University of Amsterdam
Programming Research Group

Instruction Sequence Notations with Probabilistic Instructions

J.A. Bergstra
C.A. Middelburg

Report PRG0906 June 2009

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

tel. +31 20 525.7591
e-mail: janb@science.uva.nl

C.A. Middelburg

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

e-mail: kmiddelb@science.uva.nl

Programming Research Group Electronic Report Series

Instruction Sequence Notations with Probabilistic Instructions

J.A. Bergstra and C.A. Middelburg

Informatics Institute, Faculty of Science, University of Amsterdam,
Science Park 107, 1098 XG Amsterdam, the Netherlands
J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl

Abstract. This paper concerns probabilistic instruction sequences. We use the term probabilistic instruction sequence for an instruction sequence that contains probabilistic instructions, i.e. instructions that are themselves probabilistic by nature, rather than an instruction sequence of which the instructions are intended to be processed in a probabilistic way. We propose several kinds of probabilistic instructions, provide an informal operational meaning for each of them, and discuss related work. On purpose, we refrain from providing an ad hoc formal meaning for the proposed kinds of instructions.

Keywords: instruction sequence, probabilistic instruction, probabilistic service, probabilistic process algebra thesis, projectionism.

1998 ACM Computing Classification: D.1.4, F.1.1, F.1.2.

1 Introduction

In this paper, we take the first step on a new subject in a line of research whose working hypothesis is that instruction sequence is a central notion of computer science (see e.g. [6, 10, 8, 7]). In this line of research, an instruction sequence under execution is considered to produce a behaviour to be controlled by some execution environment: each step performed actuates the processing of an instruction by the execution environment and a reply returned at completion of the processing determines how the behaviour proceeds. The term service is used for a component of a system that provides an execution environment for instruction sequences, and a model of systems that provide execution environments for instruction sequences is called an execution architecture. This paper is concerned with probabilistic instruction sequences.

We use the term probabilistic instruction sequence for an instruction sequence that contains probabilistic instructions, i.e. instructions that are themselves probabilistic by nature, rather than an instruction sequence of which the instructions are intended to be processed in a probabilistic way, e.g. by means of probabilistic services. We will propose several kinds of probabilistic instructions, provide an informal operational meaning for each of them, and discuss related work. We will refrain from a formal semantic analysis of the proposed kinds of probabilistic instructions. Moreover, we will not claim any form of completeness for the proposed kinds of probabilistic instructions. Other convincing kinds might be found in the future. We will leave unanalysed the topic of probabilistic instruction sequence processing, which includes all phenomena concerning services and concerning execution architectures for which probabilistic analysis is necessary.

Viewed from the perspective of machine-execution, execution of a probabilistic instruction sequence using an execution architecture without probabilistic features can only be a metaphor. Execution of a deterministic instruction sequence using an execution architecture with probabilistic features, i.e. an execution architecture that allows for probabilistic services, is far more plausible. Thus, it looks to be that probabilistic instruction sequences find their true meaning by translation into deterministic instruction sequences for execution architectures with probabilistic features. Indeed projection semantics, the approach to define the meaning of programs which was first presented in [4], need not be compromised when probabilistic instructions are taken into account.

This paper is organized as follows. First, we set out the scope of the paper (Section 2) and review the special notation and terminology used in the paper (Section 3). Next, we propose several kinds of probabilistic instructions (Sections 4 and 5). Then, we formulate a thesis on the behaviours produced by probabilistic instruction sequences under execution (Section 6) and discuss projectionism in the light of probabilistic instruction sequences (Section 7). Finally, we discuss related work (Section 8) and make some concluding remarks (Section 9).

2 On the Scope of this Paper

We go into the scope of the paper to clarify and motivate its restrictions.

We will propose several kinds of probabilistic instructions, chosen because of their superficial similarity with kinds of deterministic instructions known from PGA [4], PGLD with indirect jumps [6], C [12], and other similar notations and not because any computational intuition about them is known or assumed. For each of these kinds, we will provide an informal operational meaning. Moreover, we will show that the proposed unbounded probabilistic jump instructions can be simulated by means of bounded probabilistic test instructions and bounded deterministic jump instructions. We will also refer to related work that introduces something similar to what we call a probabilistic instruction and connect the proposed kinds of probabilistic instructions with similar features found in related work.

We will refrain from a formal semantic analysis of the proposed kinds of probabilistic instructions. The reasons for doing so are as follows:

- In the non-probabilistic case, the subject reduces to the semantics of program algebra. Although it seems obvious at first sight, different models, reflecting different levels of abstraction, can and have been distinguished (see e.g. [4]). Probabilities introduce a further ramification.

– What we consider sensible is to analyse this double ramification fully. What we consider less useful is to provide one specific collection of design decisions and working out its details as a proof of concept.
– We notice that for process algebra the ramification of semantic options after the incorporation of probabilistic features is remarkable, and even frustrating (see e.g. [16, 19]). There is no reason to expect that the situation is much simpler here.
– Once that a semantic strategy is mainly judged on its preparedness for a setting with multi-threading, the subject becomes intrinsically complex (like the preparedness for a setting with arbitrary interleaving complicates the semantic modelling of deterministic processes in process algebra).
– We believe that a choice for a catalogue of kinds of probabilistic instructions can be made beforehand. Even if that choice will turn out to be wrong, because prolonged forthcoming semantic analysis may give rise to new, more natural, kinds of probabilistic instructions, it can at this stage best be driven by direct intuitions.

