World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:4, No:1, 2010

PZ: A Z-based Formalism for Modeling Probabilistic Behavior

Hassan Haghighi
Faculty of Electrical and Computer Engineering,
Shahid Beheshti University,
Tehran, Iran
h_haghighi@sbu.ac.ir

Abstract—Probabilistic techniques in computer programs are becoming more and more widely used. Therefore, there is a big interest in the formal specification, verification, and development of probabilistic programs. In our work-in-progress project, we are attempting to make a constructive framework for developing probabilistic programs formally. The main contribution of this paper is to introduce an intermediate artifact of our work, a Z-based formalism called PZ, by which one can build set theoretical models of probabilistic programs. We propose to use a constructive set theory, called CZ set theory, to interpret the specifications written in PZ. Since CZ has an interpretation in Martin-Löf's theory of types, this idea enables us to derive probabilistic programs from correctness proofs of their PZ specifications.

Keywords—formal specification, formal program development, probabilistic programs, CZ set theory, type theory.

I. Introduction

METHODS for modelling probabilistic programs go back to the early work in [3] introducing *probabilistic predicate transformers* as a framework for reasoning about *imperative* probabilistic programs. From that time on, a wide variety of logics have been developed as possible bases for verifying probabilistic systems (A survey of this work can be found in [8]); the *expectation transformer* approach in particular integrates traditional assertional-styles of program verification with probability. An *expectation* is a generalized predicate suitable for expressing quantitative properties such as *the probability of achieving a postcondition*.

In [9], [11], and [12], Morgan et al. have replaced the programming logic of *weakest preconditions* for Dijkstra's *GCL* (*Guarded Command Language*) by a new logic called *greatest pre-expectation*. In this way, probabilistic nondeterminism is introduced into *GCL* and thus a means is provided with which probabilistic programs can be rigorously developed and verified. Although the semantics has been designed to work at the level of program code, it has an in-built notion of program refinement which encourages a prover to move between various levels of abstraction. Unlike many publications of Morgan et al. that handle probabilistic choice in *imperative* settings, there are several studies considering probabilistic choice in *functional* languages; for example, see [1] and [14].

As far as we know, much of the work in the literature, such as the above mentioned work, has focused on the *verification* of probabilistic programs; however, besides a considerable trend in *verifying* probabilistic programs, there is a big interest in the *formal specification and development* of such programs. In our work-in-progress project, we are attempting to make a *constructive* framework for deriving probabilistic programs from *correctness proofs of their formal specifications*. In Fig 1., we have shown the architecture of this framework.

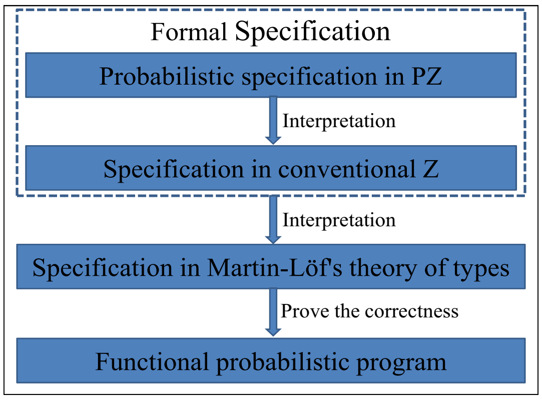
Formal Specification
Probabilistic specification in PZ
Interpretation
Specification in conventional Z
Interpretation
Specification in Martin-Löf's theory of types
Prove the correctness
Functional probabilistic program

Fig. 1. The constructive framework for developing probabilistic programs

In this framework, we will use a *Z-based* formalism [15], [18], called *PZ* (*Probabilistic Z*), to write specifications of probabilistic programs. Then we will interpret the resulting PZ specifications into their counterparts in the Z notation itself. Of course, to interpret the obtained specifications in Z, we will use a *constructive* set theory, called CZ set theory [10], instead of the classical set theory Z. We choose CZ since it has an interpretation [10] in *Martin-Löf's theory of types* [7]; this enables us to translate our Z-style specification of a probabilistic program into its counterpart in Martin-Löf's theory of types and then drive a functional program from a correctness proof of the resulting type theoretical specification. In this way, we will provide a completely formal way for developing probabilistic programs.

The main contribution of the current paper is to introduce PZ and show how it can be interpreted in conventional Z. We give an interpretation of PZ that can constructively lead to programs preserving the initially specified probabilistic behavior. To build PZ, we first augment the Z notation with a new notion of operation schemas, called *probabilistic schema*, intended to specify probabilistic operations. Also, since the schema calculus operations of Z will no longer work on probabilistic schemas, we define a new set of operators for the schema calculus operations *negation*, *conjunction*, *disjunction*, *existential quantifier*, *universal quantifier*, and *sequential composition* which properly work on probabilistic schemas as well as ordinary operation schemas. Augmenting Z with the notion of probabilistic schemas and new operations for the schema calculus results in the PZ formalism.

The paper is organized in the following way. In section 2, we give a brief overview of the CZ set theory and its interpretation in Martin-Löf's theory of types. Of course, in order to follow the main issue of the paper, i.e. modeling probabilistic behavior, we omit most of technical details in the program development stage, specially when transforming CZ specifications into type theory and using type theoretical rules for extracting functional programs from resulting type theoretical specifications. Thus, readers who are not familiar with type theory or are not interested in following the technical details of CZ and its interpretation in type theory can skip section 2.

In section 3, we begin to introduce the PZ notation by defining the notion of probabilistic schemas and showing how they can be used to model probabilistic operations. We also give an interpretation of probabilistic schemas in conventional Z. Since the proposed interpretation of probabilistic schemas is not sufficient for the purpose of program construction, in section 4 we give a new interpretation of probabilistic schemas leading constructively to functional programs that can implement the initially specified probabilistic behavior. In section 5, we show that the operations of the Z schema calculus will no longer work on probabilistic schemas. We thus introduce a new set of schema calculus operations into PZ that can be applied to probabilistic schemas as well as ordinary operation schemas. The last section is devoted to the conclusion and directions for future work.

II. Preliminaries

To employ both the facilities of Z as a specification medium and the abilities of constructive theories in program development, in [10], the CZ set theory has been introduced which provides constructive interpretations for the specification constructs of the Z notation. Also, the CZ set theory has been interpreted in Martin-Löf's theory of types. Since we map all the new specification constructs of PZ into Z itself, we can still use the interpretation of CZ in type theory to extract functional, probabilistic programs from their formal specifications written in PZ. In this section, we give a brief description of the CZ set theory and its interpretation in type theory.

A. CZ Set Theory

The constructive set theory CZ has been introduced in [10] to provide constructive interpretations for Z specifications. All proof rules of the *classical* set theory *ZF* (*Zermelo-Fraenkel*) can be used in CZ except *classical negation* since this rule is derived from the axiom of *excluded middle*. The axioms of CZ shadow those of the classical theory; indeed, most axioms remain intact. However, three axioms including *separation*, *foundation*, and *power set* have been modified to satisfy *constructive* scruples. Also, modifying the power set axiom yields a new axiom concerning the *cartesian product* set constructor. To indicate the constructive nature of CZ, we give the modified version of the power set axiom here. Other axioms of CZ can be found in [10].

decidable power set: $\forall x \cdot \exists z \cdot \forall y \cdot y \in z \Leftrightarrow y \sqsubseteq x$

In the above axiom, the relation $y \sqsubseteq x$ indicates that y is a *decidable subset* of x. $y \sqsubseteq x$ iff $y \subseteq x$ and $\forall u \in x \cdot u \in y \vee \neg(u \in y)$. In the Z (ZF without the axiom of *replacement* [10]) set theory, the power set is not restricted: any kind of subset is permitted, not just the decidable ones. It is the most important difference between Z and CZ set theories. CZ only permits subsets which can be constructed in the sense that we can determine their membership relative to their superset. Intuitively, the decidable subsets can be identified with decision procedures which test for membership.

