INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600
A Predicative Model for Probabilistic Specifications and Programs

by

Victor Kwan

A thesis submitted in conformity with the requirements for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 1998 by Victor Kwan
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni desextraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-40745-4
Abstract

A Predicative Model for Probabilistic Specifications and Programs

Victor Kwan
Master of Science
Graduate Department of Computer Science
University of Toronto
1998

The incorporation of probability into several theories of programming has been formalized by Morgan et al. [17, 18]. He et al. [3] present two different semantic models for extending Dijkstra’s guarded command language with an additional operator, namely probabilistic choice.

The aim of this thesis is in the same line as the work mentioned above—to extend Hehner’s theory of programming [5] (referred to as the standard model) to incorporate probabilistic behavior. The extended theory (referred to as the probabilistic model) makes it possible for the development of probabilistic programs from probabilistic specifications.

In the probabilistic model, a probabilistic specification is a boolean expression as in the standard model, but the variables of the boolean expression are no longer the prestate and the poststate. Instead, they are the prestate, and a function: the probability distribution over all possible poststates. Refinement is implication, as in the standard model. Probabilistic programs are probabilistic specifications. The probabilistic programming language is an extension of the standard programming language, and contains a construct for probabilistic choice. Various issues of interest, such as nondeterminism and timing, are discussed.

The work in this thesis is consistent with the framework proposed by Hoare and He [9] for unifying theories of programming.
Acknowledgements

I thank my supervisor, Ric Hehner, for the many valuable insights and ideas he had contributed to the material in this thesis. It had been a memorable learning experience to work with him. I also thank him for the advice and assistance he had given me on non-academic matters, particularly his understanding for the difficulties I had faced when I returned from an extended period of sick leave.

I thank my second reader, Mike Molloy, for reading and commenting on the thesis.

Finally, special thanks to my family, especially mom, who had cared for me during my sick leave.
Contents

1 Introduction .......................................................... 1
1.1 Motivation ......................................................... 1
1.2 Applications of probabilistic programs .......................... 2
   1.2.1 Randomized algorithms .................................... 2
   1.2.2 Explicit probabilistic behavior ......................... 3
   1.2.3 Quantifiable unreliability ............................... 3
1.3 Related Work ..................................................... 4
1.4 The thesis ......................................................... 6

2 Basic probability theory ........................................... 7
2.1 Probability distributions ........................................ 7
   2.1.1 Event expressions ....................................... 10
2.2 Random variables .............................................. 11
   2.2.1 Expected value of a random variable ................. 12
   2.2.2 Probability density functions ......................... 12

3 The probabilistic model of specifications ....................... 14
3.1 The model of computation ..................................... 14
3.2 Probabilistic specifications ................................... 15
   3.2.1 Definitions ............................................. 16
   3.2.2 Adding variables to the state ......................... 19
   3.2.3 Specifications with time and worst-execution time bounds .......................... 21
   3.2.4 Expected values and expected time bounds ........... 21
   3.2.5 Specification abbreviations ............................ 22
3.3 General specifications ........................................ 23
3.4 Specification notations ....................................... 24
   3.4.1 The specifications ok, x := e and if b then R else S ........ 25
   3.4.2 The specification \[ R \land S \lor (R \lor \neg S) \] ............. 25
A  The standard model
   A.1 The model of computation 74
   A.2 Specifications 75
      A.2.1 Definitions 75
      A.2.2 Specifications with time 76
   A.3 Specification notations 76
      A.3.1 The specification ok 76
      A.3.2 The specification $x := e$ 76
      A.3.3 The expression if $b$ then $R$ else $S$ 77
      A.3.4 The specification $R$ or $S$ 77
      A.3.5 The specification $R . S$ 77
   A.4 Refinement 77
      A.4.1 Refinement laws 78
   A.5 Programs 79
      A.5.1 The programming language 79
   A.6 Recursive time 80
      A.6.1 Timing expressions 81
      A.6.2 Termination 82
   A.7 Expanding the programming language 83
      A.7.1 Variable declaration 84
      A.7.2 While loop 84
      A.7.3 For loop 85

B  Axioms and laws 86
   B.1 Probability theory 86
      B.1.1 Probability distributions and events 86
      B.1.2 Expected values and probability density functions 87
   B.2 Specification notations 87
      B.2.1 Probabilistic composition (constant alternatives) 87
      B.2.2 Convex composition 89
      B.2.3 Probabilistic composition (variable alternatives) 89
      B.2.4 Dependent composition 90
      B.2.5 Variable declaration 91
      B.2.6 Random number generators 91
   B.3 Refinement 91
   B.4 Miscellaneous 92
C Proofs of selected laws

C.1 Proof of PC-10 ................................................................. 93
C.2 Proof of PC-11 ................................................................. 94
C.3 Proof of PV-7 ................................................................. 98
C.4 Proof of PV-5 ................................................................. 99

D Notations and precedence

D.1 Notations and names ....................................................... 101
D.2 Precedence ................................................................. 103

Bibliography ................................................................. 104
Chapter 1

Introduction

1.1 Motivation

A mathematical theory of programming is an essential tool for the understanding and design of programs. Without a theory, it is very difficult to write large or complex programs correctly. Many programs in the software industry are developed by programmers who have little knowledge of any theory, and consequently, many of them are bug-ridden. A good theory of programming helps us to derive precise mathematical specifications from informal ones, and to develop correct and efficient programs from the specifications. Even though we may not want to apply a theory every time we write a program, we shall have the techniques at our disposal when they are needed. Moreover, a theory benefits us in that it gives us a dissection of programs from the mathematical point of view, a philosophy that cannot be attained otherwise.

There are several theories of programming. Hoare's Logic [7] and Dijkstra's theory of weakest precondition predicate transformers [1] have provided a firm basis for research in this area, and have led to a formalism for the stepwise refinement approach to program development based on Dijkstra's semantics; this formalism (known as the Refinement Calculus) is described in Morgan's book [15]. The Vienna Development Method [11] and the Z specification language [22] have also been used to advantage by industry.

The theory developed by Hehner, presented in his book [5], is simpler and more comprehensive than any of those mentioned above. In his theory, specifications are boolean expressions, refinement is universally quantified implication, and programs are specifications; the programming language is a sublanguage of the specification language. It applies to both terminating and nonterminating computation, to both sequential and parallel computation, to both stand-alone and interactive computation. It also includes
time bounds, both for algorithm classification and for tightly constrained real-time applications.

However, some aspects and paradigms of programming that have widespread applications are not covered by some of the theories mentioned above. One of such paradigms, probability, is not covered by any of the theories mentioned above. A probabilistic program differs from a standard program in that at some stage during an execution of the program, a probabilistic choice of action can be made.

Probabilistic programs have important use in all kinds of industry, such as network protocols, distributed systems, numerical softwares, simulated systems modeling, and even computer games. Researches relating this area have been conducted by people from all disciplines of computer science, and have experienced a rapid growth in recent years. The level of interest is still rising.

The aim of this thesis is to extend Hehner's theory of programming to incorporate probabilistic specifications, so that rigorous reasoning can be done for the verification and development of probabilistic programs. Several people have worked on the same goal but for other theories of programming; some of their work is summarized in Section 1.3.

1.2 Applications of probabilistic programs

The goal of this thesis is to extend Hehner's theory of programming, so that rigorous reasoning can be done in situations such as those described below.

1.2.1 Randomized algorithms

For many applications, even though the problem to be solved (the specification) does not mention probability explicitly, a randomized algorithm may be either the simplest or the most efficient solution available, and sometimes both. The use of randomized algorithms has been undergoing tremendous development in many areas of applications.

There are several types of randomized algorithms:

- Some randomized algorithms guarantee a worst-execution time bound, but may give a wrong result with a small probability. These algorithms are sometimes called Monte Carlo algorithms. Example: primality testing.

- Some randomized algorithms always give a correct result, but the execution time is not guaranteed, though termination is guaranteed even on the worst-case input (so the probability of nontermination is zero); they are useful if they guarantee a low
expected time bound. These algorithms are sometimes called *Las Vegas algorithms*. Example: symmetry breaking in distributed algorithms.

- Some randomized algorithms always give a correct result, and guarantee a worst-execution time bound. Example: order statistics.

- Numerical randomized algorithms can be used to compute approximate solution of a numerical problem. Example: randomized approximation of definite integrals.

A comprehensive study of randomized algorithms and their applications is presented in a book by Motwani and Raghavan [19].

### 1.2.2 Explicit probabilistic behavior

Probabilistic programs can be used to simulate physical systems that exhibit probabilistic or stochastic behavior. Such programs are sometimes called *Monte Carlo simulations*. Simulations in this manner are used for various purposes, such as to derive or estimate properties of the physical systems simulated. In the simplest case, you may want to write a program to simulate the behavior of 100 fair coin tosses, and estimate the expected number of heads. In this case a mathematical analysis is simple and preferable, but it may not be so when the simulation is of complex physical phenomena such as radiation transport in the earth's atmosphere or galactic formation.

