

Loop Invariants in Probabilistic Programs

Bachelor Thesis

Justin Winkens

October 14, 2015

Lehrstuhl für Informatik 2 - System Modeling and Verification

First Supervisor: Prof. Dr. Ir. Joost-Pieter Katoen

Second Supervisor: apl. Prof. Dr. Thomas Noll

Advisor: Friedrich Gretz

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht.

Aachen, 5. August 2015

Justin Winkens

Abstract

This bachelor thesis is focused on the analysis of *probabilistic programs*. Probabilistic programs are able to model stochastic processes by using probabilistic choices, which makes them applicable in many research areas. To prove certain properties of a program it is often necessary to provide a so called *probabilistic loop invariant*, which remains the main obstacle in the verification of probabilistic programs and is undecidable in theory.

In this thesis the *weakest pre-expectation semantics* developed by McIver and Morgan is used to analyze probabilistic programs written in the *probabilistic Guarded Command Language*. First we model several discrete probability distributions as probabilistic programs and prove basic properties of them using *fixed-point iteration*. Based on the results a strategy to find probabilistic loop invariants is developed and used to find invariants for our programs.

Afterwards our invariants and programs as well as invariants from other researchers are combined to a *benchmark suite* to evaluate the probabilistic invariant synthesis tool PRINSYS. In the course of this *evaluation* a computational *bottleneck* is discovered that currently hinders PRINSYS from analyzing certain programs. We characterize this class of programs and suggest means to enhance performance and usability of PRINSYS.

Contents

1	Introduction	1
2	Preliminaries	3
2.1	The Probabilistic Guarded Command Language	3
2.2	Expectations	4
2.3	Weakest Pre-Expectation Semantics	5
2.4	Probabilistic Loop Invariants	6
2.5	PRINSYS	8
3	Finding Invariants	9
3.1	Fixed-Point Iteration	9
3.2	Invariants for Expected Outcomes	9
3.2.1	Binomial Distribution	10
3.2.2	Negative Binomial Distribution	15
3.2.3	Hypergeometric Distribution	20
3.2.4	Uniform Distribution	23
3.3	Invariants for Individual Outcomes	26
3.3.1	Geometric Distribution	26
3.3.2	Binomial Distribution	29
3.3.3	Negative Binomial Distribution	32
3.3.4	Hypergeometric Distribution	35
4	Evaluating PRINSYS	39
4.1	PRINSYS	39
4.2	Acquiring a Benchmark Suite for PRINSYS	42
4.3	Benchmarking PRINSYS	46
4.3.1	Benchmarking Runtimes	47
4.3.2	Benchmarking Expression Size	50
4.4	Challenge	52
4.4.1	Choice Coordination Problem	52

5	Conclusion	55
5.1	Summary	55
5.2	Future Work	55
A	Appendix	58
A.1	Geometric Distribution (Expected Outcome)	58
A.2	Synch-ccp.pgcl	60
A.3	Programs of Yu-Fang Chen et al. [3]	63
A.4	PRINSYS Worst Case Example	65

Chapter 1

Introduction

In the last few years *probabilistic programs* drew more and more interest due to their ability to model stochastic processes, enhance average performance of known algorithms and approximate hard problems. For example consider the quick sort algorithm. Although it is well known quick sort takes $O(n \log n)$ comparisons to sort n items on average, it still has a worst case complexity of $O(n^2)$. However, it has been shown (e.g. [5]) that using *probabilistic choices* to determine the next pivot makes the worst case less likely as n grows and according to [20] the probability to use a quadratic number of comparisons on a large array is much less than to “be struck by lightning”.

In general, whenever *uncertainty* is an issue probabilistic programs can be used to model these situations and gain insights into the systems behavior as well as solve problems deterministic algorithms fail to solve efficiently. Nice examples of such solutions are the “TrueSkill” ranking system by Microsoft [11], self-testing/correcting of numerical functions [1], modeling of genome evolution in biology [8], algorithms for distributed systems (e.g. choice coordination problem [19]) or the recently published probabilistic programming language “Picture” [13] in the field of machine learning and computer vision that manages to accomplish computer vision tasks in 50 lines of code, competing with other solutions taking thousands of lines of code.

But as the interest in probabilistic programs grows and those programs get introduced into possibly *critical environments*, there is an inherent need to *reason* about the programs behavior, *guarantee* bounds of characteristic variables or *quantify* probabilities. This is where probabilistic verification comes into play. A lot of this verification process can be automated but for example finding so called *loop invariants* is undecidable in theory. Finding invariants remains the main obstacle in practice and can only be done manually by a user. Nonetheless recent effort yielded the tool PRINSYS [10] that can support the user in finding those loop invariants and verify certain properties of a program.

This thesis is focused on finding probabilistic invariants manually providing further insight in how to compute loop invariants and what loop invariants generally look like. Furthermore these invariants will be used to evaluate PRINSYS for its capabilities and limitations as well as to suggest possibilities to improve its performance.

Outline In Chapter 2 we are going to introduce the required preliminaries, such as *syntax and semantics* of probabilistic programs and the PRINSYS tool. In Chapter 3 we are going to *analyze* various programs, thus providing *case studies* for PRINSYS evaluation. Finally, we are using the results to *evaluate* PRINSYS and show where the current implementation contains bottlenecks to be resolved. Furthermore a *challenge* is proposed to illustrate the gap between where we are know and where we want to get in the future.

Chapter 2

Preliminaries

In order to model and analyze probabilistic systems we are going to use the *probabilistic Guarded Command Language* (PGCL [17]), an extension of Dijkstra’s Guarded Command Language [6] by a *probabilistic choice statement*. In the following we are going to take a brief look at the syntax of PGCL. Then we will briefly discuss how to analyze pGCL programs by using the *weakest pre-expectation semantics* introduced by McIver and Morgan in [17]. Thereafter we will characterize probabilistic loop invariants and introduce PRINSYS.

2.1 The Probabilistic Guarded Command Language

The probabilistic Guarded Command Language (PGCL) is rather simple and only consists of 7 commands. As in imperative programming languages like JAVA or C every statement has to end with a semicolon allowing sequential execution of subprograms. Additionally, all program variables need to be declared by using the keyword *var* (e.g. *var x*);

skip

The *skip* statement is the equivalent of the empty statement and therefore does nothing at all.

abort

The *abort* statement indicates that from now on the program behaves arbitrary. We could describe the programs behavior as “undefined”, hence may or may not leave the program execution in an inconsistent state. It might as well not terminate at all.

$x := E$

The command $x := E$ is the assignment command and assigns the result of the arithmetic expression E to variable x .

$\text{if}(G) \{P\} \text{else} \{Q\}$

This command is equivalent to the well known if-then-else statement. If guard G holds program P is executed, else Q is executed.

$$\text{while}(G) \{P\}$$

This command is equivalent to the classic *while* construct of other programming languages. As long as guard G holds program P will be executed repeatedly.

Additional to classic programming language PGCL provides two more statements.

$$\{P\} \square \{Q\}$$

The above command is called *nondeterministic choice statement*. It nondeterministically decides to execute either P or Q . Note that we can write “ $(P \square Q)$,” and will omit “;” for statements P and Q if they are single statements.

$$\{P\} [p] \{Q\}$$

Adding a parameter p the nondeterministic choice statement turns into a *probabilistic choice statement*. This statement executes program P with probability p and program Q with probability $(1 - p)$ respectively. Note that we can write “ $(P [p] Q)$,” and will omit “;” for statements P and Q if they are single statements.

For example a PGCL program could look like this:

```
1 var x, flip;
2
3 x:=0;
4 flip:=0;
5 while(flip=0){
6     (x:=x-1[0.25]x:=x+1);
7     if(x>10){
8         flip:=1;
9     }else{
10        skip;
11    }
12 }
```

Listing 2.1: A simple PGCL program

This program decreases random variable x with probability 0.25 (i.e. 25%) and increases it with probability $1 - 0.25 = 0.75$ (i.e. 75%). If x surpasses 10 variable $flip$ will be set to 1, therefore terminating the loop and the program. Note that the probabilistic choice statement does not necessarily have to contain a number like 0.25, but instead can contain a *parameter* (e.g. p) that is not instantiated and symbolizes an arbitrary probability or a probability that changes during program execution. We further define a *program state* as a valuation of the program variables. For example in between line 4 and 5 of the PGCL program in Lst. 2.1 the current program state can be characterized as $x = 0$, $flip = 0$.

2.2 Expectations

In non-probabilistic programs every initial program state leads to a deterministically given final program state. In probabilistic programs every initial program state can have multiple different final states, because program execution is influenced by probabilistic choices. This

implies that after the first probabilistic choice statement the current program state is not given by a constant value but as an *expectation* of the valuation of program variables. For example consider the simple probabilistic program

$$(x := 7 [0.25] x := 1) ;$$

that assigns 7 to x with probability 0.25 and 1 with probability 0.75. From an initial state of that program we can not predict deterministically if x is going to be valued as 7 or 1 in a final state, because it depends on randomness what value it actually takes. Instead we will define the valuation of a random variable x in an initial state as its expected value. Thus the valuation of x w.r.t. the above program is $0.25 \cdot 7 + 0.75 \cdot 1 = 2.5$. We will call this value *pre-expectation* w.r.t. *post-expectation* x . Thus we are able to annotate a program as

$$\langle preE \rangle \text{ prog } \langle postE \rangle$$

where $preE$ is a pre-expectation, $prog$ a program and $postE$ a post-expectation. Note that both $preE$ and $postE$ are functions that map program states to non-negative real numbers. We say that the above expression *holds* if the expected value of $postE$ before executing $prog$ is at least $preE$. In this case:

$$\langle 2.5 \rangle (x := 7 [0.25] x := 1) ; \langle x \rangle$$

2.3 Weakest Pre-Expectation Semantics

In order to analyze pGCL programs like Lst. 2.1 we need formalisms to capture a program's *properties* and *transform* them during program execution. As we have seen in the previous section a program's properties can be expressed as post- and pre-expectations. A formalism for transforming expectations is the subject of this section. For the Guarded Command Language, Dijkstra already introduced the *weakest precondition semantics* (wp [6]) to assign *meaning* to a program. As McIver and Morgan introduced the probabilistic choice statement to GCL they also added a way to analyze said statement to the already existing weakest precondition semantics [17].

As a probabilistic choice statement is introduced, normal *predicate transformers* do not suffice to analyze probabilistic programs since we are not looking at a definitive program outcome, but instead *distributions over all possible outcomes*. Therefore McIver and Morgan turned the weakest precondition semantics from a predicate transformer into an *expectation transformer* by switching from predicates to *real valued* expectations. This allows us to analyze pGCL programs for their properties. For program P and post-expectation $postE$ (i.e. *non-negative* variable valuations) $wp(P, postE)$ is a function yielding the greatest pre-expectation $preE$ for which $\langle preE \rangle P \langle postE \rangle$ holds, i.e. the greatest lower bound of $preE$ w.r.t. $postE$ and program P .

The weakest pre-expectation function according to McIver and Morgan is defined by the following rules: Let P, Q denote arbitrary programs, G a guard, E an arithmetic expression and f a post-expectation.

$$wp(\text{skip}, f) = f$$

Of course, as *skip* does nothing, the variable valuation stays the same, hence our pre-expectation of post-expectation f remains f .

$$wp(\text{abort}, f) = 0$$

The *abort* statement leaves the program in an inconsistent state and therefore does not provide any pre-expectation of post-expectation f .

$$wp(x := E, f) = f[x/E]$$

An assignment replaces every occurrence of x in post-expectation f with expression E .

$$wp(\text{if}(G) \{P\} \text{ else } \{Q\}, f) = [G] \cdot wp(P, f) + [\neg G] \cdot wp(Q, f)$$

$[\cdot]$ denotes the cast of the boolean truth value to an integer, i.e. $[true] = 1$ and $[false] = 0$. Thus, if G holds wp will yield the weakest pre-expectation of f with respect to program P , otherwise with respect to Q .

$$wp(\{P\} \square \{Q\}, f) = \min\{wp(P, f), wp(Q, f)\}$$

The nondeterministic choice operator is resolved by returning the *minimum* over both possible paths. This is based on a worst case assumption, that means we will pick the lowest value achievable on those two paths.

$$wp(\{P\} [p] \{Q\}, f) = p \cdot wp(P, f) + (1 - p) \cdot wp(Q, f)$$

On the other hand the probabilistic choice operator is resolved by multiplying the path taken with its *probability*. Thus we will multiply the result of program path P with p and Q with $(1 - p)$. Generally speaking, wp will return the expected value over both paths.

$$wp(\text{while}(G) \{P\}, f) = \text{lfp}_F([G] \cdot wp(P, F) + [\neg G] \cdot f)$$

As we can see wp resolves loops by returning the *least fixed-point* over the loop's execution as pre-expectation. How to find the least fixed-point will be the subject of Chapter 3.

2.4 Probabilistic Loop Invariants

In the previous section we introduced the weakest pre-expectation semantics to reason about a program's behavior and properties. If we want to verify that $\langle preE \rangle prog \langle postE \rangle$ holds for a given pre-expectation, program and post-expectation we just need to compute the weakest pre-expectation $wp(prog, postE)$ and check if $preE \leq wp(prog, postE)$ (where " \leq " is interpreted pointwise).

For loop-free programs the pre-expectation returned by wp is given by purely syntactic rules and hence can be automated. However, to resolve a loop we have to calculate its least fixed-point. This means, regarding the definition in Section 2.3, we are looking for a fixed-point F of the form

$$F = [G] \cdot wp(P, F) + [\neg G] \cdot f \tag{2.1}$$

where P is the loop's body, G the loop guard and f a post-expectation. This tells us if guard G is not satisfied, i.e. the loop terminates or is not entered at all, the result should be the post-expectation. On the other hand if G holds wp is executed with the fixed-point as post-expectation and thus yields the fixed-point itself. Consequently this recursion of $wp(P, F)$ ends as soon as guard G is no longer satisfied.

Unfortunately fixed-point iteration is a very expensive task, but can be avoided by finding a probabilistic loop invariant.

Theorem 1. [17] *Let g be a pre-expectation, f a post-expectation, G a guard and ‘body’ a loop-free probabilistic program. To verify*

$$\langle g \rangle \text{while } (G) \{ \text{body} \} \langle f \rangle$$

*it is sufficient to find a **probabilistic loop invariant** \mathcal{I} which is an expectation that satisfies the following conditions:*

1. (consecution) $g \leq \mathcal{I} \wedge [\neg G] \cdot \mathcal{I} \leq f$
2. (invariance) $[G] \cdot \mathcal{I} \leq wp(\text{body}, \mathcal{I})$ (2.2)
3. (soundness) – the number of iterations is finite;
 - or \mathcal{I} is bounded from above by a fixed constant;
 - or $wp(\text{body}, [G] \cdot \mathcal{I})$ tends to zero as the number of iterations increases

Contrary to predicate transformers, where invariants are implications between predicates, a probabilistic loop invariant is an inequality between expectations. Intuitively this means that a probabilistic loop invariant is an expectation that does not decrease during loop execution and underestimates our post-expectation f . So our invariant actually approximates the result of wp from below. However, underestimating the outcome of the program is not desired, because we would like invariants to capture the behavior of the program precisely. For the sake of simplicity we will refer to probabilistic loop invariants as *invariants*.

Example 1. Geometric Distribution [4]. *The following PGCL program models the geometric distribution:*

```
x:=0; flip:=0; while (flip=0) { (flip:=1[p] x:=x+1); }
```

The mean value of the geometric distribution is defined to be $\frac{1-p}{p}$. If we want to prove this property for the above program (denoted as ‘prog’), we have to check if $\langle \frac{1-p}{p} \rangle \text{prog} \langle x \rangle$ holds. We can either do this by calculating $wp(\text{prog}, x)$ and check if $\text{pre}E \leq wp(\text{prog}, x)$ holds or by providing an invariant \mathcal{I} that satisfies the ‘invariance’, ‘consecution’ and ‘soundness’ conditions above and is strong enough to prove the property. An invariant¹ is given by:

¹This invariant is not trivial but let us assume that it is given.

$$\mathcal{I} = [flip = 0] \cdot (x + \frac{1-p}{p}) + [flip \neq 0] \cdot x$$

If we insert this invariant and set $flip$ and x to 0 we can see that the result is $\frac{1-p}{p}$ and exactly the pre-expectation we wanted to achieve. Thus $\langle \frac{1-p}{p} \rangle prog \langle x \rangle$ holds and the program indeed models the geometric distribution. Furthermore, to satisfy the ‘consecutive’ condition we convince ourselves that

$$[flip \neq 0] \cdot \mathcal{I} = [flip \neq 0] \cdot x \leq x \text{ .}$$

For completeness we also show that \mathcal{I} is actually invariant by satisfying (2.2):

$$\begin{aligned} wp(\text{body}, \mathcal{I}) &= wp((flip := 1[p]x := x + 1), [flip = 0] \cdot (x + \frac{1-p}{p}) \\ &\quad + [flip \neq 0] \cdot x) \\ &= p \cdot x + (1-p) \cdot ([flip = 0] \cdot (x + 1 + \frac{1-p}{p}) + [flip \neq 0] \cdot (x + 1)) \\ &= x + [flip = 0] \cdot \frac{1-p}{p} + [flip \neq 0] \cdot (1-p) \\ &\geq [flip = 0] \cdot (x + \frac{1-p}{p}) = [G] \cdot \mathcal{I} \end{aligned}$$

As we can see we need to know the exact pre-expectation beforehand if we want to verify a program’s property with an invariant. Technically also 0 is an invariant, because it trivially satisfies condition (2.2), but is not strong enough to prove anything about the program because expectations are defined to be non-negative and hence $0 \leq wp(P, postE)$ always holds.

