
Pushdown and Expectation Transformer Semantics of Probabilistic Recursive Programs with Nested Conditioning

Johannes Lehmann

Master's Thesis at RWTH Aachen University,
Chair for Software Modelling and Verification
Faculty of Mathematics, Computer Science and Natural
Sciences

January 10, 2023

First examiner: Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen
Second examiner: apl. Prof. Dr. Thomas Noll
Advisor: Tobias Winkler

Communicated by Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen

Acknowledgements

I am grateful to Prof. Dr. Joost-Pieter Katoen introducing me to the topic of probabilistic verification through his great lectures and for accepting the role of first examiner. I am also grateful to Prof. Dr. Thomas Noll for accepting the role of second examiner. I would like to thank my advisor Tobias Winkler for helping me find an interesting topic, for giving me helpful advice in countless meetings and for providing very useful and detailed feedback on an earlier version of this thesis.

Abstract

Conditioning is a common feature in probabilistic programming languages. In particular, nested conditioning is useful, as it allows expressing "Reasoning about reasoning". We present an imperative language with recursion and nested conditioning. We define operational semantics based on probabilistic pushdown automata, where the stack of the automaton is used both for recursion and conditioning. The construction supports unbounded recursion. We also provide weakest-preexpectation transformer semantics for our language and show that this semantics is equivalent to the operational semantics for terminal reachability and termination probabilities. We show that conditioning does not increase the expressiveness of the language and compare this to existing results. In particular, we show that two definitions of conditioning exist that differ in how they handle termination. Based on the operational semantics, we have implemented nested conditioning in the model checker Pray. This implementation is demonstrated using two examples.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 3 |
| 1.1 | Related Work | 4 |
| 2 | The Conditioned Recursive Probabilistic Programming Language | 7 |
| 2.1 | Overview | 7 |
| 2.1.1 | Global Conditioning | 8 |
| 2.1.2 | Local Conditioning With Query Blocks | 9 |
| 2.1.3 | Recursively Nested Conditioning | 10 |
| 2.2 | Syntax | 11 |
| 2.2.1 | Instructions | 11 |
| 2.2.2 | Program Fragments | 12 |
| 2.2.3 | Syntactic Sugar | 13 |
| 2.2.4 | Programs | 13 |
| 2.3 | Illegal Programs | 14 |
| 3 | Operational Semantics | 17 |
| 3.1 | Probabilistic Pushdown Automata | 17 |
| 3.1.1 | Analysing Probabilistic Pushdown Automata | 18 |
| 3.2 | Construction Rules | 18 |
| 3.2.1 | Properties of the pPDA | 24 |
| 3.2.2 | Reachability Probabilities | 24 |
| 3.3 | Examples | 27 |
| 3.3.1 | Function Calls and Concatenation | 27 |
| 3.3.2 | Recursion | 28 |
| 3.3.3 | Conditioning | 29 |
| 3.3.4 | Converting a pPDA Into a CRPPL Program | 29 |
| 3.4 | Illegal Programs | 31 |
| 3.5 | Limitations | 32 |
| 3.5.1 | Expected Runtime | 32 |
| 3.5.2 | Probabilistic Model Checking | 33 |
| 3.5.3 | Termination | 33 |
| 4 | Weakest Preexpectation Semantics | 35 |
| 4.1 | Expectation Transformers | 35 |
| 4.1.1 | Domain Theory | 37 |
| 4.2 | Scoping | 39 |
| 4.2.1 | Increasing and Decreasing Scope | 39 |

| | | |
|----------|--|-----------|
| 4.3 | The wp and wlp Transformer | 41 |
| 4.3.1 | Standard Cases | 42 |
| 4.3.2 | Observe and Query | 43 |
| 4.3.3 | While | 44 |
| 4.3.4 | Function Calls | 44 |
| 4.4 | Continuity of wp and wlp | 46 |
| 4.5 | Correspondence Between pPDA and wp Semantics | 51 |
| 4.5.1 | Induction Base | 52 |
| 4.5.2 | Induction Step | 54 |
| 5 | Expressiveness | 67 |
| 5.1 | Transformation | 67 |
| 5.1.1 | De-Conditioning Function t | 68 |
| 5.1.2 | Function Body Transformation Function t' | 70 |
| 5.2 | Comparison to Other Model Classes | 70 |
| 5.2.1 | Model Classes | 70 |
| 5.2.2 | Two Notions of Conditioning | 72 |
| 5.2.3 | Conditioning Over Words or States | 73 |
| 5.2.4 | Closure Results From [11] | 73 |
| 5.2.5 | Closure Under Post-Conditioning With States | 75 |
| 5.2.6 | Closure Under Algorithmic Conditioning With States | 76 |
| 5.2.7 | Closure Under Algorithmic Conditioning With Words | 77 |
| 5.2.8 | Summary and Discussion | 79 |
| 6 | Implementation | 81 |
| 6.1 | Description of the Implementation | 81 |
| 6.1.1 | Introduction to PRAY | 81 |
| 6.1.2 | Conditioning in PRAY | 82 |
| 6.2 | Evaluation | 82 |
| 6.2.1 | Schelling Games | 82 |
| 6.2.2 | Counting Game | 85 |
| 6.2.3 | Discussion and Limitations | 88 |
| 7 | Conclusion | 89 |
| 7.1 | Future Work | 89 |

Chapter 1

Introduction

Probabilistic programming languages support two features that traditional programming languages do not: They can make probabilistic choices and they can condition on the values of variables. The combination of these two features allows them to model a wide range of probabilistic processes.

We distinguish between *global* and *nested* conditioning. With global conditioning, all probabilistic choices are always affected by conditioning. Nested conditioning, on the other hand, allows the programmer to specify blocks that restrict which probabilistic choices are affected by conditioning. Multiple of these blocks can be nested.

Nested conditioning allows us model “reasoning about reasoning”. Here, the goal is to model what Person A thinks Person B is thinking. This requires an inner block for Person B’s thought process and an outer block for Person A. This can be nested further if Person B is in turn reasoning about Person A (or if Person A at least believes that Person B is reasoning about Person A). Nested conditioning is thus combined naturally with recursion. In this work, we present an imperative programming language that supports both recursion and nested conditioning.

In the first chapter, we introduce the *Conditionined Recursive Probabilistic Programming Language* (CRPPL). This language supports variables and assignments, conditional and probabilistic choice, while loops, recursion and nested conditioning. We require variables to have finite domain. This ensures that the program can be modelled by a finite-state pPDA. We introduce the language with a series of examples and give the formal syntax. We also investigate an edge case where conditioning can lead to an undefined final distribution.

We then define the operational semantics of CRPPL. These are based on probabilistic pushdown automata. Pushdown automata are a natural model for recursive programs, since the automaton stack can be used to store the call stack of the program. We handle conditioning using rejection sampling: If a condition is violated, we restart at the beginning of the surrounding query block. In order to find the surrounding query block, we use the automaton stack.

One can then compute termination and reachability probabilities of the automaton. Computing expected runtimes and performing model checking, e.g. for probabilistic computation tree logic formulas, are also possible for probabilistic pushdown automata, but we show that our semantics only behaves correctly for termination and reachability.

We then give the denotational semantics of CRPPL by presenting a weakest pre-expectation transformer [16]. Weakest pre-expectation semantics extend Dijkstra’s weakest pre-condition calculus [4] to probabilistic programs. The semantics allows us to determine the expected value of a given expression over the program’s variables after the program has terminated.

Like most weakest pre-expectation transformers, ours is defined inductively over the structure of the program. To handle conditioning, we separately track whether any condition was violated. When we reach the end of a query block, we then normalise the expectation using this information. This approach is inspired by [17], where weakest pre-expectation semantics for global conditioning are given. Our approach differs in that the normalisation is performed after every query block, instead of once for the entire program. Similar to [17], our approach properly handles non-termination.

Our weakest pre-expectation semantics also supports (mutual) recursion with separately scoped variables for each recursive call. Our method for this is based on [15], but can handle multiple functions. To create a new scope, we prepend a special symbol \boxtimes to each existing variable, and to leave a scope, we remove one such symbol from each variable.

We then show that the operational semantics and weakest pre-expectation semantics are equivalent, i.e. that the termination and reachability probabilities computed by the semantics are equal.

After that, we analyse whether conditioning increases the expressiveness of our language. We present a transformation that removes conditioning from any CRPPL program while preserving termination and reachability probabilities. Similar to the operational semantics, this transformation performs rejection sampling. This transformation shows that our language is closed under conditioning. We compare our results to the results from [11], where it is shown that several other model classes are *not* closed under conditioning. It turns out that there are in fact two different notions of conditioning that lead to different closure results: One notion involves conditioning statements in the program, whereas the other runs the program without conditioning and then conditions on the final distribution. These differ in particular in how they deal with termination. We analyse several model classes with respect to both types of conditioning.

In the final chapter, we present an implementation in the model checker PRAY. This implementation is based on the operational semantics and, in particular, also uses the stack to handle recursion and query blocks. We demonstrate our implementation on several examples. One example is also used to show that the rejection sampling can lead to unintuitive termination behaviour.

1.1 Related Work

Many probabilistic programming languages only support bounded recursion. Bounded recursion allows computing the distribution “bottom-up”, where each level of recursion forms a layer and each layer only depends on the layer below. This is not possible for unbounded recursion, as there would be infinitely many layers. In [21], a technique is presented that solves this problem and thus supports unbounded recursion: They build a graph of these dependencies and then identify strongly connected components (i.e. cyclic dependencies). For each of

these, they construct an equation system and solve it numerically.

Our approach can handle recursive function calls, but does not support recursive data structures, such as trees and linked lists. Recently, a technique was presented that can additionally handle recursive data structures [2]. To achieve this, they use two techniques called *defunctionalisation* and *refunctionalisation*. The idea of defunctionalisation is to postpone the generation to the point of use. Thus, instead of needing to store the entire recursive data structure, they are generated on-the-fly when they are used. For example, instead of first generating a linked list and later performing some operation on each list element, we generate each list element only when it is used. Therefore, the list can be represented simply by a position counter. Whereas previously, each list element consisted of a value and a list, each list element now consists of a value and a counter representing the rest of the list. Refunctionalisation does the opposite, by moving the work to the point of generation.

If successful, these techniques result in a program that still has recursive function calls, but only non-recursive types. They then perform exact inference on this transformed program. For this, they construct an equation system and solve it numerically.

Nested query blocks are available in several other programming languages. The first notable example is *Church* [7]. It's noteworthy that nested conditioning in church is handled without special language features – instead, it is implemented as a normal function in the language. This approach works in languages that support higher-order functions. As our language lacks these, we instead implement nested conditioning with special instructions. Nested conditioning is also supported in the logic programming language *ProbLog* [14], a probabilistic extension of *Prolog*. To support nested queries, they instantiate a separate *ProbLog* engine for each query.

Semantics of nested conditioning are treated in [25]. They present a small-step reduction operating on configurations that consist of an entropy value and an expression. The entropy value is used to resolve probabilistic choice and can be split into multiple, independent components for programs with multiple probabilistic choices. Each step is assigned a weight to handle scoring (where scoring is a generalisation of conditioning where each path is assigned a certain weight, instead of being blocked or not blocked). Without nested conditioning, probabilities are normalised (using the weight of each path) after termination. However, query requires this normalisation to occur within a program, and thus means that the normalisation and the small-step reduction depend on each other. To ensure this is still well-defined, they use step-indexing for both the small-step reduction and normalisation. They limit the number of steps in the operational semantics and during normalisation to n and then take the limit as n approaches ∞ .

Chapter 2

The Conditioned Recursive Probabilistic Programming Language

In this chapter, we introduce the *conditioned recursive probabilistic programming language* (CRPPL). In Section 2.1, we give an intuition-based introduction to the key features of our language. In particular, we demonstrate why it is useful to limit the scope of conditioning to query blocks. We then show how nested conditioning can be combined with (unbounded) recursion to model “reasoning about reasoning”.

In the second part of the chapter, we take a more formal approach. We inductively define the syntax of the language. Additionally, we look at a class of programs that are syntactically correct, but nonetheless nonsensical, as the probability that an observation is violated is 1.

Later, we define the formal semantics of the language in Chapter 3 (operational semantics) and Chapter 4 (denotational semantics).

2.1 Overview

Our language is an imperative language with C-like syntax. Each program consists of several functions, one of them designated the `main` function. Each function may contain variable assignments, conditional choice (“if-then-else”), loops and calls to other functions. One statement not usually found in C-like languages is probabilistic choice, which has the form

$$\{ P_1 \} [p] \{ P_2 \},$$

where p is an expression that evaluates to a number in $[0, 1]$ and P_1 and P_2 are two sub-programs. With probability p , we execute P_1 and with probability $1 - p$, we execute P_2 .

The instructions `skip` and `abort` are also usually not found in C-like languages. While `skip` does nothing, is useful for cases where an instruction is needed, but where no action should be performed. The instruction `abort` be-

has like `while true { skip }`, i.e. it enters an infinite loop. We use it instead of such a loop because it is much easier to specify the semantics for `abort`.

For simplicity, the formal definition of our language omits a `return` statement and instead stores the return value in a special variable `out`. This allows us to return values to the caller, but prevents the complicated control flow that `return` can cause. This difference is discussed in more detail in Section 2.2.1. Note that the formal semantics are defined for programs where functions always have a return value. In this introduction, we also allow `void` functions to improve clarity. It's easy to transform a program with `void` functions into one where every function returns a value by returning e.g. 0 in those cases and not using the return value.

We generally restrict ourselves to programs with variables from finite domains such as Booleans or statically bounded integers. This ensures that we can later transform the program into a finite-state probabilistic pushdown automaton. In the following examples, variables are always bounded, even though we do not explicitly specify those bounds.

2.1.1 Global Conditioning

There are two ways of adding conditioning to a program: We can either run the program without modifications and specify a set of acceptable end configurations, or we can add explicit conditioning instructions directly to the program¹. Specifying the condition separately is feasible for some simple situations, but it becomes unwieldy in more complex scenarios, e.g. when we need to condition on variables in recursively nested scopes. We therefore choose the other approach and express conditioning as part of the program. For this, we add a special instruction to our program:

`observe b`

Here, b is a Boolean condition. If b holds, then the instruction behaves like `skip`. If, on the other hand, b does not hold, then the path is *blocked*. Blocked paths do not contribute to the final distribution of a program. A blocked path is different from a diverging path: The latter still contributes to the final distribution by increasing the probability of divergence. For example, consider the following program:

```
void main() {
  int x; int y;
  { x := 1 } [0.5] {x := 2};
  { y := 1 } [0.5] {y := 2};
  observe(x + y == 3)
}
```

The program has two probabilistic choice instructions and thus four paths, each with probability $\frac{1}{4}$. The final variable valuations of these paths are $\{x \mapsto 1, y \mapsto 1\}$, $\{x \mapsto 1, y \mapsto 2\}$, $\{x \mapsto 2, y \mapsto 1\}$ and $\{x \mapsto 2, y \mapsto 2\}$. However, only the paths with valuations $\{x \mapsto 1, y \mapsto 2\}$ and $\{x \mapsto 2, y \mapsto 1\}$ pass the `observe` statement, whereas the others are blocked by it. Therefore, the probability that $x = 1$ and $y = 2$ after termination is $\frac{1}{2}$.

¹These two approaches are compared in detail in Chapter 5.

Conditioning and Termination

Conditioning can also change the termination probability. Consider the following program:

```
void main() {
  int x;
  { abort } [0.5] {{ x := 1 } [0.5] {x := 2}};
  observe(x == 1);
}
```

This program has three paths: One path diverges (due to the `abort`) with probability $\frac{1}{2}$, and two other paths terminate with $\{x \mapsto 1\}$ and $\{x \mapsto 2\}$ respectively, each with probability $\frac{1}{4}$. However, the path with $\{x \mapsto 2\}$ is blocked by the `observe` statement and thus does not contribute to the final distribution. The total probability of the non-blocked paths is $\frac{1}{2} + \frac{1}{4} = \frac{3}{4}$. Therefore, the final distribution of the program is as follows: With probability $\frac{1/2}{3/4} = \frac{2}{3}$, the program diverges, and with probability $\frac{1/4}{3/4} = \frac{1}{3}$, the program terminates with $\{x \mapsto 1\}$. Conditioning thus decreased the termination probability by blocking a terminating run. This behaviour is consistent with existing semantics, such as the one in [17].

Conversely, conditioning can also increase the termination probability by blocking a diverging run. However, we cannot condition on termination directly, i.e. it is generally not possible to eliminate all non-terminating runs of a program. This is because `observe` statements are only executed if they are reached. A run that never reaches an `observe` statement (e.g. because it entered an infinite loop before the `observe` statement) is not blocked and thus affects the final distribution.

2.1.2 Local Conditioning With Query Blocks

Instead of blocking a path, we can achieve the same behaviour by looping back every blocked path to the initial state of the program. This results in exactly the same final distribution², but reveals an issue with our approach: Every path always restarts at the beginning of the program, which can lead to undesirable effects.

Consider the following function:

```
void main() {
  int x = uniform(10);
  F(x)
}
```

Here, `uniform(10)` is a function that returns $0, 1, \dots, 9$ with probability $\frac{1}{10}$ each. Since variables are passed by value, `F` cannot change the value of `x` in the scope of `main`. Therefore, one would expect `x` to be distributed uniformly in the final distribution of the program (unless `F` diverges).

However, that is not necessarily the case. Consider the following definition of `F`:

²This does not hold if almost all paths in a program are blocked. In this case, the blocked-path perspective yields an undefined distribution, as we would have to divide by 0. On the other hand, the approach where we loop back to the beginning yields a distribution where the probability of divergence is 1.

```

void F(int x) {
    int y = uniform(10);
    observe(x >= y)
}

```

There are 10 paths for each value of x . For $x = 0$, 9 of those 10 paths are blocked, as only $y = 0$ satisfies the condition $x \geq y$. For $x = 9$, on the other hand, no paths are blocked, since $x \geq y$ for all $y \in \{0, \dots, 9\}$. Since conditioning is global, these blocked paths affect the final distribution of the program: $x = 9$ occurs with probability $\frac{10}{55}$, whereas $x = 1$ only has probability $\frac{1}{55}$.

This may be desirable in some cases, but poses problems in many cases. It means that a function with a certain behaviour won't necessarily exhibit the same behaviour when called from elsewhere. It would be very helpful to specify the area affected by conditioning, i.e. the location where execution is restarted if a path is blocked. For this, we introduce query blocks. Consider this alternate definition of F :

```

void F(int x) {
    query {
        int y = uniform(10);
        observe(x >= y)
    }
}

```

With this definition, the conditioning only affects the distribution of y , since y is sampled inside of the query block. On the other hand, x is sampled outside of it and thus remains uniformly distributed.

This approach to conditioning is common in functional languages such as Church [7]. This is also where the name of the `query` block is taken from. Usually, conditioning in these languages is not part of the language, but simply implemented as a function that performs rejection sampling (this is not possible for our language, as it lacks higher-order variables). A similar approach is taken in [3], where conditioning is global, but a special instruction transforms a program into a sampler (so any conditioning that happens inside this sampler does not “leak” out). On the other hand, imperative programs usually implement global conditioning (see e.g. [17]).

Query blocks don't have to be restricted to a single function. If some function F contains a query block, calls some other function G from within that query block, and an observe violation occurs in G , execution simply restarts at the beginning of F 's query block. This naturally raises the question of how conditioning and recursion interact. This behaviour is similar to try-catch blocks and exceptions: When an observe statement is violated or an exception is thrown, the program stack is unwound until a query block or try-catch block is reached.

2.1.3 Recursively Nested Conditioning

While query blocks can be useful even if they are not nested at all (as demonstrated in the previous example), it is often more interesting to nest them. In particular, one can express interesting behaviour by nesting them in recursive functions. Consider for example two players that are playing a game where they alternate taking turns. To determine the optimal move, a player not only has

to think about how their move will change the game state, but also how the opponent will react – and this in turn depends on what the opponent thinks the player will do, and so on.

This can be described naturally with nested conditioning in a function of the form

```
int player_1(int state) {
  query {
    int action = sample_action(state);
    int new_state = update_state(state, action);
    if !is_terminal(new_state) {
      int outcome = player_2(new_state);
      observe(outcome == 1);
      out = outcome;
    } else {
      out = get_winner(new_state);
    }
  }
}
```

Here, `sample_action` probabilistically returns a legal action for the player based on the current state. Based on this action and the current state, we compute the new state using `update_state`. If the new state is not a terminal state, we then perform a recursive call to `player_2` (which is defined similarly to `player_1`). This recursive call then returns the outcome of the game and we condition on winning. The specific implementation of `sample_action`, `update_state`, `is_terminal` and `get_winner` depends on the game.

The framework presented above may have issues if the game played can take unboundedly many steps. If one player can force a win, it is also necessary to modify the observe statement so that not all runs are blocked.

We use the above framework to determine the winner of a simple counting game in Section 6.2.2. There, we also discuss some further modifications to improve the quality of inference. In particular, it is of great importance to consider which probabilistic choices should be affected by conditioning. In the above example, both `sample_action` and `player_2` are probabilistic functions. As they are inside of the query block, they are affected by conditioning. We found that inference yields better results if only `sample_action` is affected by conditioning and present a method for achieving this in the second part of Section 6.2.2.

2.2 Syntax

To give the syntax of the program, we first inductively define an instruction. A program is then made up of several functions, where each function consists of a name, several parameter variables and an instruction that is executed when the function is called.

2.2.1 Instructions

Definition 2.1 (Instruction). An instruction is constructed using the following grammar:

| | | | |
|-------------|-----|---|-------------------------|
| INST | ::= | skip | (effectless program) |
| | | abort | (diverging program) |
| | | $x := e$ | (assignment) |
| | | INST ; INST | (concatenation) |
| | | if b { INST } else { INST } | (conditional branching) |
| | | { INST } [p] { INST } | (probabilistic choice) |
| | | while b { INST } | (loop) |
| | | $x := F(\vec{e})$ | (function call) |
| | | observe b | (conditioning) |
| | | query { INST } | (conditioning scope) |

Here, x is a rational variable, F is a function name, e and b are a rational and Boolean expression, respectively, p is an arithmetic expression that evaluates to a probability, and \vec{e} is a vector of rational expressions. Rational expressions are composed of the variables of the current function, rational constants and the binary arithmetic operators $+$, $-$, $*$ and $/$. For simplicity, we assume that divisions by 0 are mapped to ∞ . We refer to the set of rational, arithmetic expressions as AE . Boolean expressions are composed of rational expressions, the binary constants **true** and **false**, the unary operator **!**, the binary operators **&&** and **||** and the binary relations **==**, **!=**, **<**, **>**, **<=** and **>=**.

Returning Values from Functions

Notice that there is no **return** instruction in the above grammar. Instead, a variable **out** is available in every function. When a function wants to set its return value, it assigns a value to **out**. When the end of the function is reached, the value of **out** is returned to the caller. Apart from this, **out** behaves like a normal variable.

2.2.2 Program Fragments

We refer to the set of all instructions as *Insts* and to the set of all variable names as *Vars*. We distinguish between *program fragments*, which result directly from the syntax given in Section 2.1, and *programs*, which restrict where observe statements can be located. We first define program fragments and then introduce the additional restrictions for programs. We mainly define program fragments because they naturally occur in inductive proofs.

Definition 2.2 (Program fragment). A program fragment P is defined as a 4-tuple $(Func, Code, Params, main)$, where

- $Func$ is a finite set of function identifiers,
- $Code: Func \rightarrow Insts$ maps each function identifier to an instruction,
- $Params: Func \rightarrow Vars^*$ maps each function identifier to a list of variables that we call *parameters* and
- $main \in Func$ is the initial function (with $|Params(main)| = 0$).

Throughout the remaining chapters, we write $Func = \{F_1, \dots, F_n\}$, where F_i has m_i parameters.

Instead of specifying the elements of the program (fragment) four-tuple, we usually give the entire program in code form. A function F is written as

$$F(\text{Params}(F)) \{ \text{Code}(F) \}$$

and a program (fragment) as the concatenation of its functions. We usually call the main function `main`. Lines starting with `//` are comments.

2.2.3 Syntactic Sugar

To make examples more concise, we use a few additional constructions. These are not part of the language syntax, but can easily be converted into legal programs.

- Instead of `if b { P } else { skip }`, we write `if b { P }`.
- When we are not interested in the return value of a function, we write $F(\vec{e})$. This can be replaced by `temp := F(\vec{e})`, where `temp` is an otherwise unused variable.
- We also allow function calls in expressions. An expression containing function calls can be converted into a syntactically correct expression by replacing each call $F(\vec{e})$ by a different new variable x and adding the assignment $x := F(\vec{e})$ immediately before the expression. As function calls do not have side effects, the order of calls does not affect the behaviour of the program.

2.2.4 Programs

The syntax above allows us to place observe statements anywhere. However, observe statements need a surrounding query block to define their scope. Otherwise, it would be unclear where execution should resume after an observe violation³. Observe statements can be directly contained in a query block or indirectly, where a function is called from within a query block and that other function then contains an observe statement. More formally, we define this as follows:

Definition 2.3 (Query Function). Let $P = (\text{Func}, \text{Code}, \text{Params}, \text{main})$ be a program fragment. The set of *query functions* is the largest set $Q \subseteq \text{Func}$ such that

1. $\text{main} \notin Q$ and
2. for every $F_i \in Q$, every call $x := F_i(\vec{e})$ is either in some function $F_j \in Q$ or the call is contained in a query block.

Example 2.4. In the following program fragment, F2 and F3 are query functions, whereas `main`, F1 and F4 are not. `main` is not a query function because it is the program's main function, F1 is called from `main` without a query block, and F4 is called from F1 without a query block.

³One could implicitly assume that there is a query block around the code of the main function, but this would needlessly complicate the semantics, particularly in Chapter 4.

```

main() { F1(); query { F2() } }
F1() { query {F3()}; F4() }
F2() { F3() }
F3() { F4() }
F4() { skip }

```

Using the notion of query functions, we can formally define which program fragments are programs.

Definition 2.5 (Program). We call a program fragment a program if every observe statement is either contained in a query function or a query block.

Theorem 2.6. The set of query functions can be computed in polynomial time.

Proof. The following algorithm computes the set of all query functions.

1. Let X be the set of all functions except *main*.
2. For every $F \in X$, check whether it is called from some function $F' \notin X$ where the call is not contained in a query block. If so, remove F from X .
3. Repeat Step 2 until X no longer changes.

Step 2 is run at most n times (where n is the number of functions). In the beginning, X has size $n - 1$ and in each iteration, at least one function is removed from X . Thus, after $n - 1$ iterations, X must be empty and the algorithm terminates after one additional iteration. Each execution of Step 2 takes polynomial time (a naive implementation achieves quadratic runtime by going through the program once for each function). Therefore, the algorithm runs in polynomial time.

After termination, X only contains query functions. If X still contained a function that is not a query function, then X would change after executing Step 2 – but the algorithm only terminates after X no longer changes.

