Proof Rules for Probabilistic Loops

Seminar on Probabilistic Programs

David Korzeniewski

January 26, 2015

For newly developed algorithms or computer programs often a formal proof for the correctness is of interest. For standard programs, even with nondeterminism, there exist well known rules for reasoning about the programs, such as the Hoare logic [4] or predicate transformers by Dijkstra [2]. *Probabilistic programs* can be used to model randomized algorithms and also to model systems that are not absolutely reliable. Still traditional nondeterministic choices are useful to model for example implementational freedom. The techniques from [2] and [4] however do not deal with this kind of programs.

In this paper a set of rules, developed in [8] and further discussed in [7], for reasoning about probabilistic programs and especially loops in these programs, is presented and additional intuitive explanations are given. These rules are based on an extension of predicate transformers as in [2] to probabilistic programs. This extended model is called *expectation transformers* [7].

In the first section we define the programming language used for probabilistic programs in this paper, introduce the semantic model of predicate transformers and explain how it is extended to expectation transformers. In the second section a rule for partial correctness, as well as two rules for total correctness of loops are presented. These rules are based on the idea of *loop invariants*. The third section then establishes a rule to prove termination of loops in probabilistic programs based on *loop variants*.

1 Preliminaries

1.1 Programming Language pGCL

We use the programming language "probabilistic guarded command language", pGCL for short, as defined in [7], which is inspired by Dijkstra's guarded command language [2]. The syntax is given by the following simple grammar:

$$\begin{aligned} prog ::= \ & \textbf{skip} \mid \textbf{abort} \mid x := E \\ & \mid prog; prog \mid \{prog\}\square\{prog\} \mid \{prog\}[p]\{prog\} \\ & \mid \textbf{if}(cond)\{prog\}\textbf{else}\{prog\} \mid \textbf{while}(cond)\{prog\} \end{aligned}$$

The curly braces may be omitted if the subprogram is **skip**, **abort** or $x := E$. They may also be omitted if the program is structured using linebrakes and indentation.

The most simple programs are **skip** and **abort**. The former does nothing else but terminating immediately and the latter never terminates. The third atomic construct is the assignment written as $x := E$, where x is a program variable and E is an expression over the program variables.

To build more complex programs we can sequentially execute two programs, denoted by $prog_1; prog_2$, chose nondeterministically or probabilistically with probability p between two programs, denoted by $\{prog_1\}\square\{prog_2\}$ and $\{prog_1\}[p]\{prog_2\}$ respectively. The nondeterministic choice $\{prog_1\}\square\{prog_2\}$ is called demonic nondeterminism. As we have no control over what happens, we take a pessimistic

viewpoint and imagine an adversary, the demon, who always takes the choice that is worst for us. For the probabilistic choice we allow p to be an expression over the program variables, not only constants. Using this we can build a branching construct $\textbf{if}(cond)\{prog_1\}\textbf{else}\{prog_2\}$ from a probabilistic choice $\{prog_1\}[\langle cond\rangle]\{prog_2\}$, where $\langle cond\rangle = 1$ if and only if *cond* is **true** and $\langle cond\rangle = 0$ otherwise.

Finally we also have loops of the form $\textbf{while}(cond)\{prog\}$. Intuitively this does the following: At first *cond* is evaluated, if it is **true** then *prog* is executed and the loop starts over by evaluating *cond* again. If *cond* evaluates to **false**, then the program terminates immediately. Here we restrict the loop condition *cond* to boolean conditions.

We call a program *standard*, if it does not have any probabilistic choices[1], but demonic nondeterminism is still allowed. A program is called *deterministic*, if there is no demonic nondeterminism. Probabilistic choices are still allowed in deterministic programs.

The *state space* of a probabilistic program is the set of all possible combinations of variable valuations and program locations. A variable valuation is a function that assigns a value to each variable. A program location is a "pointer" to the sub-program that will be executed next, or a special location that indicates successful termination or unsuccessful termination. A run of a probabilistic program can thus be viewed as a sequence of states. A single execution step moves according to the program from one state to a next state.

Markov Decision Processes and Markov chains can be used as semantics of probabilistic programs and can be used for model checking of probabilistic programs [3]. We will occasionally use these semantics to further explain some facts or give an intuition for people with a probabilistic model checking background.