The CZ set theory can be considered as a *constructive* interpretation of the Z language. Specially, replacing instances of the power set by decidable ones provides a way for determining whether specifications specify decidable problems. In [10], it has been shown that the CZ set theory is enough for the purposes of program specification in the style of Z. In other words, the common set theoretical constructions employed in Z can be interpreted using CZ. In the next subsection, we show how one can transform CZ constructs into their counterparts in type theory.

B. Interpretation of CZ in Martin-Löf's Theory of Types

In [10], a model $\nu = < V, \dot{\in}, \doteq >$ of CZ in Martin-Löf's theory of types has been built in which each set is associated with a pair consisting of a base type together with a family of types, i.e., its elements. In the model ν, $V \cong Wx \in U.x$, where U is the *universe* whose elements are themselves types, and W is the type constructor for recursive data types [13]. The general form of elements of recursive types is that each *node* is built from a certain collection of *predecessors* of the same type. Suppose that we have a type A of *sorts* of node. For a particular kind of node $a \in A$, we specify what form the predecessors of the node take by supplying a type $B(a)$, which we can think of as the type of *names of predecessors places*. The collection of predecessors of the node a is determined by a function from $B(a)$ to the recursive type in question. The type thus constructed is $Wx \in A.B$ whose elements denoted by $sup(a, f)$, where $a \in A$ and f is a function from $B[a]$ to $Wx \in A.B$. According to an important property of W, each $\alpha \in V$ can be split into two components α^- and $\alpha^{\tilde{}}$ such that $\alpha = sup(\alpha^-, \alpha^{\tilde{}})$ where $\alpha^- \in U$ and $\alpha^{\tilde{}} \in \alpha^- \Rightarrow V$.

To complete the description of the model ν, we need to define two binary relations $\doteq$ and $\dot{\in}$:

$$\alpha \doteq \beta \cong (\Pi x \in \alpha^- \cdot \alpha^{\tilde{}} x \,\dot{\in}\, \beta) \otimes (\Pi x \in \beta^- \cdot \beta^{\tilde{}} x \,\dot{\in}\, \alpha)$$

$$\alpha \,\dot{\in}\, \beta \cong \Sigma x \in \beta^- \cdot \beta^{\tilde{}} x \doteq \alpha$$

In the above definition, the equality between sets is explained according to the extensional equality in set theories, stated by the *extensionality* axiom. Using the model ν, in [10], it has been given the type theoretical interpretations of the empty set and the set of natural numbers as two basic sets of CZ. Also, the function ξ has been defined which assigns elements of V to well-formed and atomic formulas of CZ as follows:

$$\begin{array}{l}
[\Omega]_\xi = \Omega \\
[x = y]_\xi = \xi(x) \doteq \xi(y) \\
[x \in y]_\xi = \xi(x) \mathrel{\dot{\in}} \xi(y) \\
[\phi \wedge \psi]_\xi = [\phi]_\xi \otimes [\psi]_\xi \\
[\phi \vee \psi]_\xi = [\phi]_\xi \oplus [\psi]_\xi \\
[\phi \Rightarrow \psi]_\xi = [\phi]_\xi \Rightarrow [\psi]_\xi \\
[\forall\, x \in y \cdot \phi]_\xi = \Pi\alpha \in (\xi(y))^- \cdot [\phi]_{\xi[(\xi(y))\tilde{}\alpha/x]} \\
[\forall\, x \cdot \phi]_\xi = \Pi\alpha \in V \cdot [\phi]_{\xi[\alpha/x]} \\
[\exists\, x \in y \cdot \phi]_\xi = \Sigma\alpha \in (\xi(y))^- \cdot [\phi]_{\xi[(\xi(y))\tilde{}\alpha/x]} \\
[\exists\, x \cdot \phi]_\xi = \Sigma\alpha \in V \cdot [\phi]_{\xi[\alpha/x]}
\end{array}$$

The translation given in [10] is not still sufficient to transform a Z specification into a type theoretical one: we need to interpret schemas, as a distinctive feature of the Z notation, in type theory. In [4], we presented a solution to handle Z schemas. Here, we only mention our solution for operation schemas. Suppose that an operation schema has the following general form:

$$\begin{array}{l}
Op\_Schema \cong [x_1 \in A_1;\ ...;\ x_m \in A_m; \\
\qquad\qquad\qquad y_1 \in B_1;\ ...;\ y_n \in B_n \mid \phi],
\end{array}$$

where $x_i (i : 1..m)$ are input or before state variables, $y_j (j : 1..n)$ are output or after state variables, and ϕ denotes the pre- and postconditions of the operation being specified. Now we extend the function ξ to translate Op_Schema into an element of V:

$$\begin{array}{l}
[Op\_Schema]_\xi = (\Pi\alpha_1 \in (\xi(A_1))^-, ..., \alpha_m \in (\xi(A_m))^- \cdot \\
\qquad\qquad \Sigma\beta_1 \in (\xi(B_1))^-, ..., \beta_n \in (\xi(B_n))^- \cdot \\
\qquad\qquad [\phi]_\xi)_{[(\xi(A_i))\tilde{}\alpha_i/x_i][(\xi(B_j))\tilde{}\beta_j/y_j]}
\end{array}$$

Now, given a specification in Z, we can use the function ξ to translate the specification into a type in type theory and then extract a program (a term in type theory) which meets the specification (more precisely, meets its representation in type theory).

III. Probabilistic Schema

In this section, we introduce the PZ notation as a tool to specify probabilistic programs. To achieve this goal, we first define the notion of *probabilistic schema* by which one can simply model probabilistic operations.

Definition 3.1 The general form of probabilistic schemas is as follows:

$$\begin{array}{l}
P\_Schema \cong [x_1 \in A_1;\ ...;\ x_m \in A_m; \\
y_1 \in B_1;\ ...;\ y_n \in B_n \mid \phi \wedge (p_1 : \phi_1;\ ...;\ p_l : \phi_l)],
\end{array}$$

where $x_i (i : 1..m)$ are input or before state variables, and $y_j (j : 1..n)$ are output or after state variables. Some part of the schema predicate, shown as ϕ, specifies those functionalities of the operation that are *non-probabilistic*; it specially includes the preconditions of the operation being specified. The remainder of the schema predicate has been separated into l predicates $\phi_1, ..., \phi_l$; $p_k \in R (k : 1..l)$ are probabilities and by the notation $p_k : \phi_k$, we want to say that the predicate ϕ_k holds with probability p_k. In other words, the relationship between the variables of P_Schema is stated by ϕ_k with probability p_k. For a given probabilistic schema, we assume that $p_1 + ... + p_l = 1$. We also assume that for each $k : 1..l$, $p_k \geq 0$. Notice that in the predicate part of P_Schema, l may be equal to 0. In other words, ordinary operation schemas are considered as special cases of probabilistic schemas. $\triangle$

In the next example, we use the notion of probabilistic schema to specify a simple, probabilistic operation.

Example 3.2 A fly moves along a straight line in unit increments. At each time period, it moves one unit to the left with probability 0.3, one unit to the right with probability 0.3, and stays in place with probability 0.4, independently of the past history of movements. The straight line has m units, and a spider is lurking at positions 1 and m: if the fly lands there, it is captured by the spider, and the process terminates. By the following probabilistic schema, we specify the movement of the fly when moving from one of the positions $2, ..., m-1$. In this schema, $x?$ and $y!$ are the current and the next positions of the fly, respectively. Also, $m?$ is the number of units in the straight line.

$$\begin{array}{l}
P\_FlyMove \cong [x?, m?, y! \in \mathbb{N} \mid \\
m? > 2 \wedge x? > 1 \wedge x? < m? \wedge \\
(0.3 : y! = x? - 1;\ 0.3 : y! = x? + 1;\ 0.4 : y! = x?)]
\end{array}$$ $\triangle$

Now, we present a way to transform probabilistic schemas into ordinary operation schemas of Z. The next definition introduces a function $[]^P$ that maps probabilistic schemas into ordinary operation schemas of Z. We will show later that such an interpretation of probabilistic schemas is not enough for the purpose of constructive program development. Therefore, in section 4, we change this interpretation to provide a constructive way for extracting probabilistic programs from their PZ specifications.

