A UTP semantics of pGCL as a homogeneous relation

Riccardo Bresciani and Andrew Butterfield[⋆]

Foundations and Methods Group,
Trinity College Dublin,
Dublin, Ireland
{bresciar,butrfeld}@scss.tcd.ie

Abstract. We present an encoding of the semantics of the probabilistic guarded command language (pGCL) in the Unifying Theories of Programming (UTP) framework. Our contribution is a UTP encoding that captures pGCL programs as predicate-transformers, on predicates over probability distributions on before- and after-states: these predicates capture the same information as the models traditionally used to give semantics to pGCL; in addition our formulation allows us to define a generic choice construct, that covers conditional, probabilistic and non-deterministic choice. As an example we study the Monty Hall game in this framework.

1 Introduction

The Unifying Theories of Programming (UTP) research activity seeks to bring models of a wide range of programming and specification languages under a single semantic framework in order to be able to reason formally about their integration [HJ98,DS06,But10,Qin10]. A success in this area has been the development of the *Circus* language [OCW09], which is a fusion of Z and CSP, with a UTP semantics, providing specifications using a "state-rich" process algebra along with a refinement calculus; recent extensions to *Circus* have included timed [SH03] and synchronous [GB09] variants. Recent interest in aspects of the POSIX filestore case study in the Verification Grand Challenge [FWB08] has led us to consider integrating probability into UTP, with a view to eventually having a probabilistic variant of *Circus*.

UTP is based on (state-)predicate transformers, whereas probabilistic models typically involve distributions over states, and so the best way to integrate probability into the UTP framework is not obvious. This paper presents first steps in constructing a theory of probabilistic programs that is expressed using predicate-transformers[1]. The focus here is on a UTP theory that captures the

[⋆] The present work has emanated from research supported by Science Foundation Ireland grant 08/RFP/CMS1277 and, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre.

[1] So probabilistic programs are predicates too (with apologies to C.A.R. Hoare [Hoa85]).

semantics of the probabilistic guarded command language (pGCL) [MM04], by means of predicates involving a homogeneous relation among distributions over states.

This paper is structured as follows: we describe the background to both UTP and pGCL (§2); discuss the motivation for and technical details of our observable variables (§3); give the semantics of pGCL in our framework (§4); and conclude (§5).

2 Background

2.1 UTP

UTP follows the key principle that "programs are predicates" [Hoa85]: theories in UTP are expressed as second-order predicates over a pre-defined collection of free observation variables, referred to as the *alphabet* of the theory. The predicates are generally used to describe a relation between a before-state and an after-state, the latter typically characterised by dashed versions of the observation variables. For example, a program using two variables `x` and `y` might be characterised by having the set $\{x, x', y, y'\}$ as an alphabet, and the meaning of the assignment `x := y+3` would be described by the predicate

$$x' = y + 3 \wedge y' = y.$$

In effect UTP uses predicate calculus in a disciplined way to build up a relational calculus for reasoning about programs.

In addition to observations of the values of program variables, often we need to introduce observations of other aspects of program execution via so-called auxiliary variables. So, for example, in order to reason about total correctness, we need to introduce boolean observations that record the starting (ok) and termination (ok') of a program, resulting in the above assignment having the following semantics:

$$ok \Rightarrow ok' \wedge x' = y + 3 \wedge y' = y$$

(if started, it will terminate, and the final value of `x` will equal the initial value of `y` plus three, with `y` unchanged).

A problem with allowing arbitrary predicate calculus statements to give semantics is that it is possible to write unhelpful predicates such as $\neg ok \Rightarrow ok'$, which describes a "program" that must terminate when not started. In order to avoid assertions that are either nonsense or infeasible, UTP adopts the notion of "healthiness conditions" which are monotonic idempotent predicate transformers whose fixpoints characterise sensible (healthy) predicates. Collections of healthy predicates typically form a sub-lattice of the original predicate lattice under the reverse implication ordering [HJ98, Chp. 3]. Key in UTP is a general notion of program refinement as the universal closure of reverse implication[2]:

$$S \sqsubseteq P \;\widehat{=}\; [P \Rightarrow S]$$

[2] Square brackets denote universal closure, *i.e.* $[P]$ asserts that P is true for all values of its free variables.

$$
\begin{array}{rcl}
\mathtt{wp.abort}.PostE & \mathrel{\hat{=}} & 0 \\
\mathtt{wp.skip}.PostE & \mathrel{\hat{=}} & PostE \\
\mathtt{wp}.(\underline{x} := \underline{e}).PostE & \mathrel{\hat{=}} & PostE[{}^{\underline{e}}\!/\!_{\underline{x}}] \\
\mathtt{wp}.(prog_1 ; prog_2).PostE & \mathrel{\hat{=}} & \mathtt{wp}.prog_1.(\mathtt{wp}.prog_2.PostE) \\
\mathtt{wp}.(prog_1 \lhd c \rhd prog_2).PostE & \mathrel{\hat{=}} & (\mathtt{wp}.prog_1.PostE)|_c + (\mathtt{wp}.prog_2.PostE)|_{\neg c} \\
\mathtt{wp}.(prog_1 \sqcap prog_2).PostE & \mathrel{\hat{=}} & \min\{\mathtt{wp}.prog_1.PostE, \mathtt{wp}.prog_2.PostE\} \\
\mathtt{wp}.(prog_1 \; {}_p\!\oplus prog_2).PostE & \mathrel{\hat{=}} & p \cdot \mathtt{wp}.prog_1.PostE + (1-p) \cdot \mathtt{wp}.prog_2.PostE
\end{array}
$$

Fig. 1. `wp`-semantics of pGCL, adapted from [MM04, p. 26].
Notation: $[{}^{\underline{e}}\!/\!_{\underline{x}}]$ denotes free occurrences of $\underline{x}$ replaced by $\underline{e}$; $|_c$ denotes expectation limited to states satisfying c.