1.2 Weakest Precondition and Weakest Preexpectation

Having the syntax defined we will now define the weakest precondition semantics for standard programs and the weakest preexpection for general probabilistic programs. We also introduce the model of predicate and expectation transformers.

For standard programs, we partition the programs state space in *good* and *bad* states. If the program terminates and is in a good state upon termination, then we consider the run successful, else we consider it a failure. The *weakest precondition* for the set of good states is the set of states from which the computation is guaranteed to terminate in a good state regardless of the choices of the demon. A subset of the state space is called a *predicate*. In practice we often write predicates as boolean expressions over program variables. A program can be viewed as a *predicate transformer*. It takes as input a predicate, the good states, and transforms it into a new predicate, the weakest precondition of the good states.

Notation 1. *Let prog be a standard program, and post be a standard predicate. We denote the weakest precondition of prog for post by wp.prog.post.*

Note that the input of the predicate transformer are target states of the computation and the output are the corresponding initial states. This reversed view matches the intended use of this model. If we want to prove the correctness of a program, we start at the desired final states and want to find all initial states from where we are guarantied to reach one of the desired final states.

For probabilistic programs we generalize predicates, which we can also identify by a characteristic function of the set which maps states to $\{0, 1\}$, to *probabilistic predicates* by allowing mappings from states to the interval $[0, 1]$. One can think of it as assigning a profit to the outcomes instead of just winning or loosing. Probabilistic predicates may be given as real valued expressions over program variables.

[1] with the exception of **if**-constructs

Running a deterministic program with a unique initial state can be viewed as a random experiment. The possible outcomes are terminal states of its state space and non-termination. For a deterministic program we have a unique distribution μ that assigns each terminal state the probability that we reach this terminal state with a run of the program. Now let the random variable X be the profit of the reached terminal state given by a probabilistic predicate g. If the program does not terminate we assign $X = 0$. The expected profit after the program has run, is given by $g(\sigma)$ for each state σ. For the unique initial state the profit we expect when running the program is the expected value of X with respect to the distribution μ. By determining the distribution for each possible initial state and taking the expected value, we can derive a new probabilistic predicate h that assigns each state the profit we expect before the program has run. So we could say the program transform the *postexpectation* g into a *preexpectation* h. Thus this model for programs is called *expectation transformer*.

For larger programs, that are composed of multiple sub-programs, we can now chain the expectation transformations. For example the preexpectation of $prog_1; prog_2$ for a predicate *pred* can be obtained by taking the preexpectation of $prog_1$ for the preexpecation of $prog_2$ for the predicate *pred*. Similarly we can give rules for transforming a predicate stepwise into a preexpectation for a large composed program.

For nondeterministic programs things get a little more involved. We cannot give a unique distribution μ for the probabilities of reaching certain terminal states. These distributions now depend on the resolution of nondeterminism. As before we assume the demon as an adversary that chooses the worst alternative for us. Since we want to maximize the profit, the demon will choose the alternative that minimizes the profit. Thus before the program has run, we have to assume the worst, which is the minimal expected value of X over all possible distributions. The function that assigns each state this minimal expected value is called the *weakest preexpectation*. Sometimes it is also called largest preexpectation[7], as it is the larges value we may expect beforehand without knowledge about what nondeterministic choices will be taken.

So in short the weakest preexpectation is the largest profit we may expect for a program and given profit-function defined at least over the terminal states. If the postexpectation or profit-function is a standard predicate, then the weakest preexpectation coincides with the least probability of reaching a state where the predicate is true.

Notation 2. *Let prog be a probabilistic program, and post be a probabilistic predicate. We denote the weakest preexpectation of prog for post by wp.prog.post.*

We now define some relations and functions on probabilistic predicates that are needed later. In the following probabilistic predicates are denoted by lower case letters $g, h \ldots$ Furthermore we use real numbers to denote predicates that are constant functions, in particular 1 and 0 for the predicate that is 1 or 0 everywhere respectively.

For a standard predicate *pred*, i.e. a state set, we define $\langle pred \rangle$ as the probabilistic predicate, where the expectation is 1 for all states in *pred* and 0 otherwise. Using this function we can now use standard predicates as probabilistic predicates.