On the other hand, wp yields the least fixed-point w.r.t. a given post-expectation, hence the pre-expectation is given to us. Furthermore let F be a loop’s least fixed-point, P the loop’s body and f a post-expectation:

$$\begin{aligned} [-G] \cdot F &= [-G] \cdot ([G] \cdot wp(P, F) + [-G] \cdot f) = [-G] \cdot f \leq f \\ &\quad \text{and} \\ [G] \cdot F &= [G] \cdot ([G] \cdot wp(P, F) + [-G] \cdot f) = [G] \cdot wp(P, F) \leq wp(P, F) \end{aligned}$$

Consequently, every fixed-point of a loop is also a loop invariant, but obviously not every invariant is also a loop’s fixed-point. However, the former fact will later give us a strategy to find invariants.

2.5 PRINSYS

PRINSYS² (pronounced “princess”) is a tool for generating and verifying probabilistic loop invariants [10]. Given a program PRINSYS can guide the user in finding an invariant or verify if an invariant provided by the user is actually invariant. In Chapter 4 we will investigate PRINSYS in detail, perform a benchmark and discover the current bottleneck in its implementation.

²PRINSYS can be downloaded at <http://moves.rwth-aachen.de/research/tools/prinsys/>

Chapter 3

Finding Invariants

In the previous chapter we introduced some necessary preliminaries including the weakest pre-expectation semantics. We have seen that loops are resolved by finding the least fixed-point of the loop. This fixed-point is generally very hard to find and requires a sophisticated user to do the calculations. First we are going to consider how to find the least fixed-point and after that will proceed to one of the main parts of this thesis: Calculation of various invariants providing insights into what probabilistic loop invariants look like as well as a benchmark suite for evaluating PRINSYS.

3.1 Fixed-Point Iteration

Looking back at the preliminaries we defined the weakest pre-expectation of a probabilistic loop as

$$wp(\text{while}(G) \{P\}, f) = \text{lfp}_F([G] \cdot wp(P, F) + [\neg G] \cdot f)$$

which requires us to find the *least fixed-point* of the loop. Luckily, due to Kleene's fixed-point theorem [14] this process comes down to finding the supremum of a k -fold application of wp to post-expectation 0 instead of F . Technically speaking:

$$\text{lfp}_F(\underbrace{[G] \cdot wp(P, F) + [\neg G] \cdot f}_{\Phi(F)}) = \sup_k(\underbrace{[G] \cdot wp(P, 0) + [\neg G] \cdot f}_{\Phi(0)})^k.$$

Every fixed-point obtained by k -fold application of Φ is the least fixed-point for all loops that terminate almost surely [17]. "Almost sure" termination means that the probability of the loop terminating is 1, which will be the case for all investigated programs in this thesis.

3.2 Invariants for Expected Outcomes

In the following sections we are going to model programs for various discrete probability distributions and verify properties of them by computing their weakest pre-expectation.

As we can see we are already stuck until we find the least fixed-point of Φ . As we have learned in Section 3.1 the computation of a least fixed-point comes down to finding the supremum of a k -fold application of Φ to post-expectation 0:

$$wp(P, c) = wp(c := 0, wp(i := 0, \underbrace{\sup_k([n-i-1 \geq 0] \cdot wp((c := c+1[p]skip); i := i+1, 0) + [n-i-1 \not\geq 0] \cdot c)^k)}_{\Phi(0)}))$$

We try to find this supremum starting with $\Phi(0)$:

$$\begin{aligned} \Phi(0) &= \underbrace{[n-i-1 \geq 0] \cdot wp((c := c+1[p]skip); i := i+1, 0)}_{=0} + [n-i-1 \not\geq 0] \cdot c \\ &= [n-i-1 \not\geq 0] \cdot c \end{aligned}$$

To compute $\Phi^2(0)$ we reinsert the result of $\Phi(0)$ back into Φ :

$$\begin{aligned} \Phi^2(0) &= \Phi(\Phi(0)) \\ &= [n-i-1 \geq 0] \cdot wp((c := c+1[p]skip); i := i+1, [n-i-1 \not\geq 0] \cdot c) \\ &\quad + [n-i-1 \not\geq 0] \cdot c \\ &= [n-i-1 \geq 0] \cdot wp((c := c+1[p]skip), [n-(i+1)-1 \not\geq 0] \cdot c) \\ &\quad + [n-i-1 \not\geq 0] \cdot c \\ &= [n-i-1 \geq 0] \cdot (p \cdot [n-(i+1)-1 \not\geq 0] \cdot (c+1) \\ &\quad \quad \quad + (1-p) \cdot [n-(i+1)-1 \not\geq 0] \cdot c) \\ &\quad + [n-i-1 \not\geq 0] \cdot c \\ &= p \cdot [n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0] \\ &\quad + c \cdot [n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0] \\ &\quad + c \cdot [n-i-1 \not\geq 0] \end{aligned}$$

$[n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0]$ can be written as $[1 \leq n-i < 2]$. If we assume integers we can follow that $[n-i = 1]$ holds and thus:

$$\begin{aligned} \Phi^2(0) &= (p+c) \cdot [n-i = 1] \\ &\quad + c \cdot [n-i-1 \not\geq 0] \end{aligned}$$

Note that the last line is the result of the previous iteration. From now on we will skip the easy computations, because those grow larger and larger but follow the same

principles and will only provide the results (and occasionally some intermediate steps for better understanding) of the individual fixed-point iteration steps.

$$\begin{aligned}
 \Phi^3(0) &= \Phi(\Phi^2(0)) \\
 &= p^2 \cdot [n-i = 2] \cdot (c+2) \\
 &+ 2p(1-p) \cdot [n-i = 2] \cdot (c+1) \\
 &+ p \cdot [n-i = 1] \cdot (c+1) \\
 &+ (1-p)^2 \cdot [n-i = 2] \cdot c \\
 &+ (1-p) \cdot [n-i = 1] \cdot c \\
 &+ [n-i-1 \not\equiv 0] \cdot c
 \end{aligned}$$

$$\begin{aligned}
 &= 2p \cdot [n-i = 2] \\
 &+ c \cdot [n-i = 2] \\
 &+ p \cdot [n-i = 1] \\
 &+ c \cdot [n-i = 1] \\
 &+ c \cdot [n-i-1 \not\equiv 0]
 \end{aligned}$$

$$\begin{aligned}
 &= (2p+c) \cdot [n-i = 2] \\
 &+ \Phi^2(0)
 \end{aligned}$$

$$\begin{aligned}
 \Phi^4(0) &= \Phi(\Phi^3(0)) \\
 &= (3p+c) \cdot [n-i = 3] \\
 &+ (2p+c) \cdot [n-i = 2] \\
 &+ (p+c) \cdot [n-i = 1] \\
 &+ c \cdot [n-i-1 \not\equiv 0]
 \end{aligned}$$

If we inspect this sum we can assume that $\Phi^{k+1}(0)$ is given by:

$$\Phi^{k+1}(0) = \sum_{j=1}^k ([n-i = j] \cdot (j \cdot p + c)) + c \cdot [n-i-1 \not\equiv 0]$$

If we investigate that sum we can conclude that only $\Phi^{n-i+1}(0) \neq 0$, as this is the only time where the $(n-i)^{th}$ summand occurs for which $[n-i = j]$ is true and the sum therefore is unequal to 0. Since we are looking for the supremum of a k -fold application of Φ the whole sum simplifies to the $(n-i)^{th}$ summand:

$$((n-i) \cdot p + c) + c \cdot [n-i-1 \not\equiv 0]$$

As expectations are defined to be non-negative, we have to add constraints to this expression to prevent this from happening. This *sanity check* should handle what happens if $n-i < 1$ and thus the loop is not entered. In that case $(n-i) \cdot p$ should not be considered in calculating the pre-expectation, hence we assume our fixed point to be²:

$$F = [n-i-1 \geq 0] \cdot ((n-i) \cdot p + c) + [n-i-1 \not\geq 0] \cdot c \quad (3.2)$$

In order to verify that F actually is a fixed-point of Φ we merely have to check if $\Phi(F) = F$ holds.

$$\begin{aligned} \Phi(F) &= [n-i-1 \geq 0] \cdot wp((c := c+1[p]skip); i := i+1, \\ &\quad [n-i-1 \geq 0] \cdot ((n-i) \cdot p + c) + [n-i-1 \not\geq 0] \cdot c) \\ &\quad + [n-i-1 \not\geq 0] \cdot c \\ &= [n-i-1 \geq 0] \cdot wp((c := c+1[p]skip); \\ &\quad [n-i-2 \geq 0] \cdot ((n-i-1) \cdot p + c) + [n-i-2 \not\geq 0] \cdot c) \\ &\quad + [n-i-1 \not\geq 0] \cdot c \\ &= [n-i-1 \geq 0] \cdot (p \cdot ([n-i-2 \geq 0] \cdot ((n-i-1) \cdot p + c + 1) + [n-i-2 \not\geq 0] \cdot (c+1)) \\ &\quad + (1-p) \cdot ([n-i-2 \geq 0] \cdot ((n-i-1) \cdot p + c) + [n-i-2 \not\geq 0] \cdot c)) \\ &\quad + [n-i-1 \not\geq 0] \cdot c \\ &= p \cdot ([n-i-1 \geq 0] \cdot [n-i-2 \geq 0] \cdot ((n-i-1) \cdot p + c + 1) \\ &\quad + [n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0] \cdot (c+1)) \\ &\quad + [n-i-1 \geq 0] \cdot [n-i-2 \geq 0] \cdot ((n-i-1) \cdot p + c) \\ &\quad + [n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0] \cdot c \\ &\quad - p \cdot ([n-i-1 \geq 0] \cdot [n-i-2 \geq 0] \cdot ((n-i-1) \cdot p + c) \\ &\quad + [n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0] \cdot c) \\ &\quad + [n-i-1 \not\geq 0] \cdot c \\ &= p \cdot ([n-i-1 \geq 0] \cdot [n-i-2 \geq 0] + [n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0]) \\ &\quad + [n-i-1 \geq 0] \cdot [n-i-2 \geq 0] \cdot ((n-i-1) \cdot p + c) \\ &\quad + [n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0] \cdot c \\ &\quad + [n-i-1 \not\geq 0] \cdot c \end{aligned}$$

²Note that this fixed-point (and others) can and will be simplified in Chapter 4, as we will discuss how PRINSYS can help to fix invariants if we miss some constraints.

As we predicted the expected value of c with regards to program P equals $n \cdot p$ if $n \geq 1$, i.e. the user flips the coin at least once. Of course this value changes if the variables are initialized differently, for example if c is initialized with 3 and i is initialized with a value lower than n the result would have been $(n-i) \cdot p + 3$.

Note that this fixed-point is also an invariant (see Section 2.4) and very intuitive, because it exactly captures what the program does. This is due to the fact that from each possible state in the loop's execution we can tell what the expected outcome will be by considering the current value of c and the amount of remaining iterations $n-i$. Furthermore this fixed-point is very close to what probability theory teaches us about the expected outcome of a random variable following the binomial distribution. However, contrary to formula (3.1) an invariant has to consider different initializations of the variables. This gives a first impression on how to find invariants without computing fixed-points by figuring out what the program does and express this expected behavior by one formula for any given program state.

3.2.2 Negative Binomial Distribution

Next we are going to perform a fixed-point iteration for the *negative binomial distribution* [4]. Contrary to the binomial distribution a random variable x follows the negative binomial distribution with parameters r (number of successes) and p (probability of success) if the probability that exactly k Bernoulli trials are performed to achieve r successes is given by the probability mass function

$$Pr(x = k) = \binom{k-1}{r-1} p^r (1-p)^{k-r} . \quad (3.3)$$

The following pGCL program models this distribution:

```

1 var x, c;
2
3 x:=0; //trial count
4 c:=0; //success count
5
6 while (r-c-1 >= 0) { //r: num successes needed
7     (c:=c+1 [p] skip);
8     x:=x+1;
9 }
```

Listing 3.2: negbinom.pgcl

In the program above r is the number of successes needed (given by the user), x is the current number of Bernoulli trials and c counts the number of successes, where c is incremented with probability p . With probability $(1-p)$ the Bernoulli trial is unsuccessful and c is not incremented. The loop ends as soon as $r = c$, because $r-c-1 \geq 0$ becomes false.

As the number of successes in this program is fixed we are not interested in the expected number of successes (that is c), as this number will always be equal to r when the program terminates. Instead we are interested in the expected number of Bernoulli trials performed

$$\begin{aligned}
 &= [r-c-1 \geq 0] \cdot [r-c-2 \not\geq 0] \cdot p \cdot (x+1) \\
 &+ [r-c-1 \not\geq 0] \cdot x
 \end{aligned}$$

If we assume integers we can simplify $[r-c-1 \geq 0] \cdot [r-c-2 \not\geq 0]$ to $[r-c = 1]$ and obtain:

$$\begin{aligned}
 \Phi^2(0) &= [r-c = 1] \cdot p \cdot (x+1) & \Phi^3(0) &= [r-c = 2] \cdot p^2 \cdot (x+2) \\
 &+ [r-c-1 \not\geq 0] \cdot x & &+ [r-c = 1] \cdot p \cdot (x+1+(1-p) \cdot (x+2)) \\
 & & &+ [r-c-1 \not\geq 0] \cdot x
 \end{aligned}$$

$$\begin{aligned}
 \Phi^4(0) &= [r-c = 3] \cdot p^3 \cdot (x+3) \\
 &+ [r-c = 2] \cdot p^2 \cdot (x+2+2 \cdot (1-p) \cdot (x+3)) \\
 &+ [r-c = 1] \cdot p \cdot (x+1+(1-p) \cdot (x+2)+(1-p)^2 \cdot (x+3)) \\
 &+ [r-c-1 \not\geq 0] \cdot x
 \end{aligned}$$

As we can already see this fixed-point iteration seems to turn out way harder as it does not develop by only adding a summand to the result of the last fixed-point iteration step. Instead the results of the last steps grow as well, indicating that a sum capturing the result with respect to iteration k will probably contain a sum itself. To further gain an intuition on what the result of the k^{th} iteration might look like to find the supremum we will provide two more steps.

$$\begin{aligned}
 \Phi^5(0) &= [r-c = 4] \cdot p^4 \cdot (x+4) \\
 &+ [r-c = 3] \cdot p^3 \cdot (x+3+3 \cdot (1-p) \cdot (x+4)) \\
 &+ [r-c = 2] \cdot p^2 \cdot (x+2+2 \cdot (1-p) \cdot (x+3)+3 \cdot (1-p)^2 \cdot (x+4)) \\
 &+ [r-c = 1] \cdot p \cdot (x+1+(1-p) \cdot (x+2)+(1-p)^2 \cdot (x+3)+(1-p)^3 \cdot (x+4)) \\
 &+ [r-c-1 \not\geq 0] \cdot x
 \end{aligned}$$

$$\begin{aligned}
 \Phi^6(0) &= [r-c = 5] \cdot p^5 \cdot (x+5) \\
 &+ [r-c = 4] \cdot p^4 \cdot (x+4+4 \cdot (1-p) \cdot (x+5)) \\
 &+ [r-c = 3] \cdot p^3 \cdot (x+3+3 \cdot (1-p) \cdot (x+4)+6 \cdot (1-p)^2 \cdot (x+5)) \\
 &+ [r-c = 2] \cdot p^2 \cdot (x+2+2 \cdot (1-p) \cdot (x+3)+3 \cdot (1-p)^2 \cdot (x+4)+4 \cdot (1-p)^3 \cdot (x+5)) \\
 &+ [r-c = 1] \cdot p \cdot (x+1+(1-p) \cdot (x+2)+(1-p)^2 \cdot (x+3)+(1-p)^3 \cdot (x+4) \\
 &\hspace{15em} + (1-p)^4 \cdot (c+5)) \\
 &+ [r-c-1 \not\geq 0] \cdot x
 \end{aligned}$$