For such a simulation, programming errors are even harder to be detected by mere testing than the standard (non-probabilistic) case, since any observation cannot be ruled unsatisfactory unless it is implied by the specification to have zero probability of occurrence. To determine whether observations satisfy a specification, the best we can do is to test the program many times, and conduct a statistical test of hypothesis based on the observations; and even if we do so, we are never 100% confident of our conclusion. Because of this, a theoretical approach to programming is even more important when probability is involved.

### 1.2.3 Quantifiable unreliability

The computation of a program may not behave as specified due to inherent unreliability of the medium that executes it. For example, a disk head may crash, the operating system may interrupt the process, and the computer may be accidentally shut down by nature or by human force. A specification does not exactly "specify" the behavior
of the computer unless such undesirable behavior is also taken into account, and this unreliability is sometimes best described by means of probability.

In many situations, the probability of undesirable behavior is negligibly small, and the theory of programming is applied at a level of abstraction that neglects the medium in which the program is executed. In some situations, for example, communication protocols which rely ultimately on transmission lines with inherent unreliability (conventionally described statistically), such a level of abstraction may be unacceptable, and probability may be unavoidably present in the implementation.

1.3 Related Work

The predicate transformer semantic model for probabilistic programs is first proposed by Kozen [13], and later by Jones [10]. Neither of their work considers nondeterminism, and consequently, their model could not be used for program development. This deficiency is later resolved when Morgan et al. (the Probabilistic Systems Group at Oxford) extends the model with nondeterminism [17].

Morgan’s model is briefly described as follows. It is an extension to Dijkstra’s theory of weakest precondition predicate transformers [1]. In the model, a probabilistic predicate is a nonnegative random variable over the state, and the semantics of a program is defined as a function from probabilistic postconditions (probabilistic predicates over poststates) to probabilistic preconditions (probabilistic predicates over prestates). For program $P$ and postcondition $\beta$, the probabilistic weakest precondition

$$ wp\ P\ \beta $$

is the minimum expected value of $\beta$, in terms of the prestate, when $P$ is executed. For a standard postcondition (the value of $\beta$ is either 1 or 0, where 1 corresponds to $\top$ and 0 corresponds to $\bot$), this gives the minimum probability that a computation of $P$ terminates in a poststate satisfying the postcondition. Under the probabilistic weakest precondition semantics, some of the programming notations we use (Section 3.4) can be defined as follows.

$$ wp\ ok\ \beta = \beta $$
$$ wp\ (x := e)\ \beta = (\text{substitute } e \text{ for } x \text{ in } \beta) $$
$$ wp\ (\text{if } b \text{ then } P \text{ else } Q)\ \beta = \text{if } b \text{ then } wp\ P\ \beta \text{ else } wp\ Q\ \beta $$
$$ wp\ (P \ P\ or\ Q)\ \beta = p \times (wp\ P\ \beta) + (1 - p) \times (wp\ Q\ \beta) $$
$$ wp\ (P \ or\ Q)\ \beta = \min (wp\ P\ \beta) \ (wp\ Q\ \beta) $$
$$ wp\ (P . Q)\ \beta = wp\ P\ (wp\ Q\ \beta) $$
Here is an example to illustrate how this works.

\[
wp (x := 1 \frac{1}{3} \text{ or } x := 2. \ y := 1 \frac{1}{2} \text{ or } y := 2) \ (\eta(x \leq y)) = \wp (x := 1 \frac{1}{3} \text{ or } x := 2) \ (\wp (y := 1 \frac{1}{2} \text{ or } y := 2) \ (\eta(x \leq y))) = \wp (x := 1 \frac{1}{3} \text{ or } x := 2) \ (\frac{1}{2} \eta(x \leq 1) + \frac{1}{2} \eta(x \leq 2)) = \frac{1}{6} \eta(1 \leq 1) + \frac{1}{6} \eta(1 \leq 2) + \frac{1}{3} \eta(2 \leq 1) + \frac{1}{3} \eta(2 \leq 2) = \frac{2}{3}
\]

This means that the computation establishes the (standard) postcondition \(x \leq y\) with probability at least \(2/3\).

Further work on the probabilistic predicate transformer model is reported in [14, 16], and an introduction to the model is given in [21].

In addition to the probabilistic predicate transformer model, Morgan et al. also present an extension to the failures/divergences model for Communicating Sequential Processes (CSP) (originally developed by Hoare [8]) to incorporate probabilistic choice [18].

Kozen [12] (and later Jones [10]) describes a model in which the semantics of a program is defined as a mapping from prestates to poststate distributions. The model allows probabilistic choice, but not nondeterministic choice. The extension with nondeterminism is provided by He et al. [3], who present two different semantic models for extending Dijkstra’s language of guarded commands with an operator for probabilistic choice.

He’s first model, the relational model, provides a source for the probabilistic model presented in this thesis. In He’s relational model, the semantics of a program is defined as a mapping from prestates to sets of poststate distributions. The model employs a “total correctness” semantics (as opposed to our model which employs a “partial correctness” semantics with a time variable); this means that the only thing we can say about time is the probability that it is finite (the probability of eventual termination), which also infers the probability that it is infinite (the probability of non-termination). To allow for non-termination, the state space contains an additional state, the improper state \(I\) (in He’s model the symbol \(\bot\) is used, but we have used it for boolean false). A program refines another if and only if its computation terminates more often and behaves less nondeterministic than the other; precisely, a poststate distribution is said to refine another if it assigns equal or higher probabilities to all poststates except \(I\), and program \(P\) refines program \(Q\) if and only if from every prestate, every poststate distribution \(P\) maps to refines some poststate distribution \(Q\) maps to. To allow refinement between programs to be defined as set inclusion, for any program and any prestate, the set of poststate distributions mapped to must be closed under refinement of poststate distributions (any poststate distribution that refines one in the set must also be in the set). (For
example, the nowhere-terminating computation is specified by the program which maps any prestate to the set of all poststate distributions, rather than just the poststate distribution that assigns probability 1 to I.) This is called the up-closure property; there is another closure property, convexity, which is used to model nondeterministic programs. This property will be discussed later in the thesis (Section 3.5).

In He’s second model, the lifted model, a program is a set of mappings from prestates to probabilistic distributions over poststates. Connections between He’s two models and the non-probabilistic model are also investigated in [3].

A one-to-one correspondence between programs of the relational model and a subspace of probabilistic predicate transformers (those satisfying the sublinear property) is proved, and for the probabilistic guarded command language, an isomorphism between its definitions under the relational model and the probabilistic predicate transformer model is established in [17].

Collected reports of all related work by the Probabilistic Systems Group can be found at [20].

1.4 The thesis

Chapter 2 presents the basic probability theory which is necessary for our probabilistic model. Appendix A contains a summary of Hehner’s theory for standard (non-probabilistic) specifications. The extension of his theory to incorporate probabilistic specifications is presented in Chapter 3, and the development of probabilistic programs is discussed in Chapter 4; these two chapters are the heart of the thesis. Chapter 5 concludes our work.

Appendix B contains a list of axioms and laws for the notations introduced in this thesis, and some of these laws are proved in Appendix C. Appendix D contains a glossary of all notations used in this thesis, as well as a precedence table for them. Unless otherwise stated, all notations that can be found in Hehner’s book share the same axioms and laws as listed in the book [5, p.229–243].
Chapter 2

Basic probability theory

Before we present our model for probabilistic programming, we turn to the basic probability theory which is necessary for our probabilistic model. The theory is the same as that which one can find in any textbook on probability (for example, [2]); however, it is adapted to the notations in Hehner's book [5] on which this thesis is based.

2.1 Probability distributions

A probability experiment is a process that yields one of a given bunch of possible outcomes, called the sample space of the experiment. If a probability experiment is conducted repeatedly, the likelihood of each outcome is constant over successive trials of the experiment. A probability experiment can be described by a probability distribution over its sample space, defined as follows. In this thesis, we only consider discrete (finite or countably infinite) sample spaces. The bunch of all real numbers between 0 and 1 inclusive, \( i : \text{real} \cdot 0 \leq i \leq 1 \), is denoted \( 0 \ldots 1 \).

**Definition 2.1.** Let \( B \) and \( D \) be bunches, where \( D \) is discrete, and \( B : D \). A probability distribution \( \mathcal{P} \) over \( D \) is a real function with domain \( D \) (\( \Delta \mathcal{P} = D \)), satisfying the following probability axioms:

\[
\begin{align*}
\mathcal{P}B & \geq 0 \\
\mathcal{P}D & = 1 \\
\mathcal{P}B & = \Sigma \varphi : B \cdot \mathcal{P}\varphi
\end{align*}
\]

---

1In Hehner's theory, bunches are introduced as an alternative data structure to sets. A bunch represents an unpackaged collection of objects (in contrast, a set represents a packaged collection). Unlike sets, bunches reduce properly to the deterministic case (a bunch of one element is equal to the element), which allows nondeterminism to be expressed more conveniently.