After termination, X contains every query function. This can be shown inductively. We know that initially, every query function is contained in X , because the only function not in X is *main* and this is never a query function. In every iteration, we only remove a function F if it is called from a function that is not in X (without surrounding query block). By induction hypothesis, any function not in X is also not a query function. Therefore, F does not fulfil the conditions of a query function. We can thus remove it from X without invalidating the induction hypothesis.

Therefore, a function is contained in X after termination if and only if it is a query function. □

2.3 Illegal Programs

The following instruction is syntactically correct:

```

query {
  observe(false)
}

```

Since it does not have any probabilistic behaviour, it only has a single run, and that run is blocked by the observe statement. What is the probability that the program terminates? The standard answer – summing over the probability of all terminating runs and dividing by the probability of all unblocked runs – fails us, because the probability of all runs is 0.

To avoid this situation, we exclude this kind of program. This is consistent with other definitions, such as [17]. Since this is a semantic property, we provide the formal definition in Section 3.4.

Chapter 3

Operational Semantics

For the operational semantics, we provide construction rules to transform a CRPPL program into a probabilistic pushdown automaton (pPDA). Using pPDAs for the semantics is suitable because they can model recursion using their stack. Therefore, even a program with unbounded recursion depth (and thus infinitely many different call stack contents over the course of execution) can be modelled by a finite-state pPDA.

However, the rest of the program state, i.e. the variables, is stored in the states of the pPDA, so it is possible to write programs that cannot be modelled by a finite-state pPDA. Any program where a variable assumes infinitely many different values over the course of execution induces an infinite pPDA. Restricting all variables to a bounded domain solves this problem. In implementations based on this operational semantics, one can choose to either statically define variable bounds before execution (as is done e.g. in the language of the probabilistic model checker PRISM [12]) or accept that model building might not terminate.

In this section, we first define pPDAs. We then provide construction rules for transforming a CRPPL program into a pPDA and give several examples of how the construction works. Finally, we show how termination probabilities and variable distributions in the CRPPL program correspond to reachability probabilities in the program.

3.1 Probabilistic Pushdown Automata

pPDAs extend discrete-time Markov chains with a stack or, equivalently, extend a pushdown system [5] with probabilistic transitions. Like Markov chains and pushdown systems – but unlike pushdown automata – they do not recognise a language. Instead, we formulate our properties over the states of the automaton and, in particular, we will the probability that the pPDA terminates in a given set of states.

In Chapter 5, we later briefly analyse pPDAs that *do* produce a language.

Definition 3.1. Formally, a probabilistic pushdown automaton is a 6-tuple $\mathcal{A} = (Q, \Gamma, \Delta, Prob, q_0, Z_0)$, where

- Q is a finite set of states,

- Γ is a finite set of stack symbols,
- $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ is the transition relation,
- $Prob : \Delta \rightarrow (0, 1]$ assigns a rational probability to each transition (and we write $qZ \xrightarrow{p} q'\alpha$ if $((q, Z), (q', \alpha)) \in \Delta$ and $Prob(q, Z, q', \alpha) = p$),
- $q_0 \in Q$ is the initial state and
- $Z_0 \in \Gamma$ is the initial stack symbol.

For any $q \in Q, Z \in \Gamma$, we require $\sum_{(qZ \xrightarrow{p} q'\alpha) \in \Delta} Prob((qZ \xrightarrow{p} q'\alpha)) = 1$ (but we frequently omit unreachable transitions).

A configuration (q, γ) of \mathcal{A} consists of a state $q \in Q$ and a stack $\gamma \in \Gamma^*$. From configuration $(q, Z\gamma)$, transitions of the form $qZ \xrightarrow{p} q'\alpha$ are enabled. We choose one of them probabilistically and transition to $(q', \alpha\gamma)$. Starting from configuration (q_0, Z_0) , this process is repeated until the stack is empty.

The pPDAs we construct in this section only use certain types of transitions. We distinguish between *push*, *pop* and *internal* transitions. The outgoing transitions of a state must all be of the same type.

Internal transitions with probability p are drawn as $q \xrightarrow{p} q'$ and neither depend on the current stack symbol nor modify it. Push transitions are drawn as $q \xrightarrow{\text{push } Z} q'$ and push the symbol Z to the stack. Note that they don't depend on the current stack nor have a probability, so each push state can only have one outgoing transition. Pop transitions are drawn as $q \xrightarrow{\text{pop } Z} q'$ and pop the symbol Z from the stack. A pop state can thus have one outgoing pop transition for every stack symbol Z . This distinction is similar to the *probabilistic visibly pushdown automata* from [24].

3.1.1 Analysing Probabilistic Pushdown Automata

We are mainly interested in analysing reachability and termination probabilities of a given pPDA. We further distinguish between almost-sure termination or reachability (“Is this set of states reached with probability 1?”) and quantitative reachability (“Is the set of states reached with at least (or at most) probability p ?”). As termination and reachability probabilities can be irrational, we cannot compute the precise probability and thus restrict ourselves to finding bounds. All the relevant problems are in PSPACE [1].

3.2 Construction Rules

In this section, we present rules to construct a pPDA from a given CRPPL program. The rules are introduced one-by-one with explanations. An overview of all rules can be found in Figure 3.1.

Most states have the form $\langle P, v \rangle$, where P is an instruction and v is a variable mapping. Formally, $v : \text{Vars} \rightarrow \mathbb{Q}$ is a partial function that assigns a rational value to each variable that is currently in scope.

As part of the construction, we generate additional instructions that the user is not allowed to write, but that are otherwise handled like normal instructions. Furthermore, to indicate termination, we use states of the form $\langle \downarrow, v \rangle$, where v once again is a variable mapping.

Skip

The effectless program **skip** does not change the program state, so we simply go to the terminal state with the same valuation.

$$\frac{}{\langle \mathbf{skip}, v \rangle \xrightarrow{1} \langle \downarrow, v \rangle}$$

Abort

The diverging program **abort** never terminates. We therefore add a transition back to **abort**. For the terminal reachability probabilities, the variable valuation does not matter, since an aborting program never contributes to it.

$$\frac{}{\langle \mathbf{abort}, v \rangle \xrightarrow{1} \langle \mathbf{abort}, v \rangle}$$

Assignment

For the assignment $x := e$, we transition to the terminal state with variable valuation $v[x/s]$, where s is the value of expression e under variable valuation v . For example, if $v = \{x \mapsto 3, y \mapsto 5\}$, then we transition from $\langle x := 2x + y, v \rangle$ to $\langle \downarrow, \{x \mapsto 11, y \mapsto 5\} \rangle$.

$$\frac{v(e) = s}{\langle x := e, v \rangle \xrightarrow{1} \langle \downarrow, v[x/s] \rangle}$$

Probabilistic Choice

For the probabilistic program $\{ P_1 \} [p] \{ P_2 \}$, we transition to P_1 with probability p and to P_2 with probability $1 - p$. The value of v is not changed.

$$\frac{v(p) = x}{\langle \{ P_1 \} [p] \{ P_2 \}, v \rangle \xrightarrow{x} \langle P_1, v \rangle} \quad \frac{v(p) = x}{\langle \{ P_1 \} [p] \{ P_2 \}, v \rangle \xrightarrow{1-x} \langle P_2, v \rangle}$$

If-Else Block

For program **if** b $\{ P_1 \}$ **else** $\{ P_2 \}$, we evaluate b under the current variable valuation and transition to P_1 if $b = \mathit{true}$ and to P_2 otherwise.

$$\frac{v(b) = \mathit{true}}{\langle \mathbf{if} \ b \ \{ P_1 \} \ \mathbf{else} \ \{ P_2 \}, v \rangle \xrightarrow{1} \langle P_1, v \rangle}$$
$$\frac{v(b) = \mathit{false}}{\langle \mathbf{if} \ b \ \{ P_1 \} \ \mathbf{else} \ \{ P_2 \}, v \rangle \xrightarrow{1} \langle P_2, v \rangle}$$

While Loop

For while loop `while b { P' }`, we once again have to consider two cases. If the loop condition b is *true*, then we execute the loop body P' . We concatenate this with the `while` loop to ensure that the loop body is run again until the loop condition no longer holds. If the loop condition doesn't hold, we go to the terminal state with the same variable valuation.

$$\frac{v(b) = true}{\langle \text{while } b \{ P' \}, v \rangle \xrightarrow{1} \langle P'; \text{while } b \{ P' \}, v \rangle}$$

$$\frac{v(b) = false}{\langle \text{while } b \{ P' \}, v \rangle \xrightarrow{1} \langle \downarrow, v \rangle}$$

Concatenation (internal)

For concatenation $P_1; P_2$, we “run” P_1 from v , yielding some P'_1 with valuation v' , and then concatenate this result with P_2 .

$$\frac{\langle P_1, v \rangle \xrightarrow{p} \langle P'_1, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{p} \langle P'_1; P_2, v' \rangle}$$

The only exception to this is if P_1 terminates. Since \downarrow is not an instruction, it does not make sense to go to state $\langle \downarrow; P_2, v' \rangle$. Instead, we simply continue execution with P_2 , since P_1 has terminated.

$$\frac{\langle P_1, v \rangle \xrightarrow{p} \langle \downarrow, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{p} \langle P_2, v' \rangle}$$

Function Call

Handling a function call $x := F(\vec{e})$ is slightly more involved. First, we evaluate the parameters \vec{e} passed to the function, yielding a variable assignment v' mapping each parameter to its value. The code associated with F is indicated by $Code(F)$, so we create a transition to $\langle Code(F), v' \rangle$. This assumes that non-parameter variables of F are implicitly initialised to 0 in F .

However, this only handles the invocation of the function. We also need to ensure that we return to the caller after execution has finished, restore its variable valuation and assign the return value of F to x .

There are different ways to approach this. The simplest would be to append the remaining code of the caller after $Code(F)$. However, this would mean that infinite recursion can only be modelled by an infinite-state pPDA. Our approach avoids this: We push the symbol $[P, v]$ onto the stack. This indicates that – once the function has terminated – execution should resume at state $\langle P, v \rangle$.

As our program has the form $x := F(\vec{e})$, no further instructions are executed after F returns. The only task remaining after function termination is handling the return value. During function execution, the return value is stored in `out`. To avoid conflicts with the caller's `out` variable, we assume that the value is transferred to a different variable `in`. This means that we only have to execute the assignment $x := \text{in}$ once execution resumes at the caller.

$$\frac{v(\vec{e}) = v'}{\langle x := F(\vec{e}), v \rangle \xrightarrow{\text{push } [x := \text{in}, v]} \langle \text{code}(F), v' \rangle}$$

We have now stored the necessary information on the stack. It remains to actually restore it once function execution has terminated. For this, we add a new transition from all terminal states. If a symbol $[P, v']$ is on the stack, we transition to $\langle P, v'[\text{in}/s] \rangle$, where s is the value of **out**. This means that code resumes at P with valuation v' , except that **in** now contains the return value of the function. As shown above, this is then used to assign the return value to x .

$$\frac{v(\text{out}) = s}{\langle \downarrow, v \rangle \xrightarrow{\text{pop } [P, v']} \langle P, v'[\text{in}/s] \rangle}$$

It should be noted that it is possible to have multiple outgoing transitions from one $\langle \downarrow, v \rangle$ state. This happens if multiple callers call the same function with the same parameters¹. If the main function terminates, the stack is empty. In this case, we transition to the terminal state $\langle \downarrow_0, v \rangle$, which indicates that execution has finished.

$$\langle \downarrow, v \rangle \xrightarrow{\text{pop } Z_0} \langle \downarrow_0, v \rangle$$

Concatenation (Push)

In the previous section, we have created push transitions, but the concatenation rules introduced so far only work for internal transitions. Consider state $\langle P_1; P_2, v \rangle$. Similar to the internal case, we first look at the behaviour of P_1 from v . Assume this behaviour consists of a push transition $\langle P_1, v \rangle \xrightarrow{\text{push } [P', v]} \langle P'_1, v' \rangle$. This indicates that a function is called that has code P'_1 and that after this function, we should resume from $\langle P, v \rangle$.

We now want to achieve a similar behaviour, but starting from $\langle P_1; P_2, v \rangle$ instead of $\langle P_1, v \rangle$. In the internal case, we did this by going to $\langle P'_1; P_2, v' \rangle$, but this does not work in the push case: It would mean that we first execute the function body P'_1 , then P_2 , and only thereafter leave the function and resume execution at P . Instead, P_2 should come after both other steps. We achieve this by adding P_2 to the push transition instead of the target state.

$$\frac{\langle P_1, v \rangle \xrightarrow{\text{push } [P', s]} \langle P'_1, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{\text{push } [P; P_2, v]} \langle P'_1, v' \rangle}$$

Note that the same variable valuation v occurs both in the outgoing state and in the stack symbol. This is possible because all $\text{push}[\cdot]$ transitions we create have the same variable valuation in both cases.

Also note that we do not need to handle the case that there is a push transition from $\langle P_1, v \rangle$ to $\langle \downarrow, v \rangle$, as all rules that create transitions to terminal states only create internal transitions.

¹Or if they call a function with different parameters, but the variable valuation is the same after function execution

Query and Observe

We implement conditioning using *rejection sampling*, i.e., whenever an observe statement is violated, we jump back to the beginning of the surrounding query block. This ensures that we can only leave the query block when no observe statements were violated. The challenge comes from determining where to jump back to – especially when the location cannot be determined at “compile time”. This can happen if a function is called in the query block and the observe violation occurs inside of this other function.

The approach we use is based on the stack. We already push symbols of the form $[P, v]$ to the stack to indicate that, on return, execution should resume at $\langle P, v \rangle$. For query blocks, we introduce query symbols of the form $\langle P, v \rangle$ to indicate that, on observe violation, execution should resume at $\langle P, v \rangle$.

As stated above, execution should resume by rerunning the query block, so for query program `query { P' }` with valuation v , we push the symbol $\langle P', v \rangle$ to the stack. We then execute the content of the query block. Finally, we must ensure that the query symbol does not remain on the stack after we leave it. For this, we execute a special instruction `pop(Z)` which removes the symbol from the stack. This is only run after P' has terminated (without violating an observe statement). The semantics of `pop(Z)` are defined below.

$$\frac{}{\langle \text{query } \{ P' \}, v \rangle \xrightarrow{\text{push } \langle \text{query } \{ P' \}, v \rangle} \langle P'; \text{pop}(\langle \text{query } \{ P' \}, v \rangle), v \rangle}$$

$$\frac{}{\langle \text{pop}(Z), v \rangle \xrightarrow{\text{pop } Z} \langle \downarrow, v \rangle}$$

Now that the stack contains information on where to jump back to, we can handle observe statements. If the observe condition holds, we simply go to the terminal state with the same valuation.

$$\frac{v(b) = \text{true}}{\langle \text{observe } b, v \rangle \xrightarrow{1} \langle \downarrow, v \rangle}$$

More interesting is the case where the observe condition is violated. We need to go back to the surrounding query block by taking the top symbol from the stack. To make this more readable, we introduce a new state $\langle \ddagger \rangle$ where this is handled.

$$\frac{v(e) = \text{false}}{\langle \text{observe } b, v \rangle \xrightarrow{1} \langle \ddagger \rangle}$$

In $\langle \ddagger \rangle$, we have to consider a few cases. If the topmost stack symbol is a query symbol $\langle P, v' \rangle$, we can resume execution at $\langle P, v' \rangle$.

$$\frac{}{\langle \ddagger \rangle \xrightarrow{\text{pop } \langle P, v' \rangle} \langle P, v' \rangle}$$

It is also possible that the topmost symbol is a function symbol. This can happen if we enter a query block, call a function (which pushes a function symbol onto the stack) and then violate an observe statement. We can discard this function symbol, since we resume execution outside of the function.

$$\frac{}{\langle \downarrow \rangle \xrightarrow{\text{pop } [P, v']} \langle \downarrow \rangle}$$

Because every observe statement must be surrounded by a query block, it is not possible to reach $\langle \downarrow \rangle$ such that only Z_0 is on the stack. However, for the inductive proofs in the following sections, we also have to construct pPDAs for program fragments, where this does not necessarily hold. Therefore, we introduce a rule that transitions to a (normally unreachable) state $\langle \downarrow \perp \rangle$ in this case. This state can be interpreted as an “unhandled observe violation”.

$$\frac{}{\langle \downarrow \rangle \xrightarrow{\text{pop } Z_0} \langle \downarrow \perp \rangle}$$

Concatenation of Query Symbols

Assume we are in state $\langle P_1; P_2, v \rangle$. We create transitions from this state based on the outgoing transitions of state $\langle P_1, v \rangle$. Since we have introduced new stack symbols for query blocks, we need to handle a few additional cases. So far, our concatenation rules can only handle function symbols of the form $[P, v]$, not query symbols of the form $\langle P, v \rangle$.

We first consider the case of a $\text{push}(\langle P, v \rangle)$ transition to some $\langle P'_1, v' \rangle$. Such a transition is generated by a query block and indicates that we should attempt to execute P'_1 , and if this fails, rerun from $\langle P, v \rangle$. We want to execute P_2 after P'_1 successfully terminated. For this, we create a transition to $\langle P'_1; P_2, v \rangle$. However, if we leave the pushed stack symbol unchanged, the following is possible: We first execute P'_1 , but encounter an observe violation. The topmost stack symbol is $\langle \text{query } \{ P'_1 \}, v \rangle$, so we resume from $\langle \text{query } \{ P'_1 \}, v \rangle$. This time, P'_1 terminates without an observe violation. We now want to execute P_2 , but this got lost during the first “abort”. To fix this, we also need to add P_2 to the stack symbol.

$$\frac{\langle P_1, v \rangle \xrightarrow{\text{push } \langle P, v \rangle} \langle P'_1, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{\text{push } \langle P; P_2, v \rangle} \langle P'_1; P_2, v' \rangle}$$

Next, assume that $\langle P_1, v \rangle$ has an outgoing pop transition $\text{pop}(\langle P^*, v^* \rangle)$. These transitions are only generated at the $\langle \downarrow \rangle$ state (which never occurs in concatenations) and by the $\text{pop}(Z)$ instruction. Since $\text{pop}(Z)$ ignores the popped symbol – after all, it is only executed on leaving a query block after no observe statement was violated – it discards the value of the stack symbol. We thus do the same in the concatenation rules.

$$\frac{\langle P_1, v \rangle \xrightarrow{\text{pop } \langle P^*, v^* \rangle} \langle P'_1, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{\text{pop } \langle P^*, v^* \rangle} \langle P'_1; P_2, v' \rangle}$$

$$\frac{\langle P_1, v \rangle \xrightarrow{\text{pop } \langle P^*, v^* \rangle} \langle \downarrow, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{\text{pop } \langle P^*, v^* \rangle} \langle P_2, v' \rangle}$$

With all rules in place, we can now formally define the pPDA that corresponds to a given program and initial variable valuation.

Definition 3.2. Let P be a program and v a variable valuation. The pPDA $Op(P, v)$ is the smallest pPDA with initial state $\langle P, v \rangle$ that fulfils all rules from Figure 3.1.

3.2.1 Properties of the pPDA

The bottom stack symbol Z_0 is only popped in transitions to $\langle \downarrow_{\perp} \rangle$ and $\langle \downarrow_0, v' \rangle$. Therefore, these are the only two states the automaton of any program fragment can terminate in (and since they are only reachable by the transition that pops Z_0 , they are always the final state of any run). This leads to the following lemma, which is useful in the induction steps of some proofs later on.

Lemma 3.3. Let P be any program fragment. Then any run in $Op(P, v)$ either

- diverges,
- reaches $\langle \downarrow_{\perp} \rangle$ with empty stack or
- reaches $\langle \downarrow_0, v' \rangle$ with empty stack for some variable valuation $v' \in V$.

3.2.2 Reachability Probabilities

Using the pPDA from the preceding section, we can now define the terminal reachability probabilities that give the semantics of the program.

Definition 3.4 (Reachability probability). Let P be a program fragment, let $v \in V$ be a variable valuation and let $V' \subseteq V$ be a set of variables. The terminal reachability probability $R[P, v](V')$ is the probability of reaching a state $\langle \downarrow_0, v' \rangle$ with $v' \in V'$. We write $R[P, v](v')$ instead of $R[P, v](\{v'\})$.

We also introduce the slightly more general notion of *weighted terminal reachability*.

Definition 3.5 (Weighted terminal reachability probability). Let $f: V \rightarrow \mathbb{R}$ be a function that assigns a real weight to each variable valuation. The weighted terminal reachability probability $W[P, v](f)$ is given by:

$$W[P, v](f) = \sum_{v' \in V} f(v') \cdot R[P, v](v')$$

Note that we are always operating on finite pPDAs, we sum only over finitely many terms.

Theorem 3.6 (Compatibility between R and W). For any program P , initial valuation $v \in V$ and set of states $V' \subseteq V$, we have

$$W[P, v](\llbracket V' \rrbracket) = R[P, v](V'),$$

where

$$\llbracket V' \rrbracket(v') = \begin{cases} 1 & \text{if } v' \in V' \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{array}{c}
\text{(skip)} \frac{}{\langle \text{skip}, v \rangle \xrightarrow{1} \langle \downarrow, v \rangle} \quad \text{(abort)} \frac{}{\langle \text{abort}, v \rangle \xrightarrow{1} \langle \text{abort}, v \rangle} \\
\\
\text{(assignment)} \frac{v(e) = s}{\langle x := e, v \rangle \xrightarrow{1} \langle \downarrow, v[x/s] \rangle} \\
\\
\text{(flip)} \frac{v(p) = x}{\langle \{ P_1 \} [p] \{ P_2 \}, v \rangle \xrightarrow{x} \langle P_1, v \rangle} \quad \frac{v(p) = x}{\langle \{ P_1 \} [p] \{ P_2 \}, v \rangle \xrightarrow{1-x} \langle P_2, v \rangle} \\
\\
\text{(if)} \frac{v(b) = \text{true}}{\langle \text{if } b \{ P_1 \} \text{ else } \{ P_2 \}, v \rangle \xrightarrow{1} \langle P_1, v \rangle} \quad \frac{v(b) = \text{false}}{\langle \text{if } b \{ P_1 \} \text{ else } \{ P_2 \}, v \rangle \xrightarrow{1} \langle P_2, v \rangle} \\
\\
\text{(while)} \frac{v(b) = \text{true}}{\langle \text{while } b \{ P' \}, v \rangle \xrightarrow{1} \langle P'; \text{ while } b \{ P' \}, v \rangle} \quad \frac{v(b) = \text{false}}{\langle \text{while } b \{ P' \}, v \rangle \xrightarrow{1} \langle \downarrow, v \rangle} \\
\\
\text{(call)} \frac{v(\bar{e}) = v'}{\langle x := F(\bar{e}), v \rangle \xrightarrow{\text{push } [x := \text{in}, v]} \langle \text{code}(F), v' \rangle} \\
\\
\text{(return)} \frac{v(\text{out}) = s}{\langle \downarrow, v \rangle \xrightarrow{\text{pop } [P, v']} \langle P, v'[\text{in}/s] \rangle} \quad \frac{}{\langle \downarrow, v \rangle \xrightarrow{\text{pop } Z_0} \langle \downarrow_0, v \rangle} \\
\\
\text{(concat. (internal))} \frac{\langle P_1, v \rangle \xrightarrow{P} \langle P'_1, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{P} \langle P'_1; P_2, v' \rangle}, \quad \frac{\langle P_1, v \rangle \xrightarrow{P} \langle \downarrow, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{P} \langle P_2, v' \rangle} \\
\\
\text{(concat. (push))} \frac{\langle P_1, v \rangle \xrightarrow{\text{push } [P, s]} \langle P'_1, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{\text{push } [P; P_2, v]} \langle P'_1, v' \rangle} \\
\\
\text{(query)} \frac{}{\langle \text{query } \{ P' \}, v \rangle \xrightarrow{\text{push } \langle \text{query } \{ P' \}, v \rangle} \langle P'; \text{pop}(\langle \text{query } \{ P' \}, v), v \rangle} \\
\\
\text{(pop (auxiliary instruction))} \frac{}{\langle \text{pop}(Z), v \rangle \xrightarrow{\text{pop } Z} \langle \downarrow, v \rangle} \\
\\
\text{(observe)} \frac{v(b) = \text{true}}{\langle \text{observe } b, v \rangle \xrightarrow{1} \langle \downarrow, v \rangle} \quad \frac{v(e) = \text{false}}{\langle \text{observe } b, v \rangle \xrightarrow{1} \langle \downarrow, v \rangle} \\
\\
(\downarrow) \frac{}{\langle \downarrow \rangle \xrightarrow{\text{pop } \langle P, v' \rangle} \langle P, v' \rangle}, \quad \frac{}{\langle \downarrow \rangle \xrightarrow{\text{pop } [P, v']} \langle \downarrow \rangle}, \quad \frac{}{\langle \downarrow \rangle \xrightarrow{\text{pop } Z_0} \langle \downarrow \perp \rangle} \\
\\
\text{(concat. (push, query))} \frac{\langle P_1, v \rangle \xrightarrow{\text{push } \langle P, v \rangle} \langle P'_1, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{\text{push } \langle P; P_2, v \rangle} \langle P'_1; P_2, v' \rangle} \\
\\
\text{(concat. (pop, query))} \frac{\langle P_1, v \rangle \xrightarrow{\text{pop } \langle P^*, v^* \rangle} \langle \downarrow, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{\text{pop } \langle P^*, v^* \rangle} \langle P_2, v' \rangle} \quad \frac{\langle P_1, v \rangle \xrightarrow{\text{pop } \langle P^*, v^* \rangle} \langle P'_1, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{\text{pop } \langle P^*, v^* \rangle} \langle P'_1; P_2, v' \rangle}
\end{array}$$

Figure 3.1: Construction rules for the pPDA $Op(C, v)$

Proof. We have

$$\begin{aligned}
W [P, v] ([V']) &= \sum_{v' \in V} [V'](v') \cdot R[P, v](v') && \text{(Definition 3.5)} \\
&= \sum_{v' \in V'} 1 \cdot R[P, v](v') && ([V'](v') = 0 \text{ for } v' \notin V') \\
&= R[P, v](V'). && \text{(Definition 3.4)}
\end{aligned}$$

□

Lemma 3.7. The probability that P terminates from initial valuation v is given by $W [P, v] (\lambda v.1)$.

Proof. By Theorem 3.3, for any program fragment P , every run on $Op (P, v)$ either diverges, reaches $\langle \downarrow_{\perp} \rangle$ or reaches $\langle \downarrow_0, v' \rangle$ with empty stack. In a well-formed program, $\langle \downarrow_{\perp} \rangle$ is unreachable. Therefore, every terminating run reaches $\langle \downarrow_0, v' \rangle$ with empty stack. Furthermore, it's easy to see that every run visits $\langle \downarrow_0, v' \rangle$ at most once, as it has no outgoing transitions and is only reachable by popping the bottom stack symbol. Therefore, the probability of termination is equal to the probability of reaching any $\langle \downarrow_0, v' \rangle$ and we get:

$$\begin{aligned}
&W [P, v] (\lambda v'.1) \\
&= \sum_{v' \in V} 1 \cdot P[P, v](v') \\
&= R[P, v](V) && \text{(No run visits two terminal states.)} \\
&= \textit{Probability of termination.} && \text{(See above)}
\end{aligned}$$

□

Weighted terminal reachability assigns 0 to programs that diverge. It is sometimes of interest to include the probability of divergence in the result (e.g. when we want to avoid a certain state, but allow both termination in another state and divergence). For this, we introduce *liberal weighted terminal reachability*.