Definition 3.3 Recall P_Schema, given in definition 3.1 as the general form of probabilistic schemas. If for all real numbers $p_1, ..., p_l$, the maximum number of digits to the right of the decimal point is d, then we have:

if P_Schema is an ordinary operation schema (i.e., when $l = 0$), then $[P\_Schema]^P = P\_Schema$;
otherwise, $[P\_Schema]^P \cong [x_1 \in A_1;\ ...;\ x_m \in A_m;$

$$\begin{array}{l}
y_1 \in B_1;\ ...;\ y_n \in B_n \mid \\
\phi \wedge (\exists\, p \in \mathbb{N} \cdot ((0 \leq p < p_1 * 10^d \wedge \phi_1) \vee \\
(p_1 * 10^d \leq p < (p_1 + p_2) * 10^d \wedge \phi_2) \vee ... \vee \\
((p_1 + ... + p_{l-1}) * 10^d \leq p < (p_1 + ... + p_l) * 10^d \wedge \phi_l)))]
\end{array}$$

$[]^P$ behaves as an identity function when applied to an ordinary operation schema, i.e., when $l = 0$; otherwise, an auxiliary variable $p \in \mathbb{N}$ is introduced into the predicate part helping us to implement the probabilistic choice between l predicates $\phi_1, ..., \phi_l$. The variable p ranges nondeterministically from 0 to $10^d - 1$, and the length of each allowable interval of its values determines how many times (of 10^d times) a predicate $\phi_k (k : 1..l)$ holds (or in fact describes the relationship between the schema variables). More precisely, in $p_k * 10^d$ cases per 10^d times, a predicate $\phi_k (k : 1..l)$ determines the behavior of the final program. In the next example, we apply the above defined

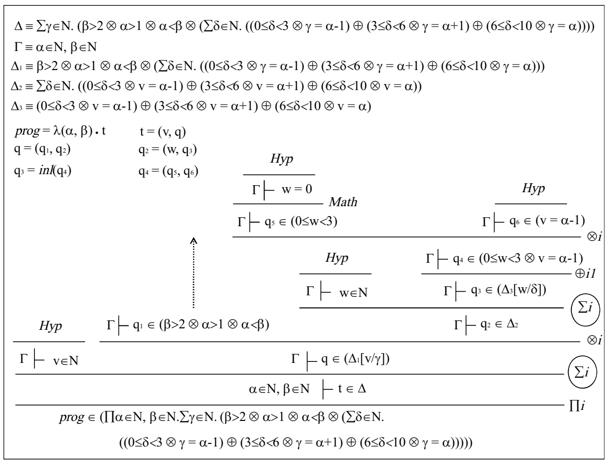
Δ ≡ ∑γ∈N. (β>2 ⊗ α>1 ⊗ α<β ⊗ (∑δ∈N. ((0≤δ<3 ⊗ γ = α-1) ⊕ (3≤δ<6 ⊗ γ = α+1) ⊕ (6≤δ<10 ⊗ γ = α))))
Γ ≡ α∈N, β∈N
Δ1 ≡ β>2 ⊗ α>1 ⊗ α<β ⊗ (∑δ∈N. ((0≤δ<3 ⊗ γ = α-1) ⊕ (3≤δ<6 ⊗ γ = α+1) ⊕ (6≤δ<10 ⊗ γ = α)))
Δ2 ≡ ∑δ∈N. ((0≤δ<3 ⊗ v = α-1) ⊕ (3≤δ<6 ⊗ v = α+1) ⊕ (6≤δ<10 ⊗ v = α))
Δ3 ≡ (0≤δ<3 ⊗ v = α-1) ⊕ (3≤δ<6 ⊗ v = α+1) ⊕ (6≤δ<10 ⊗ v = α)
prog = λ(α, β) . t
t = (v, q)
q = (q1, q2)
q2 = (w, q3)
q3 = inl(q4)
q4 = (q5, q6)
Hyp
Γ ⊢ w = 0
Math
Γ ⊢ q5 ∈ (0≤w<3)
Hyp
Γ ⊢ q6 ∈ (v = α-1)
⊗i
Hyp
Γ ⊢ w∈N
Γ ⊢ q4 ∈ (0≤w<3 ⊗ v = α-1)
⊕i1
Γ ⊢ q3 ∈ (Δ3[w/δ])
Σi
Hyp
Γ ⊢ q1 ∈ (β>2 ⊗ α>1 ⊗ α<β)
Γ ⊢ q2 ∈ Δ2
⊗i
Γ ⊢ v∈N
Γ ⊢ q ∈ (Δ1[v/γ])
Σi
α∈N, β∈N ⊢ t ∈ Δ
∏i
prog ∈ (∏α∈N, β∈N.∑γ∈N. (β>2 ⊗ α>1 ⊗ α<β ⊗ (∑δ∈N.
((0≤δ<3 ⊗ γ = α-1) ⊕ (3≤δ<6 ⊗ γ = α+1) ⊕ (6≤δ<10 ⊗ γ = α)))))

Fig. 2. Program extraction from the probabilistic schema $P\_FlyMove$.

interpretation to the probabilistic schema $P\_FlyMove$, given in example 3.2. We then use the interpretation of CZ in type theory to extract a functional program from the resulting specification.

Example 3.4 We first use the function $[]^P$ to transform $P\_FlyMove$ into an ordinary operation schema of Z as follows:

$$[P\_FlyMove]^P \cong [x?, m?, y! \in \mathbb{N} \mid$$
$$m? > 2 \wedge x? > 1 \wedge x? < m? \wedge (\exists\, p \in \mathbb{N}\cdot$$
$$((0 \leq p < 3 \wedge y! = x? - 1) \vee (3 \leq p < 6 \wedge y! = x? + 1) \vee$$
$$(6 \leq p < 10 \wedge y! = x?)))]$$

By the above schema, p *nondeterministically* takes one of 10 values $0, 1, ..., 9$. For three (i.e., in 3 cases per 10) possible values of p (i.e., $0, 1$, and 2), it has been specified that the fly moves one unit to the left. For other three (i.e., in 3 cases per 10) possible values of p (i.e., $3, 4$, and 5), it has been described that the fly moves one unit to the right. Finally, for the remaining (i.e., in 4 cases per 10) possible values of p (i.e., $6, 7, 8$, and 9), it has been indicated that the fly stays in place. Thus, it seems that if one makes a uniform choice to select one of the values $0, 1, ..., 9$ for p, s/he will be provided with a correct implementation of $P\_FlyMove$. Nevertheless, we now show that the schema $[P\_FlyMove]^P$ cannot *constructively* lead to a program implementing the probabilistic behavior specified by $P\_FlyMove$.