Program P refines S if for all observations (free variables) S holds whenever P does.

The UTP framework also uses Galois connections to link different languages and theories with different alphabets [HJ98, Chp. 4], and often these manifest themselves as further modes of refinement.

2.2 pGCL

pGCL extends GCL with an additional language construct, namely that of probabilistic choice $prog_1 \; {}_p\!\oplus prog_2$, denoting a statement that executes $prog_1$ with probability p, and $prog_2$ with probability $(1 - p)$ [MM97,MM04,MM05,NM10].

In [MM04] pGCL is given a semantics that generalises Dijkstra's weakest pre-condition semantics to what they term a *weakest pre-expectation semantics*.

An expectation is a function that assigns a weight (a non-negative real number) to program states: it is therefore a random variable. An expectation corresponding to a predicate can be defined as a random variable that maps a state to 1 if it satisfies the predicate and to 0 otherwise. Arithmetic operators and relations are extended pointwise to expectations, as is multiplication by a scalar.

If $PostE$ is a (post-)expectation after running program $prog$, then $\mathtt{wp}.prog.PostE$ is the corresponding weakest[3] (pre-)expectation before the program runs: for each state it returns the minimum expected final weight.

The weakest pre-expectation semantics for pGCL is shown in Figure 1. The key features to note in this semantics are that probabilistic choice is the obvious weighting of its alternatives' expectations, whereas demonic choice returns the pointwise minimum.

Non-determinism is crucial in order to define a sensible refinement relation[4]:

$$spec \sqsubseteq prog \;\mathrel{\hat{=}}\; \forall PostE \bullet \mathtt{wp}.spec.PostE \le \mathtt{wp}.prog.PostE$$

[3] One expectation is weaker than another if for all states it returns at most the same weight — it is the $\le$ relation lifted pointwise.

[4] We have definition of refinement that matches that of pGCL, which we do not discuss in this paper

A program *prog* refines a specification *spec* if the minimum expected weight for each state after *prog* has run is at least as much as we would get after *spec* has run.

An alternative model for pGCL is one that sees a program as a function from initial states to sets of probability distributions over the state space [HSM97,MM04]

$$S \to \mathbb{P}(S \to [0,1])$$

Programs with semantics of this form can be sequentially composed using Kleisli composition (See Appendix A), which can be interpreted as lifting the semantic domain to relations between before- and after-distributions $((S \to [0,1]) \leftrightarrow (S \to [0,1]))$ and then using relational composition [MM04, Chp. 5]. It is this form that has formed the basis for most of the prior work encoding pGCL semantics in UTP (see Section 2.3).

2.3 Probabilistic UTP

There has already been a certain amount of work looking at encoding probability in a UTP setting. He and Sanders have presented an approach to unification of probabilistic choice with standard constructs [HS06], and this work provides an example of how the laws of pGCL could be captured in UTP as predicates about program equivalence and refinement. However only an axiomatic semantics was presented, and the laws were justified via a Galois connection to an expectation-based semantic model.

Sanders and Chen then explored an approach that decomposed demonic choice into a combination of pure probabilistic choice and a unary operator that accounted for demonic behaviour [CS09]. There they commented on the lack of a satisfactory UTP theory, where probabilistic and demonic choice coexist.

A probabilistic BPEL-like language has recently been described by He [He10] that gives a UTP-style semantics for a web-based business semantics language. This language is GCL with extra constructs to handle probabilistic choice and compensations and coordination operators, including exception handling. The UTP model that is developed does not relate before- and after-variables of the same type, but instead uses predicates to encode a relationship between an initial state and a final probability distribution over states.

What all the treatments above have in common is that the UTP predicates relate an initial program variable state (σ) to a final probability distribution (δ') over states, so the relation is not homogenous. This complicates the definition of sequential composition (which has to involve some form of Kleisli composition) and also makes building links to homogeneous UTP theories more difficult. The collection of theories surrounding *Circus* are all based on homogeneous relations (before- and after-observations of the same type). This means that all of these theories have uniform definitions of many common language features, such as sequential composition. This is the main motivation for the development of a homogeneous UTP theory of pGCL.

In this paper, we present a UTP encoding of pGCL semantics as a homogenous relation between probability distributions over the set of possible states, relating a before-distribution (δ) to an after-distribution (δ').

3 Observing Distributions

In UTP we usually talk about variables and the values they map to, so a naïve (and quite straightforward) generalization to handle probability would simply consist of mapping variables to distributions over their values, and that would lead our semantic model to be a mapping from variables to value-distributions:

$$Var \to (Val \to [0..1])$$

Although such an easy generalization may look appealing, it fails to give the appropriate semantics. The reason for this is that many properties of interest depend on an "entanglement" among the variables and this is not captured by the above model.

In order to retain all of the necessary information, we have to consider distributions relating entire program states to a corresponding weight, and we have the form:

$$\delta, \delta' : (Var \to Val) \to [0..1]$$

Later on we will see how these can be related to the expectations being transformed by the semantic model of pGCL already described.

This need to bundle all the information regarding program variables into a single observation is not a major constraint. In fact in many presentations of *Circus*-like languages it is often the convention to model program variable values with a single state observation $\sigma : Var \to Val$, and to treat it as a finite map, which simplifies the treatment of alphabets to a considerable degree: our approach here towards pGCL is analogous. For the purposes of this paper, to keep things simple and to allow us to focus on the key concepts, we shall assume that the set of program variables is finite and fixed, and all states are total functions on this variable set.

