time is again:

\[
E(\text{searchall}, \text{true}) = \sum \{ p \cdot q \mid (c, p, q) \in \mathcal{O}_R(\text{searchall}, \text{true}) \} = 2.
\]

We can generalise this result to \( n = 2^k \) by asserting that the program \( \text{search} \) has an average running time of \( q = k = \log n \). Note that this is the asymptotic complexity resulting from the average-case analysis of binary search algorithms (cf. §6.2.1 of [29]).

### 5.4 Entailment

A more realistic measure of the cost of a computation should probably consider \textbf{ask} operator as a major contributor. The complexity of this operation depends on the concrete application, i.e. on the actual constraint system. It is nevertheless clear that the cost of a choice construct depends on the (number of) guards to be checked, and on complexity of checking each guard, which we can express in all generality as follows.

Let \( C(d, c) \) be the cost of asking if \( c \) is entailed by the store \( d \). There are several special cases which are of interest:

- \( C(d, c) = C(c) \) i.e. the cost depends only on the complexity of the constraint to be checked for entailment.
- \( C(d, c) = C(d) \) i.e. the cost depends only on the complexity of the store.
- \( C(d, c) = \text{cost} \) i.e. the checking cost is independent of both. This means we are only counting how often we ask the store.

Side effects (e.g. using information from previous checks) may complicate the situation even further but they are extremely implementation specific, and it is thus impossible to deal with them in general.

An augmented transition system which registers the number of asks is given in Table 6. Here we assume that each test requires the same, constant effort \( C(d, c) = 1 \). It is only in rule R2 - the choice construct — where asks may appear, therefore it is the only transitions which have a non-zero cost \( q \).

\[
\begin{align*}
\text{R1} & (\text{tell}(c), d) \rightarrow_i \langle \text{stop}, c \cup d \rangle \\
\text{R2} & (\bigwedge_{i=1}^n \text{ask}(c_i) \rightarrow p_i : A_i, d) \rightarrow_{p_j} \langle A_j, d \rangle \quad j \in [1, n] \text{ and } d \vdash c_j \\
\text{R3} & (\bigwedge_{i=1}^n p_i : A_i, c) \rightarrow_{p_{i(j)}} \langle A_{i(j)}, c' \rangle \quad j \in [1, n] \\
\text{R4} & (\exists^d A, c) \rightarrow (\exists^d A', c \cup \exists d') \\
\text{R5} & p((y), c) \rightarrow_i (A, c) \quad p(x) : -A \in P
\end{align*}
\]

**Table 6.** Augmented transition system for constant cost ask.

The definition of the observables again requires an appropriate collection of the costs (of asks) along a computational path. It is reasonable in this case to sum up the costs (i.e. the asks along each computational path), that is to define \( \Gamma = \sum_i \), and to take for \( \Omega \), the above introduced partial average (weighted sum):

\[
\Omega_{ci} q_i = \frac{\sum_{ci} p_k \cdot q_i}{\sum_{ci} p_k}
\]

A more general augmented transition system which registers the costs of ask in a more general way is given in Table 7. Here the general dependency of an ask on the complexity of the guard and the current store is expressed in rule R2 again, which states that the cost of a transition of a choice agent is always the combined sum of checking all guards.

The observables are based on the same \( \Gamma \) and \( \Omega \) as for the case of a constant cost.
6 Conclusion and Future Work

In this paper we presented an operational semantics which captures the I/O results of a program with respect to their probability and cost, the latter representing (some measure of) complexity or resource consumption of a program.

The idea of a structural operational semantics where transitions are enriched by entities capturing some intended ‘cost’ (of the transition), although new in the CCP setting, was already explored within the realm of Process Algebra (see e.g. [17]) and Stochastic Process Algebra (see e.g. [4, 33, 3, 38, 9] and references therein). Within the area of (lazy) functional programs, the idea of a counting analysis as developed in [26, 35] also seems to bear some similarities to the work presented in this paper.

We also showed how average properties can be naturally expressed by means of our augmented operational semantics. This provides a base for the specification and analysis of probabilistic properties such as the ones called in [10] long-run average properties, and for an average-case analysis of the program behaviour similar to the asymptotic average-case analysis of the complexity of algorithms.

We intend to further develop these ideas as part of a more general programme aiming at the definition of an abstract interpretation framework supporting a probabilistic program analysis as well as the analysis of probabilistic programs. Such a framework should result from the incorporation of quantitative features to the well-known Cousot one [8]. It would also be interesting to compare this framework based on algorithmic randomisation with a similar approach based on “soft constraints” [6].

Moreover, our augmented operational semantics can be used for developing a methodology for quantitative data flow analysis, where it is possible to estimate — in a way similar to statistical methods — probabilities associated to the properties of programs, instead of asserting the properties in a conservative way. In
this investigation other programming paradigms, like functional and imperative programming languages, will also be considered. Not surprisingly we expect that this will profit from various approaches in Markov chain theory [39, 40] and and Markovian process algebras [27, 5, 33, 19].

Further work will be devoted to develop a denotational counterpart of the operational semantics presented in this paper, which could be used as a base semantics for a quantitative program analysis as well. To this purpose we intend to exploit the semantics introduced in a previous work by the authors [14], which considers linear spaces structures (Banach or Hilbert spaces of measurable functions) as domain of denotations, in line with earlier important contributions in the area of probabilistic semantics like the fundamental papers of Saheb-Djahromi [34], and Kozen [30].

A first result in this direction is the denotational model for average properties which refer to static quantities. These correspond to real-valued random variables whose definition is fixed when the process starts and doesn’t change during all its execution [15]. This context also allows us to incorporate results from ergodic theory in order to analyse the relation between the expectation value $E$ and the time average $A$.

References