We will leave unanalysed the topic of probabilistic instruction sequence processing, i.e. probabilistic processing of instruction sequences, which includes all phenomena concerning services and concerning execution architectures for which probabilistic analysis is necessary. At the same time, we admit that probabilistic instruction sequence processing is a much more substantial topic than probabilistic instruction sequences, because of its machine-oriented scope. We take the line that a probabilistic instruction sequence finds its operational meaning by translation into a deterministic instruction sequence and execution using an execution architecture with probabilistic features.

3 Preliminaries

In the remainder of this paper, we will use the notation and terminology regarding instructions and instruction sequences from PGA (ProGram Algebra). The mathematical structure that we will use for quantities is a signed cancellation meadow. That is why we briefly review PGA and signed cancellation meadows in this section.

In PGA, it is assumed that a fixed but arbitrary set of *basic instructions* has been given. The *primitive instructions* of PGA are the basic instructions and in addition:

– for each basic instruction a, a *positive test instruction* $+a$;
– for each basic instruction a, a *negative test instruction* $-a$;
– for each natural number l, a *forward jump instruction* $\#l$;
– a *termination instruction* $!$.

The intuition is that the execution of a primitive instruction a produces either T or F at its completion. In the case of a positive test instruction $+a$, a is executed and execution proceeds with the next primitive instruction if T

is produced. Otherwise, the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one. If there is no next instruction to be executed, inaction occurs. In the case of a negative test instruction $-a$, the role of the value produced is reversed. In the case of a plain basic instruction a, execution always proceeds as if T is produced. The effect of a forward jump instruction $\#l$ is that execution proceeds with the l-th next instruction. If l equals 0 or the l-th next instruction does not exist, inaction occurs. The effect of the termination instruction ! is that execution terminates.

The constants of PGA are the primitive instructions and the operators of PGA are:

- the binary *concatenation* operator ; ;
- the unary *repetition* operator $^{\omega}$.

Terms are built as usual. We use infix notation for the concatenation operator and postfix notation for the repetition operator.

A closed PGA term is considered to denote a non-empty, finite or periodic infinite sequence of primitive instructions.[1] Closed PGA terms are considered equal if they denote the same instruction sequence. The axioms for instruction sequence equivalence are given in [4]. The *unfolding* equation $X^{\omega} = X \,;\, X^{\omega}$ is derivable from those equations. Moreover, each closed PGA term is derivably equal to one of the form P or $P \,;\, Q^{\omega}$, where P and Q are closed PGA terms in which the repetition operator does not occur.

In [4], PGA is extended with a *unit instruction* operator **u** which turns sequences of instructions into single instructions. The result is called $\mathrm{PGA_u}$. In [26], the meaning of $\mathrm{PGA_u}$ programs is described by a translation from $\mathrm{PGA_u}$ programs into PGA programs.

In the sequel, the following additional assumption is made: a fixed but arbitrary set of *foci* and a fixed but arbitrary set of *methods* have been given. Moreover, we will use $f.m$, where f is a focus and m is a method, as a general notation for basic instructions. In $f.m$, m is the instruction proper and f is the name of the service that is designated to process m.

The signature of signed cancellation meadows consists of the following constants and operators:

- the constants 0 and 1;
- the binary *addition* operator $+$;
- the binary *multiplication* operator $\cdot$;
- the unary *additive inverse* operator $-$;
- the unary *multiplicative inverse* operator $^{-1}$;
- the unary *signum* operator s.

Terms are build as usual. We use infix notation for the binary operators $+$ and $\cdot$, prefix notation for the unary operator $-$, and postfix notation for the unary operator $^{-1}$. We use the usual precedence convention to reduce the

[1] A periodic infinite sequence is an infinite sequence with only finitely many subsequences.

need for parentheses. We introduce subtraction and division as abbreviations: $p - q$ abbreviates $p + (-q)$ and p/q abbreviates $p \cdot (q^{-1})$. We use the notation $\underline{n}$ for numerals and the notation p^n for exponentiation with a natural number as exponent. The term $\underline{n}$ is inductively defined as follows: $\underline{0} = 0$ and $\underline{n+1} = \underline{n} + 1$. The term p^n is inductively defined as follows: $p^0 = 1$ and $p^{n+1} = p^n \cdot p$.

The constants and operators from the signature of signed cancellation meadows are adopted from rational arithmetic, which gives an appropriate intuition about these constants and operators. The equational theories of signed cancellation meadows is given in [11]. In signed cancellation meadows, the functions min and max have a simple definition (see also [11]).

A signed cancellation meadow is a cancellation meadow expanded with a signum operation. The prime example of cancellation meadows is the field of rational numbers with the multiplicative inverse operation made total by imposing that the multiplicative inverse of zero is zero, see e.g. [13].

4 Probabilistic Basic and Test Instructions

In this section, we propose several kinds of probabilistic basic and test instructions. It is assumed that a fixed but arbitrary signed cancellation meadow $\mathfrak{M}$ has been given.

We propose the following *probabilistic basic instructions*:

- $\%()$, which produces T with probability $1/2$ and F with probability $1/2$;
- $\%(q)$, which produces T with probability $\max(0, \min(1, q))$ and F with probability $1 - \max(0, \min(1, q))$, for $q \in \mathfrak{M}$.