We now look at some mathematical preliminaries regarding distributions.

Generally speaking we can define a distribution as a function χ mapping states to real numbers[5], and define its *weight* as:

$$\|\chi\| \mathrel{\widehat{=}} \sum_{\sigma \in \mathrm{dom}\,\chi} \chi(\sigma)$$

We will be working with the following two sub-classes:

- a *weighting distribution* π has the property that for every state σ we have $\pi(\sigma) \leq 1$ — we define two particular weighting distributions, ϵ and ι, as the ones mapping every state to 0 and 1 respectively. There is no limit for the distribution weight;

[5] In other words, it is a real-valued random variable — pGCL expectations are therefore distributions with the additional constraint of having only non-negative values.

- a *probability distribution* δ is a weighting distribution with the additional property that $\|\delta\| \leq 1$.

We will use the term *sub-distribution* to refer to a probability distribution where $\|\delta\| < 1$ and the term *full distribution* to refer to a probability distribution where $\|\delta\| = 1$.

Generally speaking, it is possible to operate on distributions by lifting pointwise operators such as addition, multiplication and multiplication by a scalar; analogously we can lift pointwise all traditional relations and functions on real numbers.

In the case of pointwise multiplication, it is interesting to see it as a way of "re-weighting" a distribution: we have a particular interest in the case when one of the operands is a weighting distribution π, as we will use this operation to give semantics to choice constructs. We opt for a postfix notation to write this operation, as this is an effective way of marking when pointwise multiplication happens in the operational flow: for example if we multiply the probability distribution δ by the weighting distribution π, we will write this as $\delta\langle\!\langle\pi\rangle\!\rangle$.

Given a condition (predicate on state) c, we can define the weighting distribution that maps every state where c evaluates to `true` to 1, and every other state to 0. The value of each state can be seen as the boolean value of c in that state multiplied by 1, so we overload the above notation and note this distribution as $\iota\langle\!\langle c\rangle\!\rangle$[6]. In general whenever we have the multiplication of a distribution by $\iota\langle\!\langle c\rangle\!\rangle$, we can use the postfix operator $\langle\!\langle c\rangle\!\rangle$ for short, instead of using $\langle\!\langle\iota\langle\!\langle c\rangle\!\rangle\rangle\!\rangle$.

It is worth pointing out that if we multiply a probability distribution δ by $\iota\langle\!\langle c\rangle\!\rangle$, we obtain a distribution whose weight $\|\delta\langle\!\langle c\rangle\!\rangle\|$ is exactly the probability of being in a state satisfying c.

3.1 Assignment

The challenge we now face is describing how assignment, which is very much oriented towards individual variables, is given a semantics in terms of a distribution that involves complete entanglement of those variables. In effect an assignment statement `x:=e` involves a partial entanglement of variable x with the variables mentioned in e. In general as we build up larger programs using single assignment as the basic component we observe an increasing degree of entanglement, which can often be captured as an appropriate simultaneous assignment, so we shall work at this level here.

Given a simultaneous assignment $\underline{\mathtt{v}}\mathtt{:=}\underline{\mathtt{e}}$, where underlining indicates that we have lists of variables and expressions of the same length, we denote its effect on an initial probability distribution δ by $\delta\{\!\!\{\underline{e}/\underline{v}\}\!\!\}$. The postfix operator $\{\!\!\{\underline{e}/\underline{v}\}\!\!\}$ reflects the modifications introduced by the assignment — the intuition behind this, roughly speaking, is that all states σ where the expression $\underline{e}$ evaluates to the same value $\underline{\mathit{val}} = \mathrm{eval}_\sigma(\underline{e})$ are replaced by a single state $\sigma' = (\underline{v} \mapsto \underline{\mathit{val}})$ that maps to a probability that is the sum of the probabilities of the states it replaces.

$$(\delta\{\!\!\{\underline{e}/\underline{v}\}\!\!\})(\sigma') \ \hat{=}\ \Sigma_{\{\sigma \ \mid\ \sigma'=\sigma \dagger \{\underline{v}\mapsto \mathrm{eval}_\sigma(\underline{e})\}\}}\, \delta(\sigma)$$

[6] If we see c as a predicate, then $\iota\langle\!\langle c\rangle\!\rangle$ is the corresponding expectation.

$$\begin{array}{rcl}
\texttt{abort} & \widehat{=} & \texttt{true} \\
\texttt{skip} & \widehat{=} & \delta' = \delta \\
\underline{x} := \underline{e} & \widehat{=} & \delta' = \delta\{\!|\underline{e}/\underline{x}|\!\} \\
A;B & \widehat{=} & \exists \delta_m \bullet A(\delta, \delta_m) \wedge B(\delta_m, \delta') \\
A \lhd c \rhd B & \widehat{=} & \exists \delta_A, \delta_B \bullet A(\delta\langle\!|c|\!\rangle, \delta_A) \wedge B(\delta\langle\!|\neg c|\!\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
A \;{}_p\!\oplus B & \widehat{=} & \exists \delta_A, \delta_B \bullet A\big(p \cdot \delta, \delta_A\big) \wedge B\big((1-p) \cdot \delta, \delta_B\big) \wedge \delta' = \delta_A + \delta_B
\end{array}$$

Fig. 2. UTP Semantics for the deterministic constructs of pGCL

Here we treat the state as a map, where † denotes map override; this operator essentially implements the concept of "push-forward" used in measure theory, and is therefore a linear operator.