2The notation is borrowed from the \( x \ldots y \) notation in Hehner's book [5, p. 16]. In addition to \( x \ldots y \), we introduce three new notations \( x \ldots y, \; x \ldots y \) and \( x \ldots y \) to represent intervals, and allow them to be used for any number system as long as the domain is stated or implied by context, either informally in English or formally using bunch intersection as in \( \text{nat} \{0 \ldots 7\} \). The four definitions are similar except for the inclusion or exclusion of \( x \) and \( y \); see Appendix B.
Since $\mathcal{P}$ is a function, in addition to the above axioms, all the function axioms in Hehner's theory [5, p. 235–236] also apply to $\mathcal{P}$. Note that the argument supplied to $\mathcal{P}$ can be a non-elementary bunch, so the function application axiom for probability distributions is extended to allow substitution of bunch arguments [5, p. 30–31]. For clarity, we use lowercase letters (such as $x, y$, including Greek letters such as $\varphi$) for elementary bunches, and uppercase letters (such as $A, B$) for general bunches, unless otherwise stated.

Here are some terminologies and definitions. Any bunch $B$ in the sample space $D$ is called an event.$^3$ An elementary event is an event with exactly one outcome. A compound event is an event with two or more outcomes. The null event, null (the empty bunch), is an event with no outcome. For event $B$, the complementary event of $B$, written as $\bar{B}$, is the event $\forall : D \cdot \neg \varphi : B$.

The value given by $\mathcal{P}B$ is called the probability of the event $B$, which is a measure of the likelihood of $B$ in the experiment. When an experiment is conducted repeatedly, the relative frequency of $B$ is the number of trials that the observed outcome is in $B$ divided by the total number of trials. In the description of the experiment, the probability of $B$ can be thought of as the limit of the relative frequency of $B$ over repeated trials of the experiment, ad infinitum. (The relative frequency concept of probability is intuitively meaningful, but it does not provide a precise definition of probability. In reality, “probability” cannot be defined based on observed events, because we can never conduct an experiment for an infinite number of trials. The practical definition of probability is a matter that can lead to endless philosophical arguments, and we will devote no time to it.)

Any event $B$ where $\mathcal{P}B = 1$ is called a certain event, so the event $D$ is a certain event. Any event $B$ where $\mathcal{P}B = 0$ is called an impossible event.

Here are some laws for probability distributions. Let $\mathcal{P}$ be a probability distribution over discrete sample space $D$; let $A$ and $B$ be events.

**PR-1** \[0 \leq \mathcal{P}B \leq 1\]

$^3$For a general probability distribution, where the sample space may be uncountably infinite, not every bunch in the sample space may be considered an event. The main requirement is that the set of events of a sample space be closed under complementary events, the union of a finite or countable number of events, and the intersection of a finite or countable number of events. For the general case, the third probability axiom should be

$$\mathcal{P}(\bigcup_{i=1}^{n} B_i) = \sum_{i=1}^{n} \mathcal{P}B_i$$

where $\bigcup_{i=1}^{n} B_i$ is the union of a (finite or countably infinite) sequence of events $B_0, B_1, \ldots$ that are pairwise disjoint ($B_i \cap B_j = \text{null}$ for all $i \neq j$). Formally, the sequence can be represented as a list, where each item of the list is an event packaged into a set, which is then unpackaged when applied to $\mathcal{P}$ (bunches cannot be used in the sequence because lists distribute over bunch union). Since we only consider discrete sample space, we avoid the complication and use a simpler axiom in Def. 2.1.
Although the argument supplied to a probability distribution can be a compound event, in the discrete case a probability distribution is in fact characterized by the probabilities of elementary events. Moreover, although we have defined a probability distribution $\mathcal{P}$ over $D$ on elements in the sample space $D$ only, it is intuitively clear that the probability of any element not in $D$ can be defined as zero. We can therefore add the following probability axiom, for any bunch $B$ ($B$ may not be included in $D$):

$$\mathcal{P}B = \mathcal{P}(B \cdot D)$$

This axiom extends the function application axiom for probability distributions, since $\mathcal{P}$ is now applicable to bunches not included in its domain. The axiom implies that any element not in $D$ has zero probability. If we consider any general bunch (which may not be in $D$) to be an "event", the probability of any event is equal to the probability of (the bunch of) those elements of the event that have non-zero probabilities, and all such elements must be in $D$. To describe a probability distribution, we only have to describe its domain $D$ in so far as $D$ includes all the elements with non-zero probabilities. For those elements with zero probabilities, it does not matter if we include any of them in $D$.

We therefore define two (discrete) probability distributions to be equal if and only if their probabilities are equal with respect to all elementary events in either domain:

$$\mathcal{P} = \mathcal{Q} \equiv \forall \varphi : \Delta \mathcal{P}, \Delta \mathcal{Q} \cdot \mathcal{P}\varphi = \mathcal{Q}\varphi$$

Note that this is not the same as function equality, since function equality requires $\mathcal{P}$ and $\mathcal{Q}$ to have equal domains, and give equal values when applied to any bunch in the domain (not just elementary bunches as in the above definition). This is justified since $\mathcal{P}$ and $\mathcal{Q}$ are provided to be probability distributions and must satisfy the probability axioms.

To represent a probability distribution, it suffices to write a function that uniquely characterizes it; it is not necessary to describe all properties of the probability distribution. To represent a probability distribution, $\mathcal{P}$ is only required to satisfy the following two conditions:

$$\mathcal{P}\varphi \geq 0$$
$$\sum \mathcal{P} = 1$$

And the value of $\mathcal{P}B$ for any bunch $B$ can be deduced from the probability axioms.
Example 2.1. Suppose we roll a pair of fair dice, and observe the outcome. The sample space may be represented as $[2*(\text{nat } (1,..,6))]$, or just $[2*(1,..,6)]$, and the probability distribution $\mathcal{P}$ over the sample space is

$$\mathcal{P} = \lambda[x;y]:[2*(1,..,6)] \cdot 1/36 = [2*(1,..,6)] \rightarrow 1/36$$

Alternatively, the sample space may be represented as $[2*\text{nat}]$, and the probability distribution $\mathcal{P}$ over the sample space is

$$\mathcal{P} = \lambda[x;y]:[2*\text{nat}] \cdot \text{if } 1 \leq x \leq 6 \land 1 \leq y \leq 6 \text{ then } 1/36 \text{ else } 0$$

The two probability distributions are equal.

Using the first representation of the sample space and $\mathcal{P}$, given any number $n$, the probability of the event that the sum of the two numbers on the dice equals $n$ is

$$\mathcal{P}(\{x;y|[2*(1,..,6)] \cdot x + y = n\}) = \Sigma[x;y][\{x;y|[2*(1,..,6)] \cdot x + y = n\} \cdot 1/36$$

$$= \varphi(\{x;y|[2*(1,..,6)] \cdot x + y = n\})/36$$

$$= \text{if } 2 \leq n \leq 12 \text{ then } (6 - \text{abs}(7 - n))/36 \text{ else } 0$$

The second representation of the sample space and $\mathcal{P}$ also gives the same result.  

2.1.1 Event expressions

For sample space $D$, let $B$ be an event. We have the following theorem:

$$B = \langle \varphi : D \cdot \varphi : B \rangle$$

Using the solution quantifier, any event $B$ can be expressed by the boolean expression $\varphi : B$, where $\varphi$ is the variable for the outcome. In general, to express any event $B$, we can use a boolean expression $b$ of which the outcome $\varphi$ is a variable, so that $B$ is the bunch of all outcomes satisfying $b$. We define the following notation:

Definition 2.2. Let the sample space be $D$, and the outcome be referred to by the variable $\varphi$. The notation $\langle b \rangle$, called event expression, is defined

$$\langle b \rangle = \langle \varphi : D \cdot b \rangle$$

Note that the scope of any instance of the outcome $\varphi$ that appears in $b$ is local to the expression $\langle b \rangle$.  

Example 2.2. Let the sample space be $\text{nat}$. We have

$$\langle \exists n : \text{nat} \cdot \varphi = 2n \rangle = \langle \varphi : 2 \times \text{nat} \rangle$$

$$= \langle \varphi : \text{nat} \cdot \varphi : 2 \times \text{nat} \rangle$$

$$= 2 \times \text{nat}$$

In an event expression, we may use any variable for the outcome in place of $\varphi$ as long as its name is explicitly stated. Different variable names may be used for different event expressions, but for clarity we usually choose a name that is consistent with the context of the expression; for example, if the expression is supplied to the probability distribution $\mathcal{P}'$ as argument, we usually use a primed variable to refer to the outcome. If the outcome is a list of items, its components can be referred to by variables such as $x, y, n$.  


Example 2.3. Let the sample space be \([\text{nat}; \text{nat}]\); the outcome can be represented as \([x; y]\). We have

\[
\langle x = y \rangle = \{x; y\} : [\text{nat}; \text{nat}] \cdot x = y \\
= [0; 0], [1; 1], \ldots \\
= (\lambda n \cdot [n; n]) \text{nat}
\]

In addition to \(\varphi\), the event expression also omits the sample space \(D\), the domain of the solution quantification. Usually we only want to talk about one sample space, but sometimes we need to distinguish which sample space the expression refers to (for example, when a sample space is the range of a function over another sample space, as in Section 2.2.2). The sample space may be explicitly stated (as in the above examples), or it may be determined by context: If the event expression is supplied to a probability distribution as argument (as in \(\mathcal{P}(b)\)), the sample space is the sample space (domain) of the probability distribution.