Definition 3.8 (Liberal weighted terminal reachability). Let $f: V \rightarrow \mathbb{R}$ be a function. The liberal terminal reachability probability $WL [P, v] (f)$ is given by

$$WL [P, v] (f) = 1 - W [P, v] (\lambda v'.(1 - f(v'))).$$

Lemma 3.9. For any program P , initial valuation v and function $f: V \rightarrow \mathbb{R}$, $WL [P, v] (f)$ is equal to $W [P, v] (f)$ plus the probability that $Op (P, v)$ diverges, i.e.

$$WL [P, v] (f) = W [P, v] (f) + (1 - W [P, v] (\lambda v'.1)).$$

Proof.

$$\begin{aligned}
& WL[P, v](f) \\
&= 1 - W[P, v](\lambda v'.(1 - f(v'))) \\
&= 1 - \sum_{v' \in V} (\lambda v'.(1 - f(v')))(v') \cdot P[P, v](v') \\
&= 1 - \left(\sum_{v' \in V} 1 \cdot P[P, v](v') - \sum_{v' \in V} f(v') \cdot P[P, v](v') \right) \\
&= 1 - (W[P, v](\lambda v'.1) - W[P, v](f)) \\
&= W[P, v](f) + (1 - W[P, v](\lambda v'.1))
\end{aligned}$$

□

3.3 Examples

To illustrate how the operational semantics $Op(P, v)$ of a program P with initial valuation v is constructed, we apply the rules from Figure 3.1 to three examples. In the first example, we show how function calls are handled and use the concatenation rule. In the second example, we demonstrate how recursive programs with unbounded recursion yield a finite pPDA. The last examples demonstrates how conditioning is handled.

We also present a construction to transform a pPDA into a CRPPL program.

3.3.1 Function Calls and Concatenation

In this section, we construct $Op(P, \{y \mapsto 2\})$ where P is given as follows:

```

F() { out := 5 }
main() { x := F(); x := x + y }

```

In order to construct the pPDA for the main function's code $x := F(); x := x + y$, we first need to construct the pPDA for $x := F()$, which has the following form:

$$\begin{aligned}
& \rightarrow \langle x := F(), \{y \mapsto 2\} \rangle \\
& \quad \downarrow \text{push } [x := \text{in}, \{y \mapsto 2\}] \\
& \quad \langle \text{out} := 5, \emptyset \rangle \\
& \quad \downarrow 1 \\
& \quad \langle \downarrow, \{\text{out} \mapsto 5\} \rangle \\
& \quad \downarrow \text{pop } [x := \text{in}, \{y \mapsto 2\}] \\
& \quad \langle x := \text{in}, \{\text{in} \mapsto 5, y \mapsto 2\} \rangle \\
& \quad \downarrow 1 \\
& \quad \langle \downarrow, \{x \mapsto 5, \text{in} \mapsto 5, y \mapsto 2\} \rangle
\end{aligned}$$

From this, we now construct the pPDA for $x := F(); x := x + y$ by applying the push and internal concatenation rules.

$$\begin{aligned}
&\rightarrow \langle x := F(); x := x + y, \{y \mapsto 2\} \rangle \\
&\quad \downarrow \text{push } [x := \text{in}; x := x + y, \{y \mapsto 2\}] \\
&\quad \langle \text{out} := 5, \emptyset \rangle \\
&\quad \downarrow 1 \\
&\quad \langle \downarrow, \{\text{out} \mapsto 5\} \rangle \\
&\quad \downarrow \text{pop } [x := \text{in}; x := x + y, \{y \mapsto 2\}] \\
&\langle x := \text{in}; x := x + y, \{\text{in} \mapsto 5, y \mapsto 2\} \rangle \\
&\quad \downarrow 1 \\
&\langle x := x + y, \{x \mapsto 5, \text{in} \mapsto 5, y \mapsto 2\} \rangle \\
&\quad \downarrow 1 \\
&\langle \downarrow, \{x \mapsto 7, \text{in} \mapsto 5, y \mapsto 2\} \rangle \\
&\quad \downarrow \text{pop } Z_0 \\
&\langle \downarrow_0, \{x \mapsto 7, \text{in} \mapsto 5, y \mapsto 2\} \rangle
\end{aligned}$$

3.3.2 Recursion

In this section, we demonstrate how unbounded recursion is translated into a finite-state pPDA by constructing the operational semantics of the following program:

```

F() {
  { out := F() }
  [1/3]
  { out := 1 }
}
main() { x := F() }

```

$$\begin{aligned}
&\rightarrow \langle x := F(), \emptyset \rangle \\
&\quad \downarrow \text{push } [x := \text{in}, \emptyset] \\
&\quad \langle \{ \text{out} := F() \} [\frac{1}{3}] \{ \text{out} := 1 \}, \emptyset \rangle \\
&\quad \swarrow \text{push } [\text{out} := \text{in}, \emptyset] \quad \searrow \frac{2}{3} \\
&\quad \langle \text{out} := F(), \emptyset \rangle \quad \langle \text{out} := 1, \emptyset \rangle \\
&\quad \swarrow \text{pop } [\text{out} := \text{in}, \emptyset] \quad \downarrow 1 \\
&\quad \langle \text{out} := \text{in}, \{\text{in} \mapsto 1\} \rangle \quad \langle \downarrow, \{\text{out} \mapsto 1\} \rangle \\
&\quad \swarrow \text{pop } [\text{out} := \text{in}, \emptyset] \quad \downarrow \text{pop } [x := \text{in}, \emptyset] \\
&\quad \langle \downarrow, \{\text{out} \mapsto 1, \text{in} \mapsto 1\} \rangle \quad \langle x := \text{in}, \{\text{in} \mapsto 1\} \rangle \\
&\quad \quad \downarrow 1 \\
&\quad \quad \langle \downarrow, \{x \mapsto 1, \text{in} \mapsto 1\} \rangle \\
&\quad \quad \downarrow \text{pop } Z_0 \\
&\quad \quad \langle \downarrow_0, \{x \mapsto 1, \text{in} \mapsto 1\} \rangle
\end{aligned}$$

3.3.3 Conditioning

In this section, we show how conditioning is handled in the operational semantics. While we only demonstrate a single query block, nested conditioning is handled in exactly the same way. Consider the following program P :

```
F(int x) {
  observe(x=1);
  out := 2*x
}
main() {
  query {
    { x=1 } [0.5] { x=2 };
    x := F(x)
  }
}
```

The operational semantics of P are given in Figure 3.2. We write P' instead of $\{ x:=1 \} [0.5] \{ x:=2 \}; x := F(x)$. The automaton only depicts the transitions reachable in the final automaton. In particular, we exclude the states and transitions that are only used as conditions in the concatenation rule.

3.3.4 Converting a pPDA Into a CRPPL Program

So far, we have focused on converting programs from our language into pPDAs. The reverse is also possible, so our language (restricted to bounded variables) is exactly as expressive as pPDAs. In this section, we outline how one can transform an arbitrary pPDA into a CRPPL program.

Let \mathcal{A} be a pPDA with states q_1, \dots, q_n and stack symbols Z_1, \dots, Z_m . For every stack symbol Z_j , we create a function of the following form:

```
Z_j(int state){
  int pop = 0;
  while pop == 0 {
    if state == 1 {
      T1,j
    } else if state == 2 {
      T2,j
    } ...
    else if state == n {
      Tn,j
    }
  }
  out := state
}
```

Each $T_{i,j}$ simulates the transitions from state q_i with top-most stack symbol Z_j . These differ depending on the transition type:

For **internal transitions**, i.e. transitions of form $q_i \xrightarrow{p} q_{i'}$, we simply change the state variable with the given probability using a chain of probabilistic instructions. For example, if there are three outgoing transitions to q_2 , q_3 and q_4 with probabilities $\frac{1}{2}$, $\frac{1}{3}$ and $\frac{1}{6}$, the transition code has the following form:

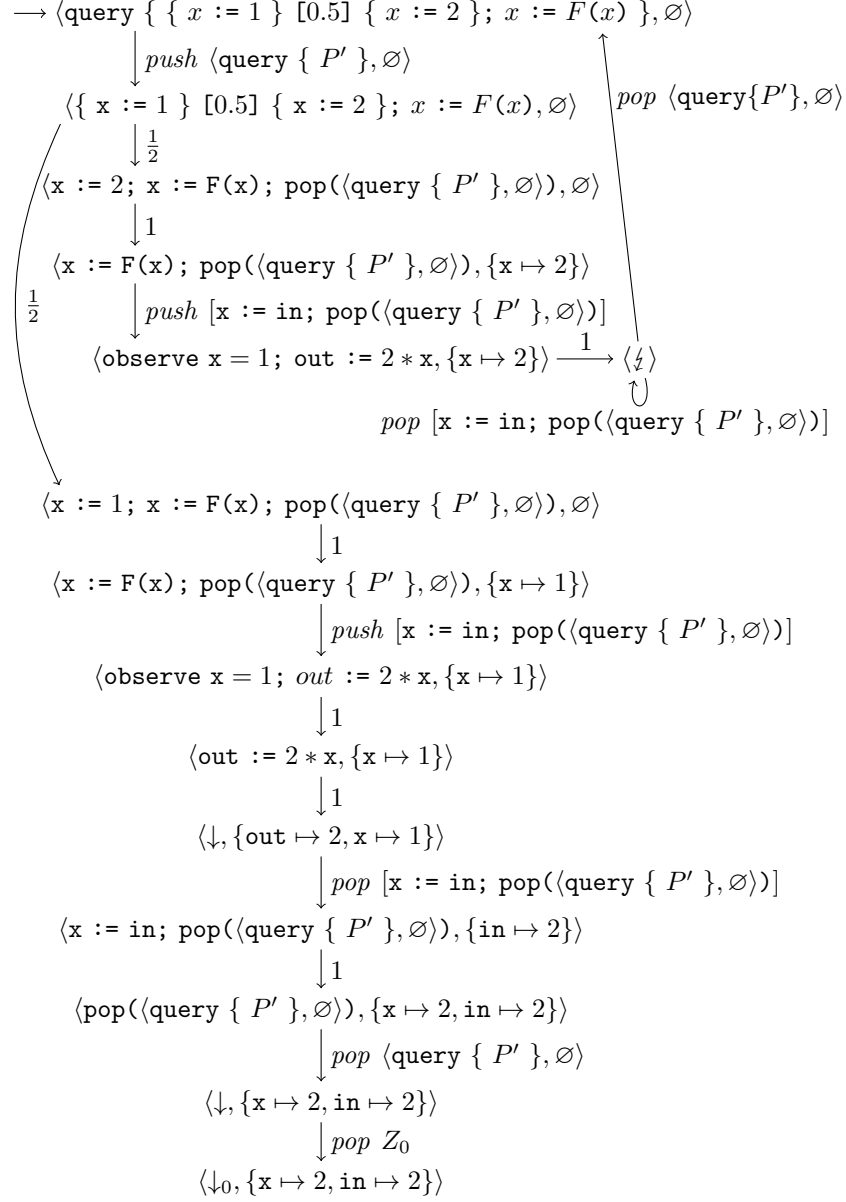


Figure 3.2: The operational semantics $Op(P, v)$ corresponding to program P from Section 3.3.3.

```
{ state = 2 } [1/2] { {state = 3} [2/3] {state = 4} }
```

For **push transitions**, i.e. transitions of form $q_i \xrightarrow{\text{push } Z_j} q_{i'}$, we call the function corresponding to Z_j with the new state $q_{i'}$. If the call terminates, we update the state accordingly. Each push state only has a single outgoing transition, so we do not have to handle other transitions:

```
state := Z_j'(i')
```

For **pop transitions**, we first determine the current top-most stack symbol. This depends only the current function and can thus be done at “compile”-time. There is a single pop transition $q_i \xrightarrow{\text{pop } Z_j} q_{i'}$ for each q_i and Z_j , so once again, we don’t have to handle multiple transitions. We update the current state and then set `pop` to 1, thereby exiting the loop and current function.

```
state := i';
pop := 1
```

We choose the function of the initial symbol as the main function of the program.

3.4 Illegal Programs

Recall the program from Section 2.3:

```
query {
  observe(false)
}
```

It is syntactically correct, but as all runs in the query block are blocked, it does not make sense to reason about its termination probability. We call such a program an *illegal program*. We can now formally give a definition of illegal programs using the operational semantics:

Definition 3.10 (Illegal Program). A program P is illegal if there is some state $\langle \text{query } \{ P' \}, v' \rangle$ that is reachable in $Op(P, v)$ with positive probability such that $\langle \perp \rangle$ is reached with probability 1 in $Op(P', v')$.

Example 3.11. The following program P is illegal.

```
F(x) {
  query { observe(x == 0) }
}
main() {
  F(0);
  F(1)
}
```

The state $\langle \text{query } \{ \text{observe } x == 0 \}, \{x \mapsto 1\} \rangle$ is reachable in $Op(P, \emptyset)$. In $Op(\text{observe } x == 0, \{x \mapsto 1\})$, \perp is reached with probability 1.

Example 3.12. The following program is an interesting edge case.

```

query {
  while true {
    {skip} [0.5] {observe(false)}
  }
}

```

There is a single run that never violates an observe statement: The (infinite) run that always chooses the left side in the probabilistic choice. The probability of this run is 0, so the probability of violating an observe statement and reaching $\langle \downarrow_{\perp} \rangle$ is 1. Therefore, the program is illegal, even though not every run violates an observe statement.

3.5 Limitations

The operational semantics produces sensible results for termination and reachability. At the same time, it is unsuitable for some other properties, such as expected runtime and pCTL model checking. This is because conditioning is implemented using “rejection sampling”. In this section, we present two examples to demonstrate the issues with these properties.

3.5.1 Expected Runtime

Consider the following program, where F is some function that takes n steps to execute. For simplicity, assume that F contains neither conditioning nor probabilistic behaviour.

```

main () {
  query {
    F();
    { skip } [0.5] { observe(false) }
  }
}

```

There is a single non-blocked run: We first enter the query block, then call F and then choose the left branch of the probabilistic choice instruction. The path that takes the right branch is blocked due to the observe statement. Therefore, the expected runtime is n (we neglect the runtime cost of `query`, `skip` and probabilist choice for simplicity).

However, our semantics produces a different expected runtime. With probability $\frac{1}{2}$, we take the left branch and therefore indeed get runtime n , but with probability $\frac{1}{2}$, we reach the blocking observe statement and restart from the query block. The total expected runtime is given by the following infinite arithmetico-geometric sequence:

$$\begin{aligned}
& \frac{1}{2} \cdot n + \frac{1}{2} \left(\frac{1}{2} \cdot 2n + \left(\frac{1}{2} \cdot 3n + \dots \right) \right) \\
&= \sum_{i \in \mathbb{N}} \left(\frac{1}{2} \right)^i \cdot i \cdot n \\
&= 2n.
\end{aligned}$$

3.5.2 Probabilistic Model Checking

We demonstrate the issue for probabilistic computation-tree logic (pCTL) [9] model checking with a similar example:

```
main () {
  query {
    label(a);
    { skip } [0.5] { observe(false) }
  }
}
```

Here, `label(a)` is a function that labels the current state with label a , which we then use to formulate our property. Consider the property

$$P_{=1}(\Box(a \implies P_{>0}(\Diamond a))),$$

which express that, whenever there is a state labelled with a , there is eventually another state labelled with a with probability greater than 0. This property should not hold in the program above, since the only non-blocked run only has a single state labelled with a . The probability of reaching another a from that state is 0.

However, due to the rejection sampling, the operational semantics does satisfy the property: When `label(a)` is reached, the `observe` statement will be violated with probability $\frac{1}{2}$, execution will restart from the query block, and thus `label(a)` is reached again.

3.5.3 Termination

The operational semantics also exhibit unintuitive termination behaviour. In some cases, it is obvious that conditioning should change termination: For example, if there is an `observe false` instruction before a diverging loop, the termination probability increases, and if such a statement blocks a terminating run, the termination probability might decrease.

However, nested conditioning can even change the termination probability in cases where the non-conditioned program almost-surely terminates. For example, consider a function F that does nothing with probability $\frac{1}{2}$ and recursively calls itself with probability $\frac{1}{2}$. This function almost-surely terminates. However, if one conditions on a sufficiently unlikely event in F , then rejection sampling will repeat the recursive call several times, until the condition is met. This can reduce the termination probability.

An example of this behaviour is given in Section 6.2.1.

Chapter 4

Weakest Preexpectation Semantics

In this section, we give denotational semantics using conditional weakest preexpectations. This extends the technique of expectation transformers [16] to conditioning and recursion. Our handling of conditioning is based on [17]. There, they separately keep track of the desired expectation and the probability that an observation was violated. We do this for every query block to support nested conditioning (whereas [17] implements global conditioning). As in [17], our approach works even for programs that do not terminate almost-surely. The handling of recursion is based on [15], where the semantics of recursion is modelled using fixed points. Nested scoping is handled by adding copies of each variable to the stack for each nested scope. Our approach extends this technique to multiple mutually-recursive functions.

There are alternative approaches to recursion. In [18], a similar approach to recursion is presented. The semantics is modelled as the supremum of bounded recursion of increasing depth (which is very similar to the fixed-point approach we take). More significantly, all variables are global, so functions do not have their separate scope.

In this chapter, we first introduce the theory underlying *wp* semantics. We then show how variable scoping is handled. After that, we define the *wp* transformer and show that it is continuous (and thus well-defined). Finally, we prove that the *wp* semantics and operational pPDA semantics correspond to each other with regard to (weighted) reachability probabilities.

4.1 Expectation Transformers

The expectation transformer semantics presented in this section operates on expectations, which are functions that assign a value to each state. Before we define these, we first need to specify the set of states that expectations are defined over.

In the operational semantics, a state consisted of a program (fragment) P , variable valuation v and the content of the stack. For the *wp* semantics, we take a slightly different approach: The state solely consists of a variable valuation. This valuation stores both the currently-in-scope variables and all other variables –

therefore removing the need for a stack. The remaining program (fragment) is not stored in the state, because the state is only used in computations where the remaining program is clear.

Definition 4.1 (Variable valuation). Let $Vars$ be the (countably infinite) set of all variable names. A *variable valuation* is a function $v: Vars \rightarrow \mathbb{R}$, where $v(\mathbf{x})$ is the current valuation of variable \mathbf{x} . We implicitly assume that $v(\mathbf{y}) = 0$ whenever the value of \mathbf{y} is not explicitly given. We usually omit these “uninteresting” variables when giving a variable valuation.

The set of all variable valuations is called V .

Definition 4.2 (Expectation). An *expectation* is a function $f: V \rightarrow \mathbb{R}_{\geq 0}^{\infty}$. The set of all expectations is called \mathbb{E} . We define the relation \sqsubseteq over \mathbb{E} such that $f \sqsubseteq g$ holds for $f, g \in \mathbb{E}$ if and only if $f(v) \leq g(v)$ holds for all $v \in V$.

A bounded expectation is a function $g: V \rightarrow \mathbb{R}_{\geq 0}^{\leq 1}$. The set of all bounded expectations is called $\mathbb{E}_{\leq 1}$. The relation \sqsubseteq is defined for bounded expectations analogously to expectations.

For any $x \in \mathbb{R}_{\geq 0}^{\infty}$, we write \mathbf{x} (in bold) to denote the constant expectation $\lambda v. x$. For Boolean expression b , we write $[b]$ to denote the expectation that is 1 for valuations that satisfy b and 0 for all others. For any $\circ \in \{+, -, \cdot, /\}$ and $f, g \in \mathbb{E}$, we write $f \circ g$ instead of $\lambda v. f(v) \circ g(v)$. Here, $x/0 = \infty$ for any $x \in \mathbb{R}_{\geq 0}^{\infty}$. For $x \in \mathbb{R}_{\geq 0}^{\infty}, f \in \mathbb{E}$, we write $x \circ f$ instead of $\lambda v. x \circ f(v)$ for any $\circ \in \{+, -, \cdot, /\}$.

We include ∞ in the image of f to ensure that any set of expectations has a supremum, which is important later.

The intuitive meaning of an expectation depends on the context. Usually, we identify interesting final states, assign a value to them in the expectation function (e.g. 1 for all interesting states and 0 for all other states). We call this the *post-expectation*, since it refers to final states. We are then interested in determining the expected value of this expectation *before* executing the program. In the above case, this would be the probability of reaching an interesting state. Since we can determine this expected value for every state, we get another expectation, which we call the *pre-expectation*.

In the above example, we have thus taken an expectation and transformed it into another expectation. We formalise this as an expectation transformer:

Definition 4.3 (Expectation Transformer). A (bounded) expectation transformer is a function $t: \mathbb{E} \rightarrow \mathbb{E}$ ($u: \mathbb{E}_{\leq 1} \rightarrow \mathbb{E}_{\leq 1}$). We define the relation \sqsubseteq over $\mathbb{E} \rightarrow \mathbb{E}$ (over $\mathbb{E}_{\leq 1} \rightarrow \mathbb{E}_{\leq 1}$) as follows. For (bounded) expectation transformers $t, t', t \sqsubseteq t'$ holds if $t(f) \sqsubseteq t'(f)$ holds for all $f \in \mathbb{E}$ ($f \in \mathbb{E}_{\leq 1}$).

Usually, we are interested in the transformer $wp[P]$ (**w**eakest **p**reexpectation) [16] for program P , which takes a post-expectation f and gives the expected value before executing program P . The bounded transformer $wlp[P]$ (**w**eakest **l**iberal **p**reexpectation) [16] gives us the expected value plus the probability of divergence. These transformers are usually defined inductively over the structure of the program.

Note that the transformers we have given as examples so far all depend on a program P . For each program, we get a different transformer. We later need to give the semantics of a function call as an expectation transformer. However, the semantics of a function call $x := F(\vec{e})$ does not just depend on the body of the function, it also depend on the k parameters $\vec{e} \in AE^k$ (for $k \in \mathbb{N}$ and

where AE is the set of arithmetic expressions, as defined in Definition 2.1) and the return variable $x \in Vars$. It is helpful to have one transformer that gives the complete semantics of a function call, so we introduce k -parametrised expectation transformers.

Definition 4.4 (k -Parametrised (bounded) expectation transformer). Let $k \in \mathbb{N}$. A k -parametrised expectation transformer $t: Vars \rightarrow AE^k \rightarrow (\mathbb{E} \rightarrow \mathbb{E})$ yields an expectation transformer $t(x)(\vec{e})$ for every $x \in Vars, \vec{e} \in AE^k$. We define the relation \sqsubseteq such that $t \sqsubseteq t'$ holds if and only if, for all $x \in Vars, \vec{e} \in AE^k$, $t(x)(\vec{e}) \sqsubseteq t'(x)(\vec{e})$ holds.

To capture the behaviour of all n functions of a program (where the functions have m_1, \dots, m_n parameters), we take an n -tuple of parametrised expectation transformers, where the i th transformer has m_i parameters. The ordering \sqsubseteq is extended component-wise, i.e. $(t_1, \dots, t_n) \sqsubseteq (t'_1, \dots, t'_n)$ if and only if $t_1 \sqsubseteq t'_1, \dots, t_n \sqsubseteq t'_n$.

k -parametrised bounded expectation transformers are defined similarly, except that \mathbb{E} is replaced by $\mathbb{E}_{\leq 1}$.

4.1.1 Domain Theory

To handle loops and recursion, our transformer makes use of fixed points. We need to ensure that these fixed points exist – otherwise, our transformer would not be well-defined. In this section, we first introduce a few prerequisites and then give a theorem that ensures the existence of fixed points.

Definition 4.5 (Complete Lattice). Let D be a set and \sqsubseteq a partial order relation over D . We call (D, \sqsubseteq) a complete lattice if every subset $C \subseteq D$ has a supremum in D .

Lemma 4.6. Let (D, \sqsubseteq) be a complete lattice. (D, \sqsubseteq) has a smallest element $\perp \in D$ and a greatest element $\top \in D$, i.e., $\perp \sqsubseteq x \sqsubseteq \top$ holds for all $x \in D$.

Proof. We choose $\perp = \sup \emptyset$. For every $x \in D$ with $\emptyset \sqsubseteq x$, we have $\sup \emptyset \sqsubseteq x$ by definition of the supremum. Since \emptyset is empty, $\emptyset \sqsubseteq x$ holds for all $x \in D$, and thus $\perp \sqsubseteq x$ holds for all $x \in D$.

We choose $\top = \sup D$. By definition of the supremum, we have $x \sqsubseteq \sup D = \top$ for all $x \in D$. As (D, \sqsubseteq) is a complete lattice and D is a subset of itself, $\sup D$ exists and is in D . \square

Lemma 4.7. Let (D, \sqsubseteq) be a complete lattice and X be a set. Then $(X \rightarrow D, \sqsubseteq')$ is a complete lattice with \sqsubseteq' defined as follows. For $f, g \in X \rightarrow D$, $f \sqsubseteq' g$ holds if and only if $f(x) \sqsubseteq g(x)$ for all $x \in X$.

Proof. Let $C \subseteq X \rightarrow D$. Then $\sup C = f \in X \rightarrow D$, where $f(x) = \sup_{g \in C} g(x)$ is well-defined because (D, \sqsubseteq) forms a complete lattice. It is an upper bound for C because $g(x) \sqsubseteq f(x)$ for any $g \in C, x \in X$. Assume it were not the least upper bound. Then there must be some upper bound f' with $f \not\sqsubseteq' f'$. This implies that there is some $x \in X$ such that $f'(x) \sqsubset f(x)$, but then $f'(x) \sqsubset \sup_{g \in C} g(x)$, so f' is no upper bound of C . \square

Lemma 4.8. Let $(D_1, \sqsubseteq_1), \dots, (D_n, \sqsubseteq_n)$ be complete lattices. Then the tuple $(D_1 \times \dots \times D_n, \sqsubseteq)$ forms a complete lattice with \sqsubseteq defined as follows. For

$(d_1, \dots, d_n), (e_1, \dots, e_n) \in (D_1 \times \dots \times D_n)$, we have $(d_1, \dots, d_n) \sqsubseteq (e_1, \dots, e_n)$ if and only if $d_1 \sqsubseteq_1 e_1, \dots, d_n \sqsubseteq_n e_n$.