Assignment preserves the overall weight of a probability distribution if $\underline{e}$ can be evaluated in every state, and if not the assignment returns a sub-distribution, where the "missing" weight accounts for the assignment failing on some states (this failure prevents a program from proceeding and causes non-termination).

These are the most significant elements and constructs that characterise our framework: this has been a presentation from a fairly high level, and it should have provided the reader with a working knowledge of the framework; a formal and rigorous definition of the elements presented so far is beyond the scope of this paper and can be found in [BB11], along with some soundness proofs.

4 UTP semantics of pGCL

We are going to express the semantics of pGCL in UTP using predicates based on a homogeneous relation among probability distributions: we will see programs as *distribution-transformers*, as they change a before-distribution δ into an after-distribution δ'.

This semantics can be related to the relational semantics and the `wp`-semantics of pGCL. [BB11]

4.1 Deterministic constructs

The semantic definitions for all deterministic constructs of pGCL are listed in Figure 2 and we will now proceed to discuss each one.

The failing program `abort` is represented by the predicate `true`, which captures the fact that it is maximally unpredictable. Program `skip` makes no changes and immediately terminates.

Assignment $\underline{x} := \underline{e}$ remaps the distribution as has already been discussed in the previous section 3.1.

Sequential composition $A;B$ is characterised by the existence of a "mid-point" distribution that is the outcome of the first program, and is then fed into the second.

We characterise conditional choice $A \lhd c \rhd B$ by using the condition (and its negation) to filter the left- and right-hand programs appropriately, and we simply sum the (now effectively disjoint) distributions. Probabilistic choice $A\ {}_p\!\oplus$ B simply uses the probability and its complement to scale the distributions for merge — this definition preserves all usual properties. In effect the predicate is only satisfied by any combination of left and right distributions that is pointwise larger than the minimum of both.

It is possible to build an isomorphism to relate the semantics of deterministic constructs described so far to the semantics proposed by Kozen [Koz81,Koz85] for probabilistic programs.

4.2 Non-deterministic choice

We are now going to address non-determinism. According to the relational semantics of pGCL from [HSM97,MM04], which sees programs as relations from a state σ to a probability distribution, we have that[7]

$$(A \sqcap B).\sigma = \cup_{p\in[0..1]}(A\ {}_p\!\oplus B).\sigma$$

If a demonic choice is performed on a state, the set of resulting distributions is that containing all possible distributions resulting from a probabilistic choice with probability p varying in the range $[0..1]$.

Seeing this, one could (reasonably?) expect the following definition for non-deterministic choice in our framework:

$$A \sqcap B \overset{?}{=} \exists p \bullet A\ {}_p\!\oplus B$$

However this definition does not work. In particular, with the above definition, we can prove the following (which is most definitely not a law of pGCL) :

$$(A \sqcap B); (C\ {}_p\!\oplus D) = (C\ {}_p\!\oplus D); (A \sqcap B) \qquad (!?)$$

It describes a demonic choice that is both history-aware, and *prescient*, and this latter ability to look into the future is undesirable, and infeasible.

The key point to note is that the first statement is talking about the possible resulting distributions starting from one single state, whereas this last definition considers all possible starting states. As a result the set of after-distributions that satisfy this definition of demonic choice (for a given before-distribution) is strictly smaller then the set of after-distributions satisfying the first statement. We can easily see this by considering that if we take the Kleisli lifting of $(A \sqcap B).\sigma$ for σ ranging over the whole state space. We obtain some after-distributions which are the result of composing programs where p is not constrained to be constant over all states, and these cases are ruled out in the proposed definition by the single quantification of p valid for all states.

The solution is therefore to take a weighting distribution π, use it with its complementary distribution $\bar{\pi} = \iota - \pi$) to weight the distributions resulting from the left- and right-hand side respectively, and existentially quantify it:

$$A \sqcap B \ \hat{=}\ \exists \pi, \delta_A, \delta_B \bullet A(\delta\langle\pi\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B$$

[7] Here we are using the point notation for function application, as in [MM04].

In this way π can range over the set of weighting distributions, and the set of after-distributions satisfying this second definition coincides with the set obtainable via the Kleisli lifting mentioned above.

A few more comments: usually we talk about demonic non-determinism when we are expecting the worst-case behaviour, to model something that behaves "as bad as it can" for any desired outcome, nevertheless our definition of non-deterministic choice *per se* mandates no such behaviour: depending on the context where it is used (*e.g.* in a framework where refinement is defined in a similar way as for pGCL), this behaviour shows up but it is not intrinsic to the definition — from this perspective we have a similar situation as in the relational model of [HSM97,MM04].

We can see that non-determinism yields a many-to-many relation: a program can be seen as a relation that associates probability before-distributions with non-disjoint sets of probability after-distributions.

The non-deterministic choice operator is idempotent according to our definition, in accordance with the pGCL semantics we take as a guide. Although some definitions of demonic choice in the literature have this property, there are others where this property does not hold: for example if on both sides we have the same program containing a probabilistic choice and this choice is resolved independently on each side *before* the non-deterministic choice is performed, then idempotency does not hold. Nonetheless idempotency does hold if the probabilistic choice is triggered *after* the non-deterministic choice is made — this is the behaviour that we can find in our framework and in pGCL,where non-deterministic choice is history-aware, but lacks prescience [HS06, p.187].