To extract a program from a correctness proof of $[P\_FlyMove]^P$, we first use the function ξ (see subsection 2.2) to translate the operation schema $[P\_FlyMove]^P$ into type theory. The resulting type theoretical specification is as follows:

$$\Pi\alpha \in \mathbb{N}, \beta \in \mathbb{N} \cdot \Sigma\gamma \in \mathbb{N} \cdot (\beta \dot{>} 2 \otimes \alpha \dot{>} 1 \otimes \alpha \dot{<} \beta \otimes$$
$$(\Sigma\delta \in \mathbb{N} \cdot ((0 \dot{\leq} \delta \dot{<} 3 \otimes \gamma \dot{=} \alpha - 1) \oplus (3 \dot{\leq} \delta \dot{<} 6 \otimes \gamma \dot{=} \alpha + 1) \oplus$$
$$(6 \dot{\leq} \delta \dot{<} 10 \otimes \gamma \dot{=} \alpha)))),$$

where α, β, γ, and δ correspond to the variables $x?$, $m?$, $y!$, and p existing in $[P\_FlyMove]^P$, respectively. Also, $b \dot{\leq} a$ and $b \dot{<} a$ are abbreviations for $\Sigma\rho \in \mathbb{N} \cdot b + \rho \dot{=} a$ and $\Sigma\rho \in \mathbb{N}_1 \cdot b + \rho \dot{=} a$, respectively. Now we can use the inference rules of type theory to prove the correctness of the above specification (or in other words, construct an object of its corresponding type). This object can be viewed as a program satisfying the schema $P\_FlyMove$. A part of such a proof is shown in Fig. 2. At the end of the proof, the following functional program has been obtained:

$$prog = \lambda(\alpha, \beta).(\alpha - 1, q),$$

where q is an intermediate proof object in the proof tree (see Fig. 2). For each valuation of $\alpha, \beta \in \mathbb{N}$, the program $prog$ *always* provides the value $\alpha - 1$ for γ, provided that $\beta > 2$, $\alpha > 1$, and $\alpha < \beta$. In this way, $prog$ cannot implement the probabilistic behavior specified by $P\_FlyMove$: according to $prog$, the fly *always* moves one unit to the left, provided that it does not move from any of the positions 1 and m. Notice that if we select another path in the proof tree (i.e., if we prove the correctness of $(3 \dot{\leq} \delta \dot{<} 6 \otimes \gamma \dot{=} \alpha + 1)$ or $(6 \dot{\leq} \delta \dot{<} 10 \otimes \gamma \dot{=} \alpha)$, rather than $(0 \dot{\leq} \delta \dot{<} 3 \otimes \gamma \dot{=} \alpha - 1)$), we will again obtain a program that cannot implement the probabilistic behavior specified by $P\_FlyMove$. The problem is due to the fact that in the proof tree, we can replace each of the variables γ and δ by *only one* of the possible values. This occurs when we use the introduction rule for dependent sum (Σ_i); see two circled Σ_i in Fig. 2. As it can be seen in the proof tree, this finally results in the single value 0 for δ and the single value $\alpha - 1$ for γ. $\triangle$

As it was shown in example 3.4, using the function $[]^P$ to interpret the probabilistic schema $P\_FlyMove$ and then proving the correctness of the resulting specification did not *constructively* lead to a probabilistic program being enable to implement the probabilistic behavior initially specified by $P\_FlyMove$. We can investigate this problem in the general case by applying $[]^P$ to the schema P_Schema, introduced earlier in definition 3.1 as the general form of probabilistic schemas:

$$[P\_Schema]^P \cong [x_1 \in A_1;\ ...;\ x_m \in A_m;$$
$$y_1 \in B_1;\ ...;\ y_n \in B_n \mid$$
$$\phi \wedge (\exists\, p \in \mathbb{N} \cdot ((0 \leq p < p_1 * 10^d \wedge \phi_1) \vee ... \vee$$
$$((p_1 + ... + p_{l-1}) * 10^d \leq p < (p_1 + ... + p_l) * 10^d \wedge \phi_l)))]$$

By the function ξ, presented in subsection 2.2, the following equality holds:

$$[[P\_Schema]^P]_\xi =$$
$$\Pi\alpha_1 \in (\xi(A_1))^-, ..., \alpha_m \in (\xi(A_m))^- \cdot$$
$$\Sigma\beta_1 \in (\xi(B_1))^-, ..., \beta_n \in (\xi(B_n))^- \cdot$$
$$([\phi \wedge (\exists\, p \in \mathbb{N} \cdot ((0 \leq p < p_1 * 10^d \wedge \phi_1) \vee ... \vee$$
$$((p_1 + ... + p_{l-1}) * 10^d \leq p < (p_1 + ... + p_l) * 10^d \wedge$$
$$\phi_l)))]_\xi)_{[(\xi(A_i))^{\neg}\alpha_i/x_i][(\xi(B_j))^{\neg}\beta_j/y_j]},$$

where $[[P\_Schema]^P]_\xi$ is the type theoretical equivalent of $[P\_Schema]^P$. Using the conventions

$A'_i = (\xi(A_i))^-$ $(i : 1..m)$, $B'_j = (\xi(B_j))^-$ $(j : 1..n)$, and

$$\phi' = ([\phi \wedge (\exists\, p \in \mathbb{N} \cdot ((0 \leq p < p_1 * 10^d \wedge \phi_1) \vee ... \vee$$
$$((p_1 + ... + p_{l-1}) * 10^d \leq p < (p_1 + ... + p_l) * 10^d \wedge$$
$$\phi_l)))]_\xi)_{[(\xi(A_i))^{\neg}\alpha_i/x_i][(\xi(B_j))^{\neg}\beta_j/y_j]},$$

$[[P\_Schema]^P]_\xi$ is equal to the following type in type theory:

$$\Pi\alpha_1 \in A'_1, ..., \alpha_m \in A'_m \cdot \Sigma\beta_1 \in B'_1, ..., \beta_n \in B'_n \cdot \phi'$$

We can now derive a program from a correctness proof of the above type theoretical specification. An initial part of such a proof is shown in Fig. 3. The extracted program is as follows:

$$prog = \lambda(\alpha_1, ..., \alpha_m) \cdot ((v_1, ..., v_n), q),$$

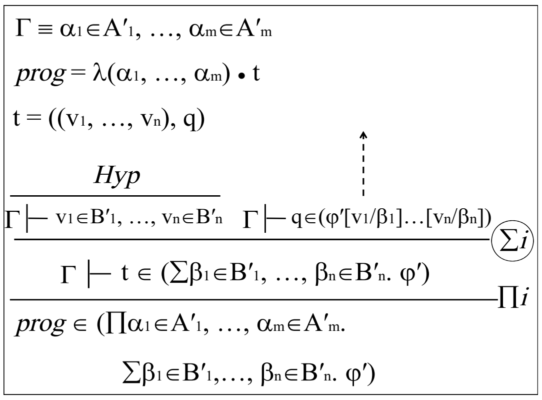
Γ ≡ α1∈A′1, …, αm∈A′m
prog = λ(α1, …, αm) • t
t = ((v1, …, vn), q)
Hyp
Γ ├─ v1∈B′1, …, vn∈B′n   Γ ├─ q∈(φ′[v1/β1]…[vn/βn])
Σi
Γ ├─ t ∈ (∑β1∈B′1, …, βn∈B′n. φ′)
∏i
prog ∈ (∏α1∈A′1, …, αm∈A′m.
∑β1∈B′1,…, βn∈B′n. φ′)

Fig. 3. Program extraction from the probabilistic schema P_Schema.

where q is the proof object of $\phi'[v_1/\beta_1]...[v_n/\beta_n]$. For each valuation of $\alpha_1 \in A'_1, ..., \alpha_m \in A'_m$, this program *always* produces the single n-ary $(v_1, ..., v_n)$. In this way, the probabilistic behavior specified by P_Schema cannot be implemented by $prog$. The origin of this problem can be realized considering the proof tree, specially where we used the introduction rule for dependent sum (Σ_i) (see the circled Σ_i in Fig. 3): according to the definition of Σ_i, we could replace the n-ary $(\beta_1, ..., \beta_n)$ by *only one* of its possible values. Although the problem originates from the rules of type theory, in the next section, we change the given interpretation of probabilistic schemas such that without the need to modify the proof rules of type theory, we will be able to construct functional programs which can preserve the initially specified probabilistic choice.