The probabilistic basic instructions have no side-effect on a state.

The basic instruction $\%()$ can be looked upon as a shorthand for $\%(1/2)$. We distinguish between $\%()$ and $\%(1/2)$ for reason of putting the emphasis on the fact that it is not necessary to bring in a notation for quantities ranging from 0 to 1 in order to design probabilistic instructions.

Once that probabilistic basic instructions of the form $\%(q)$ are chosen, an unbounded ramification of options for the notation of quantities is opened up. We will assume that closed terms over the signature of signed cancellation meadows are used to denote quantities. Instructions such as $\%(\sqrt{1+1})$ are implicit in the form $\%(q)$, assuming that it is known how to view $\sqrt{\ }$ as a notational extension of signed cancellation meadows (see e.g. [3]).

Like all basic instructions, each probabilistic basic instruction can be turned into two *probabilistic test instructions*:

- $\%()$ can be turned into $+\%()$ and $-\%()$;
- $\%(q)$ can be turned into $+\%(q)$ and $-\%(q)$.

Probabilistic primitive instructions of the form $+\%(q)$ and $-\%(q)$ can be considered probabilistic branch instructions where q is the probability that the branch is not taken and taken, respectively, and likewise the probabilistic primitive instructions $+\%()$ and $-\%()$.

We find that the primitive instructions $\%()$ and $\%(q)$ can be replaced by $\#1$ without loss of (intuitive) meaning. Of course, in a resource aware model, $\#1$ may be much cheaper than $\%(q)$, especially if q is hard to compute. Suppose that $\%(q)$ is realized at a lower level by means of $\%()$, which is possible, and suppose that q is a computable real number. The question arises whether the expectation of the time to execute $\%(q)$ is finite.

To exemplify the possibility that $\%(q)$ is realized by means of $\%()$ in the case where q is a rational number, we look at the following probabilistic instruction sequences:

$$-\%(2/3) \,;\, \#3 \,;\, a \,;\, ! \,;\, b \,;\, ! \;,$$
$$(+\%() \,;\, \#3 \,;\, a \,;\, ! \,;\, +\%() \,;\, \#3 \,;\, b \,;\, !)^{\omega} \;.$$

It is easy to see that these instruction sequences produce on execution the same behaviour: with probability $2/3$, first a is performed and then termination follows; and with probability $1/3$, first b is performed and then termination follows. In the case of computable real numbers other than rational numbers, use must be made of a service that does duty for the tape of a Turing machine (such a service is described in [10]).

Let $q \in \mathfrak{M}$, and let $\mathsf{random}(q)$ be a service with a method get whose reply is T with probability $\max(0, \min(1, q))$ and F with probability $1 - \max(0, \min(1, q))$. Then a reasonable view on the meaning of the probabilistic primitive instructions $\%(q)$, $+\%(q)$ and $-\%(q)$ is that they are translated into the deterministic primitive instructions $\mathsf{random}(q).\mathsf{get}$, $+\mathsf{random}(q).\mathsf{get}$ and $-\mathsf{random}(q).\mathsf{get}$, respectively, and executed using an execution architecture that provides the probabilistic service $\mathsf{random}(q)$. Another option is possible here: instead of a different service $\mathsf{random}(q)$ for each $q \in [0, 1]$ and a single method get, we could have a single service random with a different method $\mathsf{get}(q)$ for each $q \in [0, 1]$. In the latter case, $\%(q)$, $+\%(q)$ and $-\%(q)$ would be translated into the deterministic primitive instructions $\mathsf{random}.\mathsf{get}(q)$, $+\mathsf{random}.\mathsf{get}(q)$ and $-\mathsf{random}.\mathsf{get}(q)$.

5 Probabilistic Jump Instructions

In this section, we propose several kinds of probabilistic jump instructions. It is assumed that the signed cancellation meadow $\mathfrak{M}$ has been expanded with an operation $\mathbb{N}$ such that, for all $q \in \mathfrak{M}$, $\mathbb{N}(q) = 0$ iff $q = \underline{n}$ for some $n \in \mathbb{N}$. We write $\bar{l}$, where $l \in \mathfrak{M}$ is such that $\mathbb{N}(l) = 0$, for the unique $n \in \mathbb{N}$ such that $l = \underline{n}$. Moreover, we write $\hat{q}$, where $q \in \mathfrak{M}$, for $\max(0, \min(1, q))$.

We propose the following *probabilistic jump instructions*:

- $\#\%\mathrm{H}(k)$, having the same effect as $\#j$ with probability $1/k$ for $j \in [1, \bar{k}]$, for $k \in \mathfrak{M}$ with $\mathbb{N}(k) = 0$;
- $\#\%\mathrm{G}(q)(k)$, having the same effect as $\#j$ with probability $\hat{q} \cdot (1 - \hat{q})^{j-1}$ for $j \in [1, \bar{k}]$, for $q \in \mathfrak{M}$ and $k \in \mathfrak{M}$ with $\mathbb{N}(k) = 0$;
- $\#\%\mathrm{G}(q)l$, having the same effect as $\#\bar{l} \cdot j$ with probability $\hat{q} \cdot (1 - \hat{q})^{j-1}$ for $j \in [1, \infty)$, for $q \in \mathfrak{M}$ and $l \in \mathfrak{M}$ with $\mathbb{N}(l) = 0$.