We can reproduce prescient non-deterministic behaviour if we run the program twice with probabilistic choice on local variables, and then merge the outputs by means of a non-deterministic choice: this is a behaviour that has nothing to do with idempotency — we keep the actions of one program separate from the other's, so we are actually dealing with two *different* program instances that share the same specification.

We are now going to treat the well-known Monty Hall game as an example, which contains all of the main constructs of pGCL and shows the interaction between demonic and probabilistic choice.

The Monty Hall game In the Monty Hall game a player is challenged to guess which of the three doors in front of him hides a car. After having chosen a door among the three possible options, Monty Hall will open one of the remaining two doors: Monty Hall knows where the car is, so he is going to open one of the other two; the player is given the chance to change his guess at this point.

It is known from the literature that the player will maximize the probability of finding the car if now he changes the door he has chosen (the probability will be $^2/_3$) — this is Bertrand's box paradox (1889).

In fact the player can lose only if his first choice was the i-th door, which is hiding the car (and this happens with probability $^1/_3$), so after Monty Hall has opened the k-th door, that is one of the two hiding a goat, the switching strategy leads the player's final choice to be the j-th door, which is hiding a goat.

Nevertheless this is a winning strategy with probability $^2/_3$, as the chances of winning equal the chances of choosing a door hiding a goat, when all doors are closed. In fact choosing the j-th door forces Monty Hall to open the k-th door, and switching makes the player choose the i-th door.

The following is a short program, which uses the program constructs defined above to implement the game — in Figure 3 we give the definition for each variable, function and instruction that we are using:

$$P \;\hat{=}\; \texttt{setup;player;host;guess}$$

The variables a, b, c have values in the set $\{1,2,3\}$, therefore the state space is:

$$S = \{\sigma \mid \sigma = \underline{v} \mapsto \underline{\textit{val}}\}$$

where $\underline{v} = (a,b,c)$ and $\underline{\textit{val}} \in \{1,2,3\} \times \{1,2,3\} \times \{1,2,3\}$.

The initial distribution is a parameter of the problem: we assume its weight is 1, but make no further assumptions on the individual weight of each state.

The first instruction is made of three assignments[8], combined via non-deterministic choice:

$$\begin{array}{rcl} a := i & = & \delta' = \delta\{\!|{}^{i}/_{a}|\!\} \\ \texttt{setup} & = & \exists \pi_1, \pi_2, \pi_3 \bullet \delta' = \delta\langle\pi_1\rangle\{\!|{}^{1}/_{a}|\!\} + \delta\langle\pi_2\rangle\{\!|{}^{2}/_{a}|\!\} + \delta\langle\pi_3\rangle\{\!|{}^{3}/_{a}|\!\} \\ & & \wedge\ \pi_3 = \iota - \pi_1 - \pi_2 \end{array}$$

The second instruction is also made of three assignments, but this time they are combined via a uniform probabilistic choice:

$$\begin{array}{rcl} b := i & = & \delta' = \delta\{\!|{}^{i}/_{b}|\!\} \\ \texttt{player} & = & \delta' = {}^1/_3 \cdot \delta\{\!|{}^{1}/_{b}|\!\} + {}^1/_3 \cdot \delta\{\!|{}^{2}/_{b}|\!\} + {}^1/_3 \cdot \delta\{\!|{}^{3}/_{b}|\!\} \end{array}$$

[8] We use the notation $\{\!|{}^{e}/_{x}|\!\}$ for the assignment `x:=e`, which leaves all other variables unchanged.

$$\begin{array}{rcl@{\qquad}rcl} a & \hat{=} & \text{the position of the car} & \mathcal{S}(x,y) & \hat{=} & \min(\{1,2,3\} \smallsetminus \{x,y\}) \\ b & \hat{=} & \text{the player's guess} & \mathcal{H}_m(x) & \hat{=} & \min(\{1,2,3\} \smallsetminus \{x\}) \\ c & \hat{=} & \text{Monty Hall's hint} & \mathcal{H}_M(x) & \hat{=} & \max(\{1,2,3\} \smallsetminus \{x\}) \end{array}$$

$$\begin{array}{rcll} \texttt{setup} & \hat{=} & a := 1 \sqcap (a := 2 \sqcap a := 3) & \texttt{[1]} \\ \texttt{player} & \hat{=} & b := 1 \;{}_{\frac{1}{3}}\!\oplus (b := 2 \;{}_{\frac{1}{2}}\!\oplus b := 3) & \texttt{[2]} \\ \texttt{host} & \hat{=} & c := \mathcal{S}(a,b) \lhd (a \neq b) \rhd (c := \mathcal{H}_m(a) \sqcap c := \mathcal{H}_M(a)) & \texttt{[3]} \\ \texttt{guess} & \hat{=} & b := \mathcal{S}(b,c) & \texttt{[4]} \end{array}$$

Fig. 3. Variables, functions and instructions for the program implementing the Monty Hall game.