IV. A New Interpretation of Probabilistic Schemas

As we have shown at the end of the previous section, the current interpretation of probabilistic schemas is not sufficient for the purpose of program construction. Although the problem originates from the inference rules of type theory, we change the current interpretation of probabilistic schemas such that it *explicitly* models *all* possible values of the variable p and also all possible values of the after state and output variables of P_Schema, allowed according to the predicate part of this schema.

In this way, the process of proving correctness is forced to construct a program that involves *all* possible values of p and also all possible values of the after state and output variables; such a program will be able to implement the probabilistic behavior, initially specified by the probabilistic choice between l predicates $\phi_1, ..., \phi_l$. This approach is similar to what we presented in [5] and [6] to specify nondeterminism *explicitly* in Z. The next definition introduces a new function $[]^{NP}$ that interprets probabilistic schemas according to the new idea.

Definition 4.1 Recall P_Schema, given in definition 3.1 as the general form of probabilistic schemas. If for all real numbers $p_1, ..., p_l$, the maximum number of digits to the right of the decimal point is d, we have:

if P_Schema *is an ordinary operation schema (i.e., when* $l = 0$*), then* $[P\_Schema]^{NP} = P\_Schema$;
otherwise, $[P\_Schema]^{NP} \cong [x_1 \in A_1;\ ...;\ x_m \in A_m;$
$pvar \in seq(B_1 \times ... \times B_n \times \mathbb{N})\ |$
$\forall (y_1, ..., y_n, p) \in (B_1 \times ... \times B_n \times \mathbb{N})\cdot$
$(y_1, ..., y_n, p) \in pvar \Leftrightarrow \psi]$,
where $\psi \equiv \phi \wedge ((0 \leq p < p_1 * 10^d \wedge \phi_1)\vee$
$(p_1 * 10^d \leq p < (p_1 + p_2) * 10^d \wedge \phi_2) \vee ...\vee$
$((p_1 + ... + p_{l-1}) * 10^d \leq p < (p_1 + ... + p_l) * 10^d \wedge \phi_l))$

Like $[]^P$, the function $[]^{NP}$ behaves as an identity function when applied to an ordinary operation schema, i.e., when $l = 0$; otherwise, it promotes the *combination* of the after state and output variables and an auxiliary variable $p \in \mathbb{N}$ to a *sequence* $pvar$ of *all* possible combinations of these variables that satisfy the predicates of the schema. We have combined all of the above mentioned variables using the cartesian product of their types in order to preserve the relationship between them after the interpretation.

Theorem 4.2 Assume that for every predicate $\phi_k (k : 1..l)$ existing in the predicate part of P_Schema, each combination of values of before state and input variables with one and only one combination of values of after state and output variables satisfies ϕ_k. A program extracted from the correctness proof of the type theoretical counterpart of $[P\_Schema]^{NP}$ can implement the probabilistic behavior specified by P_Schema.

Proof. According to the predicate part of $[P\_Schema]^{NP}$, a program satisfies $[P\_Schema]^{NP}$ iff when applied to a combination of input values, it produces a sequence consisting of all allowable values of $y_1, ..., y_n, p$ and not anything else. Therefore, any formal program development method that is sound (such as the method of extracting programs from the correctness proofs of the type theoretical counterparts of Z specifications; see the soundness theorem in [10]) is forced to extract a program from $[P\_Schema]^{NP}$ that for each combination of input values, produces a sequence consisting of all possible values of $y_1, ..., y_n, p$ and not anything else. On the other hand, considering the assumption stated by the theorem statement, the resulting sequence includes 10^d elements from which $p_k * 10^d$ $(k : 1..l)$ elements implement the behavior specified by ϕ_k. Therefore, if we make a uniform choice over the elements of this sequence, we will be provided with a correct implementation of the probabilistic behavior, initially specified by P_Schema. $\triangle$

In the next example, we show the application of the function $[]^{NP}$ to the probabilistic schema $P\_FlyMove$, given in example 3.2.

Example 4.3 We use the function $[]^{NP}$ to translate the probabilistic schema $P\_FlyMove$, given in example 3.2, into an ordinary operation schema of Z:

$[P\_FlyMove]^{NP} \cong [x?, m? \in \mathbb{N};\ pvar \in seq(\mathbb{N} \times \mathbb{N})\ |$
$\forall (y!, p) \in (\mathbb{N} \times \mathbb{N})\cdot$
$(y!, p) \in pvar \Leftrightarrow (m? > 2 \wedge x? > 1 \wedge x? < m?\wedge$
$((0 \leq p < 3 \wedge y! = x? - 1)\vee$
$(3 \leq p < 6 \wedge y! = x? + 1) \vee (6 \leq p < 10 \wedge y! = x?)))]$

If we apply the function ξ (see subsection 2.2) to the above resulting schema, the following type theoretical specification is obtained:

$\Pi(\alpha,\beta) \in (\mathbb{N} \otimes \mathbb{N}) \cdot \Sigma\gamma \in List^1\ (\mathbb{N} \otimes \mathbb{N})\cdot$
$\Pi(\tau,\delta) \in (\mathbb{N} \otimes \mathbb{N})\cdot$
$(\tau,\delta)\dot{\in}\gamma \Leftrightarrow (\beta\dot{>}2 \otimes \alpha\dot{>}1 \otimes \alpha\dot{<}\beta \otimes ((0\dot{\leq}\delta\dot{<}3 \otimes \tau\doteq\alpha-1)\oplus$
$(3\dot{\leq}\delta\dot{<}6 \otimes \tau\doteq\alpha+1) \oplus (6\dot{\leq}\delta\dot{<}10 \otimes \tau\doteq\alpha))),$

where α, β, γ, τ, and δ correspond to the variables $x?$, $m?$, $pvar$, $y!$, and p existing in $[P\_FlyMove]^{NP}$, respectively. Due to the space limitation, we do not give the correctness proof of the above type theoretical specification here. Nevertheless, such a proof results in a functional program that for each combination of input values α and β, produces a sequence consisting of all allowable values of τ and δ. For example, for $\alpha = 2$ and $\beta = 4$, this program produces the sequence

$< (1,0),(1,1),(1,2),(3,3),(3,4),(3,5),(2,6),(2,7),$
$(2,8),(2,9) >$

Selecting any of the first three elements of the above sequence results in $\tau = 1$ which means that the fly moves one unit to the left. Similarly, selecting any of the three elements (3,3),(3,4) and (3,5) results in $\tau = 3$ which means that the fly moves one unit to the right. Finally, selecting any of the four elements (2,6),(2,7),(2,8) and (2,9) results in $\tau = 2$ which means that the fly stays in place. Now, we can assume that we have a programming construct in the final functional language which *uniformly* chooses between the elements existing in the above sequence. For instance, we can use a *functional* random number generator, similar to what has been introduced in [16], to make a *uniform* choice over alternatives existing in the sequence. Making such a choice, it is guaranteed that the fly will move one unit to the left with probability 0.3, one unit to the right with probability 0.3, and stays in place with probability 0.4. This behavior corresponds to the probabilistic choice specified initially by $P\_FlyMove$. $\triangle$

Although using the interpretation function $[]^{NP}$ leads to programs that can implement the probabilistic behavior, this function suffers from a main drawback: it only works correctly when applied to a probabilistic schema

$P\_Schema \cong [x_1 \in A_1;\ ...;\ x_m \in A_m;$
$y_1 \in B_1;\ ...;\ y_n \in B_n \mid \phi \wedge (p_1 : \phi_1;\ ...;\ p_l : \phi_l)]$

that obeys the following law:

for every predicate $\phi_k(k : 1..l)$, each combination of values of before state and input variables with one and only one combination of values of after state and output variables satisfies ϕ_k.