Proof. Let $C \subseteq (D_1 \times \dots \times D_n)$. Then $\sup C = (d_1, \dots, d_n)$, where $d_i = \sup_{e \in C} e_i$ is well-defined because (D_i, \sqsubseteq_i) forms a complete lattice. It is an upper bound for C because $e_i \sqsubseteq_i d_i$ holds for all $e \in C, i \in \{1, \dots, n\}$. Assume it were not the least upper bound. Then there must be some upper bound (d'_1, \dots, d'_n) with $(d_1, \dots, d_n) \not\sqsubseteq (d'_1, \dots, d'_n)$. This implies that $d'_i \sqsubseteq_i d_i$ for some $i \in \{1, \dots, n\}$, but then $d'_i \sqsubseteq_i \sup_{e \in C} e_i$, so (d'_1, \dots, d'_n) is no upper bound of C . \square

Lemma 4.9. The following sets each form a complete lattice:

1. $(\mathbb{E}, \sqsubseteq)$,
2. $(\mathbb{E}_{\leq 1}, \sqsubseteq)$,
3. $(\mathbb{E} \rightarrow \mathbb{E}, \sqsubseteq)$,
4. $(\mathbb{E}_{\leq 1} \rightarrow \mathbb{E}_{\leq 1}, \sqsubseteq)$,
5. $(\text{Vars} \rightarrow AE^k \rightarrow (EP \rightarrow EP), \sqsubseteq)$ (for $EP \in \{\mathbb{E}, \mathbb{E}_{\leq 1}\}$) and
6. $(\text{Vars} \rightarrow AE^{m_1} \rightarrow (EP \rightarrow EP), \dots, \text{Vars} \rightarrow AE^{m_n} \rightarrow (EP \rightarrow EP), \sqsubseteq)$.

Proof. We write $x \sqsubset y$ instead $x \sqsubseteq y \wedge x \neq y$.

1. $(\mathbb{E}, \sqsubseteq)$: Let $C \subseteq \mathbb{E}$. Then $\sup C = f \in \mathbb{E}$ with $f(v) = \sup_{f_C \in C} f_C(v)$. f is well-defined because $(\mathbb{R}_{\geq 0}^{\infty}, \leq)$ forms a complete lattice. It is an upper bound for C because $f_C(v) \leq f(v)$ holds for any $f_C \in C, v \in V$. Assume it were not the least upper bound. Then there must be some upper bound f' with $f \not\sqsubseteq f'$. This implies that there is some $v \in V$ with $f'(v) < f(v)$, but then $f'(v) < \sup_{f_C \in C} f_C(v)$, so f' is no upper bound of C .
2. $(\mathbb{E}_{\leq 1}, \sqsubseteq)$: The proof is analogous to $(\mathbb{E}, \sqsubseteq)$.
3. $(\mathbb{E} \rightarrow \mathbb{E}, \sqsubseteq)$: Holds by Lemma 4.7, as $(\mathbb{E}, \sqsubseteq)$ is a complete lattice.
4. $(\mathbb{E}_{\leq 1} \rightarrow \mathbb{E}_{\leq 1}, \sqsubseteq)$: The proof is analogous to $(\mathbb{E} \rightarrow \mathbb{E}, \sqsubseteq)$.
5. $(\text{Vars} \rightarrow AE^k \rightarrow (EP \rightarrow EP), \sqsubseteq)$: Holds by Lemma 4.7. $(EP \rightarrow EP, \sqsubseteq)$ is a complete lattice. Applying Lemma 4.7 shows that $(AE^k \rightarrow (EP \rightarrow EP), \sqsubseteq)$ is a complete lattice (where \sqsubseteq is extended appropriately) and applying Lemma 4.7 again yields the desired result.
6. $(\text{Vars} \rightarrow AE^{m_1} \rightarrow (EP \rightarrow EP), \dots, \text{Vars} \rightarrow AE^{m_n} \rightarrow (EP \rightarrow EP), \sqsubseteq)$
The statement holds by Lemma 4.8.

\square

Definition 4.10 (Chains). Let (D, \sqsubseteq) be a complete lattice. A subset $C \subseteq D$ is a chain if $x \sqsubseteq y$ or $y \sqsubseteq x$ holds for every $x, y \in C$.

Definition 4.11 (Monotonic Functions). Let (D, \sqsubseteq) be a partial order. A function $f: D \rightarrow D$ is monotonic if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ for all $x, y \in D$.

Definition 4.12 (Continuous Functions). Let (D, \sqsubseteq) be a complete lattice. A function $f: D \rightarrow D$ is continuous if $\text{sup}(f(D)) = f(\text{sup } C)$ holds for any chain $C \subseteq D$.

Theorem 4.13. Every continuous function is monotonic.

Proof. Let (D, \sqsubseteq) be a complete lattice and let $f: D \rightarrow D$ be a continuous function. Assume $x \sqsubseteq y$ holds for $x, y \in D$. Then $\text{sup}\{f(x), f(y)\} = f(\text{sup}\{x, y\}) = f(y)$ and thus $f(x) \sqsubseteq f(y)$. Therefore, f is monotonic. \square

Theorem 4.14 (Kleene’s Fixed Point Theorem [13]). Let (D, \sqsubseteq) be a complete lattice and let $f: D \rightarrow D$ be a continuous function. Then the least fixed point $\text{lfp } f$ and greatest fixed point $\text{gfp } f$ exist and we have

$$\text{lfp } f = \sup_{n \in \mathbb{N}} f^n(\perp).$$

4.2 Scoping

One of the challenges our programming language poses is that each recursive function call creates a new variable scope with new local variables. For example, if a function F with local variable x calls itself, there are now two values for x : an inner and outer value. Only the inner value is accessible during the nested call, but we need to preserve the value of the outer x to restore it after the inner function terminates.

In the operational semantics, this is handled by storing the currently accessible variables in the state and all other variables on the stack. For the *wp* transformer, we choose a different approach, based on [15]. The basic idea is to store all variables in the state. To distinguish between different versions of a variable, we add the scoping symbol \boxtimes before variables if they are not currently in scope.

Example 4.15. Consider the following program:

```

F(x) { G(2) }
G(x) { H(3) }
H(x) { }

main() {
  F(1)
}

```

When execution first starts in the main method, no variables are in scope, so we have an empty variable assignment. When F is called, x comes into scope, and the assignment is $\{x \mapsto 1\}$. When G is called, we add \boxtimes to the “outer” x as it goes out of scope, and add the inner x : $\{\boxtimes x \mapsto 1, x \mapsto 2\}$. When H is called, we get: $\{\boxtimes \boxtimes x \mapsto 1, \boxtimes x \mapsto 2, x \mapsto 3\}$.

4.2.1 Increasing and Decreasing Scope

In [15], operators to increment the scope of variable assignments and expressions are provided. We repeat these definitions here and extend the operator to expectations.

Definition 4.16 (Scoping Operators [15]). For a variable assignment $v: \text{Vars} \rightarrow \mathbb{R}$, the scope increment operator $[+\boxplus]$ is defined as the variable assignment

$$v[+\boxplus] := \lambda x. \begin{cases} v(y) & \text{if } x = \boxplus y \\ 0 & \text{otherwise.} \end{cases}$$

The scope decrement operator $[-\boxminus]$ is defined as the variable assignment

$$v[-\boxminus] := \lambda x. v(\boxminus x).$$

For any expression e , the operators are defined as

$$\begin{aligned} e[+\boxplus] &:= \lambda v. e(v[+\boxplus]) && \text{and} \\ e[-\boxminus] &:= \lambda v. e(v[-\boxminus]). \end{aligned}$$

Let $E \in \{\mathbb{E}, \mathbb{E}_{\leq 1}\}$. For any (bounded) expectation $f \in E$, the operators $[+\boxplus]: E \rightarrow E$ and $[-\boxminus]: E \rightarrow E$ are defined as

$$\begin{aligned} f[+\boxplus] &:= \lambda v. \begin{cases} f(v') & \text{if } v = \boxplus v' \\ 0 & \text{otherwise} \end{cases} && \text{and} \\ f[-\boxminus] &:= \lambda v. f(v[-\boxminus]). \end{aligned}$$

Example 4.17. Let $v = \{x \mapsto 7, \boxplus x \mapsto 13, \boxplus y \mapsto 2\}$ be a variable assignment. Then $v[+\boxplus] = \{\boxplus x \mapsto 7, \boxplus \boxplus x \mapsto 13, \boxplus \boxplus y \mapsto 2\}$ and $v[-\boxminus] = \{x \mapsto 13, y \mapsto 2\}$ (with all other variables mapping to 0).

Let $e = x + \boxplus y$. Then $e[+\boxplus] = y$. This may seem counter-intuitive at first (because the $[+\boxplus]$ operator removes \boxplus symbols), but it follows the above definition. Consider variable assignment $v = \{x \mapsto 1, \boxplus x \mapsto 2, y \mapsto 10, \boxplus y \mapsto 20\}$. Then $e[+\boxplus](v) = e(v[+\boxplus]) = e(\boxplus x \mapsto 1, \boxplus \boxplus x \mapsto 2, \boxplus y \mapsto 10, \boxplus \boxplus y \mapsto 20) = 0 + 10 = 10$ (because x implicitly has value 0 in $v[+\boxplus]$). Conversely, $e[-\boxminus] = \boxplus x + \boxplus \boxplus y$.

In the next section, we will use the operators on expectations and require the result to be continuous. For this, the following result is helpful:

Lemma 4.18. For $E \in \{\mathbb{E}, \mathbb{E}_{\leq 1}\}$, the operators $[+\boxplus]: E \rightarrow E$ and $[-\boxminus]: E \rightarrow E$ are continuous, i.e., for any chain $S \subseteq \mathbb{E} \times \mathbb{E}_{\leq 1}$, we have

$$\begin{aligned} \left(\sup_{f \in C} f \right) [+\boxplus] &= \sup_{f \in C} (f[+\boxplus]) && \text{and} \\ \left(\sup_{f \in C} f \right) [-\boxminus] &= \sup_{f \in C} (f[-\boxminus]). \end{aligned}$$

Proof. We first show that $(\sup_{f \in C} f) [+ \square] = \sup_{f \in C} (f [+ \square])$.

$$\begin{aligned}
& \left(\sup_{f \in C} f \right) [+ \square] \\
&= \lambda v. \begin{cases} (\sup_{f \in C} f) (v') & \text{if } v = \square v' \\ 0 & \text{otherwise} \end{cases} \\
&= \lambda v. \begin{cases} \sup_{f \in C} (f(v')) & \text{if } v = \square v' \\ 0 & \text{otherwise} \end{cases} \quad (*) \\
&= \sup_{f \in C} \lambda v. \begin{cases} f(v') & \text{if } v = \square v' \\ 0 & \text{otherwise} \end{cases} \quad (\dagger) \\
&= \sup_{f \in C} (f [+ \square])
\end{aligned}$$

The equalities (*) and (†) hold because the relation \sqsubseteq is defined component-wise for expectations.

We now show that $(\sup_{f \in C} f) [- \square] = \sup_{f \in C} (f [- \square])$.

$$\begin{aligned}
& \left(\sup_{f \in C} f \right) [- \square] \\
&= \lambda v. \left(\sup_{f \in C} f \right) (v[- \square]) \\
&= \lambda v. \sup_{f \in C} f(v[- \square]) \quad (*) \\
&= \sup_{f \in C} \lambda v. f(v[- \square]) \quad (\dagger) \\
&= \sup_{f \in C} f[- \square]
\end{aligned}$$

The equalities (*) and (†) hold analogously to the proof for $[+ \square]$. □

4.3 The wp and wlp Transformer

In this section, we present the expectation transformer $wp: \mathbb{E} \rightarrow \mathbb{E}$ and bounded expectation transformer $wlp: \mathbb{E}_{\leq 1} \rightarrow \mathbb{E}_{\leq 1}$. The full rules are given in Figure 4.1 on Page 47 and additionally depend on the auxiliary functions Ψ and Ψ' given in Figure 4.2 on Page 48.

In general, wp tracks the expected pre-expectation and wlp tracks the expected pre-expectation plus the probability of divergence. Inside query blocks, wp and wlp additionally track observe violations. They track the unnormalised expectation, i.e. the expected value after reaching a terminal state *and* passing all observe statements, not the expected value after reaching a terminal state *under the condition* of passing all observe statements. This transformation is performed in the wp and wlp semantics of query blocks.

In the first part of this section, we present the definition of wp rule by rule with explanations and examples. We will first introduce simple, standard cases

(without conditioning, loops and recursion). We then show how conditioning is defined and finally present the rules for loops and recursion, both of which use fixed points.

4.3.1 Standard Cases

The cases presented in this section are taken from [17]¹.

Skip

In the case of the effectless program `skip`, the program state is not changed and the post-expectation thus is not modified. We set

$$wp[\text{skip}](f) = wlp[\text{skip}](f) = f.$$

Abort

In the case of the diverging program `abort`, no terminal state will ever be reached and we thus set the pre-expectation to constant 0. Since we diverge, we will never violate another observe statement and the probability of divergence is 1, so we set the liberal pre-expectation to constant 1.

$$\begin{aligned} wp[\text{abort}](f) &= \mathbf{0} \\ wlp[\text{abort}](f) &= \mathbf{1} \end{aligned}$$

Assignment

For the assignment, we apply the assignment to the post-expectation by setting

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

where

$$f[x/e] = \lambda v. f(v[x/v(e)]).$$

Here, $v[x/v(e)]$ is the variable mapping where x is mapped to $v(e)$ and all other variables are mapped to the same value as in v .

Example 4.19 (Assignment). Let $P = x := x + 1$ and let $f = [x > 2]$. Then $wp[P](f) = [x > 2][x/x+1] = [x+1 > 2] = [x > 1]$. If $g = \mathbf{1}$, then $wlp[P](g) = \mathbf{1}[x+2] = \mathbf{1}$.

Concatenation

Concatenation is defined inductively. For $P = P_1; P_2$, we first compute the pre-expectation of P_2 . This is then simultaneously the post-expectation of P_1 , so we then compute the pre-expectation of P_1 based on the post-expectation of P_1 .

$$\begin{aligned} wp[P_1; P_2](f) &= wp[P_1](wp[P_2](f)) \\ wlp[P_1; P_2](f) &= wlp[P_1](wlp[P_2](f)) \end{aligned}$$

¹In [17], an expectation *pair* transformer is defined. We instead define two separate transformers as this is more natural for nested conditioning, but our transformer is otherwise identical for these standard cases.

If-Else Block

The pre-expectation of the **if-else** block is the pre-expectation of the **if** branch if the condition holds and the pre-expectation of the **else** branch otherwise. We can write this in the same “expression-style” notation as the other definitions using the indicator function of the condition:

$$\begin{aligned} wp[\mathbf{if}\ b\ \{\ P_1\ }\ \mathbf{else}\ \{\ P_2\ }](f) &= [b] \cdot wp[P_1](f) + [\neg b] \cdot wp[P_2](f) \\ wlp[\mathbf{if}\ b\ \{\ P_1\ }\ \mathbf{else}\ \{\ P_2\ }](f) &= [b] \cdot wlp[P_1](f) + [\neg b] \cdot wlp[P_2](f) \end{aligned}$$

Probabilistic Choice

Probabilistic choice works similar to **if**, except that we multiply the two blocks by the probability p and $1 - p$ instead of the indicator function of the condition:

$$\begin{aligned} wp[\{\ P_1\ }\ [p]\ \{\ P_2\ }](f) &= p \cdot wp[P_1](f) + (1 - p) \cdot wp[P_2](f) \\ wlp[\{\ P_1\ }\ [p]\ \{\ P_2\ }](f) &= p \cdot wlp[P_1](f) + (1 - p) \cdot wlp[P_2](f) \end{aligned}$$

4.3.2 Observe and Query

We handle **observe** in the same way as [17]: If the condition of an **observe** statement holds, it behaves like a **skip** program, i.e. it does not change the expectation. If, on the other hand, the condition does not hold, the run is blocked and we will not reach any final state. Recall that we track the post-expectation if a final state is reached *and* the run is not blocked, not *under the condition* that the run is not blocked. Therefore, we set the value to 0 if the condition does not hold. We do this using the indicator function of the condition.

$$wp[\mathbf{observe}\ b](f) = wlp[\mathbf{observe}\ b](f) = [b] \cdot f$$

In [17], conditioning is global and there are no query blocks. Given a post-expectation f , they first compute the weakest pre-expectation $wp[P](f)$ in a similar way to our approach. They then compute the weakest *liberal* pre-expectation for post-expectation $\mathbf{1}$. In the standard case, $wlp[P](\mathbf{1}) = \mathbf{1}$ for all programs, but this no longer holds when **observe** is handled as above. Now, $wlp[P](\mathbf{1})(v)$ is the probability that no **observe** was violated in P with initial valuation v . They thus get the desired pre-expectation by dividing $wp[P](f)$ by $wlp[P](\mathbf{1})$.

Global conditioning is equivalent to having one query block that surrounds the entire program. Therefore, every query block is handled similarly to how [17] handles global conditioning. Given post-expectation f for some query block **query** $\{\ P'\ }$, we compute $wp[P'](f)$ and – to track **observe** violations – $wlp[P'](\mathbf{1})$. We then normalise the pre-expectation by the probability that no **observe** was violated and get

$$\begin{aligned} wp[\mathbf{query}\ \{\ P'\ }](f) &= \frac{wp[P'](f)}{wlp[P'](\mathbf{1})} && \text{and} \\ wlp[\mathbf{query}\ \{\ P'\ }](f) &= \frac{wlp[P'](f)}{wlp[P'](\mathbf{1})}. \end{aligned}$$

Example 4.20 (Conditioning). In this example, we look at a simple program with post-expectation x . The computation steps are given together with the program itself and should be read bottom-to-top (note that the handling of the probabilistic choice is slightly abbreviated):

```

//  $\frac{1}{4} \cdot 2 = 2$ 
query {
  // wp:  $\frac{1}{4}[2 < 3] \cdot 0 + \frac{3}{4}[5 < 3] \cdot 1 = \frac{1}{4} \cdot 2$ 
  // wlp:  $\frac{1}{4}[2 < 3] + \frac{3}{4}[5 < 3] = \frac{1}{4} \cdot 1$ 
  {x := 2} [0.25] {x := 5}
  // wp:  $[x < 3] \cdot x$ 
  // wlp:  $[x < 3]$ 
  observe (x < 3)
  // wp:  $x$ 
  // wlp:  $1$ 
}
//  $x$ 

```

This yields pre-expectation **2**, indicating that the expected value of x after termination is 2.

4.3.3 While

To compute the pre-expectation of a program `while b { P' }` with respect to post-expectation f , we have to take two cases into account: If the loop condition b does not hold, the loop behaves like `skip`, which we express as follows: $[\neg b] \cdot f$. If, on the other hand, b does hold, we compute the pre-expectation of the inner program P' . However, we cannot simply evaluate $wp [P'] (f')$, because the loop might run multiple times. The post-expectation of one iteration is the pre-expectation of the following iteration. We define a function that takes the pre-expectation f' of one iteration as input and returns the pre-expectation of the preceding iteration. Applying this function repeatedly simulates a bounded loop with increasingly large bound² and taking the fixed point of the function yields the semantics of the unbounded loop:

$$\begin{aligned}
 wp [\text{while } b \{ P' \}] (f) &= lfp \lambda f'. ([b] \cdot wp [P'] (f') + [\neg b] \cdot f) \\
 wlp [\text{while } b \{ P' \}] (f) &= gfp \lambda f'. ([b] \cdot wlp [P'] (f') + [\neg b] \cdot f)
 \end{aligned}$$

Note that wlp not only uses wlp for the loop body, but also takes the greatest fixed point instead of the least. This is to account for non-termination.

4.3.4 Function Calls

The rules for function calls are based on [15], but extended to mutual recursion of multiple functions. To explain the semantics, we first split a function call into multiple smaller and simpler instructions. The wp semantics of each of these operations are fairly simple and their concatenation then leads to the semantics of the entire function call. Consider a function call $x := F_i(\vec{e})$. To execute it, we have to perform several steps.

²This is later proven in Lemma 4.27 as part of the correspondence proof.

- First, we have to prepare a new variable scope. In the operational semantics, we handled this by pushing the current variable valuations onto the stack. Here, we instead use the scoping techniques presented in Section 4.2. We execute the “instruction” $+\boxplus$, which applies the scope increment operator $[\boxplus]$ to the current variable valuation (c.f. Definition 4.16).
- We now need to initialise the parameters x_1, \dots, x_{m_i} of the new function. Each x_i needs to be set to the value of e_i – but this is under the valuation before $[\boxplus]$ was applied to it. We thus evaluate $e_i[-\boxplus]$ to compensate for this.
- This concludes the preparations and we can now execute F_i . For this, we simply run $Code(F_i)$.
- After execution has resumed, we need to leave the current scope, i.e. restore the old variables. Before we do that, we need to pass the return value (stored in `out`) to the outer scope. For this, we execute the assignment $\boxminus x := \text{out}$.
- Finally, we remove the scope by executing the $-\boxminus$ instruction.

Putting all this together, the semantics of a function call $x := F_i(\vec{e})$ with respect to post-expectation $f \in \mathbb{E}$ is given by

$$wp[code(F_i)](f[-\boxminus][\boxminus x'/\text{out}]) [x_1/e'_1[-\boxminus]] \dots [x_{m_i}/e'_{m_i}[-\boxminus]] [\boxplus].$$

Note in particular the reversed order of the operations. We first apply the scoping operator $[-\boxminus]$ to the post-expectation, then set the return value and then compute the pre-expectation of $code(F_i)$ with respect to this. Finally, we perform the steps described above as “preparing the new variable scope”.

The above definition has one issue, however: It does not work for recursion. There might be recursive calls in $code(F_i)$, so computing the wp semantics would not be possible because the “unfolding” would never terminate.

For this, we use a work-around. We define a special auxiliary transformer $wp_{\varphi_1, \dots, \varphi_n}[P]$, which behaves like $wp[P]$, except for function calls: The semantics of a call $y := F_j(\vec{e}')$ are simply given by φ_j . We define $wlp_{\varphi_1, \dots, \varphi_n}[P]$ similarly. This construction allows us to specify any behaviour for (recursive) function calls. Note that, as φ_j needs to handle all possible call parameters and return variables, it has type $\varphi_j: Vars \rightarrow AE^{m_j} \rightarrow (\mathbb{E} \rightarrow \mathbb{E})$.

We now perform a fixed-point construction similar to the semantics of **while**. For this, we construct a function that takes the behaviour $\varphi_1, \dots, \varphi_n$ of the n functions of the program as input. It then constructs n new functions using the construction described above. For function F_i , this function has the form

$$\begin{aligned} & \lambda x'. \lambda(e'_1, \dots, e'_{m_i}). \lambda f. \\ & \quad wp_{\varphi_1, \dots, \varphi_n}[code(F_i)](\\ & \quad \quad f[-\boxminus][\boxminus x'/\text{out}] \\ & \quad \quad \quad [x_1/e'_1[-\boxminus]] \dots [x_{m_i}/e'_{m_i}[-\boxminus]] [\boxplus]. \end{aligned}$$

We combine n of these transformers in a function Ψ , which is defined in full in Figure 4.2 on Page 48 and then use it to define the wp semantics of a function call. For wlp , we define a similar function Ψ' and get

$$\begin{aligned} wp[x := F_i(\vec{e})](f) &= (lfp \Psi)_i(x)(\vec{e}) && \text{and} \\ wlp[x := F_i(\vec{e})](f) &= (gfp \Psi')_i(x)(\vec{e}). \end{aligned}$$

4.4 Continuity of wp and wlp

The definition of wp and wlp makes use of fixed points. To ensure that wp and wlp are well-defined, we use Kleene's Fixed Point Theorem (see Theorem 4.14), but this is only applicable to continuous functions. In this section, we thus show that wp and wlp are indeed continuous.

Theorem 4.21. For every program fragment P , the expectation transformer $wp[P]$ and bounded expectation transformer $wlp[P]$ are continuous over $(\mathbb{E}, \sqsubseteq)$ and $(\mathbb{E}_{\leq 1}, \sqsubseteq)$, respectively. Furthermore, $wp_{\varphi_1, \dots, \varphi_n}[P]$ and $wlp_{\varphi_1, \dots, \varphi_n}[P]$ are continuous over the same domains if $\varphi_1, \dots, \varphi_n$ are continuous.

Proof. Let $t \in \{wp, wlp, wp_{\varphi_1, \dots, \varphi_n}, wlp_{\varphi_1, \dots, \varphi_n}\}$, where $\varphi_1, \dots, \varphi_n$ are continuous. Let $E \in \{\mathbb{E}, \mathbb{E}_{\leq 1}\}$ be the domain of t and let $C \subseteq E$ be a chain. We show inductively that

$$t[P] \left(\sup_{f \in C} f \right) = \sup_{f \in C} t[P](f).$$

Except for the cases of query and function call, all cases have already been shown in [8].

Query

$$\begin{aligned} & wp[\text{query } \{ P' \}] \left(\sup_{f \in C} f \right) \\ &= \frac{wp[P'](\sup_{f \in C} f)}{wlp[P'](\mathbf{1})} \\ &= \frac{\sup_{f \in C} wp[P'](f)}{wlp[P'](\mathbf{1})} && \text{(Induction hypothesis)} \\ &= \sup_{f \in C} \frac{wp[P'](f)}{wlp[P'](\mathbf{1})} && \text{(Division by non-negative constant, see *)} \\ &= \sup_{f \in C} wp[\text{query } \{ P' \}](f) \end{aligned}$$

(*): $wlp[P'](\mathbf{1})$ does not depend on f . Therefore, for all $f_1, f_2 \in C$, $f \sqsubseteq f'$ holds if and only if $\frac{f_1}{wlp[P'](\mathbf{1})} \sqsubseteq \frac{f_2}{wlp[P'](\mathbf{1})}$ holds. The case of $wlp[P'](\mathbf{1}) = 0$ also maintains this property, as divisions by 0 are mapped to ∞ .