The letter H in $\#\%\mathrm{H}(k)$ indicates a homogeneous probability distribution, and the letter G in $\#\%\mathrm{G}(q)(k)$ and $\#\%\mathrm{G}(q)l$ indicates a geometric probability distribution. Instructions of the forms $\#\%\mathrm{H}(k)$ and $\#\%\mathrm{G}(q)(k)$ are bounded probabilistic jump instructions, whereas instructions of the form $\#\%\mathrm{G}(q)l$ are unbounded probabilistic jump instructions.

Like in the case of the probabilistic basic instructions, we propose in addition the following probabilistic jump instructions:

- $\#\%\mathrm{G}()(k)$ as the special case of $\#\%\mathrm{G}(q)(k)$ where $q = 1/2$;
- $\#\%\mathrm{G}()l$ as the special case of $\#\%\mathrm{G}(q)l$ where $q = 1/2$.

We believe that all probabilistic jump instructions can be eliminated. In particular, we believe that unbounded probabilistic jump instructions can be eliminated. This believe can be understood as the judgement that it is reasonable to expect from a semantic model of probabilistic instruction sequences that the following identity and similar ones hold:

$$
\begin{array}{l}
+a \,;\, \#\%\mathrm{G}()2 \,;\, (+b \,;\, ! \,;\, c)^{\omega} = \\
+a \,;\, +\%() \,;\, \#8 \,;\, \#10 \,; \\
\quad (+b \,;\, \#5 \,;\, \#10 \,;\, +\%() \,;\, \#8 \,;\, \#10 \,; \\
\qquad ! \,;\, \#5 \,;\, \#10 \,;\, +\%() \,;\, \#8 \,;\, \#10 \,; \\
\qquad c \,;\, \#5 \,;\, \#10 \,;\, +\%() \,;\, \#8 \,;\, \#10)^{\omega} \;.
\end{array}
$$

Taking this identity and similar ones as our point of departure, the question arises what is the most simple model that justifies them. A more general question is whether instruction sequences with unbounded probabilistic jump instructions can be translated into ones with only probabilistic test instructions provided it does not bother us that the instruction sequences may become much longer (e.g. expectation of the length bounded, but worst case length unbounded).

6 The Probabilistic Process Algebra Thesis

In the absence of probabilistic instructions, threads as considered in thread algebra [4] can be used to model the behaviours produced by instruction sequences under execution.[2] Processes as considered in general process algebras such as ACP [1], CCS [23] and CSP [18] can be used as well, but they give rise to a more complicated modelling of the behaviours of instruction sequences under execution (see e.g. [5]).

In the presence of probabilistic instructions, we would need a probabilistic thread algebra, i.e. a variant of thread algebra that covers probabilistic behaviours. It appears that any probabilistic thread algebra is inherently more complicated to such an extent that the advantage of not using a general process algebra evaporates. Moreover, it appears that any probabilistic thread algebra

[2] In [4], basic thread algebra is introduced under the name basic polarized process algebra.

requires justification by means of an appropriate probabilistic process algebra. This leads us to the following thesis:

Thesis. *Modelling the behaviours produced by probabilistic instruction sequences under execution is a matter of using directly processes as considered in some probabilistic process algebra.*

Notice that once we move from deterministic instructions to probabilistic instructions, instruction sequence becomes an indispensable concept. Instruction sequences cannot be replaced by threads or processes without taking potentially premature design decisions. However, it is a reasonable to claim that, like for deterministic instruction sequence notations, all probabilistic instruction sequence notations can be provided with a probabilistic semantics by translation of the instruction sequences concerned into appropriate single-pass instruction sequences. Thus, the approach of projection semantics works for probabilistic instruction sequence notations as well.

A probabilistic thread algebra has to cover the interaction between instruction sequence behaviours and services. Two mechanisms are involved in that. They are called the use mechanism and the apply mechanism (see e.g. [9]). The difference between them is a matter of perspective: the former is concerned with the effect of services on behaviours of instruction sequences and therefore produces behaviours, whereas the latter is concerned with the effect of instruction sequence behaviours on services and therefore produces services. It appears that the intricacy of a probabilistic thread algebra originates in large part from the use mechanism.

7 Discussion of Projectionism

In preceding sections, we have outlined how instruction sequences with the different kinds of probabilistic instructions can be translated into instruction sequences without them. Thus, we have made it plausible that projectionism is feasible for probabilistic instruction sequences.

Projectionism is the point of view that:

- any instruction sequence P, and more general even any program P, first and for all represents a single-pass instruction sequence as considered in PGA;
- this single-pass instruction sequence, found by a translation called a projection, represents in a natural and preferred way what is supposed to take place on execution of P;
- PGA provides the preferred notation for single-pass instruction sequences.

In a rigid form, as in [4], projectionism provides a definition of what constitutes a program.

The fact that projectionism is feasible for probabilistic instruction sequences, does not imply that it is uncomplicated. To give an idea of the complications that may arise, we will sketch below found challenges for projectionism.

First, we introduce some special notation. Let N be a program notation. Then we write N2pga for the projection function that gives, for each program P in N, the closed PGA terms that denotes the single-pass instruction sequence that produces on execution the same behaviour as P.