Notice that the above law is really what has been explicitly assumed in theorem 4.2. For instance, consider the following probabilistic schema:

$P\_GetLEQ \cong [x? \in \mathbb{N};\ y! \in \mathbb{N} \mid$
$0.5 : y! < x?;\ 0.5 : y! = x?]$

P_GetLEQ does not obey the above mentioned law: we can find a value of the input variable $x?$, such as 2 or 3, that with more than one value of the output variable $y!$ satisfies the predicate $y! < x?$. Now, applying the function $[]^{NP}$ to P_GetLEQ results in the following schema:

[1]In [10], it has been shown that $seqX$ of CZ is equivalent to $List(X)$ of type theory.

$[P\_GetLEQ]^{NP} \cong [x? \in \mathbb{N};\ pvar \in seq(\mathbb{N} \times \mathbb{N}) \mid$
$\forall(y!, p!) \in (\mathbb{N} \times \mathbb{N}) \cdot (y!, p!) \in pvar \Leftrightarrow$
$((0 \leq p! < 5 \wedge y! < x?) \vee (5 \leq p! < 10 \wedge y! = x?))]$

A program satisfying the above obtained specification is not what the initial schema, i.e., P_GetLEQ, specifies. For example, for the input value 2, such a program produces the sequence

$< (0,0),(1,0),(0,1),(1,1),(0,2),(1,2),(0,3),(1,3),(0,4),$
$(1,4),(2,5),(2,6),(2,7),(2,8),(2,9) >$

Selecting any of the first 10 elements of the above sequence results in an output value less than 2. In other words, a uniform choice over the elements of this sequence selects an output value less than 2 with probability $\frac{2}{3}$. It also selects the output value 2 with probability $\frac{1}{3}$; however, who wrote the initial specification wants both the above sorts of output to be produced with the same probability. This problem is due to the nondeterministic relationship between possible values of $x?$ and $y!$ allowed by the predicate $y! < x?$ in P_GetLEQ. A similar problem occurs when a probabilistic schema involves a predicate $\phi_k(k : 1..l)$ that is unsatisfiable for a combination of values of before state and input variables. For instance, consider the following probabilistic schema by which we specify an operation that for each input value x, produces $x+1$ with probability 0.5 and produces $x-1$ with probability 0.5:

$P\_GetAdj \cong [x? \in \mathbb{N};\ y! \in \mathbb{N} \mid$
$0.5 : y! = x? + 1;\ 0.5 : y! = x? - 1]$

Notice that for the input value $x? = 0$, there exists no value for the output variable $y!$ which satisfies the predicate $y! = x? - 1$. Now, using $[]^{NP}$ to interpret P_GetAdj results in a program that for the input value $x? = 0$, produces the sequence $< (1,0),(1,1),(1,2),(1,3),(1,4) >$. A uniform choice over the elements of this sequence *always* (with probability 1) results in the value 1 for the output variable $y!$ while the specification writer wants the program to produce the output value 1 with probability 0.5 and *aborts* (without producing anything) with probability 0.5. Therefore, we have again obtained a program that does not satisfy the initial specification. Introducing a new interpretation of probabilistic schemas that solves the above mentioned problem can be an interesting topic in continuing this work.

We have so far proposed to use probabilistic schemas instead of ordinary operation schemas in order to specify probabilistic programs in the PZ notation. A distinctive feature of Z is its *schema calculus operations*. In the next section, we show these operations will no longer work correctly in the presence of probabilistic schemas. We thus introduce a new set of schema calculus operations into PZ that can be applied to probabilistic schemas as well as ordinary operation schemas.

V. A Calculus for Probabilistic Schemas

We first investigate whether we can use the operations of the Z schema calculus to manipulate probabilistic schemas. It seems that a simple way to do this is to transform probabilistic schemas into ordinary ones (using the function $[]^{NP}$) before applying the schema calculus operations of Z; in this way, we will have ordinary operation schemas that can be manipulated by the Z schema calculus operations in the standard way. However, we show that this approach may result in *unwanted* specifications; it even may make the applications of operations to schemas *undefined*. For instance, consider the probabilistic schema $P\_FlyMove$, given in example 3.2. This schema specifies a *partial* operation [18] since the effect of the operation is undefined for some input values, i.e., when $m? \leq 2$, $x? \leq 1$, or $x? \geq m?$. To describe a *total* operation, we give a new specification:

$Res ::= OK \mid Not\_Move$
$P\_P\_FlyMove \cong [x?, m?, y! \in \mathbb{N};\ r! \in Res \mid$
$m? > 2 \land x? > 1 \land x? < m? \land r! = OK \land$
$(0.3 : y! = x? - 1;\ 0.3 : y! = x? + 1;\ 0.4 : y! = x?)]$
$Exception \cong [x?, m?, y! \in \mathbb{N};\ r! \in Res \mid$
$\neg(m? > 2 \land x? > 1 \land x? < m?) \land$
$r! = Not\_Move \land y! = x?]$

Now, we can describe a total operation by applying a disjunction between two schemas $P\_P\_FlyMove$ and $Exception$ above. Before doing this, however, we first translate $P\_P\_FlyMove$ into an ordinary operation schema as follows:

$[P\_P\_FlyMove]^{NP} \cong [x?, m? \in \mathbb{N};$
$pvar \in seq(\mathbb{N} \times Res \times \mathbb{N}) \mid$
$\forall (y!, r!, p) \in (\mathbb{N} \times Res \times \mathbb{N}) \cdot (y!, r!, p) \in pvar \Leftrightarrow$
$(m? > 2 \land x? > 1 \land x? < m? \land r! = OK \land$
$((0 \leq p < 3 \land y! = x? - 1) \lor (3 \leq p < 6 \land y! = x? + 1) \lor$
$(6 \leq p < 10 \land y! = x?)))]$

Two schemas $[P\_P\_FlyMove]^{NP}$ and $Exception$ are *type compatible*, i.e., each variable common to two schemas has the same type in both of them [18]. Thus, we can apply the operator $\lor$ to these schemas. However, in the resulting schema, there is no relationship between the variables $y!$ and $r!$ coming from $Exception$ and the sequence $pvar$ coming from $[P\_P\_FlyMove]^{NP}$ whereas all the elements of $pvar$ involve instances of $y!$ and $r!$ as their first and second components, respectively. This problem originates from using the function $[]^{NP}$ that forces the output variables $y!$ and $r!$ existing in $P\_P\_FlyMove$ to be combined into a new variable, and the resulting variable to be promoted to a sequence.

Interpreting probabilistic schemas before applying the schema calculus operations may even yield undefined operations. For instance, suppose that we use $\exists y! \in \mathbb{N} \cdot P\_P\_FlyMove$ to hide $y!$ in the resulting schema. If we use the function $[]^{NP}$ to interpret $P\_P\_FlyMove$ before applying the existential quantifier, we lose $y!$ since it is combined with some other schema variables and then promoted to a sequence; in this way, the quantification over $y!$ becomes undefined.

Similar problems occur when we transform probabilistic schemas into ordinary ones before applying the other schema calculus operations, such as conjunction, universal quantifier, and sequential composition: By using $[]^{NP}$ to interpret probabilistic schemas, the relationship between instances of a variable that exist in the declaration part of various schemas (or exist in the list of quantified variables and the declaration part of the quantified schema when applying universal or existential quantifier) may be lost; hence, applying schema calculus operations to the resulting schemas may be undefined or result in unwanted specifications.