The relations $\Leftarrow$ and $\Rrightarrow$ compare two predicates. They are informally defined as "everywhere no less than" and "everywhere no more than" respectively, i.e. for every state the expectation on the left hand side is not less or not greater than the expectation of the right hand side. They serve as probabilistic counterparts to implication. Implication means that one predicate is a subset of the other, with probabilistic predicates we use that the value of states is not greater than with the other. This way we again have that weaker predicates, with lower expected profit "implying" stronger ones with higher or equal expected profit. In the case of standard predicates they actually resemble implication.

We denote pointwise addition, subtraction and multiplication by the usual symbols $+$, $-$, $*$. Pointwise means, for each state the operation is applied to the expectations assigned by the operands.

The functions minimum, maximum and complement/one-minus, denoted by $\sqcap, \sqcup$ and $\overline{g}$, operate pointwise, so for each state the expectation of $g \sqcap h$ is the minimum of the expectations in g and h

and analogous for maximum and one-minus. They often occur as probabilistic counterparts to "and", "or" and "not" respectively, but this is only an intuition to help understanding formulas using these.

Another "replacement" for boolean "and" is the operator & defined as $g \,\&\, h := (g + h - 1) \sqcup 0$. This has the important property that $wp.prog.(g \,\&\, h) \Leftarrow wlp.prog.(g) \,\&\, wp.prog.(h)$ holds, we say it sub-distributes over weakest preexpectation.[2]

1.3 Program Semantics

Now that we have defined the syntax of the language and have an intuition on what the weakest preexpectation of a program is, we define the *wp*-semantics for probabilistic programs.

$$\begin{aligned}
wp.abort.g &:= 0 \\
wp.skip.g &:= g \\
wp.(x := expr).g &:= g[x \mapsto expr] \\
wp.(prog; prog').g &:= wp.prog.(wp.prog'.g) \\
wp.(prog \square prog').g &:= wp.prog.g \sqcap wp.prog'.g \\
wp.(prog[p]prog').g &:= p * wp.prog.g + \overline{p} * wp.prog'.g
\end{aligned}$$

These definitions are consistent with the intuitive semantics for programs and the intuition of weakest preexpectation provided in previous sections. The program *abort* does not terminate and thus we cannot expect anything, *skip* does nothing, so the expectations do not change. With the assignment, we have the assigned expression substituted for the left hand side of the assignment. Sequential composition of programs is defined as functional composition of *wp*. For nondeterministic choices we assume the worst case and thus everywhere take the minimal expectation (recall that $\sqcap$ denotes pointwise minimum) and for probabilistic choices we add up the expectations, weighted with the probabilities.

The semantics of loops are a bit more complicated, as they may run for an arbitrary number of iterations and could even not terminate at all. To capture this, a simple syntactic definition is not sufficient. Instead it is defined via a least fixed point.

Let the program *loop* be defined as

$$loop := \textbf{while } (\langle G \rangle)\{\ body\ \}$$

where G, the loop guard, is a standard predicate.

The weakest precondition for *loop* is defined as

$$wp.loop.h := \mu g \cdot (\langle G \rangle \sqcap wp.body.g) \sqcup (\overline{\langle G \rangle} \sqcap h)$$

where μ denotes the least fixed point.

2 Invariants for Probabilistic Loops

For standard programs we use *invariants* to show correctness. An invariant in the standard setting is a predicate I satisfying $I \wedge G \Rightarrow wp.body.I$, where G is the loop guard and *body* is the loop body. From that we can then can conclude that $I \Rightarrow wp.loop.(I \wedge \neg G)$. For probabilistic programs we will show a number of similar results. At first we show a very simple rule for partial loop correctness. After that we will extend it to total loop correctness, but with some restrictions on the loop body and the invariant.

[2] *wlp* will be defined later. It is *wp* but non-termination is viewed as success.

2.1 Partial Loop Correctness

For partial loop correctness we introduce the concept of *weakest liberal preexpectations*. This is a flavor of *wp* where not terminating or aborting is also considered *good*. Otherwise it is similar to weakest preexpectation.