And now we are going to use a trick. We will write the individual summands as a sum of sums (and more sums), where the sums in the end of each line will provide the different

coefficients to the summands.

$$\begin{aligned}
 \Phi^6(0) &= [r-c = 5] \cdot p^5 \cdot \sum_{t=0}^0 ((1-p)^t \cdot (x+5+t)) \cdot \sum_{s=1}^{t+1} \sum_{l=1}^s \sum_{u=1}^l (u) \\
 &+ [r-c = 4] \cdot p^4 \cdot \sum_{t=0}^1 ((1-p)^t \cdot (x+4+t)) \cdot \sum_{s=1}^{t+1} \sum_{l=1}^s (l) \\
 &+ [r-c = 3] \cdot p^3 \cdot \sum_{t=0}^2 ((1-p)^t \cdot (x+3+t)) \cdot \sum_{s=1}^{t+1} (s) \\
 &+ [r-c = 2] \cdot p^2 \cdot \sum_{t=0}^3 ((1-p)^t \cdot (x+2+t) \cdot (t+1)) \\
 &+ [r-c = 1] \cdot p^1 \cdot \sum_{t=0}^4 ((1-p)^t \cdot (x+1+t) \cdot 1) \\
 &+ [r-c-1 \not\geq 0]x
 \end{aligned}$$

Every single one of those sums converges as the number of iterations grows. Finally we will get:

$$\begin{aligned}
 \Phi^{k+1}(0) &= [r-c = k] \cdot p^k \cdot \left(\frac{x \cdot p + k}{p^{k+1}} \right) \\
 &\vdots \\
 &+ [r-c = 2] \cdot p^2 \cdot \left(\frac{x \cdot p + 2}{p^3} \right) \\
 &+ [r-c = 1] \cdot p^1 \cdot \left(\frac{x \cdot p + 1}{p^2} \right) \\
 &+ [r-c-1 \not\geq 0] \cdot x
 \end{aligned}$$

$$\Phi^{k+1}(0) = \sum_{j=1}^k ([r-c = j] \cdot \frac{x \cdot p + j}{p}) + [r-c-1 \not\geq 0] \cdot x$$

Now we are ready to determine our fixed-point F .

$$F = [r-c-1 \geq 0] \cdot \frac{x \cdot p + r - c}{p} + [r-c-1 \not\geq 0] \cdot x \tag{3.4}$$

To verify that this is a fixed-point we are going to check if $\Phi(F) = F$.

$$\begin{aligned}
 \Phi(F) &= [r-c-1 \geq 0] \cdot wp((c := c+1[p]skip); x := x+1, \\
 &\quad [r-c-1 \geq 0] \cdot \frac{x \cdot p + r - c}{p} + [r-c-1 \not\geq 0] \cdot x) \\
 &+ [r-c-1 \not\geq 0] \cdot x
 \end{aligned}$$

$$\begin{aligned}
 &= [r-c-1 \geq 0] \cdot (p \cdot ([r-c-2 \geq 0] \cdot \frac{(x+1) \cdot p+r-c-1}{p} + [r-c-2 \not\geq 0] \cdot (x+1)) \\
 &\quad + (1-p) \cdot ([r-c-1 \geq 0] \cdot \frac{(x+1) \cdot p+r-c}{p} + [r-c-1 \not\geq 0] \cdot (x+1))) \\
 &+ [r-c-1 \not\geq 0] \cdot x \\
 \\
 &= [r-c-1 \geq 0] \cdot [r-c-2 \geq 0] \cdot ((x+1) \cdot p+r-c-1) \\
 &+ \underbrace{[r-c-1 \geq 0] \cdot [r-c-2 \not\geq 0]}_{=[r-c-1=0]} \cdot p \cdot (x+1) \\
 &+ [r-c-1 \geq 0] \cdot \frac{(x+1) \cdot p+r-c}{p} \\
 &- [r-c-1 \geq 0] \cdot ((x+1) \cdot p+r-c) \\
 &+ [r-c-1 \not\geq 0] \cdot x
 \end{aligned}$$

Due to constraint $[r-c-1 \geq 0] \cdot [r-c-2 \not\geq 0]$ (which is equal to $[r-c-1 = 0]$ if we assume integers) we can add a ‘free’ $r-c-1$ to line two.

$$\begin{aligned}
 \Phi(F) &= [r-c-2 \geq 0] \cdot ((x+1) \cdot p+r-c-1) \\
 &+ [r-c-1 = 0] \cdot ((x+1) \cdot p+r-c-1) \\
 &+ [r-c-1 \geq 0] \cdot \frac{(x+1) \cdot p+r-c}{p} \\
 &- [r-c-1 \geq 0] \cdot ((x+1) \cdot p+r-c) \\
 &+ [r-c-1 \not\geq 0] \cdot x
 \end{aligned}$$

If we now factorize $((x+1) \cdot p+r-c-1)$ and assume integers we can see that

$$[r-c-2 \geq 0] + [r-c-1 = 0] = [r-c-1 \geq 0]$$

and thus

$$\begin{aligned}
 \Phi(F) &= [r-c-1 \geq 0] \cdot ((x+1) \cdot p+r-c-1) \\
 &+ [r-c-1 \geq 0] \cdot \frac{(x+1) \cdot p+r-c}{p} \\
 &- [r-c-1 \geq 0] \cdot ((x+1) \cdot p+r-c) \\
 &+ [r-c-1 \not\geq 0] \cdot x \\
 \\
 &= [r-c-1 \geq 0] \cdot ((x+1) \cdot p+r-c-1 + \frac{(x+1) \cdot p+r-c}{p} - ((x+1) \cdot p+r-c)) \\
 &+ [r-c-1 \not\geq 0] \cdot x
 \end{aligned}$$

$$\begin{aligned}
 &= [r-c-1 \geq 0] \cdot \frac{x \cdot p^{r-c}}{p} + [r-c-1 \not\geq 0] \cdot x \\
 &= F
 \end{aligned}$$

Hence F is a fixed-point of Φ . Inserting into our weakest pre-expectation calculation yields:

$$\begin{aligned}
 wp(P, x) &= wp(x := 0, wp(c := 0, \\
 &\quad \sup_k ([r-c-1 \geq 0] \cdot wp((c := c+1[p] skip); x := x+1, 0) \\
 &\quad \quad \underbrace{+ [r-c-1 \not\geq 0] \cdot x)^k}_{\Phi(0)}) \\
 &= wp(x := 0, wp(c := 0, [r-c-1 \geq 0] \cdot \frac{x \cdot p^{r-c}}{p} + [r-c-1 \not\geq 0] \cdot x)) \\
 &= wp(x := 0, [r-1 \geq 0] \frac{x \cdot p^{r-1}}{p} + [r-1 \not\geq 0] \cdot x) \\
 &= [r-1 \geq 0] \cdot \frac{r}{p}
 \end{aligned}$$

As desired the expected value of x with regards to program P is $\frac{r}{p}$ if $r \geq 1$, i.e. at least one success is needed. Again, this value might change due to different initializations of the variables, but it will follow all rules of probability theory associated with the negative binomial distribution. Note that once more this fixed-point (and therefore invariant) is really intuitive, i.e. is very similar to what probability theory teaches us about the expected outcome of a random variable following the negative binomial distribution, but additionally considers different initializations and program states. This enables us at any given program state during the loop's execution to give an exact expectation of the loop's outcome.

3.2.3 Hypergeometric Distribution

A particularly interesting distribution is the *hypergeometric distribution* [4]. Consider N elements of which a subset of M elements have a certain desired property. A random variable x follows the hypergeometric distribution with parameters N (number of elements), $M \subseteq N$ (elements with a certain property) and n (sample size) if the probability to draw exactly k elements with the desired property in a sample is given by the probability mass function

$$Pr(x = k) = \frac{\binom{M}{k} \cdot \binom{N-M}{n-k}}{\binom{N}{n}} . \quad (3.5)$$

$$\Phi(0) = [n-x-1 \not\geq 0] \cdot c$$

$$\begin{aligned} \Phi^2(0) &= \Phi(\Phi(0)) \\ &= \left(\frac{M-c}{N-x} + c\right) \cdot [n-x = 1] \\ &\quad + c \cdot [n-x-1 \not\geq 0] \end{aligned}$$

$$\begin{aligned} \Phi^3(0) &= \Phi(\Phi^2(0)) \\ &= \left(2 \cdot \frac{M-c}{N-x} + c\right) \cdot [n-x = 2] \\ &\quad + \left(\frac{M-c}{N-x} + c\right) \cdot [n-x = 1] \\ &\quad + c \cdot [n-x-1 \not\geq 0] \end{aligned}$$

$$\begin{aligned} \Phi^4(0) &= \Phi(\Phi^3(0)) \\ &= \left(3 \cdot \frac{M-c}{N-x} + c\right) \cdot [n-x = 3] \\ &\quad + \left(2 \cdot \frac{M-c}{N-x} + c\right) \cdot [n-x = 2] \\ &\quad + \left(\frac{M-c}{N-x} + c\right) \cdot [n-x = 1] \\ &\quad + c \cdot [n-x-1 \not\geq 0] \end{aligned}$$

The careful reader will notice that these fixed-point iteration steps are the exact same steps as in the binomial distribution fixed-point iteration in Section 3.2.1, except the probability parameter p has been switched to $\frac{M-c}{N-x}$. Apparently, changing the probability parameter during execution has no effect on the verification process, because the fixed-point describes the behavior of the loop for every given state and the probability to draw an element of M is $\frac{M-c}{N-x}$ in any state of the loop. Thus, we can conclude that $\Phi^{k+1}(0)$ is given by:

$$\Phi^{k+1}(0) = \sum_{j=1}^k ([n-x = j] \cdot (j \cdot \frac{M-c}{N-x} + c)) + [n-x-1 \not\geq 0] \cdot c$$

Analogous to our reasoning in Section 3.2.1 our fixed-point is:

$$F = [n-x-1 \geq 0] \cdot ((n-x) \cdot \frac{M-c}{N-x} + c) + [n-x-1 \not\geq 0] \cdot c$$

If we insert this back into our weakest pre-expectation calculation (ultimately setting x and c to 0) we will obtain $n \cdot \frac{M}{N}$ if the user draws at least one element. We will omit proof of $\Phi(F) = F$ here, as it works completely analogous to the binomial distribution. Again this fixed-point is very intuitive and we could have guessed it beforehand by expressing $n \cdot \frac{M}{N}$ w.r.t. different initializations of the variables. Then we would only need to check if $\Phi(F) = F$.

This gives us a strategy to find invariants. As we have seen in Section 2.4 every fixed-point is also an invariant, because it satisfies all conditions of Theorem 1. Instead of performing a lengthy fixed-point iteration to prove a property we can use that property to find an invariant by thinking of an expression \mathcal{I} that predicts the program's behavior in any given state w.r.t. the pre- and post-expectation and is a fixed-point of the loop.

Then we only check if $\Phi(\mathcal{I}) = \mathcal{I}$ and thus indeed found an invariant that lets us prove the property. On the other hand, if $\Phi(\mathcal{I}) \neq \mathcal{I}$ we may be able to fix the invariant candidate based on the knowledge gained by failing to prove the equation.

However, if we have no idea how our fixed-point might look like or what your program does, fixed-point iteration provides a nice view on how the program behaves. Nonetheless, fixed-point iteration might not always work as it is possible for the results to just blow up in size, making it almost impossible to find a formula describing Φ^k . We will see this in the next section where we investigate an example we are unable to analyze via fixed-point iteration.

3.2.4 Uniform Distribution

A random variable x follows the *uniform distribution* [4] if the probability of $x = k$, where $a \leq k \leq b$ is given by the probability mass function

$$P(x = k) = \frac{1}{b-a} .$$

Since PGCL does not offer a probabilistic choice that draws from an interval, we are left with Bernoulli trials to model a uniform distribution. An algorithm using Bernoulli trials to model a uniform distribution can be found in [15]. Translated to PGCL the algorithm looks like this:

```

1 var v,c,running;
2 v:=1;
3 c:=0;
4 running:=0;
5 while(running=0){
6     v:=2*v;
7     (c:=2*c+1[0.5]c:=2*c);
8     if(v-n>=0){
9         if{n-c>0}{ //terminate
10            running:=1;
11        }
12        else{ //roll back
13            v:=v-n;
14            c:=c-n;
15        }
16    }
17 }
```

Listing 3.4: uniform.pgcl

In this program v is an auxiliary variable that is doubled in every iteration until it exceeds a value n that is given by the user and sets the range of possible values to $\{0, \dots, n-1\}$, c is the random variable that is returned at the end of the program. If v exceeds n and c is in range $\{0, \dots, n-1\}$ the loop guard will be set to 1 and the program terminates, else if c is not in this range n is subtracted from both v and c to jump back into range and try again.

Because this program is very hard to understand a program graph can be found in Fig. 3.1 illustrating the process to show that this algorithm indeed models a uniform

distribution. As mentioned earlier fixed-point iteration fails on this example, as the size of the results grows very fast with each step. To give you an impression we will look at the result of the third step:

$$\begin{aligned}
 \Phi^3(0) = & [running = 0][\frac{n}{2} > v \geq \frac{n}{4}][c < \frac{n-3}{4}](c + \frac{3}{4}) \\
 & + [running = 0][\frac{n}{2} > v \geq \frac{n}{4}][c < \frac{n-2}{4}](c + \frac{2}{4}) \\
 & + [running = 0][\frac{n}{2} > v \geq \frac{n}{4}][c < \frac{n-1}{4}](c + \frac{1}{4}) \\
 & + [running = 0][\frac{n}{2} > v \geq \frac{n}{4}][c < \frac{n}{4}]c \\
 & + [running = 0][v \geq \frac{3n}{4}][\frac{3n-3}{4} > c \geq \frac{n-1}{2}](c - \frac{n}{2} + \frac{3}{4}) \\
 & + [running = 0][v \geq \frac{3n}{4}][\frac{3n-2}{4} > c \geq \frac{n-1}{2}](c - \frac{n}{2} + \frac{2}{4}) \\
 & + [running = 0][v \geq \frac{3n}{4}][\frac{3n-1}{4} > c \geq \frac{n}{2}](c - \frac{n}{2} + \frac{1}{4}) \\
 & + [running = 0][v \geq \frac{3n}{4}][\frac{3n}{4} > c \geq \frac{n}{2}](c - \frac{n}{2}) \\
 & + [running = 0][v \geq \frac{n}{2}][c < \frac{n-1}{2}](c + \frac{1}{2}) \\
 & + [running = 0][v \geq \frac{n}{2}][c < \frac{n}{2}]c \\
 & + [running \neq 0]c
 \end{aligned}$$