Unfortunately, another problem will occur if we try the reverse path, i.e., applying the schema calculus operations to probabilistic schemas before interpreting them by $[]^{NP}$. For instance, suppose that we apply the operator $\lor$ to the schemas $P\_P\_FlyMove$ and $Exception$ before interpreting $P\_P\_FlyMove$:

$P\_T\_FlyMove \cong P\_P\_FlyMove \lor Exception \cong$
$[x?, m?, y! \in \mathbb{N};\ r! \in Res \mid (m? > 2 \land x? > 1 \land x? < m? \land$
$r! = OK \land$
$(0.3 : y! = x? - 1;\ 0.3 : y! = x? + 1;\ 0.4 : y! = x?)) \lor$
$(\neg(m? > 2 \land x? > 1 \land x? < m?) \land$
$r! = Not\_Move \land y! = x?)]$

$P\_T\_FlyMove$ does not correspond to the general form of probabilistic schemas (see definition 3.1). Therefore, we are not allowed to apply the function $[]^{NP}$ to interpret $P\_T\_FlyMove$. It seems that we can solve this problem by manually transforming the resulting schema into the general form of probabilistic schemas or even changing the definition of $[]^{NP}$ to cover schemas such as $P\_T\_FlyMove$; however, having such a method in mind, in various situations we encounter various cases for each of which we must provide a specially manual way. For instance, consider the following simple specification:

$[y! \in \mathbb{N} \mid 0.5 : y! = 0;\ 0.5 : y! = 1] \lor$
$[y! \in \mathbb{N} \mid 0.5 : y! = 2;\ 0.5 : y! = 3]$

The above disjunction results in the following schema:

$S_1 \cong [y! \in \mathbb{N} \mid (0.5 : y! = 0;\ 0.5 : y! = 1) \lor$
$(0.5 : y! = 2;\ 0.5 : y! = 3)]$

By the above specification, we are in fact interested in the following schema:

$S_2 \cong [y! \in \mathbb{N} \mid 0.25 : y! = 0;\ 0.25 : y! = 1;$
$0.25 : y! = 2;\ 0.25 : y! = 3]$

However, transforming manually S_1 into S_2 requires extra analysis that becomes more complicated when we are to handle more complex probabilistic schemas.

We have so far seen that both of the mentioned possible paths (interpreting probabilistic schemas before applying the schema calculus operations or the reverse path) to employ the operations of the Z schema calculus in PZ do not work when we want to manipulate probabilistic schemas. Now, we present a usable approach in which the application of the schema calculus operations and the interpretation of probabilistic schemas occur in an interleaved manner. Suppose that the interpretation function $[]^{NP}$ operates in a two-step process. More precisely, suppose that $[]^{NP}$ is equivalent to the composition of two functions $[]^{NP_1}$ and $[]^{NP_2}$; the former approximately behaves like the function $[]^{P}$ introduced in section 3 (see definition 3.3); unlike $[]^{P}$, $[]^{NP_1}$ introduces the variable p into the declaration part of the schema. Here is the formal definition of $[]^{NP_1}$:

Definition 5.1 Recall P_Schema, given in definition 3.1 as the general form of probabilistic schemas. Also assume that for all real numbers $p_1, ..., p_l$, the maximum number of digits to the right of the decimal point is d. Thus we have:

if P_Schema is an ordinary operation schema, then
$[P\_Schema]^{NP_1} = P\_Schema$;
otherwise, $[P\_Schema]^{NP_1} \cong$
$[x_1 \in A_1;\ ...;\ x_m \in A_m;\ y_1 \in B_1;\ ...;\ y_n \in B_n;$

$p! \in \&\mathbb{N} \mid \phi \wedge ((0 \leq p! < p_1 * 10^d \wedge \phi_1) \vee$
$(p_1 * 10^d \leq p! < (p_1 + p_2) * 10^d \wedge \phi_2) \vee ... \vee$
$((p_1 + ... + p_{l-1}) * 10^d \leq p! < (p_1 + ... + p_l) * 10^d \wedge \phi_l))]$

In definition 5.1, we have used the symbol $\&$ in the declaration of $p!$ in order to be able to distinguish between probabilistic schemas and ordinary operation schemas when we want to apply the function $[]^{NP_2}$. The function $[]^{NP_2}$ takes a schema and promotes the combination of its output and after state variables to a sequence, provided that it includes an output variable declared by the symbol $\&$. In the following, we formalize the definition of $[]^{NP_2}$:

Definition 5.2 Suppose that $[]^{NP_2}$ applies to the following operation schema:

$Op\_Schema \cong [x_1 \in A_1;\ ...;\ x_m \in A_m;$
$y_1 \in B_1;\ ...;\ y_n \in B_n \mid \phi]$,

where $x_i (i : 1..m)$ are input or before state variables, and $y_j (j : 1..n)$ are output or after state variables. Now, we have:

if OP_Schema has no output variable declared by $\&$, then $[OP\_Schema]^{NP_2} = OP\_Schema$;
otherwise, $[OP\_Schema]^{NP_2} \cong [x_1 \in A_1;\ ...;\ x_m \in A_m;$
$pvar \in seq(B_1 \times ... \times B_n) \mid$
$\forall (y_1, ..., y_n) \in (B_1 \times ... \times B_n) \cdot (y_1, ..., y_n) \in pvar \Leftrightarrow \phi]$

It can be easily justified that $[]^{NP} = [[]^{NP_1}]^{NP_2}$. Now, to manipulate probabilistic schemas by the operations of the Z schema calculus, we propose to apply these operations between the applications of $[]^{NP_1}$ and $[]^{NP_2}$. An informal illustration of the correctness of this approach is as follows: $[]^{NP_1}$ transforms a probabilistic schema into an ordinary one according to the probabilities involved in its predicate part; however, $[]^{NP_1}$ do not promote the combination of the output and after state variables to a sequence. Therefore, we can apply the operations of the Z schema calculus to the resulting schema; this does not yield unwanted specifications or undefined operations. At the final stage, we apply $[]^{NP_2}$ to the resulting schema in order to guarantee that the final program can implement the initially specified probabilistic behavior.

To implement the above idea, we introduce a new set of schema calculus operations into PZ that can be applied to probabilistic schemas appropriately. In the Z notation [15], [18], there exist operators $\neg$, $\wedge$, $\vee$, $\exists$, $\forall$, and $\fatsemi$ for the schema calculus operations *negation*, *conjunction*, *disjunction*, *existential quantifier*, *universal quantifier*, and *sequential composition*, respectively. Here, we define a new set of operators consisting of $\neg_p$, $\wedge_p$, $\vee_p$, $\exists_p$, $\forall_p$, and $\fatsemi_p$ instead of the previous ones:

Definition 5.3 Let PS_1 and PS_2 be two probabilistic schemas. Now, we have:

$\neg PS_1 \cong [\neg [PS_1]^{NP_1}]^{NP_2}$
$PS_1 \wp_p PS_2 \cong [([PS_1]^{NP_1} \wp [PS_2]^{NP_1})]^{NP_2} \qquad \wp \in \{\wedge, \vee, \fatsemi\}$
$\varrho_p d_h \cdot PS_1 \cong [\varrho d_h \cdot [PS_1]^{NP_1}]^{NP_2} \qquad \varrho \in \{\exists, \forall\}$,

where d_h is the declaration of quantified variables.