Definition 1. *The **weakest liberal preexpectation** wlp.prog.g of a program prog and a probabilistic predicate g is defined as* $wlp.abort.g := 1$, $wlp.loop.g := \eta h \cdot (\langle G\rangle \sqcap wlp.body.h) \sqcup (\overline{\langle G\rangle} \sqcap g)$, *where* η *denotes the largest fixed point.*

For all other programs it is the same as the weakest preexpectation, i.e. $wlp.prog.g := wp.prog.g$ *if prog is not abort or a loop.*

The definition of invariants corresponds to the definition for the standard case, just replacing standard predicates with probabilistic ones.

Definition 2. *A wlp-*invariant *of a loop is a probabilistic predicate I that satisfies* $\langle G\rangle \sqcap I \Rrightarrow wlp.body.I$.

The given formula reads as "the current value of I is everywhere not less than the expected value of I after the loop body". So instead of having some property that is literally invariant, we take a property that is expected not to decrease.

As with standard programs, there is no simple way of deriving convenient invariants. In [7] the following heuristic for standard postexpectations is suggested. Let $\langle post\rangle$ be a standard predicate. Let p be a lower bound for the probability of staying in states where *post* holds through subsequent iterations of the loop, from a state where *post* holds. The expression $p * \langle post\rangle$ may be a good starting point for finding a loop invariant. It can be interpreted as "if *post* already holds the expected value for *post* after the loop body is p".

Using the definition of *wlp*-invariants, we can now easily prove the following rule for partial loop correctness.

Lemma 1. *Let I be a wlp-invariant of loop, then we have* $I \Rrightarrow wlp.loop.(\overline{\langle G\rangle} \sqcap I)$.

Proof. To show that the invariant I is "everywhere no greater than" $wlp.loop.(\overline{\langle G\rangle} \sqcap I)$ we use the property $x \leq f(x)$ implies $x \leq \eta \cdot f$ of largest fixed points. So we substitute i for h in the definition of *wlp*.

$$
\begin{array}{lll}
 & (\langle G\rangle \sqcap wlp.body.I) \sqcup (\overline{\langle G\rangle} \sqcap (\overline{\langle G\rangle} \sqcap I)) & \\
\Lleftarrow & (\langle G\rangle \sqcap (\langle G\rangle \sqcap I)) \sqcup (\overline{\langle G\rangle} \sqcap (\overline{\langle G\rangle} \sqcap I)) & I \text{ is wlp-invariant} \\
\equiv & I & G \text{ is standard}
\end{array}
$$

Therefore by the above mentioned property of largest fixed points we have that I is everywhere no greater than the largest fixed point, which is the *wlp*. □

2.2 Total Loop Correctness

In this section we will first improve Lemma 1 by using a *wlp*-invariant combined with a terminiation criterion to get a simple rule for total correctness. After that we further refine this by using a *wp*-invariant that is everywhere no less than a termination criterion which then results in a rule similar to Lemma 1, but for total correctness.

For standard programs we prove total loop correctness by requiring partial loop correctness and certain termination. Partial loop correctness was treated in the previous section. Termination, at least

in the standard case, is just $wp.prog.(\langle true\rangle) = wp.prog.1$. So for standard programs we can use that $wlp.prog.g \sqcap wp.prog.1 \Rrightarrow wp.prog.g$ holds.

For probabilistic programs, we still use $wp.prog.1$ which equals to the expectation of eventually terminating. Just taking the minimum however is not sufficient, as we will show by a simple example shortly. Since the proof for the standard case requires subdistributivity, we will use the & operator, which has this property. The general approach is otherwise identical to the standard case.

Notation 3. *For a given loop, we denote the termination expectation by* $T := wp.loop.1$.

Lemma 2. *Let* I *be a wlp-invariant for loop. Then*

$$I \& T \Rrightarrow wp.loop.(\overline{\langle G\rangle} \sqcap I)$$

Proof.

$$\begin{aligned}
& wp.loop.(\overline{\langle G\rangle} \sqcap I) \\
\equiv & wp.loop.((\overline{\langle G\rangle} \sqcap I) \,\&\, 1) \\
\Lleftarrow & wlp.loop.(\overline{\langle G\rangle} \sqcap I) \,\&\, wp.loop.1 && \&\text{ sub-distributes} \\
\equiv & wlp.loop.(\overline{\langle G\rangle} \sqcap I) \,\&\, T && \text{def. of T} \\
\Lleftarrow & I \& T && \text{Lemma 1}
\end{aligned}$$

□

If either I or T is standard, then $I \& T \equiv I \sqcap T$ holds. Thus in these cases the above lemma is useful. In general however, as mentioned before, the equivalence does not hold. A simple example can be used to show that $I \sqcap T$ is too weak in general.