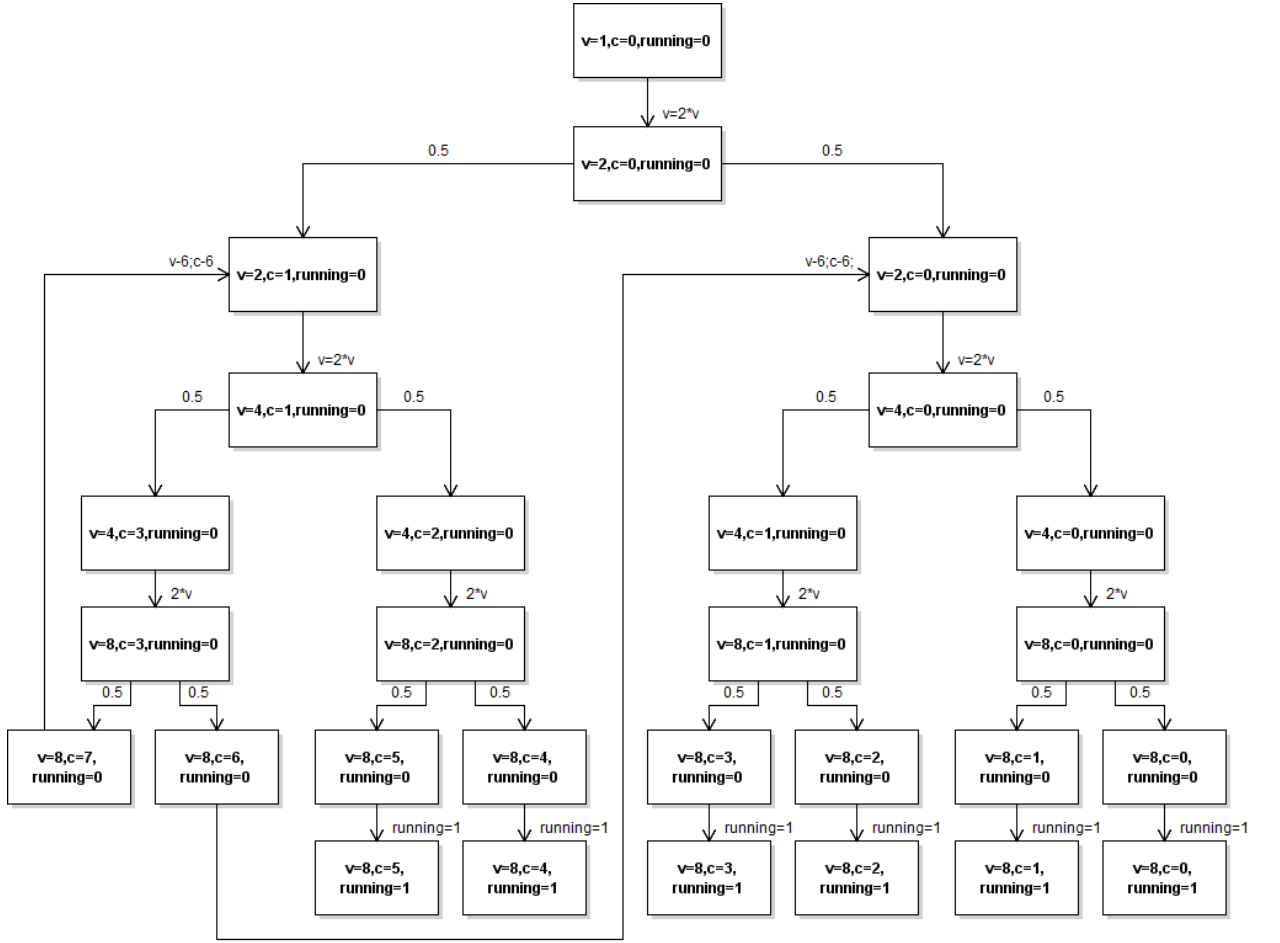
If investigate this result w.r.t. the graph in Fig. 3.1 we are able to assume that the constraints model the different nodes in the graph and add the expected value of c w.r.t. the leaf nodes that are reachable from that node. For example consider the first 4 lines:

$$\begin{aligned}
 & [running = 0][\frac{n}{2} > v \geq \frac{n}{4}][c < \frac{n-3}{4}](c + \frac{3}{4}) \\
 & + [running = 0][\frac{n}{2} > v \geq \frac{n}{4}][c < \frac{n-2}{4}](c + \frac{2}{4}) \\
 & + [running = 0][\frac{n}{2} > v \geq \frac{n}{4}][c < \frac{n-1}{4}](c + \frac{1}{4}) \\
 & + [running = 0][\frac{n}{2} > v \geq \frac{n}{4}][c < \frac{n}{4}]c
 \end{aligned}$$

All these constraints become true for the upper right node labeled $v = 2$, $c = 0$, $running = 0$. Inserting these values into the 4 lines above yields:

$$0 + \frac{3}{4} + 0 + \frac{2}{4} + 0 + \frac{1}{4} + 0 = \frac{6}{4} = 1.5$$

This is exactly the mean value of the leaf nodes reachable from that node, i.e. $\frac{0+1+2+3}{4}$. While we proceed iterating the constraints will more and more define the different levels in the graph while the graph grows. As there are only a finite number of states for a fixed range n this fixed-point iteration will at one point just add redundant information w.r.t.


 Figure 3.1: uniform.pgcl program graph for $n = 6$

the fixed parameter n , that means the whole sum is going to describe a lot more nodes than needed for range $\{0, \dots, n - 1\}$. These nodes will either be valued with 0 due to unsatisfied constraints, or split up the result into more summands.

Summing up, we can conclude that fixed-point iteration is a very *powerful* tool for proving a program's properties. However, there are cases where fixed-point iteration becomes explicitly *hard*, *error-prone* and might not lead to a conclusive result at all. Furthermore we have seen that we may be able to guess a fixed-point of the loop. This gives us a *strategy* to find invariants by looking for a fixed-point that captures the program's behavior w.r.t. a property we want to prove and merely show that it is a fixed-point of Φ . We will benefit from this in the next sections as we are going to revisit our programs to look at post-expectations of the form $[x=k]$, i.e. the pre-expectation should be the probability of x having value k at the end of the program. As we will see, those invariants will not be

linear anymore but *exponential*.

3.3 Invariants for Individual Outcomes

From now on, since we learned that we may be able to guess the program's fixed-points by looking at known properties of the different distributions we will leave fixed-point iterations behind and instead try to guess the fixed-point F of the loop and verify it by ensuring that $\Phi(F) = F$. This has one major reason: As we are investigating post-expectations of the form $[x = k]$ we will see that all our invariants will turn from linear to exponential, for example because the respective distribution is described by binomial coefficients. Therefore fixed-point iteration becomes harder to perform manually. On the other hand, guessing a fixed-point and proving that it indeed is a fixed-point is a lot easier.

The pre-expectation of post-expectation $[x = k]$ is the probability that at the end of the program $x = k$ holds. This will verify that our programs indeed model the respective distribution, because the pre-expectation of the program should be the probability mass function of the distribution. To begin with, we are looking at the geometric distribution since this one is particularly easy.

3.3.1 Geometric Distribution

The *geometric distribution* [4] can be described as the number of failed Bernoulli trials until success. Formally, a random variable x follows the geometric distribution if the probability that exactly k failed trials occur until success is given by the probability mass function

$$P(x = k) = (1 - p)^k \cdot p . \quad (3.6)$$

The following PGCL program models this distribution:

```
1 var x, flip;
2 x:=0;
3 flip:=0;
4 while(flip = 0){ //until success
5     (flip:=1[p]x:=x+1); //flip coin
6 }
```

Listing 3.5: geometric.pgcl

In this program x is the number of failed trials. As soon as a successful trial occurs variable *flip* is set to 1 and the loop terminates. If we compute the weakest pre-expectation for post-expectation x , i.e. the expected number of trials needed until success, a fixed-point is given by

$$F = [flip = 0] \cdot (x + \frac{1-p}{p}) + [flip \neq 0] \cdot x .$$

As this result is given in detail in published work we will not prove this here.³ Instead we are interested in post-expectation $[x = k]$, i.e. that x has exactly k as a value at the

³Proof can be found in Appendix A.1

end of the program and therefore the pre-expectation should be the probability for that to happen.

As this probability is described by (3.6) we will use this formula to guess a fixed-point and invariant instead of performing a fixed-point iteration to prove the program's property. Since k is the only variable in this formula and describes the number of failed trials we subtract the number x of failed trials that already happened and obtain a formula that describes the probability of $[x = k]$ in any state of the loops execution:

$$(1 - p)^{k-x} \cdot p$$

As we have learned earlier this expectation requires sanity checks, for example to distinct cases where the loop is entered and where it is not entered or to prevent negativity of expectations. Obviously, the expression above only calculates the correct probability if the loop is entered, i.e. $[flip = 0]$ is true. On the other hand, if $[flip \neq 0]$ holds the probability of $x = k$ should not be 0, because x might have been initialized with value k , hence the probability that $x = k$ is 1. This will lead to the following guess for an invariant:

$$\mathcal{I} = [flip = 0] \cdot (1 - p)^{k-x} \cdot p + [flip \neq 0] \cdot [x = k]$$

Unfortunately this is not enough to prove that $\Phi(\mathcal{I}) = \mathcal{I}$, i.e. \mathcal{I} is a fixed-point that satisfies Theorem 1. We will need one more constraint to prove this equality, namely that $[k - x \geq 0]$. In other words this sanity check ensures that x is not greater than k , because $x > k$ would mean that there is no possibility for us to achieve exactly k failed trials since we already have more than that. Thus, the probability should be 0. Finally we obtain our invariant candidate:

$$\mathcal{I} = [flip = 0 \wedge k - x \geq 0] \cdot (1 - p)^{k-x} \cdot p + [flip \neq 0] \cdot [x = k]$$

As mentioned before, we are not going through fixed-point iteration anymore and instead will only try to prove that our invariant candidate is a fixed-point of the loop. This saves us a lot of work, as we are able to consistently guess the invariant by looking at the probability mass function, express it w.r.t. different variable initializations and provide proper constraints to prove that it is a fixed-point.

$$\begin{aligned} \Phi(\mathcal{I}) &= \Phi([flip = 0 \wedge k - x \geq 0] \cdot (1 - p)^{k-x} \cdot p + [flip \neq 0] \cdot [x = k]) \\ &= [flip = 0] \cdot wp((flip := 1[p]x := x + 1), \\ &\quad [flip = 0 \wedge k - x \geq 0] \cdot (1 - p)^{k-x} \cdot p + [flip \neq 0] \cdot [x = k]) \\ &\quad + [flip \neq 0] \cdot [x = k] \\ &= [flip = 0] \cdot (p \cdot [x = k] + (1 - p) \cdot ([flip = 0 \wedge k - x - 1 \geq 0] \cdot (1 - p)^{k-x-1} \cdot p \\ &\quad + [flip \neq 0] \cdot [x + 1 = k])) \\ &\quad + [flip \neq 0] \cdot [x = k] \end{aligned}$$

$$\begin{aligned}
 &= [flip = 0] \cdot ([x = k] \cdot p \\
 &\quad + [flip = 0 \wedge k-x-1 \geq 0] \cdot (1-p)^{k-x} \cdot p \\
 &\quad + [flip \neq 0][x+1 = k] \cdot (1-p)) \\
 &+ [flip \neq 0] \cdot [x = k]
 \end{aligned}$$

$$\begin{aligned}
 &= [flip = 0] \cdot [x = k] \cdot p \\
 &+ [flip = 0] \cdot [k-x-1 \geq 0] \cdot (1-p)^{k-x} \cdot p \\
 &+ \underbrace{[flip = 0] \cdot [flip \neq 0]}_{=0} \cdot [x+1 = k] \cdot (1-p) \\
 &+ [flip \neq 0] \cdot [x = k]
 \end{aligned}$$

$$\begin{aligned}
 &= [flip = 0] \cdot [0 = k-x] \cdot p \\
 &+ [flip = 0] \cdot [k-x \geq 1] \cdot (1-p)^{k-x} \cdot p \\
 &+ [flip \neq 0] \cdot [x = k]
 \end{aligned}$$

In line one of the last result $k-x=0$ holds, hence we know that we can write p as $1 \cdot p = (1-p)^0 \cdot p = (1-p)^{k-x} \cdot p$ and we obtain:

$$\begin{aligned}
 \Phi(\mathcal{I}) &= [flip = 0] \cdot [0 = k-x] \cdot (1-p)^{k-x} \cdot p \\
 &+ [flip = 0] \cdot [k-x \geq 1] \cdot (1-p)^{k-x} \cdot p \\
 &+ [flip \neq 0] \cdot [x = k]
 \end{aligned}$$

This is the where we can see why we added $[k-x \geq 0]$ as a constraint, because now we can follow that

$$[k-x = 0] + [k-x \geq 1] = [k-x = 0 \vee k-x \geq 1] = [k-x \geq 0]$$

and thus conclude:

$$\Phi(\mathcal{I}) = [flip = 0 \wedge k-x \geq 0] \cdot (1-p)^{k-x} \cdot p + [flip \neq 0] \cdot [x = k] = F$$

Consequently \mathcal{I} is a fixed-point and therefore invariant. If we set x and $flip$ to 0 we obtain the probability mass function of the geometric distribution if $k \geq 0$, hence our program indeed models a random variable following the geometric distribution. All we needed to do was to alter the probability mass function to consider different initializations and provide a sanity check that allows us to prove that our invariant is also a fixed-point of the loop. One could even state that finding the arithmetic expression describing the invariant is not hard at all but finding the constraint to prove that it is a fixed-point is the true challenge. Finding those constraints comes down to a *trial and error* approach. Unfortunately we cannot simply add a lot of constraints that we assume to hold for our guess, but have to find the exact set that works, because any additional constraints might make the simplification steps harder or impossible.

3.3.2 Binomial Distribution

Next, we are going to revisit the binomial distribution and try to guess an invariant for post-expectation $[c = k]$. Looking at the program in Lst. 3.1 and probability mass function (3.1) we can assume that our invariant candidate to prove that our program models a binomial distribution is something along the lines of

$$\mathcal{I} = [\alpha] \cdot \binom{n-i}{k-c} \cdot (1-p)^{n-i-(k-c)} \cdot p^{k-c} + [\beta] \cdot [c = k]$$

where α and β are constraints to be determined that suffice as a sanity check to prove $\Phi(\mathcal{I}) = \mathcal{I}$. First of all, we can safely assume that α should contain the loop guard and β the negated loop guard. Furthermore α should rule out that $c > k$, that is we already achieved more successful trials than we want to achieve and hence it is impossible to achieve exactly $c = k$. Therefore we will choose our invariant candidate to be:

$$\mathcal{I} = [n-i-1 \geq 0 \wedge k-c >= 0] \cdot \binom{n-i}{k-c} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} + [n-i-1 \not\geq 0] \cdot [c = k]$$

We are going to check if $\Phi(\mathcal{I}) = \mathcal{I}$:

$$\begin{aligned} \Phi(\mathcal{I}) &= [n-i-1 \geq 0] \cdot wp((c := c+1[p]skip); i := i+1, \\ &\quad [n-i-1 \geq 0 \wedge k-c >= 0] \cdot \binom{n-i}{k-c} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \\ &\quad + [n-i-1 \not\geq 0] \cdot [c = k]) \\ &\quad + [n-i-1 \not\geq 0] \cdot [c = k] \\ &= [n-i-1 \geq 0] \cdot wp((c := c+1[p]skip), \\ &\quad [n-i-2 \geq 0 \wedge k-c >= 0] \cdot \binom{n-i-1}{k-c} \cdot (1-p)^{n-i-1-k+c} \cdot p^{k-c} \\ &\quad + [n-i-2 \not\geq 0] \cdot [c = k]) \\ &\quad + [n-i-1 \not\geq 0] \cdot [c = k] \\ &= [n-i-1 \geq 0] \cdot (p \cdot ([n-i-2 \geq 0 \wedge k-c-1 >= 0] \\ &\quad \cdot \binom{n-i-1}{k-c-1} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c-1} \\ &\quad + [n-i-2 \not\geq 0] \cdot [c+1 = k]) \\ &\quad + (1-p) \cdot ([n-i-2 \geq 0 \wedge k-c >= 0] \\ &\quad \cdot \binom{n-i-1}{k-c} \cdot (1-p)^{n-i-1-k+c} \cdot p^{k-c} \\ &\quad + [n-i-2 \not\geq 0] \cdot [c = k])) \\ &\quad + [n-i-1 \not\geq 0] \cdot [c = k] \end{aligned}$$

$$\begin{aligned}
 &= [n-i-1 \geq 0] \cdot [n-i-2 \geq 0] \cdot [k-c-1 \geq 0] \cdot \binom{n-i-1}{k-c-1} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \\
 &+ [n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0] \cdot [k-c-1 = 0] \cdot p
 \end{aligned} \tag{1}$$

$$\begin{aligned}
 &+ [n-i-1 \geq 0] \cdot [n-i-2 \geq 0] \cdot [k-c \geq 0] \cdot \binom{n-i-1}{k-c} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \\
 &+ [n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0] \cdot [k-c = 0] \cdot (1-p) \\
 &+ [n-i-1 \not\geq 0] \cdot [c = k]
 \end{aligned} \tag{2}$$

We take a closer look at (1) and (2):

- (1) With $[n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0]$ and assuming integers we obtain $[n-i = 1]$. Combining this with $[k-c-1 = 0]$ we can conclude:

$$\begin{aligned}
 1 \cdot p &= \underbrace{\binom{n-i-1}{0}}_{=1} \cdot p^1 \cdot (1-p)^0 \\
 &= \binom{n-i-1}{k-c-1} \cdot p^{k-c} \cdot (1-p)^{\overbrace{n-i}^{=1} + \overbrace{(-k+c)}^{=-1}}
 \end{aligned}$$