The advantage of the notation \(\langle b \rangle\) is that we can express events using operators over boolean expressions rather than bunches. For example, if events \(A, B\) are expressed by \(\langle a \rangle, \langle b \rangle\) respectively, the event \(A \land B\) can be expressed by \(\langle a \lor b \rangle\), the event \(A \lor B\) can be expressed by \(\langle a \land b \rangle\), and the event \(\overline{B}\) can be expressed by \(\langle \neg b \rangle\). Moreover, the event expression \(\langle x = y \rangle\) in Ex. 2.3 is more formal than \([0; 0], [1; 1], \ldots\), and more succinct than \((\lambda n \cdot [n; n]) \text{nat}\) (which is a function distributing over the bunch \(\text{nat}\)).

Using event expressions, some of the axioms and laws above can be restated as follows, where \(a\) and \(b\) are boolean expressions.

\begin{align*}
\text{PR-1} & \quad 0 \leq \mathcal{P}(b) \leq 1 \\
\text{PR-2} & \quad \mathcal{P}(b) = \sum \varphi : D \cdot \text{if } b \text{ then } \mathcal{P}\varphi \text{ else } 0 \\
\text{PR-3} & \quad \mathcal{P}(\top) = 1 \\
\text{PR-4} & \quad \mathcal{P}(a) + \mathcal{P}(b) = \mathcal{P}(a \lor b) + \mathcal{P}(a \land b) \\
& \quad \mathcal{P}(b) + \mathcal{P}(\neg b) = 1 \\
\text{PR-5} & \quad \forall \varphi : D \cdot a \Rightarrow b \Rightarrow \mathcal{P}(a) \leq \mathcal{P}(b) \\
\text{PR-6} & \quad \mathcal{P}(a \land b) = 1 \Rightarrow \mathcal{P}(a) = 1 \land \mathcal{P}(b) = 1 \\
& \quad \mathcal{P}(a \lor b) = 0 \Rightarrow \mathcal{P}(a) = 0 \land \mathcal{P}(b) = 0
\end{align*}

2.2 Random variables

A random variable \(\mathcal{X}\) is a real function whose domain is the sample space of a probability experiment. For example, if the sample space is \([\text{nat}; \text{nat}]\), \(\lambda [x; y] : [\text{nat}; \text{nat}] \cdot x^2 y / 7\) is a random variable.
When the sample space and the variables that refer to the outcome are explicitly stated, as an abbreviation we sometimes just write the body of \( \mathcal{X} \) to mean \( \mathcal{X} \). For example, the random variable above can be abbreviated as \( x^2 y/7 \).

### 2.2.1 Expected value of a random variable

Given a probability distribution, the expected value of a random variable is defined as follows.

**Definition 2.3.** Let \( \mathcal{P} \) be a probability distribution over sample space \( D \), and let \( \mathcal{X} \) be a random variable. The expected value of \( \mathcal{X} \) over \( \mathcal{P} \), \( \mathcal{E}\mathcal{P}\mathcal{X} \), is defined

\[
\mathcal{E}\mathcal{P}\mathcal{X} = \sum_{\varphi} \varphi : D \times \mathcal{X}\varphi \times \mathcal{P}\varphi
\]

In the definition, the symbol \( \mathcal{E} \) can be regarded as a real function of two functional parameters: a probability distribution, and a random variable. \( \mathcal{E}\mathcal{P} \) is thus a real function of one functional parameter, a random variable. Conventionally, when we talk about expectations we sometimes only provide the random variable (preferably put in parentheses when local variables are introduced to represent the random variable but are omitted in abbreviation), and the probability distribution is explicitly stated. For a given probability distribution \( \mathcal{P} \), we can introduce a symbol such as \( \mathcal{E} \) to represent \( \mathcal{E}\mathcal{P} \). We call \( \mathcal{E} \) the expectation function for \( \mathcal{P} \).

**Example 2.4.** Consider the experiment in Ex. 2.1 where we roll a pair of fair dice, and let \([x; y]\) represent the outcome. The expected value of the random variable \( \text{abs}(x - y) \) is

\[
\mathcal{E}(\text{abs}(x - y)) = \sum_{[x; y]} [2*(1,6)] \cdot 1/36 \times \text{abs}(x - y) = 35/18
\]

Note that in the above example, \( x \) and \( y \) are just local variables introduced in the abbreviation \( \text{abs}(x - y) \), and are not global variables of the expression \( \mathcal{E}(\text{abs}(x - y)) \).

### 2.2.2 Probability density functions

A random variable associates each possible outcome of a probability experiment with a real number; given any real number, we are interested in the probability of the event that the outcome is associated with that number. More generally, we may consider any function whose domain is the sample space of the experiment, and we are interested in the probability distribution induced on the range of the function.\(^4\) We have the following definition:

\(^4\)Conventionally, probability density functions are defined for random variables only, and the probability density function of random variable \( \mathcal{X} \) is often just called “the probability distribution for \( \mathcal{X} \)”. Here we take a generalization which will be used in the thesis, and use separate terminologies for probability density functions and probability distributions.
**Definition 2.4.** Let \( \mathcal{P} \) be a probability distribution over sample space \( D \); let \( \mathcal{X} \) be a function with domain \( D \). The **probability density function** of \( \mathcal{X} \) with respect to \( \mathcal{P} \), \( \mathcal{X} \circ \mathcal{P} \), is a function defined by the following axioms:

\[
\begin{align*}
\mathcal{X} D & : \Delta(\mathcal{X} \circ \mathcal{P}) \\
(\mathcal{X} \circ \mathcal{P}) x & = \mathcal{P}(\mathcal{X} \varphi = x)
\end{align*}
\]

From the probability axioms, \((\mathcal{X} \circ \mathcal{P}) x \geq 0\) and \( \Sigma x : \Delta(\mathcal{X} \circ \mathcal{P}) : (\mathcal{X} \circ \mathcal{P}) x = 1 \). Given \( \mathcal{P} \) and \( \mathcal{X} \), the probability density function \( \mathcal{X} \circ \mathcal{P} \) thus uniquely characterizes a probability distribution over \( \Delta(\mathcal{X} \circ \mathcal{P}) \).

Whenever \( x \) is not included in \( \mathcal{X} D \), the event \( (\mathcal{X} \varphi = x) \) is the null event, so we have \((\mathcal{X} \circ \mathcal{P}) x = 0\) from the second axiom. This can be reflected by the fact that the domain of a probability distribution may freely include or exclude any element with probability zero; although we only require that the range of \( \mathcal{X} \) be included in the domain of \( \mathcal{X} \circ \mathcal{P} \) in the first axiom, \( \mathcal{X} \circ \mathcal{P} \) does in fact characterize a unique probability distribution no matter what its domain is (as long as it includes \( \mathcal{X} D \)).

**Example 2.5.** Consider (again) the probability experiment in Ex. 2.1, where we roll a pair of fair dice, and let \( [x; y] \) represent the outcome. The probability density function of the random variable \( x + y \) may be written as

\[
(x + y) \circ \mathcal{P} = \lambda n : 2..12 \cdot (6 - \text{abs}(7 - n))/36
\]

It may also be written equivalently as

\[
(x + y) \circ \mathcal{P} = \lambda n : \text{nat} \cdot \text{if } 2 \leq n \leq 12 \text{ then } (6 - \text{abs}(7 - n))/36 \text{ else } 0
\]

Note that in the above example, \( (x + y) \) is just an abbreviation; \( x \) and \( y \) are not global variables. Conventionally, one may also abbreviate the probability density function \( \mathcal{X} \circ \mathcal{P} \) by writing the random variable as the subscript of the probability distribution, as \( \mathcal{P}_\mathcal{X} \). Another convention is to write \( \mathcal{X} = x \) as an abbreviation of the event \( (\mathcal{X} \varphi = x) \). So one may write

\[
\mathcal{P}_\mathcal{X} x = \mathcal{P}(\mathcal{X} = x)
\]

Given probability distribution \( \mathcal{P} \) and random variable \( \mathcal{X} \), the expected value of \( X \) can be equivalently defined

\[
\epsilon \mathcal{P} \mathcal{X} = \Sigma x : \mathcal{X} D \cdot x \times (\mathcal{X} \circ \mathcal{P}) x
\]
Chapter 3

The probabilistic model of specifications

In this chapter, we present an extension of Hehner’s theory to incorporate probability into specifications. We discuss the probabilistic model of computation, and present a formalism for specifying probabilistic computational behavior. We then present a general form of specifications that allows us to combine standard specifications and probabilistic specifications freely at the same time. We introduce some notations for probabilistic specifications, some of them redefined from the standard model. We present some laws for these notations; in addition, we define a class of specifications that has some additional properties regarding these notations. We also discuss the issue of nondeterminism, and explain why it is accounted dissimilarly for specifications and for computations.

3.1 The model of computation

When we execute a program (probabilistic or not), we provide a prestate $\sigma$ as input, and the execution delivers a poststate $\sigma'$ as output; in this respect a probabilistic program is the same as a standard program. However, a probabilistic program differs from a standard program in that the poststate delivered may depend upon probabilistic choices made during the course of the computation. When a computation makes a probabilistic choice between two or more alternatives, we can depict the likelihood or probability of each alternative, but we cannot say for sure which alternative is selected (unless the probability is 1).