Example: Take invariant $I := \langle n = 0\rangle/2 + \langle n = 1\rangle$ in the program *loop*, defined by

```
while(n≥0) {
  if(n=0) {
    n:= -1 [0.5] n:= +1
  }
  else {
    skip
  }
}
```

We have

$$\begin{aligned}
T &\equiv \langle n < 0\rangle + \langle n = 0\rangle/2 \\
I \sqcap T &\equiv \langle n = 0\rangle/2 \\
I \sqcap \overline{\langle G\rangle} &\equiv 0,
\end{aligned}$$

but $I \sqcap \overline{\langle G\rangle} \equiv \langle n = 2\rangle/2 \not\Rrightarrow wp.loop.0 \equiv wp.loop.(\overline{\langle G\rangle} \sqcap I)$

Lemma 2 can be improved by taking a *wp*-invariant instead of a *wlp*-invariant. From that we then build a larger invariant which we use with Lemma 2.

Theorem 3. *If* I *is a wp-invariant of loop and* $I \Rrightarrow T$ *then*

$$I \Rrightarrow wp.loop.(\overline{\langle G\rangle} \sqcap I)$$

Proof. Let $I' := I + 1 - T$. We show that I' is a *wlp* invariant of loop.

$$
\begin{aligned}
& wlp.body.I' \\
\equiv\ & wlp.body.(I + 1 - T) && \text{definition } i' \\
\equiv\ & wlp.body.I + wlp.body.1 - wp.body.T && \text{distributivity of } + \text{ with det. } body \\
\equiv\ & wlp.body.I + 1 - wp.body.T && \text{since in general } wlp.prog.1 \equiv 1 \\
\Leftarrow\ & \langle G\rangle \sqcap (wlp.body.I + 1 - wp.body.T) && \langle G\rangle \sqcap \text{ can only decrease the value} \\
\equiv\ & \langle G\rangle \sqcap wlp.body.I + \langle G\rangle - \langle G\rangle \sqcap wp.body.T && G \text{ standard} \\
\equiv\ & \langle G\rangle \sqcap wlp.body.I + \langle G\rangle - \langle G\rangle \sqcap T && T \text{ is strictly } wp \text{ invariant} \\
\Leftarrow\ & \langle G\rangle \sqcap (\langle G\rangle \sqcap I) + \langle G\rangle - \langle G\rangle \sqcap T && I \text{ is } wp\text{-invariant} \\
\equiv\ & \langle G\rangle \sqcap (I + 1 - T) && G \text{ standard} \\
\equiv\ & \langle G\rangle \sqcap I' && \text{definition of } I'
\end{aligned}
$$

The above steps rely on the fact that + distributes over *wp*, but that is only true if *body* is deterministic. However for every program, there is a deterministic program with the same *wp* [7]. Thus the above proof can be adapted for nondeterministic programs.

We now apply lemma 2 and get

$$I \equiv I' \& T \Rrightarrow wp.loop.(\overline{\langle G\rangle} \sqcap I')$$

Since G is standard we have $\overline{\langle G\rangle} \sqcap I' \equiv \overline{\langle G\rangle} \sqcap (I + 1 - T) \equiv \overline{\langle G\rangle} \sqcap I + \overline{\langle G\rangle} - \overline{\langle G\rangle} \sqcap T$. As $\overline{\langle G\rangle}$ implies immediate termination we have $\overline{\langle G\rangle} \equiv \overline{\langle G\rangle} \sqcap T$ and thus $\overline{\langle G\rangle} \sqcap I' \equiv \overline{\langle G\rangle} \sqcap I$. So now we can conclude that

$$I \equiv I' \& T \Rrightarrow wp.loop.(\overline{\langle G\rangle} \sqcap I') \equiv wp.loop.(\overline{\langle G\rangle} \sqcap I)\ .$$

□

2.3 Example

We look at the following simple program to uniformly select a random integer l from the range 0 to $N - 1$.

```
init:
  l := 0;
  h := N;
loop:
  while ( l + 1 ≠ h) {
    p := (l+h)/2;
    l := p [(h−p)/(h−l)] h:= p
  }
```

Given an arbitrary integer C we are interested in the probability that the program terminates and we have $l = C$. If that is $\frac{1}{N}$ for all $0 \leq C < N$ and 0 otherwise, the program correctly implements uniform random selection.