- (2) With $[n-i-1 \geq 0] \cdot [n-i-2 \not\geq 0]$ and assuming integers we obtain $[n-i = 1]$. Combining this with $[k-c = 0]$ we can conclude:

$$\begin{aligned}
 1 \cdot (1-p) &= \underbrace{\binom{n-i-1}{0}}_{=1} \cdot p^0 \cdot (1-p)^1 \\
 &= \binom{n-i-1}{k-c} \cdot p^{k-c} \cdot (1-p)^{\overbrace{n-i}^{=1} + \overbrace{(-k+c)}^{=0}}
 \end{aligned}$$

Further we are going to simplify constraints, because $[n-i-1 \geq 0] \cdot [n-i-2 \geq 0]$ is equivalent to $[n-i \geq 2]$:

$$\begin{aligned}
 \Phi(\mathcal{I}) &= [n-i \geq 2] \cdot [k-c-1 \geq 0] \cdot \binom{n-i-1}{k-c-1} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \\
 &+ [n-i = 1] \cdot [k-c-1 = 0] \cdot \binom{n-i-1}{k-c-1} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \\
 &+ [n-i \geq 2] \cdot [k-c \geq 0] \cdot \binom{n-i-1}{k-c} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \\
 &+ [n-i = 1] \cdot [k-c = 0] \cdot \binom{n-i-1}{k-c} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \\
 &+ [n-i-1 \not\geq 0] \cdot [c = k]
 \end{aligned}$$

Line 1 + 2 and 3 + 4 can be simplified to:

$$\begin{aligned}\Phi(\mathcal{I}) &= [n-i \geq 1] \cdot [k-c-1 \geq 0] \cdot \binom{n-i-1}{k-c-1} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \\ &\quad + [n-i \geq 1] \cdot [k-c \geq 0] \cdot \binom{n-i-1}{k-c} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \\ &\quad + [n-i-1 \not\geq 0] \cdot [c = k]\end{aligned}$$

If we factorize $[n-i \geq 1] \cdot (1-p)^{n-i-k+c} \cdot p^{k-c}$ we obtain:

$$\begin{aligned}\Phi(\mathcal{I}) &= [n-i \geq 1] \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \\ &\quad \cdot ([k-c-1 \geq 0] \cdot \binom{n-i-1}{k-c-1} + [k-c \geq 0] \cdot \binom{n-i-1}{k-c}) \quad (*) \\ &\quad + [n-i-1 \not\geq 0] \cdot [c = k]\end{aligned}$$

For (*) we need a case distinction w.r.t. $k-c$:

1. $k-c < 0$: Since both constraints are unsatisfied the result is 0.
2. $k-c = 0$:

$$\begin{aligned}[k-c-1 \geq 0] \cdot \binom{n-i-1}{k-c-1} + [k-c \geq 0] \cdot \binom{n-i-1}{k-c} \\ = 0 + \binom{n-i-1}{0} = \binom{n-i-1}{0} \\ = \binom{n-i}{0} = \binom{n-i}{k-c}\end{aligned}$$
3. $k-c > 0$:

$$\begin{aligned}[k-c-1 \geq 0] \cdot \binom{n-i-1}{k-c-1} + [k-c \geq 0] \cdot \binom{n-i-1}{k-c} \\ = \binom{n-i-1}{k-c-1} + \binom{n-i-1}{k-c} \\ = \binom{n-i}{k-c}\end{aligned}$$

Finally, we can conclude:

$$\begin{aligned}\Phi(\mathcal{I}) &= [n-i \geq 1] \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} \cdot [k-c \geq 0] \cdot \binom{n-i}{k-c} + [n-i-1 \not\geq 0] \cdot [c = k] \\ &= [n-i-1 \geq 0 \wedge k-c \geq 0] \cdot \binom{n-i}{k-c} \cdot (1-p)^{n-i-k+c} \cdot p^{k-c} + [n-i-1 \not\geq 0] \cdot [c = k] \\ &= \mathcal{I}\end{aligned}$$

Hence our invariant candidate is a fixed-point of the loop and therefore satisfies Theorem 1. If we initialize i and c with 0 and choose $n \geq 1 \wedge k \geq 0$ we obtain the probability mass function of the binomial distribution, thus our program indeed models a random variable following the binomial distribution.

3.3.3 Negative Binomial Distribution

In this section we revisit the negative binomial distribution and find an invariant for post-expectation $[x = k]$, i.e. the probability that exactly k Bernoulli trials are performed before r successes are achieved. Looking back at the program in Lst. 3.2 and probability mass function (3.3) we attempt to alter this function, so that it defines the probability w.r.t. different initializations.

We switch r to $r - c$, because the number of successes that is still needed depends on the number of successes already achieved and k to $k - x$, because the number of trials left depends on the number of trials already performed. Hence we assume our invariant candidate to be of the form

$$\mathcal{I} = [\alpha] \cdot \binom{k-x-1}{r-c-1} \cdot p^{r-c} \cdot (1-p)^{k-x-(r-c)} + [\beta] \cdot [k = x] .$$

We safely assume that α contains the loop guard and β the negated loop guard. If we look closely at the sanity check of the loop invariants of the last two sections we can infer that both rule out conditions for which achieving the post-expectation is *impossible*. Ruling out that we already did more than k trials is not sufficient here. Instead we need a tighter bound, because we further have to ensure that we still have enough trials left to achieve r successes, i.e. $k - x \geq r - c$. Hence our invariant candidate is

$$\mathcal{I} = [r-c-1 \geq 0 \wedge k-x \geq r-c] \cdot \binom{k-x-1}{r-c-1} \cdot p^{r-c} \cdot (1-p)^{k-x-(r-c)} + [r-c-1 \not\geq 0] \cdot [k = x] .$$

Also note that condition $x > k$, which makes achieving $[x = k]$ impossible is excluded by transitivity, since $k - x \geq r - c \geq 1$. Now we are going to check if $\Phi(\mathcal{I}) = \mathcal{I}$:

$$\begin{aligned} \Phi(\mathcal{I}) &= [r-c-1 \geq 0] \cdot wp((c := c+1[p]skip); x := x+1, \\ &\quad [r-c-1 \geq 0 \wedge k-x >= r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\ &\quad + [r-c-1 \not\geq 0] \cdot [x = k]) \\ &\quad + [r-c-1 \not\geq 0] \cdot [x = k] \\ &= [r-c-1 \geq 0] \cdot wp((c := c+1[p]skip), \\ &\quad [r-c-1 \geq 0 \wedge k-x-1 >= r-c] \cdot \binom{k-x-2}{r-c-1} \cdot (1-p)^{k-x-1-r+c} \cdot p^{r-c} \\ &\quad + [r-c-1 \not\geq 0] \cdot [x+1 = k]) \\ &\quad + [r-c-1 \not\geq 0] \cdot [x = k] \end{aligned}$$

$$\begin{aligned}
 &= [r-c-1 \geq 0] \cdot (p \cdot ([r-c-2 \geq 0 \wedge k-x-1 \geq r-c-1] \\
 &\quad \cdot \binom{k-x-2}{r-c-2} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c-1} \\
 &\quad + [r-c-2 \not\geq 0] \cdot [x+1 = k]) \\
 &\quad + (1-p) \cdot ([r-c-1 \geq 0 \wedge k-x-1 \geq r-c] \\
 &\quad \cdot \binom{k-x-2}{r-c-1} \cdot (1-p)^{k-x-1-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 \not\geq 0] \cdot [x+1 = k])) \\
 &+ [r-c-1 \not\geq 0] \cdot [x = k] \\
 \\
 &= [r-c-1 \geq 0 \wedge r-c-2 \geq 0 \wedge k-x \geq r-c] \cdot \binom{k-x-2}{r-c-2} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &+ [r-c-1 \geq 0 \wedge r-c-2 \not\geq 0] \cdot [k = x+1] \cdot p \\
 &+ [r-c-1 \geq 0 \wedge k-x-1 \geq r-c] \cdot \binom{k-x-2}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &+ [r-c-1 \not\geq 0] \cdot [x = k] \\
 \\
 &= [r-c-2 \geq 0 \wedge k-x \geq r-c] \cdot \binom{k-x-2}{r-c-2} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &+ [r-c-1 \geq 0 \wedge r-c-2 \not\geq 0] \cdot [k = x+1] \cdot p \\
 &+ [r-c-1 \geq 0 \wedge k-x-1 \geq r-c] \cdot \binom{k-x-2}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &+ [r-c-1 \not\geq 0] \cdot [x = k]
 \end{aligned}$$

Splitting up the summand in line three of the last sum we obtain:

$$\begin{aligned}
 \Phi(\mathcal{I}) &= [r-c-2 \geq 0 \wedge k-x \geq r-c] \cdot \binom{k-x-2}{r-c-2} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &+ [r-c-2 \geq 0 \wedge k-x-1 \geq r-c] \cdot \binom{k-x-2}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &+ [r-c-1 = 0 \wedge k-x-1 \geq r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &+ [r-c-1 \geq 0 \wedge r-c-2 \not\geq 0] \cdot [k = x+1] \cdot p \\
 &+ [r-c-1 \not\geq 0] \cdot [x = k]
 \end{aligned}$$

We reason analogous to the case distinction in Section 3.3.2 and obtain:

$$\begin{aligned}
 \Phi(\mathcal{I}) &= [r-c-2 \geq 0 \wedge k-x \geq r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 = 0 \wedge k-x-1 \geq r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 \geq 0 \wedge r-c-2 \not\geq 0] \cdot [k = x+1] \cdot p \\
 &\quad + [r-c-1 \not\geq 0] \cdot [x = k]
 \end{aligned} \tag{*}$$

(*) can be transformed due to constraints $[r-c-1 \geq 0 \wedge r-c-2 \not\geq 0] = [r-c=1]$ and $[k-x-1=0]$ to

$$\begin{aligned}
 1 \cdot 1 \cdot p &= \binom{0}{0} \cdot (1-p)^0 \cdot p^1 \\
 &= \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c}
 \end{aligned}$$

and thus:

$$\begin{aligned}
 \Phi(\mathcal{I}) &= [r-c-2 \geq 0 \wedge k-x \geq r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 = 0 \wedge k-x-1 \geq r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 = 0 \wedge k = x+1] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 \not\geq 0] \cdot [x = k]
 \end{aligned}$$

Due to $[r-c-1=0]$ we know that $[k=x+1] = [k=x+r-c] = [k-x=r-c]$:

$$\begin{aligned}
 \Phi(\mathcal{I}) &= [r-c-2 \geq 0 \wedge k-x \geq r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 = 0 \wedge k-x \geq r-c+1] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 = 0 \wedge k-x = r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 \not\geq 0] \cdot [x = k] \\
 &= [r-c-2 \geq 0 \wedge k-x \geq r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 = 0 \wedge k-x \geq r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &\quad + [r-c-1 \not\geq 0] \cdot [x = k]
 \end{aligned}$$

$$\begin{aligned}
 &= [r-c-1 \geq 0 \wedge k-x \geq r-c] \cdot \binom{k-x-1}{r-c-1} \cdot (1-p)^{k-x-r+c} \cdot p^{r-c} \\
 &+ [r-c-1 \not\geq 0] \cdot [x = k] \\
 &= \mathcal{I}
 \end{aligned}$$

We conclude that our guess was correct and \mathcal{I} is a fixed-point of the loop and therefore invariant. If we set x and c to 0 and choose $r \geq 1 \wedge k \geq r$ we obtain the probability mass function of the negative binomial distribution. Thus the given program indeed models a random variable following the negative binomial distribution.

As we can see, in all our examples we were able to find the necessary constraints to prove $\Phi(\mathcal{I}) = \mathcal{I}$ by excluding conditions which make it impossible to achieve the given post-expectation. This rule of thumb makes it possible for us to confidently guess invariants. However, proving that our invariant candidate is a fixed-point of the loop is highly non-trivial as we will see in the next section.

3.3.4 Hypergeometric Distribution

At the end of this chapter we are going to see that not only fixed-point iteration is hard, but performing the last step (i.e. showing that $\Phi(\mathcal{I}) = \mathcal{I}$) can be hard as well. The Hypergeometric distribution will serve as an example for that. If we look back at probability mass function (3.5) and the program in Lst. 3.3 we can assume that our invariant candidate to post-expectation $[c=k]$ is of the form

$$\mathcal{I} = [\alpha] \cdot \frac{\binom{M-c}{k-c} \cdot \binom{N-x-M+c}{n-x-k+c}}{\binom{N-x}{n-x}} + [\beta] \cdot [c = k] .$$

Applying our rule of thumb we first add the (negated) loop guard to α (β resp.) as usual, however there is a lot of things to be excluded so achieving $[c = k]$ does not become impossible:

- The remaining number of elements with a certain property must be lower or equal to the total number of remaining elements (i.e. $M - c \leq N - x$)
- The remaining number of elements with a certain property must be greater or equal to the number of elements with the certain property we still need to achieve $[c = k]$ (i.e. $M - c \geq k - c$).
- The number of elements with a certain property we have already drawn should not be greater than than the number we want to achieve (i.e. $k - c \geq 0$).
- The number of total elements remaining must be larger than the sample size remaining to be drawn (i.e. $N - x \geq n - x$).

Adding these constraints to our invariant candidate yields:

$$\mathcal{I} = [N-x \geq M-c \geq k-c \geq 0 \wedge N-x \geq n-x-1 \geq 0] \frac{\binom{M-c}{k-c} \cdot \binom{N-x-M+c}{n-x-k+c}}{\binom{N-x}{n-x}} \\ + [n-x-1 \not\geq 0][c = k]$$

If we want to prove that \mathcal{I} is a fixed-point of the loop we have to prove that the following equality holds

$$[N-x \geq M-c \geq k-c \geq 0 \wedge N-x \geq n-x-1 \geq 0] \frac{\binom{M-c}{k-c} \cdot \binom{N-x-M+c}{n-x-k+c}}{\binom{N-x}{n-x}} + [n-x-1 \not\geq 0][c = k]$$

=

$$[N-x \geq M-c \geq k-c \geq 1 \wedge N-x \geq n-x-1 \geq 2] \frac{(M-c) \binom{M-c-1}{k-c-1} \cdot \binom{N-x-M+c}{n-x-k+c}}{(N-x) \binom{N-x-1}{n-x-1}} \\ + [n-x=1][c+1=k] \\ + [n-x-1 \geq 0] \left(1 - \frac{M-c}{N-x}\right) [N-x \geq M-c \geq k-c \geq 1 \wedge N-x \geq n-x-1 \geq 2] \\ \cdot \frac{\binom{M-c}{k-c} \cdot \binom{N-x-1-M+c}{n-x-1-k+c}}{\binom{N-x-1}{n-x-1}} \\ + [n-x-1 \geq 0][n-x-2 \not\geq 0][c=k] \left(1 - \frac{M-c}{N-x}\right) \\ + [n-x-1 \not\geq 0][c=k]$$

which is impossible to do with a reasonable amount of effort, so this will stand as an open challenge to be solved by some automated technique and illustrates why *automatization* is needed.

Summing this chapter up, we have seen that fixed-point iteration can be a very powerful tool to prove a program's properties but is a very expensive task in general. Instead of performing a fixed-point iteration to prove a property an invariant can be provided that is easier to find than performing a fixed-point iteration.

Normally, invariants are allowed to underestimate the pre-expectation which is not desired if we want to prove that a program has a certain property. Our strategy for finding invariants is to guess a fixed-point of the loop, because fixed-points are always invariant. The advantage of using a fixed-point of the loop as an invariant is that it captures the program's behavior exactly and thus describes the pre-expectation precisely.

Furthermore, if we know the underlying probability distribution (and thus the pre-expectation) we are able to confidently guess an invariant by altering well known formulae to consider different initializations of the variables. Then we only have to look for proper constraints to prove that it is a fixed-point. Additionally a rule of thumb has been introduced that lets us find those constraints. Nonetheless, as a program becomes more and more complicated it appears to be impossible to prove our candidates manually, and even automated solution will fail to do so. A list of all our invariants can be found in Table 3.1.