Given a prestate as input, a computation can be considered a probability experiment, where the poststate is the outcome of the experiment. The state space $D_\sigma$ is the bunch of all possible outcomes. As in conventional probability theory, to talk about probabilities in the experiment, we describe a probability distribution (Def. 2.1) over all possible
outcomes. Since the outcome is the poststate $\sigma'$, we call the probability distribution the \textit{poststate distribution}, denoted $P'$, a probability distribution over the state space $D_\sigma$.

In our probabilistic model of computation, the input is still the prestate $\sigma$ as in the standard model, but we consider the "output" to be the poststate distribution $P'$ instead of the poststate $\sigma'$. This is justified because our interest is the probability of each poststate in a computation, not the actual poststate delivered in an individual execution; the latter can be regarded as the outcome of an individual trial of the experiment described by $P'$. Note here we distinguish between the use of the words "output" and "outcome": one can write specifications about the poststate distribution $P'$ (the "output" in our model of computation) to describe a probability experiment, but one cannot specify which "outcome" is to be the result from the experiment.

If a computation is nondeterministic in the standard model, we do not know which poststate (chosen from a bunch of possible poststates) is delivered. If the meaning of "determinism" is taken from the standard model, a probabilistic computation may well be considered nondeterministic. However, in our probabilistic model, our meaning of "determinism" differs from the standard model in that it is not based on the poststate delivered, but rather on the probability of each poststate; in other words, it is based on the poststate distribution. In this sense, probabilistic choice is deterministic, since the probability of each alternative is known. Nondeterminism in the probabilistic model thus applies to a computation that is not only nondeterministic in the poststate delivered, but also in the poststate distribution. The modeling of computation as a probability experiment applies only to probabilistic programs that are deterministic. If nondeterministic computation is allowed in the probabilistic model, it can be considered a nondeterministic choice of probability experiments (given a prestate as input); the choice may be made during an execution, and may vary depending on the prestate (this is called \textit{runtime} nondeterminism).

To talk about time, we simply add a time variable $t$ to the state as in the standard model. With regards to time, no extra consideration is necessary for the probabilistic model.

\textbf{3.2 Probabilistic specifications}

A specification is a denotation of observable computational behavior. In the standard model, an observation is taken from an individual execution of the specification's implementation, where the prestate and the poststate can be observed. In the probabilistic model, we are interested in the probability of each poststate, which can be described by
the poststate distribution. In reality, we can never "observe" probabilities, but we can execute an implementation repeatedly (given the same prestate each execution), and observe the relative frequency of each poststate. If we repeat the executions over and over ad infinitum, the limit of this relative frequency will be the probability of the poststate.\(^1\)

So a probabilistic specification can be regarded as a denotation of computational behavior based on "observations" each taken from an infinite number of executions: here an "observation" is not meant to be realistic, but hypothetical. Practically, it can be used as the hypothesis in a statistical test, against which a realistic observation (with respect to the relative frequency of each poststate) taken from a finite number of executions (on the same prestate) can be tested.

### 3.2.1 Definitions

In our model of computation, the input is the prestate \(\sigma\), and the output is the poststate distribution \(\mathcal{P}'\). A probabilistic specification is thus a boolean expression whose variables are the prestate \(\sigma\) and the poststate distribution \(\mathcal{P}'\).

Hereafter, we sometimes just say "specification" to mean a probabilistic specification, and we say "standard specification" to refer to a specification as defined in the standard model (i.e. a boolean expression of the prestate and the poststate).

As in the standard model, we use variables such as \(x, y\) to refer to the prestate \(\sigma\), and variables such as \(x', y'\) to refer to the poststate \(\sigma'\). Note that \(\sigma'\) is not a variable of a probabilistic specification. Whenever \(\sigma'\) is mentioned, its scope must be local to some expression in the specification, such as in an event expression supplied to \(\mathcal{P}'\) as argument.

Here are some definitions, which are very similar to their standard counterparts in Def. A.1.

**Definition 3.1.** For probabilistic specification \(S\),

\[
\begin{align*}
S \text{ is satisfiable for prestate } \sigma & : \exists \mathcal{P}' \cdot S \\
S \text{ is deterministic for prestate } \sigma & : \phi(\langle \mathcal{P}' \cdot S = 1 \rangle) \\
S \text{ is nondeterministic for prestate } \sigma & : \phi(\langle \mathcal{P}' \cdot S \geq 2 \rangle) \\
S \text{ is implementable: } & \forall \sigma \cdot \exists \mathcal{P}' \cdot S
\end{align*}
\]

In the examples below, let \(\sigma = [x; y], D_x = D_y = \text{nat.}\)

\(^1\)This assumes that the poststate distribution is constant over repeated executions, which is only true for a deterministic implementation. If we allow nondeterministic choice, the poststate distribution may differ from execution to execution, though the relative frequency of each poststate over repeated executions can still be observed. The limit of this relative frequency as the number of executions goes to infinity may not exist, but its lower and upper bounds will be given by the poststate distributions that may be nondeterministically chosen: the lower bound will be the minimum probability of the poststate among all possible poststate distributions, while the upper bound will be the maximum.
Example 3.1. The specification

\[ P'(x' = y' = 0) = P'(x' = y' = 1) = 1/2 \]

says that under the poststate distribution \( P' \), the probabilities of both the elementary events \( x' = y' = 0 \) and \( x' = y' = 1 \) are 1/2. This implies that the probability of any other elementary event is 0. A computation must therefore set \( x \) and \( y \) to either 0 or 1 (the same value for both variables), with equal probability (of 1/2) for both values. The specification is deterministic for every prestate. \( \diamond \)

Example 3.2. The specification

\[ P'(y' = 0 \lor y' = 1) = 1/3 \]

says that the probability of the compound event \( y' = 0 \lor y' = 1 \) is exactly 1/3. Since \( P'(y' = 0 \lor y' = 1) \) is equal to \( P'(y' = 0) + P'(y' = 1) \), the specification can be satisfied by a computation that sets \( y \) to any value, so that the probabilities for the values 0 and 1 sum to 1/3, and the probabilities for other values sum to 2/3; it may set \( x \) to any value. The specification is nondeterministic for every prestate. \( \diamond \)

In the above examples, the poststate distribution does not depend on the prestate; here is an example where it does.

Example 3.3. Consider the specification

\[ P'(y' = x + 1 \land x' = y) \geq 3/4 \]

In this specification, \( x, \ y \) are global variables, but \( x', \ y' \) are local to the event expression \( y' = x + 1 \land x' = y \), which is the bunch \( [y; x + 1] \) (an elementary event that depends on the prestate). Since the order of the variables in \( \sigma \) is given, the above specification can be equivalently written as

\[ P'[y; x + 1] \geq 3/4 \]

but it may be confusing what the expressions \( x + 1 \) and \( y \) refer to. The event expression notation \( P'(b) \) is preferred because it is clearer in meaning, and its semantics is independent of the order of variables in \( \sigma \). If we conform to the event expression notation in the whole specification, the order of variables in \( \sigma \) can be neglected (though the variable names and their domains must be explicitly stated). There is an even more important advantage of event expressions, which will be discussed in Section 3.2.2.

We can consider \( y' = x + 1 \land x' = y \) to be a standard specification. In other words, the above specification says that given any prestate, the computation must satisfy the standard specification \( y' = x + 1 \land x' = y \) with probability at least 3/4. \( \diamond \)

In general, for standard specification \( S \), the probabilistic specification

\[ P'(S) \geq p \]

says that given any prestate, the computation must satisfy \( S \) with probability at least \( p \). In particular, for \( p = 1 \), the computation must always satisfy \( S \). In the standard model,
where no probability is involved, if an implementation of $S$ is repeatedly executed, it will always satisfy $S$. Therefore, any specification $S$ in the standard model can be converted to the specification $P'_1(S) = 1$ in the probabilistic model (this will be discussed more in Section 3.3).

All the above examples are implementable. Here is an example of an unimplementable specification.

**Example 3.4.** The specification

$$P'(x = 0 \land x' = y' = 0) = 1/2$$

says that the probability of the event $\langle x = 0 \land x' = y' = 0 \rangle$ is exactly $1/2$. This cannot be satisfied by any computation that starts with $x \neq 0$, since if $x \neq 0$ the event $\langle x = 0 \land x' = y' = 0 \rangle$ can never occur. We may also see this formally as follows:

\[
\begin{align*}
P'(x = 0 \land x' = y' = 0) & = 1/2 \\
= P'(if \ x = 0 \ then \ x' = y' = 0 \ else \bot) & = 1/2 \\
= P'(if \ x = 0 \ then \ x' = y' = 0 \ else \langle \bot \rangle) & = 1/2 \\
= (if \ x = 0 \ then \ P'(x' = y' = 0) \ else \ P'(\bot)) & = 1/2 \\
= if \ x = 0 \ then \ P'(x' = y' = 0) & = 1/2 \ else \ 0 = 1/2 \\
= x = 0 \land P'(x' = y' = 0) & = 1/2
\end{align*}
\]

This specification is unsatisfiable for any prestate where $x \neq 0$, hence unimplementable. Note if we change the probability in the specification from $1/2$ to $0$, we have an implementable specification, since with a similar reasoning, we have

\[
\begin{align*}
P'(x = 0 \land x' = y' = 0) & = 0 \iff if \ x = 0 \ then \ P'(x' = y' = 0) = 0 \ else \ T
\end{align*}
\]

which is satisfiable for any prestate. So an unimplementable standard specification may be implementable if its probability of being satisfied is $0$ (though it is not always the case), but it is always unimplementable if its probability of being satisfied is greater than $0$ (since for at least one prestate, we have $P'(\bot) > 0$ contradicting the probability axioms).