We have an if-statement in the third instruction, so we have:

$$\begin{array}{rcl}
c \coloneqq \mathcal{S}(a,b) & = & \delta' = \delta\{\!\!\{{}^{\mathcal{S}(a,b)}/_{c}\}\!\!\} \\
c \coloneqq \mathcal{H}_m(a) & = & \delta' = \delta\{\!\!\{{}^{\mathcal{H}_m(a)}/_{c}\}\!\!\} \\
c \coloneqq \mathcal{H}_M(a) & = & \delta' = \delta\{\!\!\{{}^{\mathcal{H}_M(a)}/_{c}\}\!\!\} \\
c \coloneqq \mathcal{H}_m(a) \sqcap c \coloneqq \mathcal{H}_M(a) & = & \exists \pi_{\mathcal{H}} \bullet \delta' = \delta\langle \pi_{\mathcal{H}} \rangle \{\!\!\{{}^{\mathcal{H}_m(a)}/_{c}\}\!\!\} + \delta\langle \bar{\pi}_{\mathcal{H}} \rangle \{\!\!\{{}^{\mathcal{H}_M(a)}/_{c}\}\!\!\} \\
\texttt{host} & = & \exists \pi_{\mathcal{H}} \bullet \delta' = \delta\langle a \neq b \rangle \{\!\!\{{}^{\mathcal{S}(a,b)}/_{c}\}\!\!\} + \\
 & & \quad + \delta\langle a = b \rangle \langle \pi_{\mathcal{H}} \rangle \{\!\!\{{}^{\mathcal{H}_m(a)}/_{c}\}\!\!\} + \delta\langle a = b \rangle \langle \iota - \pi_{\mathcal{H}} \rangle \{\!\!\{{}^{\mathcal{H}_M(a)}/_{c}\}\!\!\}
\end{array}$$

Finally the fourth instruction gives

$$b \coloneqq \mathcal{S}(b,c) \quad = \quad \delta' = \delta\{\!\!\{{}^{\mathcal{S}(b,c)}/_{b}\}\!\!\}$$

If we compose sequentially the four instructions (and jump to conclusions, full details are available in [BB11]), we obtain the following expression for the final probability distribution, which describes the program output:

$$\begin{aligned}
\delta' = & \sum_{i \neq j} {}^{1}/_{3} \cdot \delta\langle \pi_i \rangle \{\!\!\{{}^{i}/_{a}\}\!\!\} \{\!\!\{{}^{j}/_{b}\}\!\!\} \langle a \neq b \rangle \{\!\!\{{}^{\mathcal{S}(a,b)}/_{c}\}\!\!\} \{\!\!\{{}^{\mathcal{S}(b,c)}/_{b}\}\!\!\} \\
& + \sum {}^{1}/_{3} \cdot \delta\langle \pi_i \rangle \{\!\!\{{}^{i}/_{a}\}\!\!\} \{\!\!\{{}^{i}/_{b}\}\!\!\} \langle a = b \rangle \langle \pi_{\texttt{host}} \rangle \{\!\!\{{}^{\mathcal{H}(a)}/_{c}\}\!\!\} \{\!\!\{{}^{\mathcal{S}(b,c)}/_{b}\}\!\!\}
\end{aligned}$$

where i, j range over $\{1, 2, 3\}$ and $\pi_{\texttt{host}}$ ranges over $\{\pi_{\mathcal{H}}, \bar{\pi}_{\mathcal{H}}\}$ — and $\mathcal{H}$ will be $\mathcal{H}_m$ or $\mathcal{H}_M$ depending on $\pi_{\texttt{host}}$.

To evaluate the probability of winning, which is the probability of $a = b$, we have to evaluate $\|\delta'\langle a = b \rangle\|$; if we recall that $\iota\langle a = b \rangle$ represents the expectation of the predicate $a = b$, we can see that we are computing its expected value.

In the above expression we can distinguish two kinds of terms, and if we work on each one under the winning condition we obtain:

$$\begin{array}{rcl}
\delta\langle \pi_i \rangle \{\!\!\{{}^{i}/_{a}\}\!\!\} \{\!\!\{{}^{j}/_{b}\}\!\!\} \langle a \neq b \rangle \{\!\!\{{}^{\mathcal{S}(a,b)}/_{c}\}\!\!\} \{\!\!\{{}^{\mathcal{S}(b,c)}/_{b}\}\!\!\} \langle a = b \rangle & = & \delta\langle \pi_i \rangle \{\!\!\{{}^{i,j}/_{a,b}\}\!\!\} \{\!\!\{{}^{\mathcal{S}(a,b),a}/_{c,b}\}\!\!\} \\
\delta\langle \pi_i \rangle \{\!\!\{{}^{i}/_{a}\}\!\!\} \{\!\!\{{}^{i}/_{b}\}\!\!\} \langle a = b \rangle \langle \pi_{\texttt{host}} \rangle \{\!\!\{{}^{\mathcal{H}(a)}/_{c}\}\!\!\} \{\!\!\{{}^{\mathcal{S}(b,c)}/_{b}\}\!\!\} \langle a = b \rangle & = & \epsilon
\end{array}$$

The terms of the second kind will give no contribution to the overall weight of $\delta'\langle a = b \rangle$ (and in fact they account for the case when the player's first guess was the right one), whereas all others contribute with ${}^{1}/_{3} \cdot \|\delta\langle \pi_i \rangle \{\!\!\{{}^{i,j}/_{a,b}\}\!\!\} \{\!\!\{{}^{\mathcal{S}(a,b),a}/_{c,b}\}\!\!\}\|$ (and of course these account for the case when the player had first chosen a door hiding a goat).

As both remapping operations use expressions defined everywhere, and thanks to the fact that in this condition the remap operators preserves the weight of a distribution, we have that:

$$\|\delta\langle \pi_i \rangle \{\!\!\{{}^{i,j}/_{a,b}\}\!\!\} \{\!\!\{{}^{\mathcal{S}(a,b),a}/_{c,b}\}\!\!\}\| = \|\delta\langle \pi_i \rangle\|$$

Therefore we have:

$$\|\delta'\langle a = b \rangle\| \; = \|2 \cdot ({}^{1}/_{3} \cdot \delta\langle \pi_1 \rangle + {}^{1}/_{3} \cdot \delta\langle \pi_2 \rangle + {}^{1}/_{3} \cdot \delta\langle \pi_3 \rangle)\| = {}^{2}/_{3} \cdot \|\delta\|$$

We have assumed that the weight of the initial distribution is 1, so the weight of all winning states is $2/3$ — it is now clear why we did not need to make any other assumption, as this is all that matters, as all the variables undergo at least an assignment during the run of the program. $2/3$ is also the expected value for each of the initial states, so the pre-expectation assigning this weight to every state corresponds to the post-expectation of the predicate $\iota\langle a = b\rangle$.