Program	invariant for postExp:= var	invariant for postExp:= [var=k]
geometric	$[flip = 0](x + \frac{1-p}{p}) + [flip \neq 0]x$	$[flip = 0 \wedge k-x \geq 0](1-p)^{k-x}p + [flip \neq 0][x = k]$
binomial	$[n-i-1 \geq 0]((n-i)p+c) + [n-i-1 < 0]c$	$[n-i-1 \geq 0 \wedge k-c \geq 0] \cdot \binom{n-i}{k-c} \cdot (1-p)^{n-i-k+c} p^{k-c}$ $+ [n-i-1 \not\geq 0] \cdot [c = k]$
negbinomial	$[r-c-1 \geq 0] \frac{xp+r-c}{p} + [r-c-1 < 0]x$	$[r-c-1 \geq 0 \wedge k-x \geq r-c] \binom{k-x-1}{r-c-1} p^{r-c} (1-p)^{k-x-r+c}$ $+ [r-c-1 < 0][x = k]$
hypergeometric	$[n-x-1 \geq 0] \left((n-x) \frac{M-c}{N-x} + c \right) + [n-x-1 < 0]c$	$-\setminus-$

Table 3.1: Table of all invariants found

Chapter 4

Evaluating PRINSYS

In this chapter we are going to benchmark PRINSYS and detect its bottlenecks. First of all we are going to introduce the core procedures of PRINSYS. Thereafter we are going to enrich our current set of invariants with invariants found by other researchers to create a *benchmark suite*. Afterwards we will use said benchmark suite to detect where the current implementation of PRINSYS has its *limits* and why. Finally a challenge is proposed that illustrates the gap between what PRINSYS is presently able to do and what we want it to be able to do in the future.

4.1 PRINSYS

PRINSYS (pronounced “princess” [10]) is a *probabilistic invariant synthesis tool* that uses a *constraint-based approach* proposed by Katoen et al. [12] to infer probabilistic loop invariants. Furthermore, provided with a program and an invariant candidate PRINSYS is able to check whether or not the given expression is an invariant for the respective program.

The general approach proposed in [12] is based on speculatively annotating the loop with a so called *invariant template*, a parametrized invariant candidate. Using constraint solvers PRINSYS then determines parameters for which the template is indeed a loop invariant, i.e. satisfies condition (2.2), page 7.

Example 2. Dueling Cowboys [9, 17]. *We consider a modified version of the dueling cowboys program. The program models the classic situation of two cowboys A and B taking turns shooting at each other until one of them is hit. In this version both cowboys are of equal skill and thus have the same probability p to hit his opponent. Furthermore a counter is introduced that counts the number of shots.*

```
1 var turn, alive, x;
2
3 x:=0; //shot count
4 (turn:=A[]turn:=B); //decide who starts
5 dead:=0;
6 while (dead=0) {
7     if (turn=A) {
8         (dead:=1[p]turn:=B);
9     }
10    else {
11        (dead:=1[p]turn:=A);
12    }
13    x:=x+1;
14 }
15
```

Listing 4.1: modified dueling cowboys

We are interested in the number of shots taken until one of the duelists is dead, i.e. the expected value of x . The interested reader will notice that this program models a variant of the geometric distribution where the trials are performed by two players and the successful trial is counted as well, hence the expected value of random variable x should be $\frac{1}{p}$. If we want to prove this using PRINSYS we will start by finding a template to use. Because we already obtained an invariant for the geometric distribution we will choose our template to be

$$\mathcal{T}_\alpha = [\text{dead} \neq 0 \wedge x \geq 0] \cdot x + [\text{dead} = 0 \wedge x \geq 0] \cdot (x + \alpha)$$

where α is an unknown real valued parameter. Depending on instantiation of α the instance of template \mathcal{T}_α may or may not satisfy the invariance condition (2.2). PRINSYS is able to characterize all α s that make \mathcal{T}_α invariant. Provided with the above program and template PRINSYS will return

$$\alpha \cdot p - \alpha \leq 0 \wedge \alpha \cdot p - 1 \leq 0 .$$

which simplifies to

$$0 \leq \alpha \leq \frac{1}{p} .$$

To obtain the greatest pre-expectation possible we will choose the greatest α and obtain an invariant:

$$\mathcal{T}_{\frac{1}{p}} = [\text{dead} \neq 0 \wedge x \geq 0] \cdot x + [\text{dead} = 0 \wedge x \geq 0] \cdot (x + \frac{1}{p})$$

This result can be used to prove that variable x has an expected outcome of $\frac{1}{p}$ which corresponds to the mean value of a random variable following the geometric distribution.

In order to distill the parameters for which the template is invariant PRINSYS follows a specific workflow [10].

Step 1. PRINSYS parses template \mathcal{T}_α and program P and generates a control flow graph. Afterwards PRINSYS traverses the graph of the program and computes $wp(P, \mathcal{T}_\alpha)$ to set up the inequality given by the invariance condition (2.2). Theoretically this inequality is of the form

$$\underbrace{[O_1] \cdot u_1 + \dots + [O_m] \cdot u_m}_{[G] \cdot \mathcal{T}_\alpha} \leq \underbrace{[R_1] \cdot v_1 + \dots + [R_n] \cdot v_n}_{wp(P, \mathcal{T}_\alpha)}$$

where O_i, R_i are constraints, u_i, v_i are arithmetic expressions and $[G]$ is the loop's guard. All unsatisfiable and trivially 0 summands generated are removed. In order to satisfy this inequality we need to find all α s for which the point-wise inequality holds. PRINSYS will do this by pairwise comparisons of the summands of the left-hand side and the right-hand side. Since the constraints of the summands may overlap the expectations first have to be rewritten into disjoint normal form.

Step 2. PRINSYS transforms the expectation of the left-hand side and the right-hand side of the inequality above into disjoint normal form (DNF). An expectation is in DNF if it is of the form $[P_1] \cdot f_1 + \dots + [P_n] \cdot f_n$ where P_1, \dots, P_n are disjoint, i.e. at most one of them evaluates to true on any valuation. The transformation is performed according to the following theorem:

Theorem 2. [10] *Given an expectation of the form:*

$$f = [P_1] \cdot w_1 + \dots + [P_n] \cdot w_n.$$

Then an equivalent expectation in DNF can be written as:

$$\sum_{I \in \mathcal{P}(\bar{n}) \setminus \emptyset} ([\bigwedge_{i \in I} P_i \wedge \neg(\bigwedge_{j \in \bar{n} \setminus I} P_j)] \cdot (\sum_{i \in I} w_i))$$

where \bar{n} is the index set $\{1, \dots, n\}$ and $\mathcal{P}(\cdot)$ denotes the power set.

All unsatisfiable and trivially zero summands generated are removed.

Step 3. The resulting inequality in DNF is encoded as a first-order formula by transforming it according to the following theorem:

Theorem 3. [10] *Given two expectation f, g in DNF over variables x_1, \dots, x_k*

$$f = [O_1] \cdot u_1 + \dots + [O_m] \cdot u_m, \quad g = [R_1] \cdot v_1 + \dots + [R_n] \cdot v_n.$$

The inequality $f \leq g$ holds if and only if

$$\begin{aligned} \forall x_1, \dots, x_k \in \mathbb{R} : & \bigwedge_{i \in \bar{m}} \bigwedge_{j \in \bar{n}} (O_i \wedge R_j \Rightarrow (u_i - v_j \leq 0)) \\ & \wedge \bigwedge_{i \in \bar{m}} (O_i \wedge (\bigwedge_{j \in \bar{n}} \neg R_j) \Rightarrow u_i \leq 0) \\ & \wedge \bigwedge_{j \in \bar{n}} (R_j \wedge (\bigwedge_{i \in \bar{m}} \neg O_i) \Rightarrow 0 \leq v_j) \end{aligned}$$

holds, where \bar{x} is the set of indices $\{1, 2, \dots, x\}$.

This generated first-order formula is passed to REDLOG[7] that uses quantifier elimination to simplify the formula. The result returned by REDLOG is then presented to the user by PRINSYS (e.g. see Example 2) and may contain redundant information that can be further simplified manually by the user.

In technical terms the following steps take place in the current implementation of PRINSYS:

1. PRINSYS sets up the inequality to verify the invariance condition:
 - (a) Template \mathcal{T} and program P are parsed.
 - (b) \mathcal{T} is multiplied with the loop guard and stored in variable ‘lesserExp’, as it is the lesser side of the invariance condition.
 - (c) Method ‘getWLP()’ computes $wp(\text{body}, \mathcal{T})$ and stores the result in ‘greaterExpr’, as it is the greater side of the invariance condition.
 - (d) Method ‘removeUnsatAndZero()’ removes all unsatisfiable and trivially zero summands from greaterExpr.
2. both expressions are passed to method ‘transformToFormula()’:
 - (a) Method ‘toDNF()’ transforms both lesserExpr and greaterExpr to disjoint normal form.
 - (b) removeUnsatAndZero() ‘cleans’ both lesserExpr and greaterExpr.
3. lesserExpr and greaterExpr are encoded to a FO-formula and forwarded to REDLOG.

Remark Provided with an invariant candidate instead of a parametrized template PRINSYS will check if the candidate is indeed invariant by performing the steps above. Ideally PRINSYS will return ‘true’ if the candidate satisfies the invariance condition (2.2). However, often times REDLOG is not able to entirely simplify the first-order formula to the point where it can return ‘true’, but instead returns a statement that can be identified as a tautology by a user manually with a reasonable amount of effort.

4.2 Acquiring a Benchmark Suite for PRINSYS

Before we start to evaluate PRINSYS we are going to enrich our current set of invariants by invariants found by other researchers in this field and trivial invariants for hard programs, as not all of our invariants found in Chapter 3 can be used with PRINSYS. In fact, we will only be able to verify three of our invariants as the current implementation does neither support input of a fraction for the probability parameter in the probabilistic choice statement (as used in Lst. 3.3), nor does it support input of binomials or exponentials in the template. The latter is due to the fact that it is currently unknown if the theory of real numbers with exponential functions is decidable at all [16]. Thus our invariant for the hypergeometric

```

1 var h, t, jump, step, count;
2 h:=0;
3 t:=30;
4 count := 0;
5 //h is hare and t is turtle
6 while(h-t<=0){
7     (jump:=0[0.5]jump:=1);
8     if(jump = 0){ //uniform0-10
9         {{{step:=0;}[0.5]{step:=1;}}[2/5]{{{step:=2;}
10            [1/3]
11            {{step:=3;}[0.5]{step:=4;}}}}
12            [5/11]
13            {{{step:=5;}[1/3]{{step:=6;}[0.5]{step:=7;}}}}
14            [0.5]
15            {{step:=8;}[1/3]{{step:=9;}[0.5]{step:=10;}}}}
16         //jump
17         h := h+step;
18     }
19     t:=t+1;
20     count:=count+1;
21 }

```

Listing 4.2: HARE-TURTLE.pgcl

distribution as well as all invariants in column “invariant for postExp:= $[var := k]$ ” of Table 3.1 are not supported by PRINSYS.

First we are going to adapt programs and invariants by Yu-Fang Chen et al. presented in [3]. The programs can be found in Appendix A.3. Although they “failed to verify any of [their] non-linear examples with PRINSYS” we will be able to remedy this issue by providing proper constraints for the respective invariants and altering the loop guards of the programs.

Furthermore we will adapt the HARE-TURTLE program by Aleksandar Chakarov et al. presented in [2] together with a trivial invariant. The program in Lst. 4.2 models the classic story of a race between a hare and a turtle, where the turtle is given a head start. The hare jumps with a fifty percent chance a distance between zero and ten steps while the turtle consistently moves one step per turn. A trivial invariant for this program is $[h \geq 0 \wedge t \geq 0 \wedge count \geq 0]$ because this condition obviously holds through the entire program. The complete benchmark suite can be found in Table 4.1.

Before we can use these invariants for benchmarking we have to discuss the syntax of templates for PRINSYS.

Program	Invariant
geometric	$[flip = 0](x + \frac{1-p}{p}) + [flip \neq 0]x$
binomial	$[n - i - 1 \geq 0]((n - i)p + c) + [n - i - 1 < 0]c$
negbinomial	$[r - c - 1 \geq 0]\frac{xp+r-c}{p} + [r - c - 1 < 0]x$
gamblersruin	$z + xy - x^2$
geo1	$x + 3zy$
geo2	$\frac{25}{2}z - 5z^2 + x + \frac{5}{2}nx$
bin1	$x + \frac{1}{4}ny$
bin2	$x + \frac{1}{8}n^2 + \frac{1}{8}n - \frac{3}{4}ny$
sum	$x + \frac{1}{4}n^2 + \frac{1}{4}n$
prod	$\frac{1}{4}n^2 + \frac{1}{2}nx + \frac{1}{2}ny + xy - \frac{1}{4}n$
coin1	$-x - \frac{1}{2}y^2 + \frac{3}{2}y + 1$
coin2	$-x^2 + 2x + \frac{1}{2}y^2 - \frac{3}{2}y + 1$
coin3	$n - \frac{28}{3}x^2 + \frac{16}{3}xy + \frac{20}{3}x + \frac{4}{3}y^2 - 4y + \frac{8}{3}$
HARETURTLE	$[h \geq 0 \wedge t \geq 0 \wedge count \geq 0]$

Table 4.1: PRINSYS benchmark suite

Definition 1. PRINSYS Template Syntax. PRINSYS templates have to be of the form

$$[constraint] * (expression) + \dots + [constraint] * (expression)$$

where each constraint is of the form

$$[expression \text{ rel } 0 \text{ con } \dots \text{ con } expression \text{ rel } 0]$$

where $rel \in \{>=, <=, !=, =\}$ and $con \in \{and, or\}$. Note that each of the comparisons always has to compare to 0 and also ‘true’ and ‘false’ instead of comparisons are allowed. Each expression is a polynomial with no variables in the exponents. If an expression contains a fraction, the whole expression has to be converted to a fraction with a common denominator. All dividers have to be surrounded by braces.

Example 3. Geometric Distribution. In Section 3.3.1 we have seen that an invariant for the program in Lst. 3.5 is given by

$$[flip = 0](x + \frac{1-p}{p}) + [flip \neq 0]x .$$

This invariant will not be accepted by PRINSYS as the first expression of the sum is of the form $x + \text{fraction}$. To be successfully parsed by PRINSYS the input has to be rewritten to

$$[flip = 0] * ((p * x + 1 - p)/(p)) + [flip! = 0] * (x) .$$

If a user enters this invariant together with the respective program PRINSYS will produce the output:

false

This indicates that our invariant is not invariant and this may seem perplexing because we know for sure that is an invariant. This is due to the fact that we assumed x to be positive, but PRINSYS tries to prove the invariance condition over all real numbers. Therefore we have to add a constraint that ensures that values of x are positive. Thus the correct invariant is given by

$$[flip = 0 \text{ and } x \geq 0] * ((p * x + 1 - p)/(p)) + [flip! = 0 \text{ and } x \geq 0] * (x)$$

which yields the output¹:

$$p * *3 - p \leq 0$$

This statement is obviously true, because $0 \leq p \leq 1$. As mentioned earlier a tautology is a common output if parametrized probabilistic choice statements are involved.

We can conclude that often times we have to alter the invariant we obtained manually, e.g. by adding a sanity check. This is because we often assume facts (i.e. positivity, integers) which are not automatically assumed by PRINSYS, since it will try to verify the invariance condition over the set of real numbers. The fact that PRINSYS occasionally does not yield ‘true’ but a tautology can be unfortunate, because it is possible for the tautology to blow up in size for complex programs. This makes them complicated to read and identifying them as a tautology suddenly becomes hard.

Next we are going to see that we sometimes will be able to obtain a simplified representation of our invariant by including certain summands into others.

Example 4. Binomial Distribution. As an example for simplification consider the binomial distribution and our invariant found in Section 3.2.1:

$$[n - i - 1 \geq 0]((n - i)p + c) + [n - i - 1 < 0]c$$

First we rewrite this invariant to PRINSYS syntax:

$$[n - i - 1 \geq 0] * (n * p - i * p + c) + [n - i - 1 < 0] * (c)$$

Entering this into PRINSYS will yield ‘false’, because we assumed integers, but if we change the constraint of the first summand to $[n - i \geq 0]$ the second summand is just the $[n - i = 0]$ case of the first summand. Thus our invariant is simplified to

$$[n - i \geq 0] * (n * p - i * p + c)$$

and yields the result

¹ $p * *3$ simply means p^3

true

because the invariant now holds for real numbers as well.

Example 5. Negative Binomial Distribution. As an example for a required sanity check consider our invariant for the negative binomial distribution found in Section 3.2.2. Similar to Example 4 it can be simplified into a single summand:

$$[r - c \geq 0] * ((x * p + r - c) / (p))$$

Passed to PRINSYS this will yield the result

$$p - 1 = 0 \text{ or } p = 0$$

which is obviously ‘false’, because $0 \leq p \leq 1$. We can remedy this by adding a sanity check, i.e. that both $x \geq 0$ and $c \geq 0$:

$$[r - c \geq 0 \text{ and } x \geq 0 \text{ and } c \geq 0] * ((x * p + r - c) / (p)),$$

Forwarded to PRINSYS this will produce the output

$$(p * r - p \geq 0 \text{ or } p = 0 \text{ or } r - 1 < 0) \\ \text{and } (p ** 2 - p \leq 0 \text{ or } r - 1 < 0)$$

which is not trivially ‘true’, but if we look at it in detail we can show that both clauses are ‘true’ and therefore the and-conjunction is ‘true’:

$$(p * r - p \geq 0 \text{ or } p = 0 \text{ or } r - 1 < 0)$$

The first or-clause holds for all $r \geq 1$ and the third or-clause for all $r < 1$, thus the statement always holds.

$$(p ** 2 - p \leq 0 \text{ or } r - 1 < 0)$$

We know that $0 \leq p \leq 1$ holds and hence the statement holds.