The proofs for wlp , $wp_{\varphi_1, \dots, \varphi_n}$ and $wlp_{\varphi_1, \dots, \varphi_n}$ are analogous.

| P | $wp [P] (f)$ |
|---|---|
| skip | f |
| abort | $\mathbf{0}$ |
| $x := e$ | $f[x/E]$ |
| $P_1 ; P_2$ | $wp [P_1] (wp [P_2] (f))$ |
| if $b \{ P_1 \}$ else $\{ P_2 \}$ | $[b] \cdot wp [P_1] (f) + [\neg b] \cdot wp [P_2] (f)$ |
| $\{ P_1 \} [p] \{ P_2 \}$ | $p \cdot wp [P_1] (f) + (1 - p) \cdot wp [P_2] (f)$ |
| while $b \{ P' \}$ | $(lfp \ \lambda f'. ([b] \cdot wp [P'] (f') + [\neg b] \cdot f))$ |
| $x := F_i(\vec{e})$ | $(lfp \ \Psi)_i(x)(\vec{e})(f)$ (see Figure 4.2) |
| observe b | $[b] \cdot f$ |
| query $\{ P' \}$ | $\frac{wp [P'] (f)}{wlp [P'] (\mathbf{1})}$ |

| P | $wlp [P] (f)$ |
|---|--|
| abort | $\mathbf{1}$ |
| $P_1 ; P_2$ | $wlp [P_1] (wlp [P_2] (f))$ |
| if $b \{ P_1 \}$ else $\{ P_2 \}$ | $[b] \cdot wlp [P_1] (f) + [\neg b] \cdot wlp [P_2] (f)$ |
| $\{ P_1 \} [p] \{ P_2 \}$ | $p \cdot wlp [P_1] (f) + (1 - p) \cdot wlp [P_2] (f)$ |
| while $b \{ P' \}$ | $(gfp \ \lambda f'. ([b] \cdot wlp [P'] (f') + [\neg b] \cdot f))$ |
| $x := F_i(\vec{e})$ | $(gfp \ \Psi)_i(x)(\vec{e})(f)$ (see Figure 4.2) |
| query $\{ P' \}$ | $\frac{wlp [P'] (f)}{wlp [P'] (\mathbf{1})}$ |

In all other cases, the definition is the same as for $wp [P] (f)$

| P | $wp_{\varphi_1, \dots, \varphi_n} [P] (f)$ |
|---------------------|--|
| $x := F_i(\vec{e})$ | $\varphi_i(x)(\vec{e})(f)$ |

In all other cases, the definition is the same as for $wp [P] (f)$

| P | $wlp_{\varphi_1, \dots, \varphi_n} [P] (f)$ |
|---------------------|---|
| $x := F_i(\vec{e})$ | $\varphi_i(x)(\vec{e})(f)$ |

In all other cases, the definition is the same as for $wlp [P] (f)$

Figure 4.1: Definition of expectation transformer $wp: \mathbb{E} \rightarrow \mathbb{E}$, bounded expectation transformer $wlp: \mathbb{E}_{\leq 1} \rightarrow \mathbb{E}_{\leq 1}$ and the associated auxiliary (bounded) expectation transformers.

Ψ is an n -tuple of m_i -parameterised expectation transformers (where $i \in \{1, \dots, n\}$) and is defined as follows:

$$\begin{aligned} \Psi(\varphi_1, \dots, \varphi_n) = & \\ & \left(\lambda x'. \lambda(e'_1, \dots, e'_{m_1}). \lambda f. \right. \\ & \quad wp_{\varphi_1, \dots, \varphi_n}[\text{code}(F_1)](\\ & \quad \quad f[-\Box][\Box x' / \text{out}] \\ & \quad \left. \right) [x_1/e'_1[-\Box]] \dots [x_{m_1}/e'_{m_1}[-\Box]][+\Box], \\ & \dots, \\ & \lambda x'. \lambda(e'_1, \dots, e'_{m_n}). \lambda f. \\ & \quad wp_{\varphi_1, \dots, \varphi_n}[\text{code}(F_n)](\\ & \quad \quad f[-\Box][\Box x' / \text{out}] \\ & \quad \left. \right) [x_1/e'_1[-\Box]] \dots [x_{m_n}/e'_{m_n}[-\Box]][+\Box], \\ & \left. \right). \end{aligned}$$

$\Psi'(\varphi_1, \dots, \varphi_n)$ is defined similarly, except that occurrences of $wp_{\varphi_1, \dots, \varphi_n}$ are replaced by $wlp_{\varphi_1, \dots, \varphi_n}$.

Figure 4.2: Definition of the functions Ψ and Ψ' that give the semantics of the functions of a given program.

Function call

The proof for function calls consists of two parts:

1. We show that Ψ is continuous. From this, it follows that $\text{lfp}\Psi$ exists.
2. We then show that $(\text{lfp}\Psi)_i(x)(\vec{e})$ is continuous. For this, we prove that $(\Psi(\perp)^n)_i(x)(\vec{e})$ is continuous for any $n \in \mathbb{N}$.

Lemma 4.22. The transformers $wp_{\varphi_1, \dots, \varphi_n}$ and $wlp_{\varphi_1, \dots, \varphi_n}$ are continuous with respect to their parameters $\varphi_1, \dots, \varphi_n$, i.e., for any chain $C \subseteq D$, we have

$$\begin{aligned} wp_{\text{sup } C}[P] &= \sup_{(\varphi_1, \dots, \varphi_n) \in C} wp_{\varphi_1, \dots, \varphi_n}[P] && \text{and} \\ wlp_{\text{sup } C}[P] &= \sup_{(\varphi_1, \dots, \varphi_n) \in C} wlp_{\varphi_1, \dots, \varphi_n}[P]. \end{aligned}$$

Proof. We show the statement by structural induction over P . All cases except for $P = x := F_i(\vec{e})$ are trivial, since they do not depend on the parameters $\varphi_1, \dots, \varphi_n$. We thus only present this remaining case. Note that – unlike for the wp transformer – this is not an induction step, but a base case. Let $f \in \mathbb{E}$.

$$\begin{aligned}
& wp_{\sup C}[x := F_i(\vec{e})](f) \\
&= \left(\sup_{(\varphi_1, \dots, \varphi_n) \in C} \varphi_1, \dots, \varphi_n \right)_i (x)(\vec{e})(f) \quad (*) \\
&= \left(\sup_{(\varphi_1, \dots, \varphi_n) \in C} \varphi_i \right) (x)(\vec{e})(f) \quad (\dagger) \\
&= \sup_{(\varphi_1, \dots, \varphi_n) \in C} (\varphi_i) (x)(\vec{e})(f) \\
&= \sup_{(\varphi_1, \dots, \varphi_n) \in C} wp_{\varphi_1, \dots, \varphi_n}[x := F_i(\vec{e})](f)
\end{aligned}$$

(*): This holds since \sqsubseteq is defined component-wise on $(Vars \rightarrow AE^{m_1} \rightarrow (\mathbb{E} \rightarrow \mathbb{E}), \dots, Vars \rightarrow AE^{m_n} \rightarrow (\mathbb{E} \rightarrow \mathbb{E}))$.

(†): This holds since \sqsubseteq is defined component-wise on $Vars \rightarrow AE^{m_i} \rightarrow (\mathbb{E} \rightarrow \mathbb{E})$.

The proof for wlp is similar. \square

Lemma 4.23. Ψ is continuous.

Proof.

$$\begin{aligned}
\Psi(\sup C) &= (\lambda x'. \lambda(e'_1, \dots, e'_{m_1}). \lambda f. \\
&\quad wp_{\sup C}[code(F_1)](f[-\square][\square x'/\text{out}]) \\
&\quad [x_1/e'_1[-\square]] \dots [x_{m_1}/e'_{m_1}[-\square]][+\square], \\
&\quad \dots, \\
&\quad \lambda x'. \lambda(e'_1, \dots, e'_{m_n}). \lambda f. \\
&\quad wp_{\sup C}[code(F_n)](f[-\square][\square x'/\text{out}]) \\
&\quad [x_1/e'_1[-\square]] \dots [x_{m_n}/e'_{m_n}[-\square]][+\square]) \\
&= (\lambda x'. \lambda(e'_1, \dots, e'_{m_1}). \lambda f. \quad (\text{Lemma 4.22}) \\
&\quad \sup_{(\varphi_1, \dots, \varphi_n) \in C} (wp_{\varphi_1, \dots, \varphi_n}[code(F_1)](f[-\square][\square x'/\text{out}])) \\
&\quad [x_1/e'_1[-\square]] \dots [x_{m_1}/e'_{m_1}[-\square]][+\square], \\
&\quad \dots, \\
&\quad \lambda x'. \lambda(e'_1, \dots, e'_{m_n}). \lambda f. \\
&\quad \sup_{(\varphi_1, \dots, \varphi_n) \in C} (wp_{\varphi_1, \dots, \varphi_n}[code(F_n)](f[-\square][\square x'/\text{out}])) \\
&\quad [x_1/e'_1[-\square]] \dots [x_{m_n}/e'_{m_n}[-\square]][+\square])
\end{aligned}$$

Continued on the next page.

$$\begin{aligned}
&= \left(\lambda x' . \lambda (e'_1, \dots, e'_{m_1}) . \lambda f . \right. \\
&\quad \sup_{(\varphi_1, \dots, \varphi_n) \in C} \left(wp_{\varphi_1, \dots, \varphi_n} [code(F_1)] (f[-\square][\square x' / \text{out}]) \right. \\
&\quad \quad [x_1/e'_1[-\square]] \dots [x_{m_1}/e'_{m_1}[-\square]] [+ \square], \\
&\quad \quad \dots, \\
&\quad \quad \lambda x' . \lambda (e'_1, \dots, e'_{m_n}) . \lambda f . \\
&\quad \quad \sup_{(\varphi_1, \dots, \varphi_n) \in C} \left(wp_{\varphi_1, \dots, \varphi_n} [code(F_n)] (f[-\square][\square x' / \text{out}]) \right. \\
&\quad \quad \quad \left. [x_1/e'_1[-\square]] \dots [x_{m_n}/e'_{m_n}[-\square]] [+ \square] \right) \\
&= \sup_{(\varphi_1, \dots, \varphi_n) \in C} \left(\lambda x' . \lambda (e'_1, \dots, e'_{m_1}) . \lambda f . \right. \\
&\quad \quad wp_{\varphi_1, \dots, \varphi_n} [code(F_1)] (f[-\square][\square x' / \text{out}]) \\
&\quad \quad [x_1/e'_1[-\square]] \dots [x_{m_1}/e'_{m_1}[-\square]] [+ \square], \\
&\quad \quad \dots, \\
&\quad \quad \lambda x' . \lambda (e'_1, \dots, e'_{m_n}) . \lambda f . \\
&\quad \quad wp_{\varphi_1, \dots, \varphi_n} [code(F_n)] (f[-\square][\square x' / \text{out}]) \\
&\quad \quad \quad \left. [x_1/e'_1[-\square]] \dots [x_{m_n}/e'_{m_n}[-\square]] [+ \square] \right) \\
&= \sup_{(\varphi_1, \dots, \varphi_n) \in C} \Psi(\varphi_1, \dots, \varphi_n)
\end{aligned}$$

□

Lemma 4.24. The expectation transformer $(lfp\Psi)_i(x)(\vec{e})$ is continuous.

Proof. For any $i \in \{1, \dots, n\}$, $x \in \text{Vars}$ and $\vec{e} \in \text{Exp}^{m_i}$, we show inductively over $k \in \mathbb{N}$ that the transformer $\Psi^k(\perp)_i(x)(\vec{e})$ is continuous. From this, it follows that $(lfp\Psi)_i(x)(\vec{e})$ is continuous as well.

Induction base: $(\Psi^0(\perp))_i(x)(\vec{e}) = \perp_i(x)(\vec{e}) = \perp$ is continuous, as all post-expectations are mapped to same pre-expectation \perp .

Induction step: Let $\Psi^k(\perp)_i(x)(\vec{e})$ be continuous. Then $\Psi^{k+1}(\perp)_i(x)(\vec{e})$ is

continuous as well.

$$\begin{aligned}
& \Psi^{k+1}(\perp)_i(x)(\vec{e}(\sup_{f \in C} f)) \\
&= wp_{\Psi^k(\perp)}[code(F_1)]((\sup_{f \in C} f)[- \square][\square x' / \text{out}]) \\
&\quad [x_1/e'_1[- \square]] \dots [x_{m_1}/e'_{m_1}[- \square]][+ \square] \\
&= wp_{\Psi^k(\perp)}[code(F_1)](\sup_{f \in C} f[- \square][\square x' / \text{out}]) \quad (\text{Lemma 4.18}) \\
&\quad [x_1/e'_1[- \square]] \dots [x_{m_1}/e'_{m_1}[- \square]][+ \square] \\
&= \sup_{f \in C} \left(wp_{\Psi^k(\perp)}[code(F_1)](f[- \square][\square x' / \text{out}]) \right) \quad (\text{Induction hypothesis}) \\
&\quad [x_1/e'_1[- \square]] \dots [x_{m_1}/e'_{m_1}[- \square]][+ \square] \\
&= \sup_{f \in C} wp_{\Psi^k(\perp)}[code(F_1)](f[- \square][\square x' / \text{out}]) \quad (\text{Lemma 4.18}) \\
&\quad [x_1/e'_1[- \square]] \dots [x_{m_1}/e'_{m_1}[- \square]][+ \square] \\
&= \sup_{f \in C} \Psi^{k+1}(\perp)_i(x)(\vec{e}(f))
\end{aligned}$$

□

The proof for the continuity of $wlp[x := F_i(\vec{e})]$ is similar.

For $wp_{\varphi_1, \dots, \varphi_n}$ and $wlp_{\varphi_1, \dots, \varphi_n}$, we have

$$\begin{aligned}
& wp_{\varphi_1, \dots, \varphi_n}[x := F_i(\vec{e})](f) = \varphi_i(f) \quad \text{and} \\
& wlp_{\varphi_1, \dots, \varphi_n}[x := F_i(\vec{e})](f) = \varphi_i(f),
\end{aligned}$$

and φ_i is continuous by assumption.

4.5 Correspondence Between pPDA and wp Semantics

In this section, we show that the wp semantics defined in the previous section is compatible with the operational semantics defined in Chapter 3.

For this, recall the notion of *weighted reachability* from Definition 3.5 and *liberal weighted reachability* from Definition 3.8. For a program P and initial valuation v , we assign a rational number to each terminal state in $Op(P, v)$ and compute the expected value of this number after reaching a terminal state (plus the probability of termination in the case of liberal weighted reachability).

Before we can show the equivalence of weighted reachability and wp semantics, we have to deal with one issue: The state $\langle \downarrow \perp \rangle$ in $Op(P, v)$ does not correspond to any variable valuation, but is a terminal state and thus is assigned a value in weighted reachability. Therefore, weighted reachability assignments do not quite match expectations. To mitigate this, we introduce the extension function $\ell: \mathbb{E} \rightarrow (\mathbb{E} \cup \langle \downarrow \perp \rangle)$ with

$$\ell(f) = \lambda v. \begin{cases} 0 & \text{if } v = \langle \downarrow \perp \rangle \\ f(v) & \text{otherwise} \end{cases},$$

which is extended to bounded expectations in the natural way.

Theorem 4.25. For any program fragment P , expectation $f \in \mathbb{E}$ and bounded expectation $g \in \mathbb{E}_{\leq 1}$, we have

$$\begin{aligned}\ell(wp[P](f)) &= \lambda v. W[P, v](\ell(f)) && \text{and} \\ \ell(wlp[P](g)) &= \lambda v. WL[P, v](\ell(g)).\end{aligned}$$

We show the statement by structural induction over the program's syntax.

4.5.1 Induction Base

Skip

If $P = \text{skip}$, then $Op(P, v)$ has the form

$$\langle \text{skip}, v \rangle \xrightarrow{1} \langle \downarrow, v \rangle \quad \langle \not\downarrow \perp \rangle$$

and we have

$$\begin{aligned}& \lambda v. W[P, v](\ell(f)) \\ &= \lambda v. \ell(f)(v) \\ &= \ell(f) \\ &= \ell(wp[P](f)).\end{aligned}$$

The proof for wlp is analogous.

Abort

If $P = \text{abort}$, then $Op(P, v)$ has the form

$$\langle \text{abort}, v \rangle \xrightarrow{1} \langle \downarrow, v \rangle \quad \langle \not\downarrow \perp \rangle$$

and we have

$$\begin{aligned}& \lambda v. W[P, v](\ell(f)) \\ &= \ell(\mathbf{0}) \\ &= \ell(wp[P](f)).\end{aligned}$$

For wlp , we have

$$\begin{aligned}& \lambda v. WL[P, v](\ell(g)) \\ &= \ell(\mathbf{1}) && (*) \\ &= \ell(wlp[P](g)).\end{aligned}$$

(*): By setting this to $\ell(\mathbf{1})$, we assign 1 to all states except for $\not\downarrow \perp$. This is because $\langle \not\downarrow \perp \rangle$ does not have a self loop (see above) and thus does not diverge. For the other states, the program diverges with probability 1, so the liberal weighted terminal reachability probability is 1.

Assignment

If $P = x := e$, then $Op(P, v)$ has the form

$$\langle x := e, v \rangle \xrightarrow{1} \langle \downarrow, v[x/e] \rangle \quad \langle \downarrow \perp \rangle$$

and we have

$$\begin{aligned} & \lambda v. W[P, v](\ell(f)) \\ &= \lambda v. \ell(f)(v[x/e]) \\ &= \ell(f)[x/e] \\ &= \ell(f[x/e]) \\ &= \ell(wp[P](f)). \end{aligned}$$

The proof for wlp is analogous.

Conditioning

For $P = \text{observe } b$, we first shown an auxiliary statement: For any $v \in V$, we have

$$\begin{aligned} W[P, v](\ell(f)) &= [v(b)] \cdot \ell(f)(v) && \text{and} \\ WL[P, v](\ell(f)) &= [v(b)] \cdot \ell(f)(v) \end{aligned}$$

We distinguish between two cases. If $v(b) = \text{true}$, then $Op(P, v)$ has the form

$$\langle \text{observe } b, v \rangle \xrightarrow{1} \langle \downarrow, v \rangle$$

and we have

$$\begin{aligned} & W[P, v](\ell(f)) \\ &= \ell(f)(v) \\ &= [v(b)] \cdot \ell(f). \quad (b \text{ is } \text{true} \text{ under } v, \text{ so } [v(b)] = 1) \end{aligned}$$

If $v(b) = \text{false}$, then $Op(P, v)$ has the form

$$\langle \text{observe } b, v \rangle \xrightarrow{1} \langle \downarrow \rangle \xrightarrow{\text{pop } Z_0} \langle \downarrow \perp \rangle$$

Because no symbols are pushed to the stack, the bottom stack symbol Z_0 is still the top of the stack when $\langle \downarrow \rangle$ is reached and the run continues to $\downarrow \perp$. We therefore have

$$\begin{aligned} & W[P, v](\ell(f)) \\ &= 0 \\ &= [v(b)] \cdot \ell(f). \quad (b \text{ is } \text{false} \text{ under } v, \text{ so } [v(b)] = 0) \end{aligned}$$

The proof for WL is analogous, since the probability of divergence is 0 in the programs shown above. We then show the main statement:

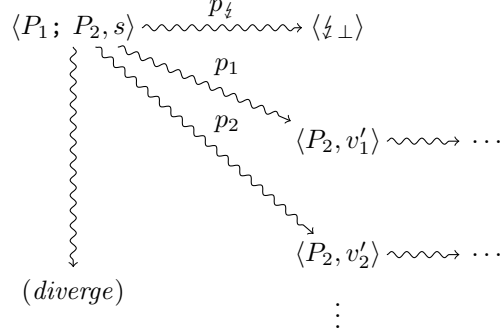
$$\begin{aligned} & \lambda v. W[P, v](\ell(f)) \\ &= \lambda v. v(b) \cdot \ell(f)(v) && \text{(Shown above)} \\ &= [b] \cdot \ell(f) \\ &= \ell([b] \cdot f) \\ &= \ell(wp[P](f)). \end{aligned}$$

The proof for wlp is analogous.

4.5.2 Induction Step

Concatenation

If $P = P_1; P_2$, then $Op(P, v)$ has the form



We first show why the automaton always has this form, i.e. why there is no transition to some $\langle P', v' \rangle$ with $P' \neq P_2$. By Theorem 3.3, any terminating run in $Op(P_1, v)$ either reaches $\langle \downarrow, \perp \rangle$ or $\langle \downarrow, v' \rangle$ with empty stack for some variable valuation v' . We have to show that this implies that any terminating run in $Op(P_1; P_2, v)$ either reaches $\langle \downarrow, \perp \rangle$ or $\langle P_2, v' \rangle$.

For this, consider the operational rules for concatenation. Internal concatenation and the query push and pop rules go to target states that contain program P_2 and are thus of no concern. It remains to show that P_2 is preserved during function calls. Recall the rule for function calls:

$$\frac{\langle P_1, v \rangle \xrightarrow{push [P, s]} \langle P'_1, v' \rangle}{\langle P_1; P_2, v \rangle \xrightarrow{push [P; P_2, v]} \langle P'_1, v' \rangle}$$

This rule removes P_2 from the destination state. However, by Theorem 3.3, $\langle \downarrow, v' \rangle$ is reached with empty stack. Therefore, the $[P; P_2, v]$ symbol has to be popped before reaching $\langle \downarrow, v' \rangle$. This happens only in the following rule:

$$\frac{v(\text{out}) = s}{\langle \downarrow, v \rangle \xrightarrow{pop [P, v']} \langle P, v'[\text{in}/s] \rangle}$$

After this transition is taken with $[P; P_2, v]$ on top of the stack, the state $\langle P; P_2, v' \rangle$ is reached for some variable valuation v' . Therefore, in every case, P_2 remains in the state.

For any expectation g , we have

$$\begin{aligned} W[P_1; P_2](g) &= \sum_{i \in \mathbb{N}} p_i \cdot W[P_2, v'_i](g) \\ &= W[P_1, v](\lambda v'. W[P_2, v](g)) \end{aligned}$$

and

$$\begin{aligned} WL[P_1; P_2](g) &= \sum_{i \in \mathbb{N}} p_i \cdot WL[P_2, v'_i](g) + \text{Probability of divergence} \\ &= WL[P_1, v](\lambda v'. WL[P_2, v](g)). \end{aligned}$$

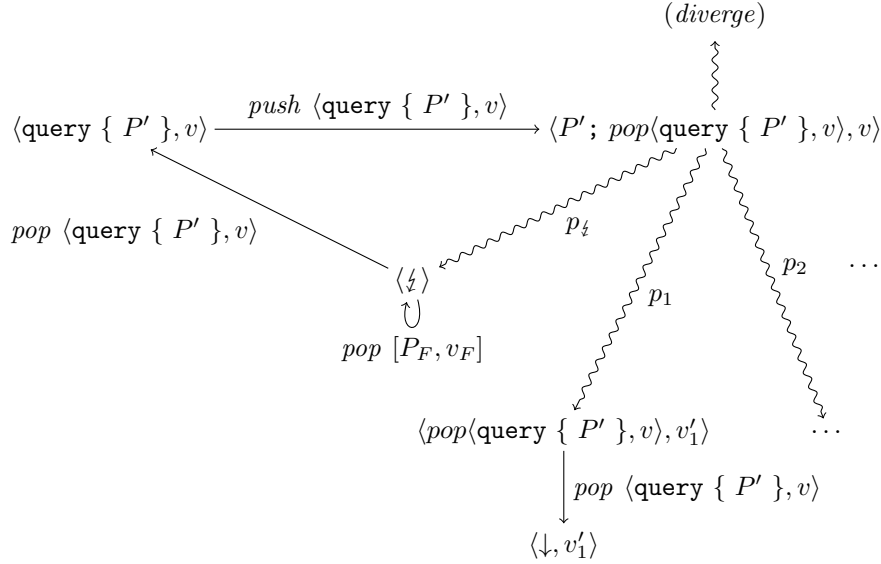
and we have

$$\begin{aligned}
& \lambda v. W [P, v] (\ell(f)) \\
&= \lambda v. p \cdot W [P_1, v] (\ell(f)) + (1 - p) \cdot W [P_2, v] (\ell(f)) \\
&= p \cdot \lambda v. W [P_1, v] (\ell(f)) + (1 - p) \cdot \lambda v. W [P_2, v] (\ell(f)) \\
&= p \cdot \ell(wp [P_1] (f)) + (1 - p) \cdot \ell(wp [P_2] (f)) \quad (\text{Induction hypothesis}) \\
&= \ell(p \cdot wp [P_1] (f) + (1 - p) \cdot wp [P_2] (f)) \\
&= \ell(wp [P] (f))
\end{aligned}$$

The proof for wlp is analogous.

Query

If $P = \text{query } \{ P' \}$, then $Op(P, v)$ has the form



The p_i only represents the observe violations that reach $\langle \downarrow \rangle$ in $Op(P', v)$. If P' contains nested query blocks then there may be additional transitions that go to $\langle \downarrow \rangle$, which are not depicted, since they have already been handled inductively in our proof. This leaves two cases when reaching $\langle \downarrow \rangle$ in $Op(\text{query } \{ P' \}, v)$:

1. If the topmost stack symbol is a function symbol $[P_F, v_F]$, it is popped without changing the state.
2. Otherwise, the topmost stack symbol is $\langle \text{query } \{ P' \}, v \rangle$. Nested query blocks may push other query symbols onto the stack, but it is not possible for some other query symbol $\langle P_Q, v_Q \rangle$ with $P_Q \neq P'$ or $v_Q \neq v$ to be the topmost stack symbol, because every query symbol is always popped before leaving the surrounding query block.

Note that $\langle \downarrow \rangle$ is unreachable. Additionally,

$$\begin{aligned}
& \lambda v. W [P'; \text{pop} \langle \text{query } \{ P' \}, v \rangle, v] (f) = \lambda v. W [P', v] (f) \quad \text{and} \\
& \lambda v. WL [P'; \text{pop} \langle \text{query } \{ P' \}, v \rangle, v] (f) = \lambda v. WL [P', v] (f)
\end{aligned}$$

holds, because the additional pop transition does not change the probability of reaching the terminal states.

Finite paths in $Op(\text{query } \{ P' \}, v)$ first take the $\langle \downarrow \rangle$ loop finitely often, with each loop corresponding to a run terminating at $\langle \downarrow \perp \rangle$ in $Op(P', v)$. After finitely many loops, each path then takes one of the other transitions from $\langle P' \rangle$; $pop(\text{query } \{ P' \}, v, v)$ towards some $\langle \downarrow, v'_i \rangle$. We use this to show the main statement:

$$\begin{aligned}
& \lambda v. W[\text{query } \{ P' \}, v](\ell(f)) \\
&= \lambda v. \sum_{k=0}^{\infty} W[P', v](\langle \downarrow \perp \rangle)^k \cdot W[P', v](f) && \text{(See above)} \\
&= \lambda v. \frac{1}{1 - W[P', v](\langle \downarrow \perp \rangle)} W[P', v](f) && \text{(Geometric series)} \\
&= \lambda v. \frac{1}{WL[P', v](\langle \neg \downarrow \perp \rangle)} W[P', v](f) && \text{(Def. of WL)} \\
&= \lambda v. \frac{W[P', v](f)}{WL[P', v](\langle \neg \downarrow \perp \rangle)} \\
&= \lambda v. \frac{W[P', v](f)}{WL[P', v](\ell(\mathbf{1}))} && \text{(Def. of } \ell) \\
&= \ell \left(\frac{wp[P'](f)}{wlp[P'](\mathbf{1})} \right) && \text{(Induction hypothesis)} \\
&= \ell(wp[\text{query } \{ P' \}](f))
\end{aligned}$$

The proof for wlp is analogous.

While

To show correspondence for while loops, we introduce the concept of n -bounded loop unrolling.

Definition 4.26. For $n \in \mathbb{N}$, the n -bounded loop unrolling $\text{while}_n b \{ P' \}$ is defined as $\text{if } b \{ P' \}; \text{while}_{n-1} b \{ P' \} \}$ else $\{ \text{skip} \}$ for $n > 0$ and as abort for $n = 0$.

Since this is purely syntactic sugar, the wp and operational semantics are already defined for n -bounded loop unrolling. We now show the equivalence in three steps, similar to [17]: First, we show that wp semantics of (unbounded) loops are equivalent to the supremum of the semantics of bounded loops. We then show that the wp semantics of bounded loops correspond to the operational semantics. Finally, we show that the operational semantics of (unbounded) loops are equivalent to the supremum of the semantics of bounded loops.