At first we look for a loop invariant using the previously mentioned heuristic $p * \langle post\rangle$. We want that $l = C$ holds after the loop, which is equivalent to $l \leq C < h$ when the loop has terminated. From any state in the loop where $l \leq C < h$ holds, the probability of actually choosing C is $\frac{1}{h-l}$. Thus we define

$$I := \begin{cases} \langle l \leq C < h\rangle/(h - l) & \text{if } h \neq l \\ 0 & else \end{cases}$$

and will now show that this is a valid loop invariant, by proving that $I \sqcap \langle l+1 \neq h \rangle \Rrightarrow wp.body.(I)$.

$$
\begin{aligned}
& wp.body.(I) \\
\equiv\ & wp.(p := (l+h)/2).(\frac{h-p}{h-l}\langle p \leq C < h\rangle/(h-p) \\
& + \frac{p-l}{h-l}\langle l \leq C < p\rangle/(p-l) \\
&) && \text{apply } wp.(l := p[\frac{h-p}{h-l}]h := p) \\
\Leftarrow\ & \text{(by treating the special cases with division by zero appropriately)} \\
& wp.(p := (l+h)/2).\langle l < p < h\rangle \sqcap I) \\
\equiv\ & \langle l < (h+l)/2 < h\rangle \sqcap I && \text{apply } wp \\
\Leftarrow\ & \langle l+1 \neq h\rangle \sqcap I
\end{aligned}
$$

Using the standard variant $l - h$ we can show, that termination is certain. Thus we have $T \equiv 1$ and therefore $I \Rrightarrow T$. So the wp-invariant I fulfills all preconditions of Theorem 3.

$$I \Rrightarrow wp.loop.(\langle l+1 = h\rangle \sqcap I) \equiv wp.loop.(\langle l = C\rangle)$$

The last step is to calculate the wp of the whole program:

$$\langle 0 \leq C < N\rangle/N \equiv wp.init.(I) \Rrightarrow wp.prog.\langle l = C\rangle$$

From that we can immediately see that the program correctly uniformly chooses a random element from the integer range $0, \ldots, N-1$. □

3 Variants for Probabilistic Loops

For standard programs a well known technique to show termination is to provide a loop *variant*. That is an expression over program variables which is guaranteed to decrease in every iteration and is bounded from below. Thus for any state eventually the lower bound is reached and the loop terminates.

For probabilistic programs we can also use variants to show almost sure termination, if the state space of the program is finite. The only additional requirement for the variant will be that it has to be bounded from below and from above.

3.1 0-1 Law for Termination

A simplified, informal formulation of the *0-1 law for termination* for finite state space programs is "termination is almost sure, unless there is a state with termination probability is 0 in the loop". A state is considered in the loop, if the program location is some part of the loop. We will refer to states with termination probability 0 as *trap states*. For infinite state systems, however, instead of trap states a sequence of states with termination probabilities converging to 0 are also sufficient, as the execution can be trapped inside that sequence. In general we have to say "termination is almost sure, unless the infimum over the termination probabilities of the states inside the loop is 0".

With Markov chains, which are an alternative semantics for probabilistic programs, we have similar statements about long run probabilities. On the long run probability mass is only left in bottom strongly connected components, for finite Markov chains. So either a subset of the state space contains a bottom strongly connected component, or the probability of eventually leaving that subset is 1. For

infinite Markov chains we again also have to take care of sequences where the escape probability converges to 0.

The following lemma is the formal version of the 0-1 law for termination. It is a slightly more powerful formulation than the informal version above, as it also gives some information about states where termination is not almost sure.