If we proceed with all programs and invariants we obtain a benchmark suite for PRINSYS that can be found in Table 4.2.

4.3 Benchmarking PRINSYS

Sometimes PRINSYS appears to not *terminate* and/or *crash* when attempting to prove the invariance condition for a template or invariant candidate. Finding out what is the limiting factor in PRINSYS’ current implementation is subject to this section.

Program	Invariant
geometric	$[flip = 0 \text{ and } x \geq 0] * ((p * x + 1 - p)/(p)) + [flip! = 0 \text{ and } x \geq 0] * (x)$
binomial	$[n - i \geq 0] * (n * p - i * p + c)$
negbinomial	$[r - c \geq 0 \text{ and } x \geq 0 \text{ and } c \geq 0] * ((x * p + r - c)/(p))$
gamblersruin	$[x - 1 \geq 0 \text{ and } x - y + 1 \leq 0] * (z + x * y - x^2)$
geo1	$[x \geq 0 \text{ and } z - 1 \leq 0 \text{ and } z \geq 0 \text{ and } y \geq 0] * (x + 3 * z * y)$
geo2	$[z \geq 0 \text{ and } z - 1 \leq 0 \text{ and } x \geq 0 \text{ and } y \geq 0] * (12.5 * z - 5 * z * z + x + 2.5 * n * x)$
bin1	$[y \geq 0 \text{ and } n - 1 \geq 0] * (x + 0.25 * n * y)$
bin2	$[y \geq 0 \text{ and } n - 1 \geq 0] * (x + 0.125 * n * n + 0.125 * n - 0.75 * n * y)$
sum	$[n - 1 \geq 0] * (x + 0.25 * n * n + 0.25 * n)$
prod	$[n - 1 \geq 0 \text{ and } x \geq 0 \text{ and } y \geq 0] * (0.25 * n * n + 0.5 * n * n + 0.5 * n * y + x * y - 0.25 * n)$
coin ¹	$[true] * (1 - x - 0.5 * y * y + 1.5 * y)$
coin ²	$[true] * (1 - x * x + 2 * x + 0.5 * y * y - 1.5 * y)$
coin ³	$[x \geq 0 \text{ and } x - 1 \leq 0 \text{ and } y \geq 0 \text{ and } y - 1 \leq 0 \text{ and } n \geq 0] * ((3 * n - 28 * x * x + 16 * x * y + 20 * x + 4 * y * y - 12 * y + 8)/(3))$
HARETURTLE	$[h \geq 0 \text{ and } t \geq 0 \text{ and } count \geq 0] * (1)$

Table 4.2: PRINSYS benchmark suite (fixed)

4.3.1 Benchmarking Runtimes

To benchmark the runtimes of PRINSYS we will use a simple *time logger*. This time logger is able to hand out *timers* that can be requested by any method providing its method name. If a method signals that it has finished the time logger stops the timer associated with that method and writes the method's name as well its runtime into a log file. A sample output in a log file could look like this:

```
[2015-06-18 13:35:53]:Method tools.reduce.sat() took
203milliseconds to complete!
```

Using this time logger to benchmark the runtimes of PRINSYS will produce the results that can be found in Table 4.3.

Program	Time	Program	Time
geometric	3.2sec	binomial	1.35sec
negbinomial	1.0sec	gamblersruin	1.04sec
geo1	1.19sec	geo2	1.16sec
bin1	1.9sec	bin2	1.3sec
sum	1.2sec	prod	1.1sec
coin1	1.85sec	coin2	2.1sec
coin3	3.77sec	HARETURTLE	12min 5sec

Table 4.3: PRINSYS runtime benchmark (rounded)

The first thing we notice is that the runtimes are very different depending on the program and invariant and range from 1 second to about 12 minutes.

The reason for this large gap can be found in method `removeUnsatAndZero()` that removes unsatisfiable and trivially zero summands to reduce computational overhead of methods like `toDNF()`. Method `removeUnsatAndZero()` will repeatedly invoke a SAT-solver on every single summand of the given inequality to check if it is unsatisfiable. Removing these summands is crucial and cannot be omitted because `toDNF()` uses a power set of all summands to transform an expectation into disjoint normal form and thus each additional summand will double the computational overhead.

Example 6. Geometric Distribution. *Checking the invariant of the geometric distribution with PRINSYS produces the following output of the time logger if we only benchmark the calls to the SAT-solver:*

```
//4 Elements 1.1sec
..sat() took 203milliseconds to complete!
..sat() took 451milliseconds to complete!
..sat() took 258milliseconds to complete!
..sat() took 190milliseconds to complete!
..removeUnsatandZero took 1139milliseconds to complete!

//3 Elements 0.7sec
..sat() took 282milliseconds to complete!
..sat() took 210milliseconds to complete!
..sat() took 198milliseconds to complete!
..removeUnsatandZero took 706milliseconds to complete!
```

```

//7 Elements 1.4sec
..sat() took 156milliseconds to complete!
..sat() took 180milliseconds to complete!
..sat() took 201milliseconds to complete!
..sat() took 178milliseconds to complete!
..sat() took 187milliseconds to complete!
..sat() took 194milliseconds to complete!
..sat() took 253milliseconds to complete!
..removeUnsatandZero took 1385milliseconds to complete!

```

The first four calls to the SAT-solver are performed on the right-hand side of the inequality of the invariance condition, i.e. the expression produced by `getWLP()` (see Item 1c,1d). The next three calls are on the DNF representation of the left-hand side of the inequality (see Item 2a,2b). The last 7 calls are made on the DNF representation of the result of `getWLP()` (see Item 2a,2b).

Each of the calls to the SAT-solver takes about 200ms, because the SAT-solver is called via the slow hard disk drive. Adding it up the entire verification process takes about 3.2 seconds plus some negligible small time used by REDLOG and other PRINSYS methods.²

We conclude that the runtime of the verification process primarily depends on the number of calls to the SAT-solver that have to be performed. This number varies widely depending on the program and template, as we can see in the next example.

Example 7. HARE-TURTLE *Recall the HARE-TURTLE program in Lst. 4.2 and the trivial invariant proposed. Verification of the trivial invariant takes about 12 minutes and produces the following output:*

```

//24 Elements 4.4sec
..sat() took 465milliseconds to complete!
..sat() took 177milliseconds to complete!
:
..sat() took 158milliseconds to complete!
..sat() took 157milliseconds to complete!
..removeUnsatandZero took 4354milliseconds to complete!

```

```

//1 Element 0.25 sec
..sat() took 224milliseconds to complete!
..removeUnsatandZero took 246milliseconds to complete!

```

```

//4095 Elements 12min
..sat() took 193milliseconds to complete!
..sat() took 170milliseconds to complete!

```

²All benchmarks were performed on a 3GHz quad-core processor with 4GB RAM. OS: Windows 8.1.

```
⋮  
..sat() took 158milliseconds to complete!  
..sat() took 160milliseconds to complete!  
..removeUnsatandZero took 769706milliseconds to complete!
```

The first 24 calls to the SAT-solver are performed on the result of `getWLP()` (see Item 1c,1d). The following single call is on the DNF representation of the left-hand side of the inequality (see Item 2a,2b). The subsequent 4095 calls are made on the DNF representation of the result of `getWLP()` (see Item 2a,2b). Finally PRINSYS will output 'true'.

As we can see, the majority of calls to the SAT-solver are performed on the DNF representation of the result of `getWLP()`, i.e. the result of $wp(\text{body}, \mathcal{T})$. Consequently the runtime is primarily affected by the formula produced by `getWLP()` and `toDNF()` in terms of number of summands, because the SAT-solver is called on every single summand. Determining the size of these expressions is subject of the next section.

4.3.2 Benchmarking Expression Size

As we have seen in the last section the runtime of PRINSYS primarily depends on the expression size produced by `getWLP()` and `toDNF()`. The goal of this section is to determine this expression size in terms of an upper bound that can be computed before actually starting the verification process.

The question how many summands are produced by `toDNF()` can be answered by looking back at Theorem 2, page 41. As we can see the surrounding sum produces summands by using a power set of the summands of the given expectation f without the empty set. Thus, the number of summands resulting from that process is $2^{\text{numsum}(f)} - 1$, where $\text{numsum}(\cdot)$ denotes the number of summands of the given expression. Consequently we need to find out how large $\text{numsum}(f)$ is going to be.

The number of summands on the left-hand side of the inequality will be the number of summands of the template, because it is only multiplied by the loop guard. The number of summands on the right-hand side of the inequality will be the number of summands produced by `getWLP()` on the template, i.e. the number of summands produced by $wp(\text{body}, \mathcal{T})$ without the summands that get removed by `removeUnsatAndZero()` (see Item 1c,1d).

Looking back at Example 6 `getWLP()` produces 4 summands. One of those summands is removed by `removeUnsatAndZero()` leading to a DNF representation of $2^3 - 1 = 7$ (instead of 15 if `removeUnsatAndZero()` is not used). In Example 7, 12 of the 24 summands produced by `getWLP()` are removed leading to a DNF representation of $2^{12} - 1 = 4095$ instead of $2^{24} - 1 = 16777215$. As we can see removing unsatisfiable and trivially 0 summands is a crucial step in the workflow of PRINSYS and has a large impact on the expression size. However, as we can not possibly tell how many of the summands are going to get removed by `removeUnsatAndZero()` beforehand we are satisfied with an upper bound, i.e. the number of summands of the result of `getWLP()` before cleaning.

This number of summands produced by $wp(\text{body}, \mathcal{T})$ can be computed before actually computing the sum itself, because it only depends on the program and the number of summands in the given template. Consequently we can define a function $numsum(\cdot)$ computing the number of summands produced by wp depending on the number of summands in the template.

Definition 2. numsum(\cdot) *Let R, Q be loop free (probabilistic) programs, x a program variable, E an arithmetic expression, G a boolean guard, p a probability, t an expectation and $|t|$ the number of summands of the expectation. We recursively define the number of summands generated by wp on a loop free program as:*

$$\begin{aligned} numsum(\text{skip}, |t|) &= |t| \\ numsum(x := E, |t|) &= |t| \\ numsum(R; Q, |t|) &= numsum(R, numsum(Q, |t|)) \\ numsum(\{R\} [p] \{Q\}, |t|) &= numsum(R, |t|) + numsum(Q, |t|) \\ numsum(\{R\} [] \{Q\}, |t|) &= numsum(R, |t|) + numsum(Q, |t|) \\ numsum(\text{if}(G) \{R\} \text{else} \{Q\}, |t|) &= numsum(R, |t|) + numsum(Q, |t|) \end{aligned}$$

Thus, with every decision the program makes during execution we effectively double the number of summands of the current sum at that point. Consequently we can compute the worst case runtime before starting the verification process by computing the number of calls to the SAT-solver that have to be performed at most. PRINSYS will call the method `removeUnsatAndZero()` three times:

1. On $wp(\text{body}, \mathcal{T})$.
2. On the DNF representation of $[G] \cdot \mathcal{T}$.
3. On the DNF representation of $wp(\text{body}, \mathcal{T})$.

Thus the maximum number of calls to the SAT-solver is given by:

$$\underbrace{numsum(\text{body}, |\mathcal{T}|)}_1 + \underbrace{2^{|[G] \cdot \mathcal{T}|} - 1}_2 + \underbrace{2^{numsum(\text{body}, |\mathcal{T}|)} - 1}_3$$

This possibly large number is why PRINSYS may crash or take a long time on complicated programs. As the number of calls to the SAT-solver in the worst case is given by the formula above and every expectation is stored as a String, this exponential growth results in a memory outage in a very short time if the program is very complicated. The more decisions are made during a programs execution, the larger the result of $numsum(\cdot)$ becomes. The result of the expression size benchmark can be found in Table 4.4.

Remark Using a simple example program, that can be found in Appendix A.4, PRINSYS was able to store a maximum expression size of $2^{15} - 1 = 32767$ summands and took about

Program	getWLP() size	toDNF() size	Program	getWLP() size	toDNF() size
geometric	4	7	binomial	2	3
negbinomial	2	3	gamblersruin	2	3
geo1	2	3	geo2	2	3
bin1	2	3	bin2	2	3
sum	2	3	prod	2	3
coin ¹	3	7	coin ²	3	7
coin ³	4	15	HARETURTLE	24	4095

Table 4.4: PRINSYS size benchmark

8 hours to process them. It crashed on $2^{16} - 1 = 65535$ summands due to a memory outage. The summands were of the form

$$\sum_{i=0}^k [x - i \geq 0] \cdot (x) .$$

Keep in mind that PRINSYS stores this summands as Strings and might be able to process less or more summands depending on how long the actual String is. However, it gives you an idea of the dimensions PRINSYS is able to store.

4.4 Challenge

At the end of this thesis we are going to discuss a challenging program that illustrates the gap between what PRINSYS is presently able to analyze and what we want it to be able to analyze in the future. Furthermore this challenge illustrates the *goal* of this research area.

4.4.1 Choice Coordination Problem

The following example is a problem that occurs in *distributed system*, when entities independently need to pick the same choice — for example in a network. This *choice coordination problem* can be solved by an algorithm using symmetry breaking randomness.

In our scenario we will investigate two processes that need to decide on one of two registers. The decision is indicated by writing a “2” in the specific register. Since this is a coordination problem an algorithm solving it should ensure that at the end of the algorithm only one of the two registers contains a “2”.

The pseudo-code representation of the algorithm “synch-ccp” [18] solving this problem is given in Lst. 4.3. A PGCL implementation can be found in Appendix A.2.

Automated verification could be used here to show that $[c1 + c2 \leq 3]$ is an invariant, because it is a condition that has to hold through the entire program if the algorithm

```

1 Input: Registers c0 and c1 initialized to 0.
2 Output: Exactly one of the two registers has the value 2.
3
4 0. P_i is initially scanning the register C_i and has its
5    local variable B_i initialized to 0.
6
7 1. Read the current register and obtain a bit R_i.
8
9 2. Select one of three cases.
10
11     case: 2.1 (R_i = 2)
12           halt;
13
14     case: 2.2 (R_i = 0, B_i = 1)
15           Write 2 into the current register;
16           halt;
17
18     case: 2.3 (otherwise)
19           Assign an unbiased random bit to B_i;
20           write B_i into the current register;
21
22 3. P_i exchanges its current register with P_{1-i} and returns
23    to Step 1.

```

Listing 4.3: pseudo-code for synch-ccp.pgcl

is correct. Indeed, drawing a *Markov decision process* of the program yields that in no state the violating condition $[c_1+c_2 \geq 4]$ holds, thus there is no state where both registers contain a 2. However, the PGCL implementation contains many *if-then-else* statements, which results in an expression size of 32 summands for $wp(\text{body}, [c_1 + c_2 \leq 3])$ which in return leads to a DNF representation of $2^{32} - 1 = 4294967295$ summands in the worst case, if none of the 32 summands of wp is removed. This will lead to a crash of PRINSYS as the expression size becomes multiple gigabytes large in just a few seconds.

Furthermore we can conjecture that this program is a more complicated version of the *dueling cowboys* program in [9], because both processes are ‘shooting’ for write access to decide on a register and thus could have a similar expected value for the number of ‘shots’ needed to decide on a register. Actually finding out that programs like synch-ccp can be reduced to dueling cowboys, which in return models kind of a geometric distribution would be a major step towards finding invariants, since this would mean that a user only needs to figure out the underlying probability distribution (which could be done by just plotting the result of various runs) to confidently guess an invariant candidate or template for PRINSYS by just parameterizing our invariant findings in the previous chapters.

Summing up, we have seen that an *exponential growth of the disjunctive normal form* of an expectation results in very large formulae for complicated programs, but we can give an

upper bound on how large these expectations will become a-priori and hence could inform a user that PRINSYS might be going to crash if he tries to analyze his program/template.

The current bottleneck of expression size combined with about 200ms long calls to the SAT-solver can only be shortened by a native inclusion of a SAT-solver into the PRINSYS implementation instead of calling a SAT-solver via the slow hard disk drive. The space explosion problem of the formulae stored as Strings can be addressed by calling the SAT-solver interleaved with the transformation to DNF, as summands are eliminated directly instead afterwards, thus saving space.