In the previous examples, any expression of the form $P'(S)$ in a specification involves standard specification $S$, whose variables are the prestate $\sigma$ and the poststate $\sigma'$. However, since the global variables of a probabilistic specification are the prestate $\sigma$ and the poststate distribution $P'$, and the poststate $\sigma'$ is introduced in the event expression $\langle S \rangle$ as a local variable, there is no reason that the other variable, the poststate distribution $P'$, cannot be a variable of $S$. Indeed it can, as shown in the following example.

**Example 3.5.** The specification

$$P'(x' = 1 \land P'(x' = y' = 1) = 1/3) = 1/2$$

says that the probability that the event $\langle x' = 1 \land P'(x' = y' = 1) = 1/3 \rangle$ happens is $1/2$. Part of this event says that the probability of the event $\langle x' = y' = 1 \rangle$ is $1/3$. Logically, we see that the outer event can be satisfied by a poststate only if the probability of the
inner event is 1/3, in which case the outer event equals \( x' = 1 \); if the probability of the inner event is any other value, no poststate satisfies the outer event, so it is equal to \( \bot \).

Formally, we have

\[
P'(x' = 1 \land P'(x' = y' = 1) = 1/3) = 1/2
\]

\[
P'(\text{if } P'(x' = y' = 1) = 1/3 \text{ then } x' = 1 \text{ else } \bot) = 1/2
\]

\[
P'(\text{if } P'(x' = y' = 1) = 1/3 \text{ then } (x = 1) \text{ else } (\bot)) = 1/2
\]

\[
(\text{if } P'(x' = y' = 1) = 1/3 \text{ then } P'(x = 1) \text{ else } P'(\bot)) = 1/2
\]

\[
P'(x' = y' = 1) = 1/3 \land P'(x' = 1) = 1/2
\]

Note that in the above, the distributive laws for conditional composition can be applied because the variables \( x' \) and \( y' \) in \( (x' = y' = 1) \) are local to the inner event expression, so they are not variables of the expression \( P'(x' = y' = 1) = 1/3 \).

In general, the variables of an event expression in a probabilistic specification may include the prestate \( \sigma \), the poststate \( \sigma' \) and the poststate distribution \( P' \).

### 3.2.2 Adding variables to the state

Given the state \( \sigma \), it may contain many variables, of which only a few are used in a specification (which means its satisfaction depends solely on the values of these variables). In the standard model, we may confine the state to include only the used variables, and do not have to concern ourselves about any unused variables (which means their values do not affect the satisfaction of the specification); on the other hand, we may consider any additional variables to be included in the state (which may be used as auxiliary variables in program development), without the need to modify the specification. In the probabilistic model, since the poststate distribution \( P' \) is a variable of a specification, and all state variables must be contained in the (list) variable \( \sigma' \) of the function \( P' \), extra consideration seems to be necessary.

Let \( S \) be a probabilistic specification, and let \( \sigma \) be the state for \( S \). Suppose we want to add unused variables to the state, collectively written as a list \( \zeta \), so the new state becomes \( \sigma + \zeta \). Let \( R \) be the corresponding probabilistic specification with the added unused (list) variable. Now, we must define what is meant by an "unused" variable in \( R \). For the prestate, it has the same meaning as the standard model, that is, the value of \( \zeta \) does not affect the satisfaction of \( R \), and the value of \( \sigma \) affects the satisfaction of \( R \) in the same way as it affects the satisfaction of \( S \). For the poststate, it means that the satisfaction of \( R \) is affected by the probabilities of the events that mention \( \sigma' \) only; or formally, we define a function

\[
\mathcal{V} = \lambda \sigma' + \zeta' \cdot \sigma'
\]
and the satisfaction of $R$ (whose poststate is $\sigma' + \varsigma'$) relies on the probability density function

$$\mathcal{Y} \circ \mathcal{P}' = \lambda \sigma'_0 \cdot \mathcal{P}'(\sigma' = \sigma'_0)$$

in the sense that any two poststate distributions with the same probability density function of $\mathcal{Y}$ affect the satisfaction of $R$ in the same way. For any $\sigma'_0$, $(\mathcal{Y} \circ \mathcal{P}')\sigma'_0$ in $R$ is the probability of the event that the value of $\sigma'$ in the poststate $\sigma' + \varsigma'$ is $\sigma'_0$; this equals the probability $\mathcal{P}'\sigma'_0$ in $S$ (whose poststate is $\sigma'$ only). Since the satisfaction of $R$ is only affected by the prestate variable $\sigma$ and the probability density function $\mathcal{Y} \circ \mathcal{P}'$, any prestate $\sigma_0 + \varsigma_0$ and poststate distribution $\mathcal{P}'_0$ satisfy $R$ if and only if prestate $\sigma_0$ and poststate distribution $\mathcal{Y} \circ \mathcal{P}'_0$ satisfy $S$. In other words, since we can just ignore the unused prestate variable $\varsigma$, $R$ is simply the same expression as $S$, with $\mathcal{P}'$ substituted by $\mathcal{Y} \circ \mathcal{P}'$.

If we conform to the event expression notation in the specification $S$, it is not necessary to convert $\mathcal{P}'$ to $\mathcal{Y} \circ \mathcal{P}'$ when we add unused variables to the state. To see this, suppose we have any expression involving $\mathcal{P}'(b)$ in $S$, where $b$ is any boolean expression of $\sigma$, $\sigma'$, and $\mathcal{P}'$. If we use the same expression $\mathcal{P}'(b)$ in $R$, the variable of the event expression becomes $\sigma' + \varsigma'$, where $\varsigma'$ is not mentioned in $b$. We can thus partition the event $\langle b \rangle$ into disjoint events, so that each event corresponds to a particular value of $\sigma'$ satisfying $b$ (and contains all values of $\varsigma'$); the probability of $\langle b \rangle$ is then the sum of the probabilities of the disjoint events. We have

$$\mathcal{P}'(b) = \sum \sigma'_0 : \langle \sigma \cdot b \rangle \cdot \mathcal{P}'(\sigma' = \sigma'_0)$$

$$= \sum \sigma'_0 : \langle \sigma \cdot b \rangle \cdot \mathcal{P}'(\mathcal{Y}(\sigma' + \varsigma') = \sigma'_0)$$

$$= \sum \sigma'_0 : \langle \sigma \cdot b \rangle \cdot (\mathcal{Y} \circ \mathcal{P}')\sigma'_0$$

$$= (\mathcal{Y} \circ \mathcal{P}')(b)$$

As we have shown, writing $\mathcal{P}'(b)$ is the same as writing $(\mathcal{Y} \circ \mathcal{P}')(b)$ in $R$. As in the case of standard specifications, there is no need to rewrite a probabilistic specification when unused variables are added to or removed from the state, so long as event expressions are used. As we will see, this proves to be very handy in many situations, such as the addition of a time variable to a specification, or the use of auxiliary variables in program development.

**Example 3.6.** Consider the specification $\mathcal{P}'(x' = y \land y' > 0) = 1/2$, whose state consists of integer variables $x$ and $y$. Now we add a new integer variable $z$ to the state (so the new state consists of $x$, $y$ and $z$), and the (list) variable of $\mathcal{P}'$ changes from $[x'; y']$ to $[x'; y'; z']$. Suppose we want to write a specification saying that the probability of the event $\langle x' = y \land y' > 0 \rangle$, under the new poststate distribution, is $1/2$. The specification is still $\mathcal{P}'(x' = y \land y' > 0) = 1/2$; we do not have to change anything.
3.2.3 Specifications with time and worst-execution time bounds

To talk about time, we use the variables \( t \) and \( t' \) as in the standard model. Like other variables in the poststate, \( t' \) is not a variable in a probabilistic specification, but it can appear within the local scope of some expression.

**Example 3.7.** The specification with time

\[
P'(x' = 0) \geq 9/10 \land P'(t' \leq t + x^2) = 1
\]

says that the computation sets \( x \) to 0 with probability at least 9/10, and always does so within an execution time of at most \( x^2 \).

In general, if a specification with time implies \( P'(t' \leq t + f\sigma) = 1 \) for some nonnegative function \( f \), we say that the specification has a worst-execution time bound of \( f\sigma \) for prestate \( \sigma \). (If \( f\sigma = \infty \) for prestate \( \sigma \), the worst-execution time is unbounded for \( \sigma \).)