To simplify the proof, we introduce a helper function for the definition of the wp semantics of while loops. Recall that

$$wp[\text{while } b \{ P' \}](f) = lfp\lambda(f').([b] \cdot wp[P'](f') + [\neg b] \cdot f).$$

For any Boolean expression b , expectation $f \in \mathbb{E}$ and bounded expectation

$g \in \mathbb{E}_{\leq 1}$, we define $F_{P',b,f}: \mathbb{E} \rightarrow \mathbb{E}$ and $FL_{P',b,g}: \mathbb{E}_{\leq 1} \rightarrow \mathbb{E}_{\leq 1}$ with

$$\begin{aligned} F_{P',b,f}(f') &= [b] \cdot wp[P'](f') + [\neg b] \cdot f && \text{and} \\ FL_{P',b,g}(g') &= [b] \cdot wlp[P'](g') + [\neg b] \cdot g. \end{aligned}$$

We can then write

$$\begin{aligned} wp[\mathbf{while} \ b \ \{ P' \}](f) &= \text{lfp} F_{P',b,f} && \text{and} \\ wlp[\mathbf{while} \ b \ \{ P' \}](g) &= \text{gfp} FL_{P',b,g}. \end{aligned}$$

Lemma 4.27. For any $n \in \mathbb{N}$, we have

$$\begin{aligned} wp[\mathbf{while}_n \ b \ \{ P' \}](f) &= F_{P',b,f}^n(\perp) && \text{and} \\ wlp[\mathbf{while}_n \ b \ \{ P' \}](g) &= FL_{P',b,g}^n(\top) \end{aligned}$$

Proof. We show the equality by induction. For $n = 0$, we have

$$\begin{aligned} &wp[\mathbf{while}_0 \ b \ \{ P' \}](f) \\ &= wp[\mathbf{abort}](f) && \text{(Def. of } \mathbf{while}_n) \\ &= \mathbf{0} \\ &= F_{P',b,f}^0(\perp) \end{aligned}$$

and

$$\begin{aligned} &wlp[\mathbf{while}_0 \ b \ \{ P' \}](g) \\ &= wlp[\mathbf{abort}](g) && \text{(Def. of } \mathbf{while}_n) \\ &= \mathbf{1} \\ &= FL_{P',b,g}^0(\top). \end{aligned}$$

If the statement holds for some $n \in \mathbb{N}$, we have

$$\begin{aligned} &wp[\mathbf{while}_{n+1} \ b \ \{ P' \}](f) \\ &= wp[\mathbf{if} \ b \ \{ P' ; \mathbf{while}_n \ b \ \{ P' \} \} \ \mathbf{else} \ \{ \mathbf{skip} \}](f) \\ &= [b] \cdot wp[P' ; \mathbf{while}_n \ b \ \{ P' \}](f) + [\neg b] \cdot wp[\mathbf{skip}](f) \\ &= [b] \cdot wp[P'](wp[\mathbf{while}_n \ b \ \{ P' \}](f)) + [\neg b] \cdot f \\ &= [b] \cdot wp[P'](F_{P',b,f}^n(\perp)) + [\neg b] \cdot f && \text{(Induction Hypothesis)} \\ &= F_{P',b,f}^{n+1}(\perp). \end{aligned}$$

The proof for wlp is analogous. \square

Corollary 4.28. By Kleene's Fixed-Point Theorem (Theorem 4.14), we have

$$\begin{aligned} &wp[\mathbf{while} \ b \ \{ P' \}](f) \\ &= \sup_{n \in \mathbb{N}} F_{P',b,f}^n(\perp) \\ &= \sup_{n \in \mathbb{N}} wp[\mathbf{while}_n \ b \ \{ P' \}](f). \end{aligned}$$

and

$$\begin{aligned} &wlp[\mathbf{while} \ b \ \{ P' \}](g) \\ &= \inf_{n \in \mathbb{N}} FL_{P',b,g}^n(\top) \\ &= \inf_{n \in \mathbb{N}} wlp[\mathbf{while}_n \ b \ \{ P' \}](g). \end{aligned}$$

We can now show the correspondence between operational and *wp* semantics for *n*-bounded loop unrollings.

Lemma 4.29. For any $n \in \mathbb{N}$, we have

$$\begin{aligned} & \lambda v. W[\mathbf{while}_n b \{ P' \}, v](\ell(f)) \\ &= \ell(wp[\mathbf{while}_n b \{ P' \}](f)) \end{aligned}$$

and

$$\begin{aligned} & \lambda v. WL[\mathbf{while}_n b \{ P' \}, v](\ell(g)) \\ &= \ell(wlp[\mathbf{while}_n b \{ P' \}](g)). \end{aligned}$$

Proof. We show the statement by induction. For $n = 0$, $\mathbf{while}_0 b \{ P' \}$ corresponds to **abort**, for which the equality has already been shown. Assume the statement holds for $n \in \mathbb{N}$. We show that it also holds for $n + 1$.

$$\begin{aligned} & \lambda v. W[\mathbf{while}_{n+1} b \{ P' \}, v](\ell(f)) \\ &= \lambda v. W[\mathbf{if} b \{ P' \}; \mathbf{while}_n b \{ P' \} \} \mathbf{else} \{ \mathbf{skip} \}, v](\ell(f)) && \text{(Def. of } \mathbf{while}_n \text{)} \\ &= \ell(wp[\mathbf{if} b \{ P' \}; \mathbf{while}_n b \{ P' \} \} \mathbf{else} \{ \mathbf{skip} \}, v](f)) && \text{(Induction hypothesis, see below)} \\ &= \ell(wp[\mathbf{while}_{n+1} b \{ P' \}, v](f)) \end{aligned}$$

In step two, we use the fact that we've already shown the equivalence for $\mathbf{while}_n b \{ P' \}$ (in the previous induction step) and for **skip**, **if**, concatenation and P' (in the structural induction over the program).

The proof for *wlp* is analogous. \square

Lemma 4.30. For any v , we have

$$\begin{aligned} \sup_{n \in \mathbb{N}} W[\mathbf{while}_n b \{ P' \}, v](f) &= W[\mathbf{while} b \{ P' \}, v](f) && \text{and} \\ \inf_{n \in \mathbb{N}} WL[\mathbf{while}_n b \{ P' \}, v](g) &= WL[\mathbf{while} b \{ P' \}, v](g) \end{aligned}$$

Proof. We show the statement by showing two inequalities.

1. $\sup_{n \in \mathbb{N}} W[\mathbf{while}_n b \{ P' \}, v](f) \leq W[\mathbf{while} b \{ P' \}, v](f)$:

Consider any path in $Op(\mathbf{while}_n b \{ P' \}, v)$. There are three cases to consider: If the path is infinite, then it doesn't contribute to the weighted reachability probability. If it reaches $\mathbf{while}_0 b \{ P' \}$, it is aborted and thus also doesn't contribute to the weighted reachability probability. If the loop condition b is only satisfied for at most n iterations, then it reaches a terminal state and thus contributes to the weighted reachability probability. In the latter case, the same weighted reachability probability is observed in an unbounded loop.

2. $\sup_{n \in \mathbb{N}} W[\mathbf{while}_n b \{ P' \}, v](f) \geq W[\mathbf{while} b \{ P' \}, v](f)$:

Consider any path in $Op(\mathbf{while} b \{ P' \}, v)$. If the path is infinite, it doesn't contribute to the weighted reachability probability, so assume it

is finite and terminates at some state $\langle \downarrow, s' \rangle$. There must be a $k \in \mathbb{N}$ such that the loop condition b is negative after k executions of the loop body P' . Then an equivalent path that also terminates at $\langle \downarrow, v' \rangle$ also exists in $Op(\mathbf{while}_n b \{ P' \}, v)$ for all $n \geq k$.

The proof for WL is similar. \square

Corollary 4.31. We have

$$\begin{aligned} \lambda v. W[\mathbf{while} b \{ P' \}, v](\ell(f)) &= \ell(wp[\mathbf{while} b \{ P' \}](f)) && \text{and} \\ \lambda v. WL[\mathbf{while} b \{ P' \}, v](\ell(g)) &= \ell(wlp[\mathbf{while} b \{ P' \}](g)) \end{aligned}$$

Proof. Consequence of Corollary 4.28, Lemma 4.29 and Lemma 4.30. \square

Function calls

The proof for function calls is structured similarly to the one for *while* loops. We first introduce *n-bounded recursion*, where recursive calls with a depth greater than k are replaced by **abort**. Unlike in the *while* case, where k -bounded loop unrolling was a purely syntactic operation, we define new *wp* and operational semantics for k -bounded recursion.

We then first show that the limit of the *wp* and *wlp* semantics of k -bounded recursion is equal to the *wp* and *wlp* semantics of unbounded recursion. After that, we show that the *wp* (and *wlp*) and operational semantics of k -bounded recursion correspond. Finally, we show that the limit of (liberal) weighted reachability probabilities of k -bounded recursion corresponds to (liberal) weighted reachability probabilities in unbounded recursion.

Bounded recursion We introduce the new syntax element $x := F^{\leq k}(\vec{e})$, which denotes a k -bounded function call.

Definition 4.32. For expectation $f \in \mathbb{E}$ and bounded expectation $g \in \mathbb{E}_{\leq 1}$, the *wp* and *wlp* semantics of k -bounded recursion are given by

$$\begin{aligned} wp[x := F_i^{\leq k}(\vec{e})](f) &= (\Psi^k(\perp))_i(x)(\vec{e})(f) && \text{and} \\ wlp[x := F_i^{\leq k}(\vec{e})](g) &= (\Psi'^k(\top))_i(x)(\vec{e})(g). \end{aligned}$$

Lemma 4.33. For expectation $f \in \mathbb{E}$ and bounded expectation $g \in \mathbb{E}_{\leq 1}$, we have

$$\begin{aligned} \sup_{k \in \mathbb{N}} wp[x := F_i^{\leq k}(\vec{e})](f) &= wp[x := F_i(\vec{e})](f) && \text{and} \\ \inf_{k \in \mathbb{N}} wlp[x := F_i^{\leq k}(\vec{e})](g) &= wlp[x := F_i(\vec{e})](g). \end{aligned}$$

Proof.

$$\begin{aligned}
& \sup_{k \in \mathbb{N}} wp \left[x := F_i^{\leq k}(\vec{e}) \right] (f) \\
&= \sup_{k \in \mathbb{N}} \left(\Psi^k(\perp)_i(x)(\vec{e})(f) \right) \\
&= \left(\sup_{k \in \mathbb{N}} \Psi^k(\perp) \right)_i(x)(\vec{e})(f) && (\sqsubseteq \text{ is defined component-wise}) \\
&= (\text{lfp} \Psi^k(\perp))_i(x)(\vec{e})(f) && (\text{Kleene's Fixed Point Theorem}) \\
&= wp \left[x := F_i(\vec{e}) \right] (f)
\end{aligned}$$

The proof for *wlp* is analogous. \square

Definition 4.34. Our operational model for bounded recursion differs significantly from how unbounded recursion is handled in operational semantics. Instead of using the stack for recursion, we employ a similar technique to that presented in Section 4.2. The behaviour is mainly defined by the following rules:

$$\begin{array}{c}
\frac{}{\langle +\boxplus, v \rangle \xrightarrow{1} \langle \downarrow, v[\boxplus] \rangle} \\
\frac{}{\langle -\boxplus, v \rangle \xrightarrow{1} \langle \downarrow, v[-\boxplus] \rangle} \\
\frac{k = 0}{\langle x := F_i^{\leq k}(\vec{e}), v \rangle \xrightarrow{1} \langle \mathbf{abort}, v \rangle} \\
\frac{k > 0, s(\vec{e}) = \vec{v}}{\langle x := F_i^{\leq k}(\vec{e}), v \rangle \xrightarrow{1} \langle +\boxplus; a_1; \dots; a_{m_i}; \text{code}_{k-1}(F_i); \boxplus x := \mathbf{out}; -\boxplus, v \rangle}
\end{array}$$

where we write a_i for $x_i := e_i[-\boxplus]$ to improve readability. Moreover, $\text{code}_{k+1}(F_i)$ is defined similarly to $\text{code}(F_i)$, except that every function call $y := F_j(\vec{e}')$ is replaced by $y := F_j^{\leq k}(\vec{e}')$.

Lemma 4.35. For any $k \in \mathbb{N}$, $i \in \{1, \dots, n\}$ and $\vec{e} \in \text{Exp}^{m_i}$, we have

$$\begin{aligned}
\ell \left(wp \left[x := F_i^{\leq k}(\vec{e}) \right] (f) \right) &= \lambda s. W \left[x := F_i^{\leq k}(\vec{e}), v \right] (\ell(f)) && \text{and} \\
\ell \left(wlp \left[x := F_i^{\leq k}(\vec{e}) \right] (g) \right) &= \lambda v. WL \left[x := F_i^{\leq k}(\vec{e}), v \right] (\ell(g)).
\end{aligned}$$

Proof. We show this inductively over k .

Induction base: If $k = 0$, then $Op \left(x := F_i^{\leq k}(\vec{e}), v \right)$ has the form

$$\langle x := F_i^{\leq 0}(\vec{e}), v \rangle \xrightarrow{1} \langle \mathbf{abort}, v \rangle \quad \langle \downarrow, v \rangle \quad \langle \downarrow \perp \rangle$$

\uparrow
 1

and we have

$$\begin{aligned}
\ell(wp [x := F_i^{\leq 0}(\vec{e})] (f)) &= \ell((\Psi^0(\perp))_i(x)(\vec{e})(f)) \\
&= \ell(\perp(x)(\vec{e})(f)) \\
&= \ell(\mathbf{0}) \\
&= \lambda s. W [x := F_i^{\leq 0}(\vec{e}), v] (\ell(f))
\end{aligned}$$

and

$$\begin{aligned}
\ell(wlp [x := F_i^{\leq 0}(\vec{e})] (g)) &= \ell((\Psi^0(\top))_i(x)(\vec{e})(g)) \\
&= \ell(\top(x)(\vec{e})(g)) \\
&= \ell(\mathbf{1}) \\
&= \lambda v. WL [x := F_i^{\leq 0}(\vec{e}), v] (\ell(g)),
\end{aligned}$$

as the only path in $Op(x := F_i^{\leq 0}(\vec{e}), v)$ diverges.

Induction step: Recall that we write a_i instead of $x_i := e_i[-\Box]$ with a_i .

$$\begin{aligned}
& (\lambda s. W [x := F_i^{\leq k}(\vec{e}), s] (\ell(f)), \lambda v. WL [x := F_i^{\leq k}(\vec{e})] (\ell(g))) \\
&= \lambda v. W [+ \Box; a_1; \dots; a_{m_i}; code_{k-1}(F_i); \Box x := \mathbf{out}; - \Box, v] (\ell(f)) \\
&= \lambda v. W [+ \Box; a_1; \dots; a_{m_i}; code_{k-1}(F_i); \Box x := \mathbf{out}, v] (\ell(f)[- \Box]) \\
&= \lambda v. W [+ \Box; a_1; \dots; a_{m_i}; code_{k-1}(F_i), v] (\ell(f)[- \Box][\Box x/\mathbf{out}]) \\
&= \lambda v. W [+ \Box; a_1; \dots; a_{m_i}, v] (\lambda v'. W [code_{k-1}(F_i), v'] (\ell(f)[- \Box][\Box x/\mathbf{out}])) \\
&= \lambda v. W [+ \Box; a_1; \dots; a_{m_i}, v] (\ell(wp [code_{k-1}(F_i)] (f)[- \Box][\Box x/\mathbf{out}])) \quad (\text{I.H.}) \\
&= \lambda v. W [+ \Box; a_1; \dots; a_{m_{i-1}}, v] (\ell(f)[- \Box][\Box x/\mathbf{out}][x_{m_i}/e_{m_i}[- \Box]]) \\
&= \dots \quad (\text{One step for every } a_i) \\
&= \lambda v. W [+ \Box, v] (\ell(f)[- \Box][\Box x/\mathbf{out}][x_{m_i}/e_{m_i}[- \Box]] \dots [x_1/e_1[- \Box]]) \\
&= \lambda v. W [+ \Box, v] (\ell(f)[- \Box][\Box x/\mathbf{out}][x_1/e_1[- \Box]] \dots [x_{m_i}/e_{m_i}[- \Box]]) \\
& \quad (\text{All } a_i \text{ independent}) \\
&= \ell(f[- \Box][\Box x/\mathbf{out}][x_1/e_1[- \Box]] \dots [x_{m_i}/e_{m_i}[- \Box]][+ \Box]) \\
&= \ell(wp [code_{k-1}(F_i)] (f[- \Box][\Box x/\mathbf{out}])[x_1/e_1[- \Box]] \dots [x_{m_i}/e_{m_i}[- \Box]][+ \Box]) \\
&= \ell(wp_{\Psi^{k-1}(\perp)} [code(F_i)] (f[- \Box][\Box x/\mathbf{out}])[x_1/e_1[- \Box]] \dots [x_{m_i}/e_{m_i}[- \Box]][+ \Box]) \\
& \quad (*) \\
&= \ell(\Psi^k(\perp)_i(x)(\vec{e})(f)) \\
&= \ell(wp [x := F_i^{\leq k}(\vec{e})] (f))
\end{aligned}$$

(*): One can show inductively that $wp[code_k(F_i)] = wp_{\Psi^k(\perp)}[code(F_i)]$. For any k , the definitions of $wp[P]$ and $wp_{\Psi^k(\perp)}[P]$ only differ for function calls, and

similarly, $code_k(F_i)$ and $code_k(F_i)$ only differ for function calls. We therefore only have to look at function calls to show the equivalence. For $k = 0$, we have

$$\begin{aligned}
& wp[y := F_j^{\leq 0}(\vec{e}')](f) \\
&= wp[\mathbf{abort}](f) \\
&= \perp \\
&= wp_{\perp}[y := F_j(\vec{e}')](f) \\
&= wp_{\Psi^0(\perp)}[y := F_j^{\leq k}(\vec{e}')]
\end{aligned}$$

Now let $wp[code_k(F_i)] = wp_{\Psi^k(\perp)}[code(F_i)]$. We show that the equivalence also holds for $k + 1$. Once again, we only have to consider the function call case. In $code_{k+1}(F_i)$, every function call $y := F_j(\vec{e}')$ has been replaced by $y := F_j^{\leq k}(\vec{e}')$ and we have

$$\begin{aligned}
& wp[y := F_j^{\leq k}(\vec{e}')] (f) \\
&= \Psi^k(\perp)_i(y)(\vec{e}')(f) && \text{(Definition 4.32)} \\
&= wp_{\Psi^k(\perp)}[y := F_j(\vec{e}')] (f) && \text{(Definition of } wp_{\varphi_1, \dots, \varphi_n})
\end{aligned}$$

The proof for wlp is analogous. \square

Operational semantics

Lemma 4.36. For any $i \in \{1, \dots, n\}$ and $\vec{e} \in AE^{m_i}$, we have

$$\sup_{k \in \mathbb{N}} W[x := F_i^{\leq k}(\vec{e}), s] = W[x := f_i(\vec{e}), s]$$

Proof. We show two inequalities to prove the equality. We first show that, for any $k \in \mathbb{N}$,

$$W[x := F_i^{\leq k}(\vec{e}), s] \leq W[x := f_i(\vec{e}), s].$$

To show this, consider any non-diverging run in $Op(x := F_i^{\leq k}(\vec{e}))$ (we can ignore diverging runs as they don't contribute to weighted reachability probabilities). We can construct a unique run with equal probability in $Op(x := f_i(\vec{e}), v)$. We show this construction for function calls, since the operational semantics have not changed for all other language elements.

Assume the run reaches some $x := F_i^{\leq k}(\vec{e})$ with valuation v . By Definition 4.34, with probability 1, the next state in the run is

$$\langle +\boxplus; a_1; \dots; a_{m_i}; code_{k-1}(F_i); \boxplus x := \mathbf{out}; -\boxplus, v \rangle.$$

We do not have to consider the case of $k = 0$, since we excluded diverging runs. By concatenation rule, with probability 1, we then proceed to

$$\langle a_1; \dots; a_{m_i}; code_{k-1}(F_i); \boxplus x := \mathbf{out}; -\boxplus, v[+\boxplus] \rangle.$$

After executing the assignments a_i (which have the form $x := e_i[-\boxplus]$), with probability 1, we reach the state

$$\langle code_{k-1}(F_i); \boxplus x := \mathbf{out}; -\boxplus, v[+\boxplus] \cup \{x_i \mapsto s_i \mid i \in \{1, \dots, m_i\}\} \rangle,$$

where $s_i = v[+\boxplus](e_i[-\boxminus]) = v(e_i)$. We define $v_F := \{x_i \mapsto s_i \mid i \in \{1, \dots, m_i\}\}$. To construct the run in $Op(x := f_i(\vec{e}), v)$, we create the following transition:

$$\langle x := F_i(\vec{e}), v \rangle \xrightarrow{\text{push } [x := \text{in}, v]} \langle \text{code}(F_i), v_F \rangle$$

Note how the same variable valuation v_F occurs both in the bounded and unbounded path.

The bounded path now executes $\text{code}_{k-1}(F_i)$. As the path is terminating, it eventually reaches

$$\langle \boxminus x := \text{out}; -\boxminus, v[+\boxplus] \cup v'_F \rangle$$

with probability p , for some v'_F . In the run in $Op(x := f_i(\vec{e}), v)$, we now execute $\text{code}(F_i)$ with variable valuation v_F . While the bounded path had valuation $v[+\boxplus] \cup v_F$, code can only access variables from v_F (as all others have at least one \boxminus in the name). Furthermore, we never reach the recursion depth limit, as that would cause the run to diverge. Therefore, the run in $Op(x := f_i(\vec{e}), v)$ behaves the same as the bounded run and we eventually reach

$$\langle \downarrow, v'_F \rangle$$

with the same probability p .

The bounded run continues to

$$\langle -\boxminus, v[+\boxplus][\boxminus x/s_{out}] \cup v'_F \rangle,$$

where $s_{out} = v[+\boxplus][\boxminus \text{out}] = v(x)$. Finally, we reach

$$\langle \downarrow, (v[+\boxplus][\boxminus x/s_{out}] \cup v'_F)[-\boxminus] \rangle$$

and have

$$\begin{aligned} & (v[+\boxplus][\boxminus x/s_{out}] \cup v'_F)[-\boxminus] \\ &= v[+\boxplus][\boxminus x/s_{out}][-\boxminus] \quad (v'_F \text{ only contains variables without } \boxminus) \\ &= v[x/s_{out}]. \end{aligned}$$

The run in $Op(x := f_i(\vec{e}), v)$ is at $\langle \downarrow, v'_F \rangle$ and thus has to pop a symbol from the stack. The only such symbol is $[x := \text{in}, v]$, which we pushed when making the function call. We thus transition to

$$\langle x := \text{in}, v[\text{in}/s_{out}] \rangle,$$

where $s_{out} = v'_F(\text{out})$. From this, we reach

$$\langle \downarrow, v[\text{in}/s_{out}, x/s_{out}] \rangle.$$

Note that the valuation is the same as for the bounded path, with the exception of the auxiliary variable in , which is never accessed in normal code. We've thus shown how one can construct an equivalent path in $Op(x := f_i(\vec{e}), v)$.

From this, it follows that

$$\sup_{k \in \mathbb{N}} W[x := F_i^{\leq k}(\vec{e}), s] \leq W[x := f_i(\vec{e}), s].$$

We now show that

$$\sup_{k \in \mathbb{N}} W[x := F_i^{\leq k}(\vec{e}), s] \geq W[x := f_i(\vec{e}), s].$$

Consider any finite run in $Op(x := f_i(\vec{e}))$ (we can once again ignore diverging runs). Since this run is finite, it only contains bounded recursion, so there is a $k_0 \in \mathbb{N}$ such that this run also exists in $Op(x := F_i^{\leq k}(\vec{e}))$ for all $k \geq k_0$. The construction takes the same steps as above, just in the other direction. \square

By the principle of structural induction, we have thus proved the Correspondence Theorem 4.25.

Corollary 4.37. For any program P , initial valuation v and set of valuations $V' \subseteq V$, we have

$$wp [P] ([V']) (v) = R[P, v](V').$$

Proof. Consequence of Theorem 4.25 (Correspondence Theorem) and Theorem 3.6 (compatibility between R and W). \square

Chapter 5

Expressiveness

In the first part of the chapter, we present a construction that transforms a CRPPL program with conditioning into a CRPPL program without conditioning. From this, it immediately follows that our language is closed under conditioning. In the second part of the chapter, we compare this closure result to the results from [11]. We show that there are two different, incompatible notions of conditioning and present closure results for both of them in several model classes.

5.1 Transformation

In this section, we show that the query statement does not increase the expressiveness of our language, i.e. for any program P , there is a query-free program that has the same terminal reachability probabilities. The transformation in this section does not significantly increase the size of the program and is straightforward to construct (compared to e.g. [23], where conditioning can only be eliminated by completely rewriting the model).

To formally prove that CRPPL is closed under conditioning, it is sufficient to first construct the operational semantics $Op(P, v)$ for a given program P and initial state v and then apply the pPDA-to-CRPPL construction from Section 3.3.4. This produces a CRPPL program without any conditioning that is equivalent to P with respect to terminal reachability and termination probabilities. However, the transformation presented in this section preserves the structure of the program much better and the result is thus easier to understand.

The basic idea is to replace each query block with a rejection-sampling loop. The transformation is based on the method presented in [17]. In [17], a new variable `rerun` is set to *true* whenever an observe statement is violated. The entire program is surrounded by a loop that runs until `rerun` is *false*. Additionally, precautions are taken to ensure that the program never diverges after an observe statement has been violated. For this, they disable all loops if `rerun` is *true*.

To handle nested conditioning and recursion, we make the following changes to the method from [17]:

- Only the surrounding query block must be rerun instead of the entire

program. This also needs to hold across function boundaries, i.e., if F calls G from within a query block and an observe statement is violated in G , then the query block in F needs to be rerun.

- The program must not diverge by making recursive function calls. We will ensure this by blocking all function calls once `rerun` is true.

For the transformation, we use a function $t: Insts \rightarrow Insts$ that removes conditioning from a given instruction. This function is defined inductively over the program structure. We then apply t to each function body and add some additional instructions to each transformed function to obtain the final program. We first introduce the function t with explanations interspersed. The full definition is given in Figure 5.1.