We have found the following challenges for projectionism:

- *Explosion of size.* If N2pga(P) is much longer than P, then the requirement that it represents in a natural way what is supposed to take place on execution of P is challenged. For example, if the primitive instructions of N include instructions to set and test up to n Boolean registers, then the projection to N2pga(P) may give rise to a combinatorial explosion of size. In such cases, the usual compromise is to permit single-pass instruction sequences to make use of services (see e.g. [9]).
- *Degradation of performance.* If N2pga(P)'s natural execution is much slower than P's execution, supposing a clear operational understanding of P, then the requirement that it represents in a natural way what is supposed to take place on execution of P is challenged. For example, if the primitive instructions of N include indirect jump instructions, then the projection to N2pga(P) may give rise to a degradation of performance (see e.g. [6]).
- *Incompatibility of services.* If N2pga(P) has to make use of services that are not deterministic, then the requirement that it represents in a natural way what is supposed to take place on execution of P is challenged. For example, if the primitive instructions of N include instructions of the form $+\%(q)$ or $-\%(q)$, then P cannot be projected to a single-pass instruction sequence without the use of probabilistic services. In this case, either probabilistic services must be permitted or probabilistic instruction sequences must not be considered programs.
- *Complexity of projection description.* The description of N2pga may be so complex that it defeats N2pga(P)'s purpose of being a natural explanation of what is supposed to take place on execution of P. For example, the projection semantics given for recursion in [2] suffers from this kind of complexity when compared with the conventional denotational semantics. In such cases, projectionism may be maintained conceptually, but rejected pragmatically.
- *Aesthetic degradation.* In N2pga(P), something elegant may have been replaced by nasty details. For example, if N provides guarded commands, then N2pga(P), which will be much more detailed, might be considered to exhibit signs of aesthetic degradation. This challenge is probably the most serious one, provided we accept that such elegant features belong to program notations. Of course, it may be decided to ignore aesthetic criteria altogether. However, more often than not, they have both conceptual and pragmatic importance.

One might be of the opinion that conceptual projectionism can accept explosion of size and/or degradation of performance. We do not share this opinion: both challenges require a more drastic response than a mere shift from a pragmatic to a conceptual understanding of projectionism. This drastic response may include viewing certain mechanisms as intrinsically indispensable for either execution performance or program compactness. For example, it is reasonable to consider the basic instructions of the form $\%(q)$, where q is a computable real number, indispensable if the expectations of the times to execute their realizations by means of $\%()$ are not all finite.

Nevertheless, projectionism looks to be reasonable for probabilistic programs: they can be projected adequately to deterministic single-pass instruction sequences for an execution architecture with probabilistic services.

8 Related Work

In [29], a notation for probabilistic programs is introduced in which we can write, for example, $\mathrm{random}(p{\cdot}\delta_0{+}q{\cdot}\delta_1)$. In general, $\mathrm{random}(\lambda)$ produces a value according to the probability distribution λ. In this case, δ_i is the probability distribution that gives probability 1 to i and probability 0 to other values. Thus, for $p{+}q = 1$, $p{\cdot}\delta_0{+}q{\cdot}\delta_1$ is the probability distribution that gives probability p to 0, probability q to 1, and probability 0 to other values. Clearly, $\mathrm{random}(p{\cdot}\delta_0{+}q{\cdot}\delta_1)$ corresponds to $\%(p)$. Moreover, using this kind of notation, we could write $\#(\frac{1}{k}{\cdot}(\delta_1{+}\cdots{+}\delta_{\bar{k}}))$ for $\#\%\mathrm{H}(k)$ and $\#(\hat{q}\cdot\delta_1+\hat{q}\cdot(1-\hat{q})\cdot\delta_2+\cdots+\hat{q}\cdot(1-\hat{q})^{k-1}\cdot\delta_k)$ for $\#\%\mathrm{G}(q)(k)$.

In much work on probabilistic programming, see e.g. [17, 21, 24], we find the binary probabilistic choice operator ${}_p\oplus$ (for $p \in [0,1]$). This operator chooses between its operands, taking its left operand with probability p. Clearly, $P\,{}_p\!\oplus Q$ can be taken as abbreviations for $+\%(p)\,;\mathbf{u}(P\,;\#2)\,;\mathbf{u}(Q)$. This kind of primitives dates back to [20] at least.

Quite related, but from a different perspective, is the toss primitive introduced in [14]. The intuition is that $\mathsf{toss}(bm,p)$ assigns to the Boolean memory cell bm the value T with probability $\hat{p}$ and the value F with probability $1{-}\hat{p}$. This means that $\mathsf{toss}(bm,p)$ has a side-effect on a state, which we understand as making use of a service. In other words, $\mathsf{toss}(bm,p)$ corresponds to a deterministic instruction intended to be processed by a probabilistic service.

Common in probabilistic programming are assignments of values randomly chosen from some interval of natural numbers to program variables (see e.g. [28]). Clearly, such random assignments correspond also to deterministic instructions intended to be processed by probabilistic services. Suppose that $x{=}i$ is a primitive instruction for assigning value i to program variable x. Then we can write: $\#\%\mathrm{H}(k)\,;\mathbf{u}(x{=}1\,;\#k)\,;\mathbf{u}(x{=}2\,;\#k{-}1)\,;\ldots;\mathbf{u}(x{=}k\,;\#1)$. This is a realistic representation of the assignment to x of a value randomly chosen from $\{1,\ldots,k\}$. However, it is clear that this way of representing random assignments leads to an exponential blow up in the size of any concrete instruction sequence representation, provided the concrete representation of k is its decimal representation.