As in standard specifications, time cannot decrease:

**Definition 3.2.** A probabilistic specification with time \( S \) is implementable if and only if

\[
\forall \sigma: \exists P'. S \land P'(t' \geq t) = 1
\]

3.2.4 Expected values and expected time bounds

We can talk about expected values in specifications. Given a random variable \( \mathcal{X} \) whose parameter is the poststate \( \sigma' \), the expected value of \( \mathcal{X} \) over \( P' \), \( \mathbb{E}_P'\mathcal{X} \) (Def. 2.3), is a real expression of \( P' \). We can therefore use it naturally in a specification. Since we are concerned with only one probability distribution \( P' \), we use the symbol \( \mathbb{E}' \) to represent \( \mathbb{E}_P' \), the expectation function for \( P' \). The random variable \( \mathcal{X} \) can be any function whose parameter is the poststate \( \sigma' \). A very useful random variable is \( \mathcal{X}_{t'} = D \cdot t' - t \), or simply abbreviated as \( t' - t \), which enables us to talk about expected execution time.

However, there is a point of concern. Since the domain of \( t \) includes \( \infty \), the range of \( t' - t \) is \( \mathbbm{xnat} \), the extended naturals. If we write

\[
\mathbb{E}'(t' - t) = \sum_{t_0 : \mathbbm{xnat}} t_0 \cdot t_0 \times P'(t' - t = t_0)
\]

the summation is undefined if \( P'(t' - t = \infty) = 0 \), since \( \infty \times 0 \) is undefined. Since any event that occurs with zero probability can be excluded from the sample space of a probability distribution, we do not want such event to affect the expected value of any random variable, even if the random variable evaluates to \( \infty \) for the event. So in the summation, we consider \( t_0 \times P'(t' - t = t_0) \) to be 0 whenever \( P'(t' - t = t_0) = 0 \), even if \( t_0 = \infty \). In other words, we may write

\[
\mathbb{E}'(t' - t) = \text{if } P'(t' - t = \infty) > 0 \text{ then } \infty \text{ else } \sum_{t_0 : \mathbbm{nat}} t_0 \cdot t_0 \times P'(t' - t = t_0)
\]
Example 3.8. The specification with time
\[ \mathcal{P}'(x' = 0) = 1 \land \mathcal{E}'(t' - t) \leq x^2 \]
says that the computation always sets \( x \) to 0, and does so within an expected execution time of at most \( x^2 \).

In general, if a specification with time implies \( \mathcal{E}'(t' - t) \leq f \sigma \) for some nonnegative function \( f \), we say that the specification has an \textit{expected time bound} of \( f \sigma \) for prestate \( \sigma \). (If \( f \sigma = \infty \) for prestate \( \sigma \), the expected time is unbounded for \( \sigma \).) Note that a (finite) expected time bound implies \( \mathcal{P}'(t' - t = \infty) = 0 \).

If a specification has a worst-execution time bound, it has an expected time bound which is less than or equal to the worst-execution time bound; however, if a specification has an expected time bound, the worst-execution time may be unbounded.

3.2.5 Specification abbreviations

Very often, part or all of a probabilistic specification relates to behavior which is always satisfied by a computation. When we speak of a specification in English, saying "such is always true" is the same as saying "such is true with probability 1", but for brevity we often prefer to say the former, or even simply "such is true" when the context is clear that "such" is meant to be always true. In view of this, we introduce two abbreviations for the expression \( \mathcal{P}'(S) = 1 \) (read "\( S \) is true with probability 1").

First, \( \mathcal{P}'(S) = 1 \) can be abbreviated by the expression \( (S)_a \), where the subscript \( a \) stands for "always". For clarity, this abbreviation should only be used when the event is expressed in event expression form (as above), and not in bunch form. The abbreviation \( (S)_a \) can be read "\( S \) is always true".

In our second abbreviation, we just write \( S \) for \( \mathcal{P}'(S) = 1 \). Intuitively, by using the abbreviation we just say "\( S \) is true" to mean "\( S \) is always true". However, unrestricted use of this abbreviation may cause ambiguity just as simply saying "\( S \) is true" may be ambiguous. For example, for standard specifications \( Q \) and \( R \), if we write the disjunction \( Q \lor R \) as an abbreviated specification, we cannot determine by context whether it abbreviates the specification \( (Q \lor R)_a \) or the specification \( (Q)_a \lor (R)_a \), as the two are not equivalent (the former specification is weaker than the latter). Likewise, we have a similar ambiguity for the negation \( \neg Q \) (the specification \( (\neg Q)_a \) is stronger than the specification \( \neg (Q)_a \)). On the other hand, it is always safe to write the conjunction \( Q \land R \) as an abbreviated specification, since the specifications \( (Q \land R)_a \) and \( (Q)_a \land (R)_a \) are equivalent.
To avoid any ambiguity, we take this abbreviation to be applied only for a probabilistic specification as a whole, and never for individual sub-expressions of a specification. So for the above, the abbreviated specification \( Q \lor R \) stands for \( \langle Q \lor R \rangle_a \), and not \( \langle Q \rangle_a \lor \langle R \rangle_a \); likewise, the abbreviated specification \( \neg Q \) stands for \( \langle \neg Q \rangle_a \), and not \( \neg \langle Q \rangle_a \). This interpretation also reflects the common English usage, as for example, when we say "Q is not true", we usually mean to say "Q is always not true", rather than "Q is not always true".

3.3 General specifications

The use of the second abbreviation in Section 3.2.5 is not necessarily restricted to those specifications of the form \( \mathcal{P}'(S) = 1 \) where \( S \) is a standard specification. Since the variables in an event expression may also contain the poststate distribution \( \mathcal{P}' \) (see Ex. 3.5), the abbreviation also enables us to write general specifications, defined as follows.

**Definition 3.3.** A general specification \( S \) is a boolean expression whose variables are the prestate \( \sigma \), the poststate \( \sigma' \), and the poststate distribution \( \mathcal{P}' \). \( S \) represents (abbreviates) the probabilistic specification
\[
\mathcal{P}'(S) = 1
\]

Although one might suspect that the definition of a "general" specification is contradictory to the word's meaning (since seemingly, it only represents a particular type of probabilistic specifications), we can see that every probabilistic specification can be written in the form above. For any probabilistic specification \( S \), the poststate \( \sigma' \) is not a variable of \( S \); so we have the following:

\[
\begin{align*}
\mathcal{P}'(S) &= 1 \\
&= \mathcal{P}'(\sigma : D_\sigma \cdot S) = 1 \\
&= \mathcal{P}'(\text{if } S \text{ then } D_\sigma \text{ else } \text{null}) = 1 \\
&= \mathcal{P}'(\text{if } S \text{ then } \langle T \rangle \text{ else } \langle \bot \rangle) = 1 \\
&= (\text{if } S \text{ then } \mathcal{P}'(T) \text{ else } \mathcal{P}'(\bot)) = 1 \\
&= (\text{if } S \text{ then } 1 \text{ else } 0) = 1 \\
&= S
\end{align*}
\]

Thus any probabilistic specification \( S \) is equal to \( \mathcal{P}'(S) = 1 \), or \( \langle S \rangle_a \), which can be represented by the general specification \( S \).

Since a general specification may contain all of the prestate, the poststate and the poststate distribution as variables, in some sense it is more general than a probabilistic specification, whose variables must not include the poststate (though they have actually the same expressiveness). Moreover, since any probabilistic specification \( S \) is represented
by the general specification $S$ itself, it is not necessary to distinguish between general specifications and probabilistic specifications when we write a specification; however, as far as semantics is concerned, a general specification that mentions the poststate must be converted to the corresponding probabilistic specification (if it does not mention the poststate, it converts to itself). (Later, in Section 4.1.2, we will see that this conversion is not necessary for proving refinements.)

Hereafter, we sometimes just say "specification" to mean a general specification; we say "probabilistic specification" to refer to a specification that does not contain the poststate as variable.

**Example 3.9.** For integer variables $x$ and $y$:

$$
\begin{align*}
x' &= 1 \vee (P'(y' = 2) = 2/5 \land x' = 2) \vee x' = 3 \\
\iff (x' = 1 \vee (P'(y' = 2) = 2/5 \land x' = 2) \vee x' = 3)_a \\
\iff \text{if } P'(y' = 2) = 2/5 \text{ then } (x' = 1 \lor x' = 2 \lor x' = 3)_a \text{ else } (x' = 1 \lor x' = 3)_a \\
\iff (x' = 1 \lor x' = 2 \lor x' = 3)_a \land (P'(x' = 2) > 0 \Rightarrow P'(y' = 2) = 2/5)
\end{align*}
$$

This specification can be satisfied by a computation that always sets the value of $x$ to one of 1, 2 or 3, and in case $x$ is set to 2 with non-zero probability, $y$ is set to 2 with probability $2/5$.  

A general specification is just an abbreviation for a probabilistic specification, and as discussed before, the abbreviation only applies to a specification as a whole, and not for individual sub-expressions. When we write a general specification, we must be careful of its semantics. As a rule, we should avoid the composition of standard specifications and probabilistic specifications in a disjunction (as in the above example). On the other hand, we can freely compose standard specifications and probabilistic specifications in a conjunction, since the general specifications $Q \land R$, $Q \land \langle R \rangle_a$ and $\langle Q \rangle_a \land \langle R \rangle_a$ are all the same as the probabilistic specification $\langle Q \land R \rangle_a$; this is the most frequent use of general specifications.