Lemma 4. *Let I be a wp-invariant of loop with termination condition T. If there is some fixed $0 < p \leq 1$ such that $pI \Rrightarrow T$, then also $I \Rrightarrow T$ holds.*

The fixed probability p has to be less less or equal to the infimum over the termination probabilities of all states σ where $I(\sigma) = 1$. From that observation we can then build the informal formulation from above.

Proof. It can be shown, that pI is a *wp*-invariant of *loop*, thus we can reason

$$
\begin{aligned}
& p\ I \\
\Rrightarrow & wp.loop.(\overline{\langle G \rangle} \sqcap (p\ I)) && pI \text{ is } wp \text{ invariant, } pI \Rrightarrow T, \text{ and Thm. 3} \\
\equiv & wp.loop.(p(\overline{\langle G \rangle} \sqcap I)) && \langle G \rangle \text{ standard} \\
\equiv & p\ wp.loop.(\overline{\langle G \rangle} \sqcap I) && wp \text{ can be scaled} \\
\Rrightarrow & p\ wp.loop.1 && wp \text{ is monotonic} \\
\equiv & p\ T && \text{definition of } T
\end{aligned}
$$

As $p \neq 0$ we finish the proof with

$$
\begin{aligned}
& p\ I \Rrightarrow p\ T \\
\Leftrightarrow\quad & I \Rrightarrow T
\end{aligned}
$$

□

3.2 Probabilistic Variants

For termination of standard programs *variants* are used to show certain termination. A variant is an integer valued expression over the program variables that is bounded from below and decreased in every iteration. This gives a complete method to show termination, as "the largest possible number of iterations from the current state" always gives a variant.

For probabilistic programs, this is not complete, as we can show by a simple example.

$$\textbf{while}\ ((n\ mod\ N) \neq 0)\ \{n := n + 1\ [0.5]\ n := n - 1\}$$

This program terminates almost surely. Intuitively we can argue that the probability of never getting decreases N times or increases N times in a row is 0, since the program decides to increase or decrease by flipping a coin and with probability 1 we eventually get N times the same in a row. Thus we have to terminate. However there is no standard variant for this program as independently of the current state in the loop the number of iterations before eventually terminating can be arbitrarily large.

From the 0-1 law however we can construct a general rule. We introduce an upper bound for the variant and say that with a fixed probability greater zero it has to be decreased in an iteration. The interpretation here is that we have a number of necessary iterations until termination for each state (instead of a maximal number) and with a fixed probability greater zero we will do a step towards termination. Then eventually we will take enough steps towards termination in a row to actually terminate. In the example the variant could be chosen as $V := |n\ mod\ N|$. This number will be

decreased with probability at least 0.5 in every state in the loop and is bounded from above and below by $0, N-1$. Thus everywhere the probability of eventual termination is at least 0.5^N and therefore termination is almost sure.

Lemma 5. *Let V be an integer valued expression over the program variables, defined at least over some subset of the statespace, defined by a standard predicate I. Suppose further that for loop it holds that*

1. *There are fixed integer constants L and H such that*

$$\langle G\rangle \sqcap I \Rrightarrow \langle L \leq V < H\rangle$$

2. *The subest I, as a standard predicate is wlp-invariant for loop*

3. *For some fixed probability $p > 0$ and for all integers N it holds that*

$$p\ (\langle G\rangle \sqcap I \sqcap \langle V = N\rangle) \Rrightarrow wp.body.\langle V < N\rangle.$$

Then termination of loop is almost sure from any state σ with $I(\sigma) = 1$.

Proof. By induction one can show that the assumptions imply

$$p^n(I \sqcap \langle V < L + n\rangle) \Rrightarrow T. \tag{1}$$

Using this and the weakened form of Assumption 1, $\langle G\rangle \sqcap I \Rrightarrow I \sqcap \langle G\rangle \sqcap I \Rrightarrow I \sqcap \langle V < H\rangle$, we can then reason

$$\begin{aligned}
& p^{H-L}\ I \\
\equiv\ & p^{H-L}(\langle G\rangle \sqcap I) \sqcup p^{H-L}(\overline{\langle G\rangle} \sqcap I) && G \text{ standard} \\
\equiv\ & p^{H-L}(\langle G\rangle \sqcap I) \sqcup T && \overline{\langle G\rangle} \Rrightarrow T \\
\Rrightarrow\ & p^{H-L}(\langle G\rangle \sqcap \langle V < H\rangle) \sqcup T && \text{Assumption 1 (weakened)} \\
\equiv\ & T \sqcup T && (1) \text{ above} \\
\equiv\ & T
\end{aligned}$$