The refinement oriented theory of programs uses demonic choice, usually written $\sqcap$, as a primitive (see e.g. [21, 22]). A demonic choice can be regarded as a probabilistic choice with unknown probabilities. Demonic choice could be written $+\sqcap$ in a PGA-like notation. However, a primitive instruction corresponding to demonic choice is not reasonable: no mechanism for the execution of $+\sqcap$ is conceivable. Demonic choice exists in the world of specifications, but not in the world of instruction sequences. This is definitely different with $+\%(p)$, because a mechanism for its execution is conceivable.

It appears that quantum computing has something to offer that cannot be obtained by conventional computing: it makes a stateless generator of random bits available (see e.g. [15, 25]). By that quantum computing indeed provides a justification of $+\%(1/2)$ as a probabilistic instruction.

9 Conclusions

We have made a notational proposal of probabilistic instructions with an informal semantics. By that we have contrasted probabilistic instructions in an execution architecture with deterministic services with deterministic instructions in an execution architecture with partly probabilistic services. The history of the proposed kinds of instructions can be traced.

We have refrained from an ad hoc formal semantic analysis of the proposed kinds of instructions. There are many solid semantic options, so many and so complex that another more distant analysis is necessary in advance to create a clear framework for the semantic analysis in question.

The grounds of this work are our conceptions of what a theory of probabilistic instruction sequences and a complementary theory of probabilistic instruction sequence processing (i.e. execution architectures with probabilistic services) will lead to:

- comprehensible explanations of relevant probabilistic algorithms, such as the Miller-Rabin probabilistic primality test [27], with precise descriptions of the kinds of instructions and services involved in them;
- a solid account of pseudo-random Boolean values and pseudo-random numbers;
- a thorough exposition of the different semantic options for probabilistic instruction sequences;
- explanations of relevant quantum algorithms, such as Shor's integer factorization algorithm [30], by first giving a clarifying analysis in terms of probabilistic instruction sequences or execution architectures with probabilistic services and only then showing how certain services in principle can be realized very efficiently with quantum computing.

Projectionism looks to be reasonable for probabilistic programs: they can be projected adequately to deterministic single-pass instruction sequences for an execution architecture with appropriate probabilistic services. At present, it is not entirely clear whether this extends to quantum programs.

References

1. Baeten, J.C.M., Weijland, W.P.: Process Algebra, *Cambridge Tracts in Theoretical Computer Science*, vol. 18. Cambridge University Press, Cambridge (1990)

2. Bergstra, J.A., Bethke, I.: Predictable and reliable program code: Virtual machine based projection semantics. In: J.A. Bergstra, M. Burgess (eds.) Handbook of Network and Systems Administration, pp. 653–685. Elsevier, Amsterdam (2007)
3. Bergstra, J.A., Bethke, I.: Square root meadows. `arXiv:0901.4664v1 [cs.LO]` at `http://arxiv.org/` (2009)
4. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming **51**(2), 125–156 (2002)
5. Bergstra, J.A., Middelburg, C.A.: Thread algebra with multi-level strategies. Fundamenta Informaticae **71**(2–3), 153–182 (2006)
6. Bergstra, J.A., Middelburg, C.A.: Instruction sequences with indirect jumps. Scientific Annals of Computer Science **17**, 19–46 (2007)
7. Bergstra, J.A., Middelburg, C.A.: Instruction sequences and non-uniform complexity theory. Electronic Report PRG0812, Programming Research Group, University of Amsterdam (2008). Available from `http://www.science.uva.nl/research/prog/publications.html`. Also available from `http://arxiv.org/`: `arXiv:0809.0352v1 [cs.CC]`
8. Bergstra, J.A., Middelburg, C.A.: Program algebra with a jump-shift instruction. Journal of Applied Logic **6**(4), 553–563 (2008)
9. Bergstra, J.A., Ponse, A.: Combining programs and state machines. Journal of Logic and Algebraic Programming **51**(2), 175–192 (2002)
10. Bergstra, J.A., Ponse, A.: Execution architectures for program algebra. Journal of Applied Logic **5**(1), 170–192 (2007)
11. Bergstra, J.A., Ponse, A.: A generic basis theorem for cancellation meadows. `arXiv:0803.3969v2 [math.RA]` at `http://arxiv.org/` (2008)
12. Bergstra, J.A., Ponse, A.: An instruction sequence semigroup with involutive anti-automorphisms. `arXiv:0903.1352v1 [cs.PL]` at `http://arxiv.org/` (2009)
13. Bergstra, J.A., Tucker, J.V.: The rational numbers as an abstract data type. Journal of the ACM **54**(2), Article 7 (2007)
14. Chadha, R., Cruz-Filipe, L., Mateus, P., Sernadas, A.: Reasoning about probabilistic sequential programs. Theoretical Computer Science **379**(1–2), 142–165 (2007)
15. Gay, S.J.: Quantum programming languages: Survey and bibliography. Mathematical Structures in Computer Science **16**(4), 581–600 (2006)
16. van Glabbeek, R.J., Smolka, S.A., Steffen, B.: Reactive, generative and stratified models of probabilistic processes. Information and Computation **121**(1), 59–80 (1995)
17. He Jifeng, Seidel, K., McIver, A.K.: Probabilistic models for the guarded command language. Science of Computer Programming **28**(2–3), 171–192 (1997)
18. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
19. Jonsson, B., Larsen, K.G., Yi, W.: Probabilistic extensions of process algebras. In: J.A. Bergstra, A. Ponse, S.A. Smolka (eds.) Handbook of Process Algebra, pp. 685–710. Elsevier, Amsterdam (2001)
20. Kozen, D.: A probabilistic PDL. Journal of Computer and System Sciences **30**(2), 162–178 (1985)
21. McIver, A.K., Morgan, C.C.: Demonic, angelic and unbounded probabilistic choices in sequential programs. Acta Informatica **37**(4–5), 329–354 (2001)
22. Meinicke, L., Solin, K.: Refinement algebra for probabilistic programs. Electronic Notes in Theoretical Computer Science **201**, 177–195 (2008)
23. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)