5.1.1 De-Conditioning Function t

As mentioned above, the transformation uses a new variable `rerun` to track whether an observe statement has been violated. We will later add the declaration of `rerun` to the beginning of each function¹. When an observe condition is violated, we have to set `rerun` to true:

$$t(\text{observe } b) = \text{if } !b \{ \text{rerun} := \text{true} \} \text{ else } \{ \text{skip} \}$$

Before we enter a query block, we need to check whether `rerun` is already true. If yes, then we do not enter the query block. Otherwise, we would forget at which query block we need to restart the computation. Assuming `rerun` is false, we now need to run the body of the query block at least once and until `rerun` is false. We use a `do { ... } while` loop for this. We reset `rerun` to `false` at the beginning of each iteration and then apply t to the body of the query block:

$$t(\text{query } \{ P \}) = \text{if } !\text{rerun} \{ \\ \quad \text{do } \{ \\ \quad \quad \text{rerun} = \text{false}; \\ \quad \quad t(P) \\ \quad \} \text{ while } \text{rerun} \\ \}$$

Using a `do { ... } while` loop instead of a `while` loop is syntactic sugar. The formal definition in Figure 5.1 uses a `while` loop instead.

While these are the only two language elements involved in conditioning, we have to modify a few more elements to ensure the behaviour stays consistent. This ensures the program never diverges after `rerun` has been set to true. For loops, we achieve this by modifying the loop condition.

$$t(\text{while } b \{ P \}) = \text{while } !\text{rerun} \ \&\& \ b \{ t(P) \}$$

We do the same for function calls. However, there is a second complication we need to deal with in that case: If a function is called from within a query

¹We cannot declare it as part of t , because t is defined inductively, whereas `rerun` only should be defined once per function.

$$\begin{aligned}
t(\text{skip}) &= \text{skip} \\
t(\text{abort}) &= \text{abort} \\
t(x := e) &= x := e \\
t(\{ P_1 \} [p] \{ P_2 \}) &= \{ t(P_1) \} [p] \{ t(P_2) \} \\
t(x := F(\vec{e})) &= \text{if !rerun} \{ \\
&\quad x := F(\vec{e}) \\
&\quad \text{if } x = \frac{1}{2} \{ \text{rerun} := \text{true} \} \text{ else } \{ \text{skip} \} \\
&\quad \} \\
t(\text{if } b \{ P_1 \} \text{ else } \{ P_2 \}) &= \text{if } b \{ t(P_1) \} \text{ else } \{ t(P_2) \} \\
t(\text{while } b \{ P \}) &= \text{while !rerun \&\& } b \{ t(P) \} \\
t(P_1; P_2) &= t(P_1); t(P_2) \\
t(\text{observe } b) &= \text{if !}b \{ \text{rerun} := \text{true} \} \text{ else } \{ \text{skip} \} \\
t(\text{query } \{ P \}) &= \text{if !rerun} \{ \\
&\quad \text{rerun} = \text{true}; \\
&\quad \text{while rerun} \{ \\
&\quad \quad \text{rerun} = \text{false}; \\
&\quad \quad t(P) \\
&\quad \} \\
&\}
\end{aligned}$$

Figure 5.1: Full definition of transformer $t: Insts \rightarrow Insts$ that removes conditioning from an instruction.

block and an observe violation occurs in the inner function, we need to pass this information on to the outer function. For this, we will later add the following to the end of each function (see Section 5.1.2):

$$\text{if rerun} \{ \text{out} := \frac{1}{2} \} \text{ else } \{ \text{skip} \}$$

Here, $\frac{1}{2}$ indicates a special return value. In practice, one can encode this by adding 1 to all non-negative values of `out` and defining $\frac{1}{2} = 0$. We omit this here to improve clarity. We can now check for $\frac{1}{2}$ after a function call:

$$\begin{aligned}
t(x := F(\vec{e})) &= \text{if !rerun} \{ \\
&\quad x := F(\vec{e}) \\
&\quad \text{if } x = \frac{1}{2} \{ \text{rerun} := \text{true} \} \text{ else } \{ \text{skip} \} \\
&\quad \}
\end{aligned}$$

All other language elements are either unmodified or only apply the transformation recursively to their children. The full definition of t is given in Figure 5.1.

5.1.2 Function Body Transformation Function t'

With t , we can remove conditioning from an instruction. As mentioned above, we have to add the declaration of `rerun` to the beginning of each function body. Additionally, we need to check whether an observe condition was violated in the function and, if yes, return $\frac{1}{2}$. Both steps are achieved by function $t' : Insts \rightarrow Insts$, where

$$t'(P) = \text{rerun} = \text{false}; t(P); \text{ if rerun } \{ \text{out} := \frac{1}{2} \} \text{ else } \{ \text{skip} \}.$$

Given a program $P = (Func, Code, Params, main)$, we can use t' to construct a program $P' = (Func, Code', Params, main)$ with $Code'(F) = t'(Code(F))$, where P' preserves termination and terminal reachability probabilities of P .

5.2 Comparison to Other Model Classes

In this section, we define two different forms of conditioning, both of which are used in literature. Our goal is for the definitions to be applicable to a wide range of models (whereas our current approach only works for CRPPL programs with finite-domain variables, which are as expressive as pPDAs).

We then analyse several model classes to determine whether they are closed under the two forms of conditioning. This expands on the result from the previous section, where we presented a positive closure result for pPDAs, and on [11], which shows several (mostly negative) closure results for other model classes.

Throughout this section, we will focus on global conditioning, i.e. programs that do not have nested query blocks, as it is non-obvious how to generalise nested conditioning to the other form of conditioning.

5.2.1 Model Classes

We will consider four model classes in our analysis. In all cases, these are automaton-based models, as opposed to the programming-language approach from the previous chapters and the formal grammar-based approach from [11].

Definition 5.1 (Probabilistic Finite Automaton). A probabilistic finite automaton (pFA) is a tuple $(Q, \Sigma, \Delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\Delta \subseteq Q \times (0, 1] \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the transition relation (where $(0, 1] \subseteq \mathbb{Q}$),
- $q_0 \in Q$ is the initial state and
- $F \subseteq Q$ is the set of final states.

For $q \in Q \setminus F$, we require $\sum_{(q,p,a,q') \in \Delta} p = 1$ and for $q \in F$, there must be no $(q,p,a,q') \in \Delta$.

A run of $(Q, \Sigma, \Delta, q_0, F)$ starts in q_0 . From the current state q , a transition of the form $(q,p,a,q') \in \Delta$ is then chosen with probability p . The current state is set to q' and the symbol a is appended to the output. This continues until the current state is in F , which may occur with probability less than 1.

We diverge slightly from the standard definition, given e.g. in [19], where the probabilities for each state *and symbol* must add up to 1. In the standard definition, a pFA reads the current input symbol and then probabilistically chooses a transition based on the current state and that symbol. Each word is then accepted with a certain probability. For example, it is possible that multiple words are accepted with probability 1.

For our approach, it is more intuitive to think about *generating* words, not *reading* them. The pFA probabilistically chooses a transition based on the current state and then outputs the symbol of that transition. Every run thus either produces a word or diverges. It is therefore not possible to produce two words with probability 1 each, unlike in the standard definition. Our definition is similar to Markov chains, except that transitions are labelled with a symbol from Σ or with ε .

This “generative” approach is closer to the grammars from [11] and to our previous pPDA definition (which does not read any input or produce any output at all). It is also helpful that the probabilities of the produced words (plus the probability of divergence) form a probability distribution.

Definition 5.2 (Probabilistic Pushdown Automaton). A probabilistic pushdown automaton (pPDA) is a tuple $(Q, \Sigma, \Gamma, \Delta, q_0, Z_0, F)$, where

- Γ is a finite stack alphabet,
- $\Delta \subseteq Q \times \Gamma \times (0, 1] \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^{\leq 2}$ is the transition relation (where $(0, 1] \subseteq \mathbb{Q}$),
- $Z_0 \in \Gamma$ is the initial stack symbol,
- and the definition of the other components matches that of pFAs.

The initial stack symbol Z_0 must never be removed from the stack. A configuration has the form (q, γ) for $q \in Q, \gamma \in \Gamma$. A run starts in configuration (q_0, Z_0) . From configuration $(q, Z\gamma)$, a transition of the form (q, Z, p, a, q, α) is chosen with probability p . The new configuration is $(q', \alpha\gamma)$. This is repeated until a final state is reached.

A special case of pPDAs is the restriction to a single state. This is of interest as it is equivalent to probabilistic context-free grammars [1].

Definition 5.3 (Probabilistic Basic Process Algebra). A pPDA with one state is called a probabilistic basic process algebra (pBPA). A run terminates if the stack is empty.

Definition 5.4 (Probabilistic Turing Machine). A probabilistic Turing machine (pTM) is a tuple $(Q, \Sigma, \Gamma, \Delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\Gamma \supset \Sigma$ is a finite tape alphabet, with $\sqcup \in \Gamma, \sqcup \notin \Sigma$,
- $\Delta \subseteq Q \times \Gamma \times (0, 1] \times \Gamma \times \{L, R\} \times Q$ is the transition relation (where $(0, 1] \subseteq \mathbb{Q}$),

- $q_0 \in Q$ is the initial state and
- $F \subseteq Q$ is the set of final states.

The configuration of a TM consists of a state $q \in Q$, an infinite tape of symbols from Γ and the position of a read-write head on the tape. We start in q_0 and a tape filled with \perp . A transition (q, a, p, a', D, q') indicates that, if the current state is q and the symbol under the read-write head is a , then with probability p , the current symbol is replaced by a' , the new state is q' and the read-write head moves to the left if $D = L$ and to the right if $D = R$. This continues until the current state is in F . For each $q \in Q \setminus F$ and $a \in \Gamma$, the probabilities of all transitions (q, a, p, a', D, q') must add up to 1.

Note that this means that the pTM can change its output arbitrarily often. This is in contrast to the other model classes, where a symbol cannot be changed once it has been output.

5.2.2 Two Notions of Conditioning

In this section, we present two conflicting definitions of conditioning. In the approach we have taken so far, `observe` statements are placed in the program to block certain runs. As the conditioning is part of the program, itself we call this *algorithmic conditioning*. To generalise this to the different classes of automata, we use “observe-violation transitions” instead of observe statements.

Definition 5.5 (Algorithmic Conditioning). For algorithmic conditioning, a set of transitions $\Delta' \subseteq \Delta$ is designated as *observe violations*. Any run that contains one or more transitions from Δ' is blocked. All runs that were not blocked make up the final distribution, where the probability of the (non-blocked) runs is normalised so that the sum is 1.

Note that, unlike observe statements, observe transitions do not check a condition before blocking the paths. Instead, every path that takes an observe transition is blocked.

On the other hand, in [11], the conditioning is done separately by specifying which final configurations are acceptable. Intuitively, the model is first executed and, after that, one removes all configurations from the final distribution that are not contained in the set. Therefore, we call this method *post-conditioning*. We distinguish between two types of final configuration: words and states.

Definition 5.6 (Post-Conditioning Over Words). For post-conditioning over words, a set $W \subseteq \Sigma^*$ is chosen. The probability of producing the words in W must be positive, i.e. $\mathbb{P}(W) = \sum_{w \in W} \mathbb{P}(w) > 0$. The probability of a word $w \in W$ is given by $\frac{\mathbb{P}(w)}{\mathbb{P}(W)}$. For any $w \in \Sigma^* \setminus W$, the probability is 0.

Definition 5.7 (Post-Conditioning Over States). For post-conditioning over states, a set $F' \subseteq F$ is chosen. The probability of reaching a state in F' must be positive, i.e. $\mathbb{P}(F') = \sum_{q \in F'} \mathbb{P}(q) > 0$. The probability of reaching a terminal state $q \in F'$ is given by $\frac{\mathbb{P}(q)}{\mathbb{P}(F')}$. For any $w \in F \setminus F'$, the probability is 0.

Note that post-conditioning always excludes diverging runs. In fact, post-conditioning over Σ^* or F *conditions the program on termination*. This is the

key difference between the two notions of conditioning: The final distribution of a program with algorithmic conditioning can contain diverging runs, whereas the final distribution of a program with post-conditioning consists only of terminating runs. For some model classes, it is in fact not possible to block all diverging paths with algorithmic conditioning, since it is undecidable whether the current configuration will eventually terminate.

Therefore, in general, the sets of problems that can be modelled with algorithmic and post-conditioning are incomparable: There are distributions that can only be achieved with algorithmic conditioning (e.g. any distribution that includes non-termination with positive probability) and distributions that can only be modelled with post-conditioning (these only exist for model classes that are not closed under post-conditioning).

5.2.3 Conditioning Over Words or States

In the previous section, we distinguished between post-conditioning over words and post-conditioning over states. Algorithmic conditioning does not require this distinction, since it blocks runs based on the transitions taken, not based on the produced word or terminal state.

The distinction is nonetheless relevant for the closure results of algorithmic conditioning. The reason for this is similar to the issue discussed in Section 3.5, where we showed that the operational semantics preserve reachability probabilities (of terminal states), but do not preserve expected runtime and pCTL formulae.

If we applied a simple rejection-sampling technique to the models to handle algorithmic conditioning, we similarly would preserve the distribution over final states, but would not preserve the produced words (because some sections could be repeated during the rejection-sampling). For this reason, we distinguish between conditioning over words and states in the following analysis.

5.2.4 Closure Results From [11]

In this section, we present closure results regarding post conditioning (and the distribution over produced words). This section is a summary of the proofs from [11], unless otherwise indicated.

Probabilistic Turing Machines

To reason about the expressiveness of pTMs, [11] uses the notion of enumerable and computable distributions. In enumerable distributions, the probability of each event can be expressed as the limit of a monotonically increasing, enumerable sequence of rationals (that is, it is approximated from below). In computable distributions, we additionally require that the probability of each event can be approximated from above. It is easy to see that every computable distribution is also enumerable, but the reverse does not hold.

In [11], it is shown that pTMs define exactly the enumerable distributions, whereas pTMs that almost-surely terminate define exactly the computable distributions.

Since post-conditioning always conditions the final distribution on termination, one can use this fact to show that pTMs are not closed under condition-

ing: For this, [11] takes an enumerable, but not computable distribution and condition it on termination. If the distribution is chosen properly, then the conditioned distribution is still not computable, but has termination probability 1. Therefore, no pTM can express it.

[11] notes that the class of pTMs that almost-surely terminate is closed under conditioning. They refer to [6] where a rejection-sampling technique very similar to ours is described. In general, this result is not surprising, since algorithmic and post-conditioning are equivalent for models that almost-surely terminate, and we will see in the next section that pTMs are closed under algorithmic conditioning.

Probabilistic Pushdown Automata

In [11], pPDAs are not considered, since there is no simple grammar class equivalent to pPDAs.

Probabilistic Basic Process Algebras

The class of pBPAs is of interest as it corresponds to context-free grammars. To show that it is not closed with respect to post-conditioning, [11] first shows that, if a pBPA generates a finite-support distribution, the distribution is rational-valued. However, if one allows infinite-support distributions, then pBPAs can generate some (but not all) irrationally-valued distributions. Using these two facts, one can easily show that pBPAs are not closed with respect to conditioning: They first construct an infinite-support distribution with irrational values and then condition it in such a way that it is still irrationally-valued, but has finite support.

Probabilistic Finite Automata

For pFAs, the conditioning set has to be regular, i.e. representable by a (non-probabilistic) finite automaton. Without this restriction, pFAs are not closed under conditioning (take an automaton that produces each word $w \in \Sigma^*$ with positive probability, condition on a non-regular language and the conditioned distribution cannot be produced by any pFA, since it would have to produce a non-regular language).

The proof that pFAs are closed under post-conditioning with respect to regular sets uses an auxiliary transformation: For any pFA \mathcal{A} with final distribution \mathbb{P} that terminates with positive probability, there is a pFA \mathcal{A}' that has the final distribution \mathbb{P}' with $\mathbb{P}'(w) = \frac{P(w)}{\text{Probability that } \mathcal{A} \text{ terminates}}$. \mathcal{A} thus represents the automaton that is post-conditioned with respect to termination.

The proof continues by constructing the product automaton $\mathcal{A} \times \mathcal{B}$ of \mathcal{A} (the pFA to be conditioned) and a non-deterministic finite automaton \mathcal{B} representing the post-conditioning set. The product automaton can track whether the current word is in the post-conditioning set. If it is not, we create an infinite ε -loop, which will lead to divergence. The product automaton thus terminates exactly on the words that are accepted by \mathcal{B} , and the probability of each such word is the same as in \mathcal{A} . Applying the transformation from the previous paragraph to $\mathcal{A} \times \mathcal{B}$ yields the desired automaton.

5.2.5 Closure Under Post-Conditioning With States

In the previous section, we presented the proofs that pTMs and pBPAs are not closed under post-conditioning with words. A natural question is whether a positive closure result can be obtained for post-conditioning with states.

Probabilistic Turing Machines

We can reuse the counter-example from the proof that pTMs are not closed under post-conditioning with words (taken from [11]). This distribution produced a with probability p , aa with probability r and diverged with probability $1 - p - r$. Here, p is enumerable, but not computable, and r is rational. There is a pTM M that produces this distribution. To get the counter-example for post-conditioning with states, we construct a new TM M' that first executes M and then checks the tape content. If the word on the tape is a , the machine goes to a new terminal state q_a , otherwise, it goes to q_{aa} . We then condition on $F' = \{q_a, q_{aa}\}$. Now, the probability $\mathbb{P}(q_a \mid F') = \frac{p}{p+r}$ is not computably enumerable, as shown in [11]. Therefore, no pTM can have the same final distribution as M' conditioned with F' .

Probabilistic Pushdown Automata

Similar to Section 5.2.4, we did not analyse the closure of pPDAs with respect to post-conditioning with states.

Probabilistic Basic Process Algebras

For pBPAs, the notion of conditioning with states is not useful, as they only have a single state. Before conditioning, the probability of terminating in this state is not necessarily 1, as the pBPA might diverge, but after conditioning, it is always 1 (unless the pBPA almost-surely diverges, in which case the pre-requisites for post-conditioning are not fulfilled). Because there is a pBPA that terminates with probability 1, pBPAs are trivially closed under post-conditioning with states.

Probabilistic Finite Automata

In the previous section, we already showed that pFAs are closed under post-conditioning with words. We can use a very similar construction to show that they are closed under post-conditioning with states. Let $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ be a pFA and let $F' \subseteq F$ be the post-conditioning set. Then we set

$$\Delta' = \Delta \cup \{(q, 1, \varepsilon, q) \mid q \in F \setminus F'\}$$

and define a new pFA $\mathcal{A}' = (Q, \Sigma, \Delta', q_0, F')$. Then \mathcal{A}' terminates exactly on the states in F' . As we have not removed any transitions, the probability of any run that terminates at some $q \in F'$ remains the same. Conditioning \mathcal{A}' on termination yields the desired distribution and we have presented a construction for this in Section 5.2.4. Therefore, pFAs are closed under post-conditioning with states.

5.2.6 Closure Under Algorithmic Conditioning With States

In the first part of this chapter, we have shown that CRPPL programs (which are equivalent in expressiveness to pPDAs) are closed under algorithmic conditioning with states. In this section, we show that a similar rejection-sampling approach works for the other model classes discussed, with the exception of pBPAs.

Probabilistic Turing Machines

The class of pTMs is closed under algorithmic conditioning with states. Let $M = (Q, \Sigma, \Gamma, \Delta, q_0, F)$ be a pTM and let $\Delta' \subseteq \Delta$ be the set of observe violation transitions. We construct a pTM M' with the transition relation

$$(\Delta \setminus \Delta') \cup \{(q, a, p, a', D, q_{\ddagger}) \mid (q, a, p, a', D, q') \in \Delta'\} \cup \Delta_{\ddagger},$$

where q_{\ddagger} is a new state and Δ_{\ddagger} is a set of transitions that clear the tape² and then go to q_0 . With this construction, M' restarts execution whenever an observe violation is encountered.

Probabilistic Pushdown Automata

The class of pPDAs is closed under algorithmic conditioning with respect to the distribution over terminal states. One can first translate the pPDA into an equivalent CRPPL program (see Theorem 3.3.4 – the construction can easily be extended with `observe` statements to handle observe-violation transitions and with a global `query` block around the function of the initial symbol). The operational semantics then yields a pPDA with the desired terminal distribution.

Probabilistic Basic Process Algebras

Unlike for post-conditioning – where the single state of a pBPA is always reached with probability 1 after conditioning – algorithmic conditioning with states is not trivial for pBPAs. Algorithmic conditioning can change the termination probability and thus the probability to reach the single terminal state.

However, we cannot perform rejection sampling, as this would require a second state to reset the stack. There may be other techniques to construct such a pBPA, but we did not find one. Whether pBPAs are closed under algorithmic conditioning thus remains an open problem.

Probabilistic Finite Automata

The class of pFAs is closed under algorithmic conditioning with states. The technique is similar to pTMs, except that the step of clearing the tape can be

²Because M can write \sqcup , clearing the tape requires some extra book-keeping to find the ends of the tape. For this, we replace each $(q, a, p, \sqcup, D, q') \in \Delta$ with $(q, a, p, \sqcup', D, q')$ and create a new transition $(q, \sqcup', p, a, D, q')$ for every $(q, \sqcup, p, a, D, q') \in \Delta$. Clearing the tape can then be achieved by going right until the first \sqcup is read and then going left and writing \sqcup to every cell until another \sqcup is read.

omitted. Let $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ be a pFA and let $\Delta' \subseteq \Delta$ be the set of observe violation transitions. Then the automaton $\mathcal{A}' = (Q, \Sigma, \Delta_C, q_0, F)$ with

$$\Delta_C = (\Delta \setminus \Delta') \cup \{(q, p, a, q_0) \mid (q, p, a, q') \in \Delta'\}$$

has the same distribution over final states as \mathcal{A} conditioned on Δ' .

5.2.7 Closure Under Algorithmic Conditioning With Words

Except for pTMs, techniques based on rejection sampling do not work for algorithmic conditioning with words. This is because a symbol cannot be retracted once it has been output. In this section, we therefore investigate whether other techniques exist that can simulate algorithmic conditioning with words.

Probabilistic Turing Machines

As pTMs can change the content of the tape arbitrarily often, they do not suffer from the limitation discussed above. Therefore, the technique presented in Section 5.2.6 also works for algorithmic conditioning with words. The class of pTMs is therefore closed under algorithmic conditioning with words.

Probabilistic Finite Automata

One can change the transition probability of a pFA in such a way that observe-violation transitions are never reached. First, one computes the probability $\mathbb{P}_{t_i}(\neg \downarrow)$ that a transition $t_i = q \xrightarrow{p_i, a} q'$ will not reach any observe-violation transition (if the transition itself is an observe-violation transition, it is assigned probability 0). For this, one can solve a linear equation system, analogously to computing reachability probabilities in Markov chains. We then set the probability of t_i to $p'_i := p_i \cdot \mathbb{P}_{t_i}(\neg \downarrow)$. We remove all transitions where $p'_i = 0$ and all states that have no outgoing transitions (it's easy to see that states with no outgoing transitions also have no incoming transitions). Finally, we normalise the transition probabilities so that, for each state, the probabilities of the outgoing transitions have a sum of 1.

Proof. Let \mathcal{A} be a pFA and let \mathcal{A}' be the pFA that results from \mathcal{A} with the above transformation. In this proof, we only consider the finite runs of \mathcal{A} and \mathcal{A}' , as infinite runs do not contribute to the final distribution. One can show that there is a bijection between the finite runs of \mathcal{A}' that weren't blocked and the finite runs of \mathcal{A} :

We map every finite run π of \mathcal{A} to a run π' of \mathcal{A}' that visits the same states and produces the same word. Such a run always exists: Every finite run of \mathcal{A} without an observe violation still has positive probability in \mathcal{A}' as only transitions that almost-surely reach an observe-violation transition are assigned probability 0. The function is a bijection: \mathcal{A}' does not contain any runs that \mathcal{A} does not. Every run in \mathcal{A} is either mapped to its corresponding run in \mathcal{A}' or blocked. \mathcal{A}' does not contain any blocked runs: Every finite run of \mathcal{A} with an observe violation is removed because observe-violation transitions are assigned probability 0 in the construction.

It remains to show that the probabilities of finite runs are preserved in \mathcal{A}' , except for the normalisation of the final distribution. Let

$$\pi = q_1 \xrightarrow{p_1, a_1} q_2 \xrightarrow{p_2, a_2} \dots \xrightarrow{p_{n-1}, a_{n-1}} q_n$$

be a finite run in \mathcal{A} and let

$$\pi' = q_1 \xrightarrow{p'_1, a_1} q_2 \xrightarrow{p'_2, a_2} \dots \xrightarrow{p'_{n-1}, a_{n-1}} q_n$$

be the corresponding run in \mathcal{A}' . We write $\mathbb{P}_{q_i}(\neg \zeta)$ to denote the probability that no observe-violation transition is reached from state q_i in \mathcal{A} . We then have to show that

$$\mathbb{P}(\pi') = \frac{\mathbb{P}(\pi)}{\mathbb{P}_{q_1}(\neg \zeta)}.$$

To prove this, we analyse the suffixes of π and π' . We write π_i for the suffix of π starting in q_i and define π'_i similarly. We then show that

$$\mathbb{P}(\pi'_i) = \frac{\mathbb{P}(\pi_i)}{\mathbb{P}_{q_i}(\neg \zeta)}$$

holds for all $i \in \{1, \dots, n\}$ by induction over i , starting with $i = n$ and decrementing it until we reach $i = 1$.

Induction Base: The base case of $i = n$ is trivial as all runs with length 1 have probability 1.

Induction Step: Assume the statement holds for some $i + 1 > 1$. We show that it also holds for i . Note that for every run from q_i , one of the following four cases holds:

- The run can visit q_{i+1} next and later take an observe-violation transition. We denote the probability of this with r_1 .
- The run can visit q_{i+1} next and never take an observe-violation transition. We denote the probability of this with r_2 .
- The run can visit a state different from q_{i+1} next and later take an observe-violation transition. We denote the probability of this with r_3 .
- The run can visit a state different from q_{i+1} next and never take an observe-violation transition. We denote the probability of this with r_4 .

We now have

$$\begin{aligned} \mathbb{P}_{q_i}(\neg \zeta) &= r_2 + r_4 \\ \mathbb{P}_{q_{i+1}}(\neg \zeta) &= \frac{r_2}{r_1 + r_2} \\ p_i &= r_1 + r_2 \\ p_{i'} &= \frac{r_2}{r_2 + r_4}. \end{aligned}$$

| | PC-S | PC-W | AC-S | AC-W |
|-------|------|------|------|------|
| pTMs | ✗ | ✗ | ✓ | ✓ |
| pPDAs | ? | ? | ✓ | ? |
| pBPAs | (✓) | ✗ | ? | ? |
| pFAs | ✓ | ✓* | ✓ | ✓ |

Figure 5.2: Closure results of algorithmic conditioning (AC) and post-conditioning (PC) with distribution over words (-W) and terminal states (-S). (*): pFAs are only closed under post-conditioning with words when the conditioning set is regular.