Chapter 5

Conclusion

5.1 Summary

In this thesis we have used fixed-point iterations to prove properties of various programs and though fixed-point iteration gives an impression of what the program does, it is a very expensive task which may not lead to a conclusive result. To avoid performing a fixed-point iteration an invariant can be provided, which can be easier to find than performing a fixed-point iteration if we know the pre-expectation we want to prove. The strategy is to use existing formulae for the pre-expectation to guess fixed-points of the loop, because we have shown that fixed-points are always invariant. This also ensures that our invariant does not underestimate the pre-expectation but instead describes it precisely. The main task was to find constraints that allow us to prove that our invariant candidate is a fixed-point of the loop and a rule of thumb was introduced that helps finding those constraints.

Furthermore we have benchmarked the probabilistic invariant synthesis tool PRINSYS with a benchmark suite created by composing our invariants and programs with those of other researchers. In addition we identified the computational bottleneck of PRINSYS to be the expression size generated by disjoint normal form transformation combined with the removal of unsatisfiable and trivially zero summands of the expectations generated. Finally we suggested measures that may increase the usability as well as performance of PRINSYS.

5.2 Future Work

As future work, we might consider an extension of the heuristic to find constraints for proving an invariant to be a fixed-point to post-expectations other than $[var = k]$. A tool that performs for example the first five steps of a fixed-point iteration and presents the result in a readable way to aid a user in finding invariants would be very helpful as well and could be implemented as an extension of PRINSYS. Furthermore, the suggested measures to improve PRINSYS performance by natively including a SAT-solver and interleaving it with the disjoint normal form transformation of expectations could be implemented and further optimized.

Bibliography

- [1] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/Correcting with Applications to Numerical Problems. In: *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*. STOC '90. ACM, 1990, pp. 73–83.
- [2] Aleksandar Chakarov and Sriram Sankaranarayanan. Expectation invariants for probabilistic program loops as fixed points. In: *Static Analysis*. Springer International Publishing, 2014, pp. 85–100.
- [3] Yu-Fang Chen et al. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In: *CoRR* abs/1502.04280 (2015).
- [4] E. Cramer and U. Kamps. Grundlagen der Wahrscheinlichkeitsrechnung und Statistik. Springer-Lehrbuch. Springer-Verlag Berlin Heidelberg, 2008.
- [5] Brian C. Dean. A simple expected running time analysis for randomized “divide and conquer” algorithms. In: *Discrete Applied Mathematics* (2006).
- [6] Edsger Wybe Dijkstra. A Discipline of Programming. 1st. Prentice Hall PTR, 1997.
- [7] Andreas Dolzhan and Thomas Sturm. Redlog computer algebra meets computer logic. In: *ACM SIGSAM Bulletin* (1996).
- [8] Martin Erwig and Steve Kollmansberger. Modeling Genome Evolution with a DSEL for Probabilistic Programming. In: *Proceedings of the 8th International Conference on Practical Aspects of Declarative Languages*. PADL'06. Springer-Verlag, 2006, pp. 134–149.
- [9] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational Versus Weakest Pre-expectation Semantics for the Probabilistic Guarded Command Language. In: *Perform. Eval.* 73 (Mar. 2014), pp. 110–132.
- [10] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. PRINSYS: On a Quest for Probabilistic Loop Invariants. In: *Proceedings of the 10th International Conference on Quantitative Evaluation of Systems*. QEST'13. Springer-Verlag, 2013, pp. 193–208.
- [11] Ralf Herbrich, Tom Minka, and Thore Graepel. TrueSkill(TM): A Bayesian Skill Rating System. In: *Advances in Neural Information Processing Systems 20*. MIT Press, 2007, pp. 569–576.
- [12] Joost-Pieter Katoen et al. Linear-Invariant Generation for Probabilistic Programs. In: *Static Analysis Symposium* (2011), pp. 390–406.

- [13] Tejas Kulkarni et al. Picture: A Probabilistic Programming Language for Scene Perception. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.
- [14] Jean-Louis Lassez, V. L. Nguyen, and Liz Sonenberg. Fixed Point Theorems and Semantics: A Folk Tale. In: *Inf. Process. Lett.* 14.3 (1982), pp. 112–116.
- [15] Jérémie Lumbroso. Optimal Discrete Uniform Generation from Coin Flips, and Applications. In: *CoRR* abs/1304.1916 (2013).
- [16] A. Macintyre and A.J. Wilkie. On the decidability of the real exponential field. In: *Kreiseliana: About and around Georg Kreisel* (1996), pp. 441–467.
- [17] Annabelle McIver and Carroll Morgan. Abstraction, Refinement And Proof For Probabilistic Systems. In: *Monographs in Computer Science* (2004).
- [18] Rajeev Motwani and Prabhakar Raghavan. Randomized Algorithms. Cambridge University Press, 1995, 355ff.
- [19] Michael O. Rabin. The choice coordination problem. In: *Acta Informatica* 17.2 (1982), pp. 121–134.
- [20] Robert Sedgewick and Kevin Wayne. Algorithms 4th Edition. Addison-Wesley Professional, 2011.

Appendix A

Appendix

A.1 Geometric Distribution (Expected Outcome)

$$\begin{aligned}
wp(P, x) &= wp(x := 0, wp(flip := 0, wp(while(flip = 0)\{ \\
&\quad (flip := 1[p]x := x + 1)\}, x))) \\
&= wp(x := 0, wp(flip := 0, \\
&\quad \text{lfp}_{\mathbb{F}}(\underbrace{[flip = 0] \cdot wp(flip := 1[p]x := x + 1, F) + [flip \neq 0] \cdot x}_{\Phi(F)}))) \\
&= wp(x := 0, wp(flip := 0, \\
&\quad \text{sup}_k(\underbrace{[flip = 0] \cdot wp(flip := 1[p]x := x + 1, 0) + [flip \neq 0] \cdot x^k}_{\Phi(0)}))
\end{aligned}$$

We start a fixed-point iteration beginning with $\Phi(0)$:

$$\begin{aligned}
\Phi(0) &= [flip = 0] \cdot \underbrace{wp(flip := 1[p]x := x + 1, 0)}_{=0} + [flip \neq 0] \cdot x \\
&= [flip \neq 0] \cdot x
\end{aligned}$$

$$\begin{aligned}
\Phi^2(0) &= \Phi(\Phi(0)) \\
&= [flip = 0] \cdot wp(flip := 1[p]x := x + 1, [flip \neq 0] \cdot x) + [flip \neq 0] \cdot x \\
&= [flip = 0] \cdot (p[1 \neq 0]x + (1 - p)[flip \neq 0](x + 1)) + [flip \neq 0] \cdot x \\
&= [flip = 0]px + (1 - p) \underbrace{[flip = 0][flip \neq 0](x + 1)}_{=0} + [flip \neq 0] \cdot x
\end{aligned}$$

$$= [flip = 0]px + [flip \neq 0]x$$

$$\Phi^3(0) = [flip = 0](px + p(1-p)(x+1)) + [flip \neq 0]x$$

$$\Phi^4(0) = [flip = 0](px + p(1-p)(x+1) + p(1-p)^2(x+2)) + [flip \neq 0]x$$

$$\Phi^5(0) = [flip = 0](px + p(1-p)(x+1) + p(1-p)^2(x+2) + p(1-p)^3(x+3)) \\ + [flip \neq 0]x$$

⋮

$$\Phi^k(0) = [flip = 0] \sum_{i=0}^k (1-p)^i p(x+i) + [flip \neq 0]x$$

The sum in the last equation is convergent so we can assume that a fixed-point of Φ is given by:

$$F = [flip = 0] * (x + \frac{1-p}{p}) + [flip \neq 0] * x \quad (\text{A.1})$$

To ensure that (A.1) is a fixed-point of the loop we merely have to insert the fixed-point into Φ and see if the result is the same as the input.

$$\begin{aligned} \Phi(F) &= [flip = 0] \cdot wp(flip := 1[p]x := x+1, [flip = 0](x + \frac{1-p}{p}) + [flip \neq 0]x) \\ &\quad + [flip \neq 0] \cdot x \\ &= [flip = 0] \cdot (px + (1-p) \cdot (x+1 + \frac{1-p}{p})) + [flip \neq 0] \cdot x \\ &= [flip = 0] \cdot (px + (1-p)x + (1-p) + \frac{(1-p)^2}{p}) + [flip \neq 0] \cdot x \\ &= [flip = 0] \cdot (px + x - px + (1-p) + \frac{(1-p)^2}{p}) + [flip \neq 0] \cdot x \\ &= [flip = 0] \cdot (x + 1 - p + \frac{(1-p)^2}{p}) + [flip \neq 0] \cdot x \\ &= [flip = 0] \cdot (x + 1 - p + \frac{p^2 - 2p + 1}{p}) + [flip \neq 0] \cdot x \end{aligned}$$

$$\begin{aligned}
&= [flip = 0] \cdot (x + 1 + \frac{p^2 - 2p + 1}{p} - \frac{p^2}{p}) + [flip \neq 0] \cdot x \\
&= [flip = 0] \cdot (x + 1 + \frac{1 - 2p}{p}) + [flip \neq 0] \cdot x \\
&= [flip = 0] \cdot (x + \frac{p}{p} + \frac{1 - 2p}{p}) + [flip \neq 0] \cdot x \\
&= [flip = 0] \cdot (x + \frac{1 - 2p + p}{p}) + [flip \neq 0] \cdot x \\
&= [flip = 0] \cdot (x + \frac{1 - p}{p}) + [flip \neq 0] \cdot x \\
&= F
\end{aligned}$$

As we can see, the result is the same as the input. Thus, F is a fixed-point of the loop. Now we can insert this fixed-point into our wp calculation:

$$\begin{aligned}
wp(P, x) &= wp(x := 0, wp(flip := 0, \\
&\quad \text{lfp}_{\mathbf{F}}([flip = 0] \cdot wp(flip := 1[p]x := x + 1, F) + [flip \neq 0] \cdot x))) \\
&= wp(x := 0, wp(flip := 0, [flip = 0] \cdot (x + \frac{1 - p}{p}) + [flip \neq 0] \cdot x)) \\
&= wp(x := 0, x + \frac{1 - p}{p}) \\
&= \frac{1 - p}{p}
\end{aligned}$$

This corresponds with our prediction of the mean value of a random variable following the geometric distribution.

A.2 Synch-ccp.pgcl

```

1 var turn, c0, c1, b0, b1, r0, r1, registerOfP0, registerOfP1, halt, turn;
2
3 //the two registers, initially 0
4 c0:=0;
5 c1:=0;
```

```

6
7 //variable of the processes to read register, initially 0
8 r0:=0;
9 r1:=0;
10
11 //local variabel for each process (b0 for p0, b1 for p1)
12 //initially 0
13 b0:=0;
14 b1:=1;
15
16 //the currently assigned register to the respective process
17 registerOfP0 := 0;
18 registerOfP1 := 1;
19
20 //variabel for termination
21 halt := 0;
22
23 //while there is no fixed choice
24 while(halt = 0){
25     //nondeterministically decide which processor is acting
26     //0 means p0, 1 means p1
27     (turn := 0[]turn := 1);
28
29     //if p0 is taking action
30     if(turn = 0){
31         //if his current register is c0
32         if(registerOfP0 = 0){
33             //read c0
34             r0 := c0;
35             //2 means one process has fixed a choice
36             // in that register
37             if(r0-2 =0){
38                 halt:=1;
39             }
40             //if register is 0 and the internal bit is 1
41             //fix the choice in that register and end
42             else{
43                 if(r0 = 0 and b0 - 1= 0){
44                     c0:=2;
45                     halt:=1;
46                 }
47                 else{
48                     //assign new unbiased bit to local
49                     //variabel and write it to the register
50                     (b0:=0 [0.5] b0 := 1);
51                     c0 := b0;
52                 }
53             }
54             //exchange registers with other process
55             registerOfP0 := 1;
56             registerOfP1 := 0;
57         }
58         //same for register c1

```

```
59     else{
60         r0 := c1;
61         if(r0-2 = 0){
62             halt:=1;
63         }
64         else{
65             if(r0 = 0 and b0-1 = 0){
66                 c1:=2;
67                 halt:=1;
68             }
69             else{
70                 (b0:=0 [0.5] b0 := 1);
71                 c1 := b0;
72             }
73         }
74         registerOfP0 := 0;
75         registerOfP1 := 1;
76     }
77 }
78 //same for process P1
79 else{
80     if(registerOfP1 = 0){
81         r1 := c0;
82         if(r1-2 = 0){
83             halt:=1;
84         }
85         else{
86             if(r1 = 0 and b1-1 = 0){
87                 c0:=2;
88                 halt:=1;
89             }
90             else{
91                 (b1:=0 [0.5] b1 := 1);
92                 c0 := b1;
93             }
94         }
95         registerOfP0 := 0;
96         registerOfP1 := 1;
97     }
98     else{
99         r1 := c1;
100        if(r1-2 = 0){
101            halt:=1;
102        }
103        else{
104            if(r1 = 0 and b1-1 = 0){
105                c1:=2;
106                halt:=1;
107            }
108            else{
109                (b1:=0 [0.5] b1 := 1);
110                c1 := b1;
111            }

```

```
112         }
113         registerOfP0 := 1;
114         registerOfP1 := 0;
115     }
116 }
117 }
```

Listing A.1: synch-ccp.pgcl

A.3 Programs of Yu-Fang Chen et al. [3]

The following programs are modified versions of those proposed in [3]. Since PRINSYS tries to prove the invariance condition over the real numbers it is recommended to use “ \leq, \geq ” in boolean guards instead of “ $<, >$ ”. Thus, we alter those guards by adding or subtracting 1. For example guard $n > 0$ is changed to $n - 1 \geq 0$.

Gamblers ruin

```
1 var z;
2
3 z:=0;
4
5 while(x-1>=0 and x-y+1<=0){
6     (x:=x+1[0.5]x:=x-1);
7     z:=z+1;
8 }
```

Listing A.2: gamblersruin.pgcl

Geometric distribution

```
1 var x, z;
2
3 x:=0;
4 z:=1;
5
6 while(z!=0){
7     (z:=0[0.25]x:=x+y);
8 }
```

Listing A.3: geo1.pgcl

APPENDIX A. APPENDIX

```
1 var x,y,z;
2
3 x:=0;
4 y:=0;
5 z:=1;
6
7 while (z!=0) {
8     y:=y+1;
9     (z:=0[0.25]x:=x+y);
10 }
```

Listing A.4: geo2.pgcl

Binomial distribution

```
1 var x;
2
3 x:=0;
4
5 while (n-1>=0) {
6     (x:=x+y[0.25]skip);
7     n:=n-1;
8 }
```

Listing A.5: bin1.pgcl

```
1 var x;
2
3 x:=0;
4
5 while (n-1>=0) {
6     (x:=x+n[0.25]x:=x+y);
7     n:=n-1;
8 }
```

Listing A.6: bin2.pgcl

Sum of Random Series

```
1 var x;
2
3 while (n-1>=0) {
4     (x:=x+n[0.5]skip);
5     n:=n-1;
6 }
```

Listing A.7: sum.pgcl

Product of dependent random variables

```
1 var x,y;
2
3 while(n-1>=0){
4     (x:=x+1[0.5]y:=y+1);
5     n:=n-1;
6 }
```

Listing A.8: prod.pgcl

Simulation of a fair coin

```
1 var x,y,n;
2
3 x:=0;
4 y:=0;
5 n:=0;
6
7 while(x-y=0){
8     (x:=1[0.25]x:=0);
9     (y:=1[0.25]y:=0);
10    n:=n+1;
11 }
```

Listing A.9: coin.pgcl

A.4 PRINSYS Worst Case Example

```
1 var x, flip;
2 x:=0;
3 flip:=0;
4 while(flip=0){
5     (flip:=0[p]x:=x+1);
6     (flip:=0[p]x:=x+1);
7     (flip:=0[p]x:=x+1);
8     (flip:=0[p]x:=x+1);
9 }
10
```

Listing A.10: PRINSYS worst case

Entering the above program with invariant candidate $[x \geq 0] * (x)$ yields 16 summands after wp which leads to a DNF representation of size $2^{16} - 1 = 65535$.

```
1     var x, flip;
2     x:=0;
3     flip:=0;
4     while(flip=0){
5         flip:=1;
6     }
7
```

Listing A.11: PRINSYS worst case

Entering the above program with invariant candidate

$$\sum_{i=0}^{14} [x - i \geq 0] \cdot (x)$$

will yield 15 summands after *wp* which leads to a DNF representation of exactly $2^{15} - 1 = 32767$ summands.