24. Morgan, C.C., McIver, A.K., Seidel, K.: Probabilistic predicate transformers. ACM Transactions on Programming Languages and Systems **18**(3), 325–353 (1996)
25. Perdrix, S., Jorrand, P.: Classically controlled quantum computation. Mathematical Structures in Computer Science **16**(4), 601–620 (2006)
26. Ponse, A.: Program algebra with unit instruction operators. Journal of Logic and Algebraic Programming **51**(2), 157–174 (2002)
27. Rabin, M.O.: Probabilistic algorithms. In: J.F. Traub (ed.) Algorithms and Complexity: New Directions and Recent Results, pp. 21–39. Academic Press, New York (1976)
28. Schöning, U.: A probabilistic algorithm for k-SAT based on limited local search and restart. Algorithmica **32**(4), 615–623 (2002)
29. Sharir, M., Pnueli, A., Hart, S.: Verification of probabilistic programs. SIAM Journal of Computing **13**(2), 292–314 (1984)
30. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: FOCS '94, pp. 124–134. IEEE Computer Society Press (1994)

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0905] J.A. Bergstra and C.A. Middelburg, *A Protocol for Instruction Stream Processing,* Programming Research Group - University of Amsterdam, 2009.

[PRG0904] J.A. Bergstra and C.A. Middelburg, *A Process Calculus with Finitary Comprehended Terms,* Programming Research Group - University of Amsterdam, 2009.

[PRG0903] J.A. Bergstra and C.A. Middelburg, *Transmission Protocols for Instruction Streams,* Programming Research Group - University of Amsterdam, 2009.

[PRG0902] J.A. Bergstra and C.A. Middelburg, *Meadow Enriched ACP Process Algebras,* Programming Research Group - University of Amsterdam, 2009.

[PRG0901] J.A. Bergstra and C.A. Middelburg, *Timed Tuplix Calculus and the Wesseling and van den Berg Equation,* Programming Research Group - University of Amsterdam, 2009.

[PRG0814] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences for the Production of Processes,* Programming Research Group - University of Amsterdam, 2008.

[PRG0813] J.A. Bergstra and C.A. Middelburg, *On the Expressiveness of Single-Pass Instruction Sequences,* Programming Research Group - University of Amsterdam, 2008.

[PRG0812] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences and Non-uniform Complexity Theory,* Programming Research Group - University of Amsterdam, 2008.

[PRG0811] D. Staudt, *A Case Study in Software Engineering with PSF: A Domotics Application,* Programming Research Group - University of Amsterdam, 2008.

[PRG0810] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Poly-Threading,* Programming Research Group - University of Amsterdam, 2008.

[PRG0809] J.A. Bergstra and C.A. Middelburg, *Data Linkage Dynamics with Shedding,* Programming Research Group - University of Amsterdam, 2008.

[PRG0808] B. Diertens, *A Process Algebra Software Engineering Environment,* Programming Research Group - University of Amsterdam, 2008.

[PRG0807] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *Tuplix Calculus Specifications of Financial Transfer Networks,* Programming Research Group - University of Amsterdam, 2008.

[PRG0806] J.A. Bergstra and C.A. Middelburg, *Data Linkage Algebra, Data Linkage Dynamics, and Priority Rewriting,* Programming Research Group - University of Amsterdam, 2008.

[PRG0805] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *UvA Budget Allocatie Model,* Programming Research Group - University of Amsterdam, 2008.

[PRG0804] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Sequential Poly-Threading,* Programming Research Group - University of Amsterdam, 2008.

[PRG0803] J.A. Bergstra and C.A. Middelburg, *Thread Extraction for Polyadic Instruction Sequences,* Programming Research Group - University of Amsterdam, 2008.

[PRG0802] A. Barros and T. Hou, *A Constructive Version of AIP Revisited,* Programming Research Group - University of Amsterdam, 2008.

[PRG0801] J.A. Bergstra and C.A. Middelburg, *Programming an Interpreter Using Molecular Dynamics,* Programming Research Group - University of Amsterdam, 2008.

[PRG0713] J.A. Bergstra, A. Ponse, and M.B. van der Zwaag, *Tuplix Calculus,* Programming Research Group - University of Amsterdam, 2007.

[PRG0712] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *Towards a Formalization of Budgets,* Programming Research Group - University of Amsterdam, 2007.

[PRG0711] J.A. Bergstra and C.A. Middelburg, *Program Algebra with a Jump-Shift Instruction,* Programming Research Group - University of Amsterdam, 2007.

[PRG0710] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences with Dynamically Instantiated Instructions,* Programming Research Group - University of Amsterdam, 2007.