With Assumption 2 and $p^{H-L} \neq 0$ we get $I \Rrightarrow T$ by using Lemma 4. As I is standard we thus have almost sure termination in all states where I holds. □

By Lemma 5 we have shown that the following informal rule can be used to show terminiation: Given an integer valued variant bounded from above and from below such that on each iteration the variant is decreased with at least constant probability $p > 0$, the probability for termination is 1.

An alternative variant rule can also be shown using Lemma 5: Given a variant that is bounded from below (not necessarily from above) such that on every iteration it is decreased with at least probability $p > 0$ and can never be increased during the loop, the probability for termination is 1.

To show that rule we have to apply Lemma 5 for each initial state of the loop individually with the initial value of the variant as upper bound and I modified such that it is only true for that one initial state.

This rule comes in handy, if there is no easy way of expressing an upper bound in the program variables, but a never increasing, but not always decreasing, variant is easily found. Such an example would be the following simple program:

$$\textbf{while}\ (n > 0)\ \{n := n - 1\ [0.5]\ skip\}$$

Here we cannot express an upper bound using the program variables. Although we could extend the program by a helper variable to store the initial value, it is more convenient to apply the alternative variant rule, setting the variant $V := n$ and the lower bound $L := 0$.

The presented rule is actually complete for programs with finite state space, i.e. for any program with finite state space we can provide a variant as defined above [8]. The intuitive argument is that with a finite state space the infimum over the termination probabilities is the minimum of the termination probabilities. Thus by the 0-1 law there have to be states with termination probability 0 inside the loop, if termination is not almost sure. But if termination is almost sure, then we can set p to the minimal probability of terminating over all states in the loop. We then choose V as the number of steps needed to reach a state outside of the loop with the bounds $L = 0$ and H as the maximal number of steps from any state in the loop to a state outside of the loop.

4 Conclusion

We now have a set of rules to show correctness of loops in probabilistic programs. Based on probabilistic invariants, we have rules for partial correctness (Lemma 1) and total correctness (Lemma 2 and Theorem 3). Probabilistic invariants are real valued expressions where the value is expected to not decrease during the loop. Based on the presented rules a method for automated correctness proofs has been developed [5]. That approach however still required the user to provide useful loop invariants, since as with standard programs, invariants that are useful for an actual correctness proof are not trivial to find. An algorithmic approach to invariant generation was later presented in [6].

For termination a rule based on probabilistic variants was presented (Lemma 5). A probabilistic variant is an integer valued expression, that decreases in each iteration with a fixed positive probability and cannot increase past a threshold. This rule is even complete for programs with finite state space.

So the rules from [8] and [7], which are presented and further explained in this paper, are the theoretical foundation for automated correctness proofs of probabilistic programs.

References

[1] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*. MIT press Cambridge, 2008.

[2] Edsger Wybe Dijkstra. *A discipline of programming*, volume 4. prentice-hall Englewood Cliffs, 1976.

[3] Friedrich Gretz, J Katoen, and Annabelle McIver. Operational versus weakest precondition semantics for the probabilistic guarded command language. In *Quantitative Evaluation of Systems (QEST), 2012 Ninth International Conference on*, pages 168–177. IEEE, 2012.

[4] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[5] Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in hol. *Theoretical Computer Science*, 346(1):96–112, 2005.

[6] Joost-Pieter Katoen, Annabelle K McIver, Larissa A Meinicke, and Carroll C Morgan. Linear-invariant generation for probabilistic programs. In *Static Analysis*, pages 390–406. Springer, 2011.

[7] Annabelle McIver and Charles Carroll Morgan. *Abstraction, refinement and proof for probabilistic systems*. Springer, 2006.

[8] CC Morgan. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Refinement Workshop, Workshops in Computing. Springer Verlag*, 1996.