4.3 Generic choice

Now that we have given an appropriate definition of non-deterministic choice, it is worth to remark in passing that we can see how all choice constructs follow a common pattern.

The reason is that all choice constructs can be seen as a specific instance of a generic choice construct:

$$\text{choice}\big(A, B, X\big) \;\widehat{=}\; \exists \pi, \delta_A, \delta_B \bullet \pi \in X \wedge A(\delta\langle\pi\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B$$

where $X \subseteq D_w$ and D_w is the set of all weighting distributions.

We can express all our choice constructs with appropriate choices of X:

- for $X = \{\iota\langle c\rangle\}$ we have conditional choice: $A \lhd c \rhd B = \text{choice}\big(A, B, \{\iota\langle c\rangle\}\big)$
- for $X = \{p \cdot \iota\}$ we have probabilistic choice: $A\ {}_p\!\oplus B = \text{choice}\big(A, B, \{p \cdot \iota\}\big)$
- for $X = D_w$ we have non-deterministic choice: $A \sqcap B = \text{choice}\big(A, B, D_w\big)$

Moreover we can see the disjunction of two programs as another kind of choice, where $X = \{\epsilon, \iota\}$: $A \vee B = \text{choice}\big(A, B, \{\epsilon, \iota\}\big)$

Our generic choice operator allows us to define a framework with only one choice construct, where all of the usual choice operators can be seen as syntactic sugar of a particular class of generic choices; moreover we can also use this generic construct to create new kinds of choices, other than the more traditional ones—the reader can refer to [BB11] for some examples; the potential of this generic choice operator has still to be fully explored.

4.4 The linkage between other semantic models and ours

The relational demonic semantics for pGCL [MM04, p139] is given as a function from a state to a set[9] of distributions: $S \to \mathcal{C}S$. Kleisli lifting (See Appendix A) of that model results in the relation between distributions that is described by our UTP semantics:

$$\begin{tikzcd}
\mathbb{C}S \arrow[r, "p^*"] & \mathbb{C}S \\
S \arrow[u, "\eta_S"] \arrow[ur, "p"'] &
\end{tikzcd}$$

[9] a.k.a *probabilistically closed* sets

From the lifted semantics mapping sets of distributions to such sets, we can extract the corresponding UTP relation (R) on distributions as follows:

$$R = \{(\delta, \delta') \mid \delta' \in p^*\{\delta\}\ \}$$

Things are slightly more complicated if we want to relate the `wp`-semantics from [MM04] to our semantic model. The way to do this is to observe that an expectation is a random variable (with non-negative real values), and as such it can be represented as a distribution χ in our framework. Then if χ' represents a post-expectation and A is a program, we can define the corresponding pre-expectation χ by computing the expected final weight of each state before A is run:

$$\chi(\sigma) = \min\big(\{\|\chi' \cdot \delta'\| \ \mid \ A(\eta_\sigma, \delta')\}\big)$$

Here η_σ represents a *point distribution*, which is a distribution where all states other than σ map to zero, while σ maps to 1:

$$\eta_\sigma \ \widehat{=}\ \epsilon \dagger \{\sigma \mapsto 1\}$$

So, $A(\eta_\sigma, \delta')$ is true for all δ' that can result from running A given a point distribution about σ. For each such δ' we scale with the post-expectation, and take the minimum over those. It shall be noted that this set of δ' so obtained is a singleton set for all deterministic constructs. We extract of the pointwise minimum from that set if not a singleton, as in this case we have non-determinism, and so we have to mirror the pointwise minimum used in Figure 1.

5 Conclusion and future work

We have provided an encoding of the semantics of pGCL in UTP, as a homogeneous relation on the alphabet $\{\delta, \delta'\}$, where the before and after variables are distributions over program states. The key is that our semantics models probabilistic programs as predicate transformers, so allowing us to claim that "probabilistic programs are predicates too". We have shown that we can deal with variables by name, despite their being entangled in the semantic domain, and that the laws of pGCL are provable from our semantics. In addition we have formulated our semantics in such a way as to be able to view all choices as instances of a generic choice construct, and even to be able to allow disjunction back in as a form of choice.

We have shown the linkage between our semantic model and the two models that feature in [HSM97,MM04]: this will lead to a formalization of the healthiness conditions, which characterise the predicates in our framework, and which we expect to be substantially the same, modulo an appropriate generalization, as in pGCL.

A further step forward to be taken is to explore the role of auxiliary variables such as `ok` and `ok'` that capture a behaviour such as termination: non-termination leads to probability sub-distributions, similar to what happens in pGCL, so we could manage without, but their introduction — together with other auxiliary variables such as `wait` and `wait′` — may prove of help in moving towards the encoding of reactive systems in this framework.

This is important, as the long term focus of this work is on a probabilistic variant of *Circus*, which requires semantic models for probabilistic process algebras like pCSP [MMSS96,DvGHM08] or PTSC [NS09]. These will then have to be integrated with our pGCL semantics in much the same way that the theory of Reactive Designs in UTP is the basis for the semantics of *Circus*-like languages.