[PRG0709] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences with Indirect Jumps,* Programming Research Group - University of Amsterdam, 2007.

[PRG0708] B. Diertens, *Software (Re-)Engineering with PSF III: an IDE for PSF,* Programming Research Group - University of Amsterdam, 2007.

[PRG0707] J.A. Bergstra and C.A. Middelburg, *An Interface Group for Process Components,* Programming Research Group - University of Amsterdam, 2007.

[PRG0706] J.A. Bergstra, Y. Hirschfeld, and J.V. Tucker, *Skew Meadows,* Programming Research Group - University of Amsterdam, 2007.

[PRG0705] J.A. Bergstra, Y. Hirschfeld, and J.V. Tucker, *Meadows,* Programming Research Group - University of Amsterdam, 2007.

[PRG0704] J.A. Bergstra and C.A. Middelburg, *Machine Structure Oriented Control Code Logic (Extended Version),* Programming Research Group - University of Amsterdam, 2007.

[PRG0703] J.A. Bergstra and C.A. Middelburg, *On the Operating Unit Size of Load/Store Architectures,* Programming Research Group - University of Amsterdam, 2007.

[PRG0702] J.A. Bergstra and A. Ponse, *Interface Groups and Financial Transfer Architectures,* Programming Research Group - University of Amsterdam, 2007.

[PRG0701] J.A. Bergstra, I. Bethke, and M. Burgess, *A Process Algebra Based Framework for Promise Theory,* Programming Research Group - University of Amsterdam, 2007.

[PRG0610] J.A. Bergstra and C.A. Middelburg, *Parallel Processes with Implicit Computational Capital,* Programming Research Group - University of Amsterdam, 2006.

[PRG0609] B. Diertens, *Software (Re-)Engineering with PSF II: from architecture to implementation,* Programming Research Group - University of Amsterdam, 2006.

[PRG0608] A. Ponse and M.B. van der Zwaag, *Risk Assessment for One-Counter Threads,* Programming Research Group - University of Amsterdam, 2006.

[PRG0607] J.A. Bergstra and C.A. Middelburg, *Synchronous Cooperation for Explicit Multi-Threading,* Programming Research Group - University of Amsterdam, 2006.

[PRG0606] J.A. Bergstra and M. Burgess, *Local and Global Trust Based on the Concept of Promises,* Programming Research Group - University of Amsterdam, 2006.

[PRG0605] J.A. Bergstra and J.V. Tucker, *Division Safe Calculation in Totalised Fields,* Programming Research Group - University of Amsterdam, 2006.

[PRG0604] J.A. Bergstra and A. Ponse, *Projection Semantics for Rigid Loops,* Programming Research Group - University of Amsterdam, 2006.

[PRG0603] J.A. Bergstra and I. Bethke, *Predictable and Reliable Program Code: Virtual Machine-based Projection Semantics (submitted for inclusion in the Handbook of Network and Systems Administration),* Programming Research Group - University of Amsterdam, 2006.

[PRG0602] J.A. Bergstra and A. Ponse, *Program Algebra with Repeat Instruction,* Programming Research Group - University of Amsterdam, 2006.

[PRG0601] J.A. Bergstra and A. Ponse, *Interface Groups for Analytic Execution Architectures,* Programming Research Group - University of Amsterdam, 2006.

[PRG0505] B. Diertens, *Software (Re-)Engineering with PSF,* Programming Research Group - University of Amsterdam, 2005.

[PRG0504] P.H. Rodenburg, *Piecewise Initial Algebra Semantics,* Programming Research Group - University of Amsterdam, 2005.

[PRG0503] T.D. Vu, *Metric Denotational Semantics for BPPA,* Programming Research Group - University of Amsterdam, 2005.

[PRG0502] J.A. Bergstra, I. Bethke, and A. Ponse, *Decision Problems for Pushdown Threads,* Programming Research Group - University of Amsterdam, 2005.

[PRG0501] J.A. Bergstra and A. Ponse, *A Bypass of Cohen's Impossibility Result,* Programming Research Group - University of Amsterdam, 2005.

[PRG0405] J.A. Bergstra and I. Bethke, *An Upper Bound for the Equational Specification of Finite State Services,* Programming Research Group - University of Amsterdam, 2004.

[PRG0404] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Strategic Interleaving,* Programming Research Group - University of Amsterdam, 2004.

[PRG0403] B. Diertens, *A Compiler-projection from PGLEc.MSPio to Parrot,* Programming Research Group - University of Amsterdam, 2004.

[PRG0402] J.A. Bergstra and I. Bethke, *Linear Projective Program Syntax,* Programming Research Group - University of Amsterdam, 2004.

[PRG0401] B. Diertens, *Molecular Scripting Primitives,* Programming Research Group - University of Amsterdam, 2004.

[PRG0302] B. Diertens, *A Toolset for PGA,* Programming Research Group - University of Amsterdam, 2003.

[PRG0301] J.A. Bergstra and P. Walters, *Projection Semantics for Multi-File Programs,* Programming Research Group - University of Amsterdam, 2003.

[PRG0201] I. Bethke and P. Walters, *Molecule-oriented Java Programs for Cyclic Sequences,* Programming Research Group - University of Amsterdam, 2002.

The above reports and more are available through the website: www.science.uva.nl/research/prog/

Electronic Report Series

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
the Netherlands

www.science.uva.nl/research/prog/