To show the usability of the new operations of the schema calculus, we apply $\vee_p$ to the schemas $P\_P\_FlyMove$ and $Exception$, introduced earlier in this section. By this example, we also show that in the case of disjunction between a probabilistic schema and an ordinary one, we must apply a slight change to the ordinary schema after using $[]^{NP_1}$ and before using $\vee$:

$P\_P\_FlyMove \vee_p Exception \cong$
$[([P\_P\_FlyMove]^{NP_1} \vee [Exception]^{NP_1})]^{NP_2} \cong$
$[x?, m?, pvar \in (\mathbb{N} \times Res \times \mathbb{N}) \mid$
$\forall (y!, r!, p!) \in (\mathbb{N} \times Res \times \mathbb{N}) \cdot (y!, r!, p!) \in pvar \Leftrightarrow$
$((m? > 2 \wedge x? > 1 \wedge x? < m? \wedge r! = OK \wedge$
$((0 \leq p! < 3 \wedge y! = x? - 1) \vee (3 \leq p! < 6 \wedge y! = x? + 1) \vee$
$(6 \leq p! < 10 \wedge y! = x?))) \vee$
$(\neg (m? > 2 \wedge x? > 1 \wedge x? < m?) \wedge r! = Not\_Move \wedge$
$y! = x?))]$

The above resulting schema specifies a total operation. When $m? > 2 \wedge x? > 1 \wedge x? < m?$, this operation produces a sequence consisting of all allowable values of $y!$ and $p!$ and also reports OK. When $m? \leq 2 \vee x? \leq 1 \vee x? \geq m?$, the operation assigns the value $x?$ to $y!$ and also reports Not_Move; however, the possible values of $p!$ has not been determined for this case. In other words, $p!$ can take any natural number; it violates producing a *finite* sequence for $pvar$. To solve this problem, it is enough to introduce $p!$ into the declaration part of $Exception$ and add a conjunct such as $p! = 0$, limiting the possible values of $p!$, into the predicate part of $Exception$ before using $\vee$ between $P\_P\_FlyMove$ and $Exception$. By this modification, the following schema is obtained finally:

$[x?, m?, pvar \in (\mathbb{N} \times Res \times \mathbb{N}) \mid$
$\forall (y!, r!, p!) \in (\mathbb{N} \times Res \times \mathbb{N}) \cdot (y!, r!, p!) \in pvar \Leftrightarrow$
$((m? > 2 \wedge x? > 1 \wedge x? < m? \wedge r! = OK \wedge$
$((0 \leq p! < 3 \wedge y! = x? - 1) \vee (3 \leq p! < 6 \wedge y! = x? + 1) \vee$
$(6 \leq p! < 10 \wedge y! = x?))) \vee$
$(\neg (m? > 2 \wedge x? > 1 \wedge x? < m?) \wedge r! = Not\_Move \wedge$
$y! = x? \wedge p! = 0))]$

The new schema specifies an operation that for $m? \leq 2 \vee x? \leq 1 \vee x? \geq m?$, produces the sequence $< (x?, Not\_Move, 0) >$. Notice that the recent modification is not required when we use conjunction or sequential composition operators between a probabilistic schema and an ordinary one since in these cases, we apply a conjunction between the predicate parts of two schemas; this automatically limits the possible values of $p!$.

VI. Conclusions and Future Work

In this paper, we have presented a new Z-based formalism, called PZ, by which one can build set theoretical models of probabilistic programs. We have also reviewed a *constructive* approach for formal program development that is well integrated with PZ: since we have interpreted all the new constructs of PZ in Z itself, we can still use the translation of the CZ set theory into type theory [10] to derive functional programs from correctness proofs of our PZ specifications of probabilistic programs. In this way, we are provided with a completely constructive framework for developing probabilistic programs formally (see Fig. 1).

However, the current framework suffers from following drawbacks/limitations in the program development stage:

1) We have shown that using the interpretation function $[]^{NP}$ can lead to appropriate programs provided that this function is applied to those probabilistic schemas

$P\_Schema \cong [x_1 \in A_1;\ ...;\ x_m \in A_m;$
$y_1 \in B_1;\ ...;\ y_n \in B_n \mid \phi \wedge (p_1 : \phi_1;\ ...;\ p_l : \phi_l)]$
that obey the following law:

for every predicate $\phi_k(k : 1..l)$, each combination of values of before state and input variables with one and only one combination of values of after state and output variables satisfies ϕ_k.

Introducing a new interpretation of probabilistic schemas that releases this limitation can be an interesting topic in continuing this work.

2) Programs obtained after applying our constructive framework to PZ specifications are not really probabilistic programs. Instead, for each input setting, these programs provide a sequence; now, a uniform choice over the elements of the resulting sequence implements the probabilistic behavior. To overcome this drawback and as a direction for future work, one can extend the interpretation of CZ in type theory and add new inference rules into type theory in order to provide a way to map probabilistic constructs of PZ into probabilistic choice constructs of a probabilistic functional language.

As we have stated in section 1, much of the work in the literature related to probabilistic programs has focused on developing and verifying such programs in imperative settings. Specially, in [9], [11], and [12], Morgan et al. have proposed an imperative programming language, called *pGCL*, with which probabilistic programs can be rigorously developed and verified. On the other hand, there are some attempts to translate Z specifications into specifications in refinement calculi; for example, see [2] and [17].

As another direction for future research, one can extend an existing translation of Z in a refinement calculus in order to map probabilistic schemas of PZ, introduced in this paper, to their equivalent specification statements in the refinement calculus. By introducing new refinement rules that refine new specification statements into probabilistic constructs of an imperative language, such as *pGCL*, we will be provided with a *refinement* approach, rather than a *constructive* one, to formally develop *imperative*, probabilistic programs, rather than *functional* ones, from their set theoretical specifications.

REFERENCES

[1] Audebaud, P., Paulin-Mohring, C.: Proofs of Randomized Algorithms in Coq. In: MPC 2006, LNCS, vol. 4014, pp. 49-68 (2006)
[2] King, S.: Z and the Refinement Calculus. In: VDM'90, LNCS 428, Springer-Verlag, pp. 164–188 (1990)
[3] Kozen, D.: Semantics of Probabilistic Programs. Journal of Computer and System Sciences, pp. 328–350 (1981)
[4] Haghighi, H., Mirian-Hosseinabadi, S.H.: An Approach to Nondeterminism in Translation of CZ Set Theory into Type Theory. In: FSEN 2005, ENTCS 159 (2006)
[5] Haghighi, H., Mirian-Hosseinabadi, S.H.: Nondeterminism in Constructive Z. Fundamenta Informaticae, vol. 88 (1-2), pp. 109–134 (2008)
[6] Haghighi, H.: Nondeterminism in CZ Specification Language. Ph.D. dissertation, Sharif Univ. of Technology, Iran, (2009)
[7] Martin-Löf, P.: An Intuitionistic Theory of Types: Predicative Part. (H.E. Rose, J.C. Sheperdson, Eds.), North Holland, pp. 73–118 (1975)
[8] McIver, A., Morgan, C.: Abstraction and Refinement in Probabilistic Systems. ACM SIGMETRICS Performance Evaluation Review, vol. 32 , no. 4, pp. 41–47 (2005)
[9] McIver, A., Morgan, C.: Developing and Reasoning About Probabilistic Programs in pGCL. LECTURE NOTES IN COMPUTER SCIENCE, pp. 123–155 (2006)
[10] Mirian-Hosseinabadi, S.H.: Constructive Z. Ph.D. dissertation, Essex Univ. (1997)
[11] Morgan, C., McIver, A.: pGCL: Formal Reasoning for Random Algorithms. Southern African Computer Journal (1999)
[12] Morgan, C., McIver, A., Hurd, J.: Probabilistic Guarded Commands Mechanised in HOL. Theoretical Computer Science, pp. 96–112 (2005)
[13] Nordstrom, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf's Type Theory: An Introduction. Oxford University Press (1990)
[14] Park, S., Pfenning, F., Thrun, S.: A Probabilistic Language Based Upon Sampling Functions. In: ACM Symp. on Principles of Prog. Lang., pp. 171–182 (2005)
[15] Spivey, J.M.: The Z Notation: A Reference Manual. Prentice Hall (1989)
[16] Thompson, S.: Haskell: The Craft of Functional Programming, 2nd ed. Addison-Wesley (1999)
[17] Woodcock, J.: An Introduction to Refinement in Z. In: VDM'91, LNCS 552, Springer-Verlag, vol. 2, pp. 96–117 (1991)
[18] Woodcock, J. and Davies, J.: Using Z, Specifications, Refinement and Proof. Prentice Hall (1996)