With this, we can show that

$$\begin{aligned}
& \mathbb{P}(\pi'_i) \\
= & p'_i \cdot \mathbb{P}(\pi'_{i+1}) \\
= & p'_i \cdot \frac{\mathbb{P}(\pi_{i+1})}{\mathbb{P}_{q_{i+1}}(\neg \zeta)} && \text{(Induction Hypothesis)} \\
= & \frac{r_2}{r_2 + r_4} \cdot \frac{\mathbb{P}(\pi_{i+1})}{\frac{r_2}{r_1 + r_2}} \\
= & \frac{r_2}{r_2 + r_4} \cdot \frac{(r_1 + r_2) \cdot \mathbb{P}(\pi_{i+1})}{r_2} \\
= & \frac{(r_1 + r_2) \cdot \mathbb{P}(\pi_{i+1})}{r_2 + r_4} \\
= & \frac{p_i \cdot \mathbb{P}(\pi_{i+1})}{\mathbb{P}_{q_i}(\neg \zeta)} \\
= & \frac{\mathbb{P}(\pi_i)}{\mathbb{P}_{q_i}(\neg \zeta)}.
\end{aligned}$$

Therefore, every non-blocked run of \mathcal{A} corresponds to a run of \mathcal{A}' that produces the same word with the same probability, except for the normalisation of the final distribution. \square

Other Classes

For pPDAs and pBPAs, we were neither able to find a construction to show closure nor able to prove that they are not closed under algorithmic conditioning with words.

5.2.8 Summary and Discussion

In this section, we have presented closure results for algorithmic and post-conditioning and for distributions over words and terminal states. The results are summarised in Figure 5.2. For post-conditioning, the type of distribution does not affect closure, with the exception of pBPAs (where closure is trivial as the single state of a pBPA is always reached with probability 1 after post-conditioning).

We did not find a model class that gained expressiveness with algorithmic conditioning, regardless of the type of distribution. There are, however, several open problems for algorithmic conditioning.

Which type of conditioning is more suitable depends on the application. Algorithmic conditioning closely models conditioning in probabilistic programming languages. When writing complex programs, it is usually easier to express conditioning as part of the language, as compared to a separate query language. This holds in particular for nested conditioning. Nested post-conditioning could be accomplished by specifying a different acceptance set for each query block (this set could even depend on the variables in scope outside of the query block).

For other modelling formalisms, i.e. probabilistic automata and grammars, post-conditioning is likely more intuitive. As these models are rarely written by hand and usually generated, e.g. from regular expressions or logic formulae, it is unnatural to then manually mark some transitions as observe-violation transitions. On the other hand, the post-conditioning set can be specified separately.

Algorithmic conditioning also requires support from tooling when processing a model. For example, one cannot apply standard automaton minimisation techniques to an automaton with algorithmic conditioning, as these techniques do not necessarily preserve the observe-violation transitions. On the other hand, the positive closure results shown in this section mean that one can first construct an equivalent model without conditioning and then use this model with standard tools.

If termination probabilities need to be analysed, then post-conditioning is not suitable.

The additional expressiveness post-conditioning often provides can be useful, but also creates problems. The advantage is that it allows the user to model problems they could not model in the underlying modelling formalism without conditioning. Whether this class of problems is of practical use is an open question. On the other hand, it also means that dedicated algorithms are required to perform inference on the models. With algorithmic conditioning, on the other hand, one can usually first construct an equivalent model without conditioning and then use standard algorithms.

Chapter 6

Implementation

We have implemented recursively nested conditioning in the model checker PRAY. The code can be found at <https://doi.org/10.5281/zenodo.7514504>. In this chapter, we introduce the capabilities of PRAY and outline how it is extended to support nested conditioning. We then demonstrate the tool’s capabilities and limitations with a few examples.

6.1 Description of the Implementation

6.1.1 Introduction to Pray

PRAY is an explicit-state pPDA-based model checker written in Java. It consists of three components:

- A front end that compiles a program written in the PRAY language into a stack-based¹ intermediate language. The PRAY language is similar to the language we have used throughout this work, with a few additional features to make it more usable in practice, e.g. additional ways to express probabilistic behaviour, more data types and additional control flow structures.
- A model builder that constructs a pPDA from the intermediate language program. This is handled similarly to the operational semantics from Chapter 3. In particular, function calls make use of the pPDA stack, while variable values and the program counter are stored in the state. However, PRAY only adds pop transitions for symbols that are reachable from that state, whereas we add pop transitions for all possible symbols in the operational semantics.
- A model checker then analyses the pPDA. Currently, it is possible to approximately compute termination probabilities and the terminal distribution over the variables of the main function.

¹Note that this *evaluation stack* is separate from the pPDA stack and is not currently abstracted during model building, i.e., the entire evaluation stack is part of the local state of the pPDA.

6.1.2 Conditioning in Pray

To support conditioning, we added query blocks and observe functions to the front end. In the intermediate language, we added new instructions for creating and reading query symbols. Apart from this, query blocks and observe functions are handled largely using existing intermediate language instructions, which keeps changes to the model builder minimal.

In addition to this, we also implemented a few other improvements to PRAY as part of this work: Pray can now output the final distribution of variables². Furthermore, the algorithm that detects reachable symbols for pop transitions is now much faster – often by an order of magnitude or more. We also reduced the number of cases where model building does not terminate due to the handling of the evaluation stack.

6.2 Evaluation

In this section, we present two examples where nested conditioning is used to model agents that reason about each other.

6.2.1 Schelling Games

Schelling games [20] are scenarios where two or more agents have to independently make a decision without communicating. The goal is for the agents to make the same decision. In our example, which is taken from [22], we consider two agents, Alice and Bob, who want to meet at a restaurant. However, there are two restaurants and Alice and Bob cannot communicate. Our goal is to model Alice’s thought process and determine the probability of her going to Restaurant 1.

A simple approach would be for Alice to simply pick the restaurant she prefers. In our example, we assume she has a slight preference for Restaurant 1. However, this does not necessarily maximise the probability of meeting Bob. Therefore, Alice also considers Bob’s preferences. Alice believes that Bob also prefers Restaurant 1, but thinks that his decision-making process in turn depends on her decision, and so on.

In [22], this is modelled as follows: Alice first samples her preferred restaurant probabilistically (by choosing Restaurant 1 with a higher probability). She then recursively evaluates Bob’s preferred restaurant and conditions on her choice being equal to Bob’s choice. Alice’s model of Bob’s thoughts works similarly, and Alice believes that Bob has an accurate model of her own thoughts (and so on).

A naive implementation of this does not terminate, as every call to Alice’s function makes a call to Bob’s function and vice versa. There are two ways of achieving termination. [22] keeps track of recursion depth. If the depth reaches some pre-defined value, further recursion is disabled, i.e. the agent sim-

²This feature uses data computed during the termination analysis, i.e., it did not require changes to the model checking algorithm itself.

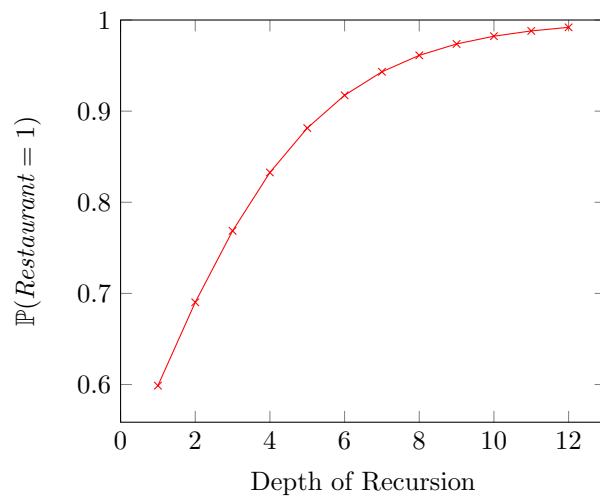


Figure 6.1: Probability that Alice goes to restaurant 1 with bounded recursion.

ply chooses from their own preference.

```

int alice(int depth) {
  query {
    int a = 1 [0.55] 2;
    int b = bob(depth - 1);
    observe(a == b);
    return a;
  }
}

int bob(int depth) {
  query {
    int b = 1 [0.55] 2;
    int a = b;
    if (depth > 0) {
      alice(depth);
    }
    observe(a == b);
    return b;
  }
}

```

The results for bounded recursion are shown in Figure 6.1. As expected, the probability approaches 1 as the recursion depth increases. The graph is identical to the one presented in [22].

Bounded recursion does not require a pPDA and could also be modelled with a Markov chain. However, using a pPDA leads to an exponentially smaller model: In both cases, the model consists of one layer per recursion step. In the case of the pPDA, each layer has the same size, as it has to store the result of the probabilistic choice $1[0.55]2$ before making the recursive call. In Markov chains, on the other hand, each layer doubles in size, as it has to store the results of the

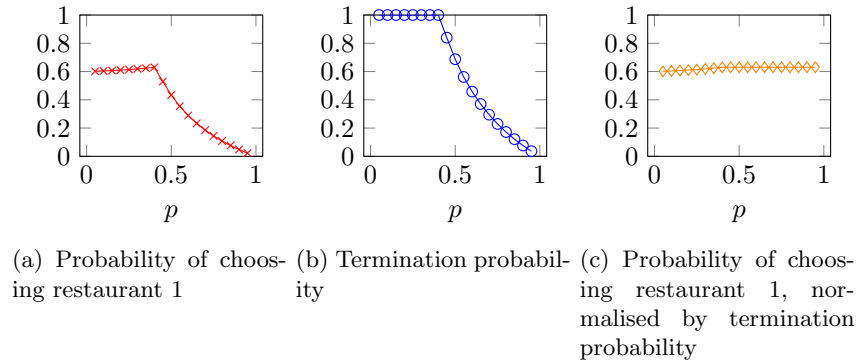


Figure 6.2: The restaurant scenario with probabilistic abortion condition.

probabilistic choices of all previous layers as well. Therefore, the pPDA grows linearly with recursion depth, whereas the Markov chain grows exponentially.

In addition to bounding the depth, we implemented a second approach that uses a probabilistic termination condition. In every function, we disable further recursion with some pre-defined probability p :

```
int alice() {
  query {
    int a = 1 [0.55] 2;
    int b = bob() [p] a;
    observe(a == b);
    return a;
  }
}
int bob() {
  query {
    int b = 1 [0.55] 2;
    int a = alice() [p] b;
    observe(a == b);
    return b;
  }
}
```

While this means that recursion is potentially unbounded, it can be expressed in a finite-state pPDA because we are using the stack for recursion and conditioning. In fact, this approach keeps model size constant for all $0 < p < 1$. The results are shown in Figure 6.2a. For small p , one can see that the probability of going to Restaurant 1 slowly increases. However, there is a sharp drop-off when p is roughly 0.5. The reason for this is shown in Figure 6.2b: For these p , the model no longer almost-surely terminates.

This is unexpected, since every function only makes a single recursive call. As long as the probability of recursion stopping is non-zero, the model should terminate almost-surely. However, our functions are in fact making more than one call due to rejection-sampling.

Note also that, even if we remove the diverging runs from the terminal distribution (see Figure 6.2c), we still do not get the desired behaviour.

6.2.2 Counting Game

In this section, we compare two approaches to model 2-player games. In both, we use conditioning to reason about the next move of the opponent. The approaches differ in which steps are performed inside of the query block.

In the game we are modelling, two players, Alice and Bob, take turns incrementing a counter. The counter starts at 0, each player increments it at least by 1 and at most by k , and the player on whose turn the counter equals or exceeds some n wins. For example, a game with $k = 3$ and $n = 7$ could look as follows: Alice starts by incrementing the counter by 2. Bob adds 1, so the counter is now 3. Alice adds 3, bringing the counter to 6. Bob now adds 1, bringing the counter to 7 and winning the game.

The game has a fairly simple strategy: If the counter reaches $n - k - 1$, then the next player has to increment it to at least $n - k$ and at most $n - 1$. In every case, the following player can reach n . Therefore, the game is won by the player who first reaches $n - k - 1$ if he or she plays optimally. This argument can be applied repeatedly, so the first player to reach $n - i \cdot (k - 1)$ wins for any i . In the above example (with $k = 3$ and $n = 7$), Alice could have forced a win by starting with $n - k - 1 = 3$, forcing Bob to play 4, 5 or 6, and thus allowing Alice to play 7 in every case. In general, Bob wins if and only if $n = i \cdot (k + 1)$ for some i .

Our goal is to compute the expected winner for a given k and n .

Naive Recursive Approach

Our naive approach is very similar to the restaurant example from Section 6.2.1. Alice probabilistically chooses her move by uniformly sampling from $\{1, \dots, k\}$. She updates the counter and then calls Bob. Bob returns the winner of the game (by choosing his move and recursively calling Alice) and Alice conditions on her winning. To make sure the program is legal, with a small probability (e.g. 10^{-3}), she also allows losing. Note that we do not have the issue of non-termination like in the restaurant example: Players have to increment the counter by at least 1 and the game ends once the counter reaches n .

We implement this as follows, with `bob(...)` implemented similarly.

```
string alice(int counter) {
  if counter >= n {
    return "Bob";
  }
  query {
    int choice = uniform(k) + 1;
    int winner = bob(value + choice);
    if winner != "Alice" {
      observe(flip(1//1000));
    }
    return winner;
  }
}
```

The winning probability for Alice using this model is depicted in Figure 6.3 for $k = 3$ and varying n . The model accurately predicts a high winning

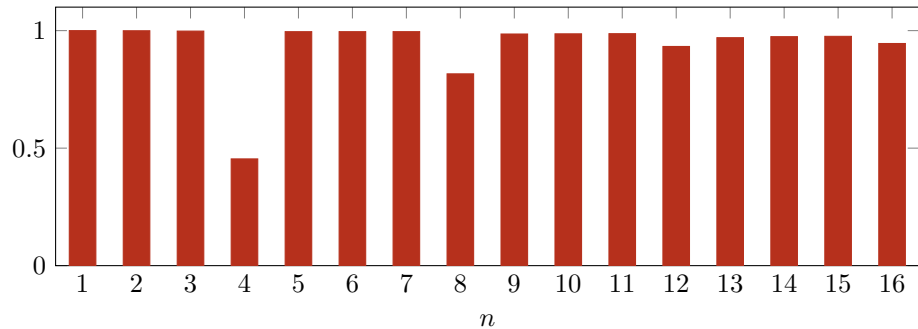


Figure 6.3: Probability of Alice winning the counting game for $k = 3$ and different n , using the naive recursive implementation. If Bob played perfectly, he would win for $n = 4$, $n = 8$, $n = 12$ and $n = 16$.

probability for the cases where Alice can win (those where n is not divisible by $k + 1$), but it overestimates the winning probability for $n = 4$, $n = 8$, \dots (where Bob has a winning strategy) significantly. This issue gets worse as n grows. For $n = 16$, the model predicts that Alice will win with 94.5% probability, even though Bob can force a win.

To explain why this occurs, consider some of the runs for $k = 3$ and $n = 4$. In this scenario, Bob should be able to force a win, but our model does not give him very good odds of doing this. Assume that Alice chooses to play 1. Then Bob has three options: He can add 1, 2 or 3. If he adds 1 or 2 and recursively calls Alice, she can force a win. Therefore, those two runs only pass his conditioning with probability $\frac{1}{1000}$. If, on the other hand, he plays 3, Bob's conditioning is not violated. However, the run is unlikely to pass Alice's conditioning. This is the key problem: All three runs are equally likely to pass conditioning³.

The approach also favours shorter runs, because in every other step, the probability of passing the observe statement is 10^{-3} . If the run has odd length $2 \cdot i + 1$, i.e. Alice makes the final move, then the run is more likely to pass if Alice wins (because there are $i + 1$ observe statements conditioning on Alice winning and just i statements for Bob).

Another perspective is this: We want Alice to probabilistically choose her move, then check the result of this move and then condition her chosen move based on the result. However, we are conditioning her chosen move *and the result* based on the result. Because `bob(...)` is itself a probabilistic function, it is affected by conditioning just as `uniform(...)` is. Therefore, the recursive call does not actually evaluate Bob's best move fairly.

Recursive Approach With Caching

Fixing this requires a somewhat clunky workaround: Alice needs to call Bob recursively *outside* of the query block, but both choosing her move and conditioning on the result need to be inside of the query block. Choosing the move needs to happen before the recursive call and conditioning on the result needs

³In practice, it is slightly more complicated, since the model of Alice exhibits the same problem, so it's not guaranteed that she will actually force a win if Bob adds 1.

to happen after the call. Because query blocks are contiguous, we cannot model this.

However, there is a workaround: We first recursively call Bob for all k possible moves by Alice and store the result in k variables. We then enter the query block, probabilistically choose a move, check the precomputed result of this move and condition on it. For $k = 3$, the implementation looks as follows:

```
string alice(int counter) {
  if counter >= n then {
    return "Bob";
  }
  string b1 = bob(counter + 1);
  string b2 = bob(counter + 2);
  string b3 = bob(counter + 3);

  query {
    string winner;
    int choice = uniform(3) + 1;
    if choice == 1 then { winner = b1; }
    if choice == 2 then { winner = b2; }
    if choice == 3 then { winner = b3; }
    if winner != "Alice" then {
      observe(flip(1//1000));
    }
    return winner;
  }
}
```

The results of this approach are shown in Figure 6.4. The model now predicts the winner more accurately for all cases. However, this comes at the cost of a much larger model and much slower model building. In our implementation (which performs some size-reduction procedures between model building and checking), the model had 176 states, 186 transitions and 44 symbols for $n = 4$ and took about 10 seconds to check, while it had 2365 states, 4738 transitions and 533 symbols for $n = 11$ and took about half an hour to check on a reasonably modern desktop computer. The growth in model size is roughly linear.

While this approach cannot compete with the performance of dedicated game-solving techniques, such as the minimax algorithm (potentially improved with techniques like alpha-beta pruning), it has some distinct advantages: For example, the technique presented can incorporate probabilistic behaviour without any extra effort. If the players were throwing dice in the game above, one would only have to change how the counter is incremented. Dedicated game-solving techniques, on the other, might require modifications to data structures and algorithms to express this probabilistic behaviour.

It might be helpful to explore ways of making this way of modelling more natural. For example, it would be useful to write the code in the normal order with special annotations to mark calls that should be cached. The caching is then implemented automatically by the compiler. Since the caching approach can lead to a very large state space, it might be worth exploring whether it is possible to achieve the same behaviour without caching.

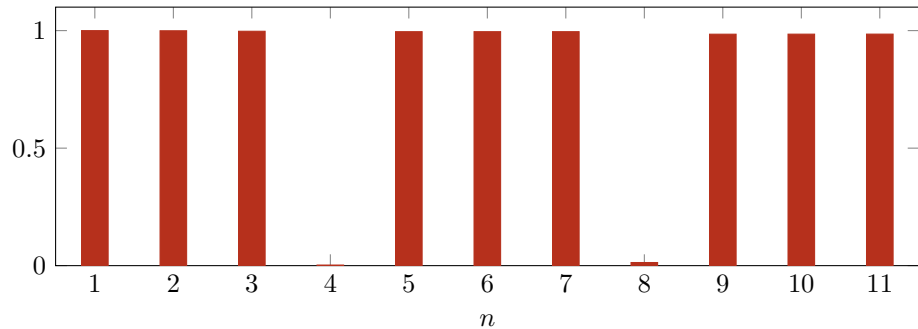


Figure 6.4: Probability of Alice winning the counting game for $k = 3$ and different n , using the recursive implementation with caching. If Bob plays perfectly, he wins for $n = 4$, $n = 8$, $n = 12$ and $n = 16$.

6.2.3 Discussion and Limitations

We have shown how nested conditioning can model human decision-making. Our examples demonstrate the advantages that nested query blocks bring over global conditioning. They also show that this makes sense even in the bounded case, because a pPDA model can be exponentially smaller than a Markov chains that models the same program.

At the same time, the unintuitive termination behaviour of the restaurant example is a potential weakness of our approach. While this behaviour can be desirable in some cases, it might be worthwhile to develop alternate semantics where the program almost-surely terminates. One approach could be to build a separate pPDA for each query block. It is not clear how one would handle circular dependencies (which occur during recursion) with this technique.

The second example has demonstrated how observe statements always condition all probabilistic decisions in the query block and don't give more fine-grained control over which variables are affected by conditioning. One way of mitigating this is to only reset some variables during rejection sampling. However, this is problematic if an early variable should be conditioned and a later variable depends on this, but should not be conditioned. In the above example, this is the case, as we want to condition Alice's choice, but not Bob's behaviour, but Bob's behaviour depends on Alice's choice.

It would have been beneficial to have a larger collection of benchmarks available. Unlike other areas of model checking, where there are often well-curated benchmark sets (see e.g. [10] for a collection of DTMCs, MDPs, etc.), there are very few benchmarks for nested conditioning. It would be very helpful to assemble such a set so that different tools can be compared more easily.

Testing was also hampered by PRAY's limited performance. It was not possible to implement and test some of the larger models from [22], as PRAY either failed to build the model before running out of memory or did not complete model checking even after several hours. While faster code cannot change the underlying complexity of computing the distribution of a pPDA, it is likely that it would still make the technique much more usable in practice.

Chapter 7

Conclusion

We introduced the CRPPL language and showed how nested conditioning can be used to model “reasoning about reasoning”. The operational semantics of CRPPL are based on pPDAs and use the stack of the pPDA both to handle recursive function calls and to store the current query block. The weakest-preexpectation semantics are defined inductively and handle conditioning by computing a pair of pre-expectations for every query block and then using the second expectation for normalisation. These two semantics produce the same reachability and termination probabilities.

We have presented two different types of conditioning – algorithmic and post-conditioning – and explored how these affect the closedness of several model classes under conditioning. For algorithmic conditioning, with the exception of pBPAs, we only obtained positive closure results. We have shown that our language is closed under conditioning by presenting a transformation that replaces query blocks with do-while loops.

We demonstrated our implementation on two examples. In both of these, we modelled “reasoning about reasoning” and demonstrated several potential problems: We showed that nested conditioning can change the termination probability even for programs that almost-surely terminate without conditioning. We also showed that there are intricacies in which probabilistic choices are affected by conditioning. If these are handled properly, then nested conditioning allows modelling games and computing optimal strategies.

7.1 Future Work

We did not consider non-determinism in our program. While [17] shows that weakest-preexpectation semantics cannot be extended to non-deterministic programs easily, even for global conditioning, the same limitation does not apply to the operational semantics. A similar approach might also be possible for our pPDA-based semantics.

The unintuitive termination behaviour and the issues with excluding recursive calls from conditioning could both be solved with memoisation, i.e. performing each recursive call only once even if rejection sampling requires the result in multiple runs. While it is possible to do this manually in our language, this is both inelegant and leads to an exponential state-space in the number of different

recursive calls to be cached. It would thus be interesting to develop operational semantics that perform memoisation by default. It is unclear, however, whether a model class exists where this is possible without significant overhead.

Bibliography

- [1] Tomáš Brázdil et al. “Analyzing probabilistic pushdown automata”. In: *Formal Methods in System Design* 43.2 (2012), pp. 124–163. DOI: 10.1007/s10703-012-0166-0.
- [2] David Chiang, Colin McDonald, and Chung-chieh Shan. “Exact Recursive Probabilistic Programming”. In: *CoRR* abs/2210.01206 (2022).
- [3] Fredrik Dahlqvist and Dexter Kozen. “Semantics of higher-order probabilistic programs with conditioning”. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 57:1–57:29. DOI: 10.1145/3371125.
- [4] Edsger Wybe Dijkstra. “Programming as a Discipline of Mathematical Nature”. In: *The American Mathematical Monthly* 81.6 (1976), pp. 608–612. DOI: 10.1080/00029890.1974.11993624.
- [5] Javier Esparza et al. “Efficient Algorithms for Model Checking Pushdown Systems”. In: *CAV*. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 232–247.
- [6] Cameron Freer, Daniel Roy, and Joshua Tenenbaum. “Towards common-sense reasoning via conditional simulation: legacies of Turing in Artificial Intelligence”. In: *Turing’s Legacy*. Vol. 42. Cambridge University Press, May 1, 2014, pp. 195–252. DOI: 10.1017/cbo9781107338579.007.
- [7] Noah D. Goodman et al. “Church: a language for generative models”. In: *UAI*. AUA Press, 2008, pp. 220–229.
- [8] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. “Operational versus weakest pre-expectation semantics for the probabilistic guarded command language”. In: *Performance Evaluation* 73 (2014), pp. 110–132. DOI: 10.1016/j.peva.2013.11.004.
- [9] Hans Hansson and Bengt Jonsson. “A logic for reasoning about time and reliability”. In: *Formal Aspects of Computing* 6.5 (1994), pp. 512–535. DOI: 10.1007/bf01211866.
- [10] Arnd Hartmanns et al. “The Quantitative Verification Benchmark Set”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 344–350. DOI: 10.1007/978-3-030-17462-0_20.
- [11] Thomas Icard. “Calibrating generative models: The probabilistic Chomsky–Schützenberger hierarchy”. In: *Journal of Mathematical Psychology* 95 (Apr. 2020), p. 102308. DOI: 10.1016/j.jmp.2019.102308.

- [12] Marta Kwiatkowska, Gethin Norman, and David Parker. “PRISM 4.0: Verification of Probabilistic Real-Time Systems”. In: *Computer Aided Verification*. Springer, 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.
- [13] 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.
- [14] Theofrastos Mantadelis and Gerda Janssens. *Nesting Probabilistic Inference*. 2011. DOI: 10.48550/ARXIV.1112.3785.
- [15] Christoph Matheja. “Automated reasoning and randomization in separation logic”. PhD thesis. 2020. DOI: 10.18154/RWTH-2020-00940.
- [16] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2005. DOI: 10.1007/b138392.
- [17] Federico Olmedo et al. “Conditioning in Probabilistic Programming”. In: *ACM Transactions on Programming Languages and Systems* 40.1 (Jan. 12, 2018), pp. 1–50. DOI: 10.1145/3156018.
- [18] Federico Olmedo et al. “Reasoning about Recursive Probabilistic Programs”. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, July 5, 2016. DOI: 10.1145/2933575.2935317.
- [19] Michael Rabin. “Probabilistic automata”. In: *Information and Control* 6.3 (Sept. 1963), pp. 230–245. DOI: 10.1016/s0019-9958(63)90290-0.
- [20] Thomas C Schelling. *The Strategy of Conflict*. Harvard university press, 1980.
- [21] Andreas Stuhlmüller and Noah D. Goodman. “A Dynamic Programming Algorithm for Inference in Recursive Probabilistic Programs”. In: *StarAI@UAI*. 2012.
- [22] A. Stuhlmüller and N.D. Goodman. “Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs”. In: *Cognitive Systems Research* 28 (2014), pp. 80–99. DOI: 10.1016/j.cogsys.2013.07.003.
- [23] Zenna Tavares et al. *The Random Conditional Distribution for Higher-Order Probabilistic Inference*. 2019. DOI: 10.48550/ARXIV.1903.10556.
- [24] Tobias Winkler, Christina Gehnen, and Joost-Pieter Katoen. “Model Checking Temporal Properties of Recursive Probabilistic Programs”. In: *Lecture Notes in Computer Science*. Springer, 2022, pp. 449–469. DOI: 10.1007/978-3-030-99253-8_23.
- [25] Yizhou Zhang and Nada Amin. “Reasoning about “reasoning about reasoning”: semantics and contextual equivalence for probabilistic programs with nested queries and recursion”. In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022), pp. 1–28. DOI: 10.1145/3498677.