Acknowledgements We wish to thank (some of) the anonymous referees who have reviewed previous versions of this paper for their insightful comments and suggestions.

References

[BB11] Riccardo Bresciani and Andrew Butterfield. Towards a UTP-style framework to deal with probabilities. Technical Report TCD-CS-2011-09, FMG, Trinity College Dublin, Ireland, August 2011.

[But10] Andrew Butterfield, editor. *Unifying Theories of Programming, Second International Symposium, UTP 2008, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers*, volume 5713 of *Lecture Notes in Computer Science*. Springer, 2010.

[CS09] Yifeng Chen and Jeff W. Sanders. Unifying probability with nondeterminism. In *FM 2009, LNCS 5850*, pages 467–482, 2009.

[DS06] Steve Dunne and Bill Stoddart, editors. *Unifying Theories of Programming, First International Symposium, UTP 2006, Walworth Castle, County Durham, UK, February 5-7, 2006, Revised Selected Papers*, volume 4010 of *Lecture Notes in Computer Science*. Springer, 2006.

[DvGHM08] Yuxin Deng, Rob J. van Glabbeek, Matthew Hennessy, and Carroll Morgan. Characterising testing preorders for finite probabilistic processes. *Logical Methods in Computer Science*, 4(4), 2008.

[FWB08] L. Freitas, J. Woodcock, and A. Butterfield. Posix and the verification grand challenge: A roadmap. *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pages 153–162, 31 2008-April 3 2008.

[GB09] Paweł Gancarski and Andrew Butterfield. The denotational semantics of slotted-Circus. In Ana Cavalcanti and Dennis Dams, editors, *FM2009: Formal Methods*, volume 5850 of *LNCS*, pages 451–466. Springer, 2009.

[He10] Jifeng He. A probabilistic BPEL-like language. In Qin [Qin10], pages 74–100.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall International Series in Computer Science, 1998.

[Hoa85] C. A. R. Hoare. Programs are predicates. In *Proceedings of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 141–155, Upper Saddle River, NJ, USA, 1985. Prentice-Hall.

[HS06] Jifeng He and Jeff W. Sanders. Unifying probability. In Dunne and Stoddart [DS06], pages 173–199.

[HSM97] Jifeng He, K. Seidel, and A. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2-3):171–192, 1997. Formal Specifications: Foundations, Methods, Tools and Applications.

[Koz81] Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

[Koz85] Dexter Kozen. A probabilistic pdl. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.

[MM97] Carroll Morgan and Annabelle McIver. A probabilistic temporal calculus based on expectations. Technical Report PRG-TR-13-97, Oxford University Computing Laboratory, 1997.

[MM04] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science).* SpringerVerlag, 2004.

[MM05] Annabelle McIver and Carroll Morgan. Abstraction and refinement in probabilistic systems. *SIGMETRICS Performance Evaluation Review*, 32(4):41–47, 2005.

[MMSS96] Carroll Morgan, Annabelle McIver, Karen Seidel, and Jeff W. Sanders. Refinement-oriented probability for CSP. *Formal Asp. Comput.*, 8(6):617–647, 1996.

[NM10] Ukachukwu Ndukwu and Annabelle McIver. An expectation transformer approach to predicate abstraction and data independence for probabilistic programs. *CoRR*, 2010.

[NS09] Ukachukwu Ndukwu and J. W. Sanders. Reasoning about a distributed probabilistic system. In Rod Downey and Prabhu Manyem, editors, *Fifteenth Computing: The Australasian Theory Symposium (CATS 2009)*, volume 94 of *CRPIT*, pages 35–42, Wellington, New Zealand, 2009. ACS.

[OCW09] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP semantics for Circus. *Formal Asp. Comput*, 21(1-2):3–32, 2009.

[Qin10] Shengchao Qin, editor. *Unifying Theories of Programming - Third International Symposium, UTP 2010, Shanghai, China, November 15-16, 2010. Proceedings*, volume 6445 of *Lecture Notes in Computer Science*. Springer, 2010.

[SH03] Adnan Sherif and Jifeng He. Towards a time model for Circus. *Lecture Notes in Computer Science*, 2495:613–624, 2003.

A Keisli Composition

Assume a semantic model of the form $S \to \mathbb{F}S$ where $\mathbb{F}$ is a type constructor (functor). The question that naturally arises is how to compose such functions, i.e., given $p : S \to \mathbb{F}T$ and $q : T \to \mathbb{F}U$, how do we compose these to get $(p;q) : S \to \mathbb{F}U$? The standard solution for this is Kleisli lifting and composition which involves two functions with the following signatures:

$$\eta_S : S \to \mathbb{F}S \qquad\qquad \_^* : (S \to \mathbb{F}T) \to (\mathbb{F}S \to \mathbb{F}T)$$

that obey the following laws:

$$\eta_S^* = id_{\mathbb{F}S} \qquad p^* \circ \eta_S = p \qquad (q^* \circ p)^* = q^* \circ p^*$$

The intuition behind these is best understood in a diagram:

$$\begin{tikzcd}
\mathbb{F}S \arrow[r, "p^*"] & \mathbb{F}T \arrow[r, "q^*"] & \mathbb{F}U \\
S \arrow[u, "\eta_S"] \arrow[ur, "p"'] & T \arrow[u, "\eta_T"] \arrow[ur, "q"'] & U \arrow[u, "\eta_U"']
\end{tikzcd}$$

The Kleisli composition of p and q is given by $q^* \circ p$, where $\circ$ denotes regular function composition.

In this paper $\mathbb{F}S = \mathbb{P}(S \to [0,1])$.