### 3.4 Specification notations

As in standard specifications, there are no restrictions on the mathematical notations available for general specifications. Here we redefine some of the basic standard notations introduced in Section A.3; although these notations are redefined, as we will see, they are just extensions of their standard definitions. Moreover, we invent two new notations for probabilistic choice.
CHAPTER 3. THE PROBABILISTIC MODEL OF SPECIFICATIONS

3.4.1 The specifications ok, $x := e$ and if $b$ then $R$ else $S$

These notations do not need to be redefined, and have the same definitions as in the standard model. Note that the standard definitions of the specifications ok and $x := e$ include the poststate as variable. When any of them is the whole specification, it represents its corresponding probabilistic specification (i.e. $(ok)_a$ or $(x := e)_a$), meaning that it can be satisfied by a computation that always behaves accordingly.

If $b$ is a boolean expression of the prestate, the specification if $b$ then $R$ else $S$ includes the poststate as variable if $R$ or $S$ does. When it is the whole specification, it represents $(if b then R else S)_a$, which is equivalent to $(if b then (R)_a else (S)_a$, so it can be satisfied by a computation that always behaves according to $R$ if $b$ is true, or always according to $S$ otherwise.

When any of these notations appears as a sub-expression of a specification, the whole specification (and not individual sub-expressions) must be considered a general specification. For example, the specification $\neg x := 2 \lor ok$ represents $(\neg x := 2 \lor ok)_a$, not $(\neg (x := 2)_a \lor (ok)_a$. If a sub-expression is indeed intended to be an abbreviation itself, the abbreviation must be explicitly converted, either by the specifier or in the definition of a notation (as in some of our notations below).

3.4.2 The specification $R \ p\gamma S$ $(R \ p\or S)$

$$R \ p\gamma S = \exists \mathcal{R}, \mathcal{S}. \quad (p > 0 \Rightarrow (\text{substitute } \mathcal{R} \text{ for } \mathcal{P}' \text{ in } (R)_a))$$
$$\land (p < 1 \Rightarrow (\text{substitute } \mathcal{S} \text{ for } \mathcal{P}' \text{ in } (S)_a))$$
$$\land \mathcal{P}' = \mathcal{R} \ p\oplus S$$

The above notation is called $p$-probabilistic composition. $p$ is a probability-valued expression ($p: 0, 1$), which may depend on the prestate (if $p$ is not in $0, 1$ for a prestate, the specification is unsatisfiable for the prestate). The existential variables $\mathcal{R}$ and $\mathcal{S}$ are probability distributions over the state space. We have not defined the expression $\mathcal{R} \ p\oplus S$, so we do it here.

**Definition 3.4.** Let $\mathcal{P}, \mathcal{Q}$ be probability distributions over the same sample space; let $p$ be a probability-valued expression. The $p$-probabilistic sum of $\mathcal{P}$ and $\mathcal{Q}$, $\mathcal{P} \ p\oplus \mathcal{Q}$ is defined as follows:

$$\mathcal{P} \ p\oplus \mathcal{Q} = p \times \mathcal{P} + (1 - p) \times \mathcal{Q}$$

It can be seen that $\mathcal{P} \ p\oplus \mathcal{Q}$ is a probability distribution. $\square$

Given the probabilistic distributions $\mathcal{P}$ and $\mathcal{Q}$, $\mathcal{P} \ p\oplus \mathcal{Q}$ describes a probability experiment constructed from the two experiments described by $\mathcal{P}$ and $\mathcal{Q}$, so that it makes a
probabilistic choice between the experiment described by $\mathcal{P}$, with probability $p$ of being selected; and the experiment described by $\mathcal{Q}$, with probability $1-p$ of being selected.\footnote{This can be reasoned as follows. For any event $(b)$, $\mathcal{P}(b)$ is the (conditional) probability of $(b)$ given that the experiment described by $\mathcal{P}$ is selected, and $\mathcal{Q}(b)$ is the (conditional) probability of $(b)$ given that the experiment described by $\mathcal{Q}$ is selected. So the probability of $(b)$ under the constructed experiment is given by $$(\mathcal{P} \oplus \mathcal{Q})(b) = p \times \mathcal{P}(b) + (1-p) \times \mathcal{Q}(b)$$ Since this must be true for every event, our definition for $\mathcal{P} \oplus \mathcal{Q}$ does characterize probabilistic choice as desired.}

We now turn back to the specification $R \mathcal{P} \mathcal{Y} S$. If $R$ and $S$ are implementable specifications, it can be satisfied by a computation that makes a probabilistic choice between the behavior according to $R$ and the behavior according to $S$; they have probabilities $p$ and $1-p$ respectively of being selected. In the notation, $R$ and $S$ may mention the poststate; by using $\langle R \rangle_a$ and $\langle S \rangle_a$ in our definition, they are converted to probabilistic specifications (Section 3.3). Any occurrences of $\mathcal{P}'$ in $\langle R \rangle_a$ and $\langle S \rangle_a$ are then renamed to $\mathcal{R}$ and $\mathcal{S}$ respectively (note that $\langle R \rangle_a$ is an abbreviation for $\mathcal{P}'(R) = 1$ and this $\mathcal{P}'$ must also be renamed), so that they can be used to construct $\mathcal{R} \mathcal{P} \oplus \mathcal{S}$, the poststate distribution for the specification $R \mathcal{P} \mathcal{Y} S$.

If either $R$ or $S$ is unsatisfiable for a prestate (thus unimplementable), so is $R \mathcal{P} \mathcal{Y} S$ for that prestate, since the computation cannot choose, not even probabilistically, to behave according to $R$ (or $S$) if no such behavior is possible. There is an exception to this when $p = 1$ or $p = 0$: when $p = 1$, if no behavior according to $S$ is possible, the computation can still satisfy $R \mathcal{P} \mathcal{Y} S$ by choosing to never behave according to $S$, and always behave according to $R$; in other words, it does not matter what $S$ is, even if $S$ is unimplementable. Similarly when $p = 0$, it does not matter whether $R$ is implementable or not. In view of this, in our definition we require that $S$ be obtained from the renaming of $\mathcal{P}'$ in $S$ only if $p < 1$, so that it can be any probability distribution if $p = 1$ (likewise for $R$ and $p = 0$).\footnote{If we require the renaming of $\mathcal{P}'$ regardless of the value of $p$, the implementability of $R \mathcal{P} \mathcal{Y} S$ will require that $S$ be implementable even if $p = 1$; such a requirement will be too strong that we will have to consider $p = 0$ and $p = 1$ as special cases for some desired laws. Moreover, as illustrated in Ex. 3.4, an unimplementable standard specification may be implementable (only) if its probability of being satisfied is 0; this is consistent with the weaker definition that we use.}

The programming notation $R \mathcal{P} \text{or} S$ is identical to $R \mathcal{P} \mathcal{Y} S$, except for precedence. (It is merely introduced in correspondence with the introduction of the notation $R$ or $S$.)

Example 3.10. For integer variable $x$:

$$\mathcal{P}'(x' = x) = \frac{4}{5} \quad x' = 3$$

$$= \exists \mathcal{R}, \mathcal{S} \cdot \mathcal{R}(x' = x) = \frac{4}{5} \land \mathcal{S}(x' = 3) = 1 \land \mathcal{P}' = \mathcal{R} \mathcal{P} \oplus \mathcal{S}$$

A further simplification of the specification in this example will be shown in Ex. 3.12. \diamond
Laws for $R \triangleright S$

In the definition of $R \triangleright S$, the existential variables $R$ and $S$ are merely introduced to construct $P'$ from the poststate distributions of $R$ and $S$. In order to express the specification in terms of probabilities of events, we may want to eliminate these variables (and the existential quantifier) and simplify the specification. Here we present some laws that can help us to do this as well as to prove other theorems.

In the following, $Q$, $R$, $S$, and $T$ are probabilistic specifications; $U$ and $V$ are standard specifications; $b$ is a boolean expression of the prestate; $p$, $q$, $u$, and $v$ are probability-valued expressions of the prestate; $c$ and $d$ are real expressions of the prestate; $\mathcal{X}$ is a function with domain $D_{\mathcal{X}}$; $\mathcal{R}_x$ and $\mathcal{S}_x$ are probability distributions over the range of $\mathcal{X}$.

\begin{align*}
\text{PC-1} & \quad R \triangleright S = S \setminus (1 - R) \triangleright R \\
\text{PC-2} & \quad (Q \triangleright R) \triangleright S = Q \triangleright (R \triangleright S)) \iff u = pq \land 1 - q = (1 - u)(1 - v) \\
\text{PC-3} & \quad R \triangleright S = R \\
\text{PC-4} & \quad R \triangleright R \iff R \\
\text{PC-5} & \quad (Q \lor R) \triangleright S = (Q \triangleright S) \lor (R \triangleright S) \\
\text{PC-6} & \quad (\text{if } b \text{ then } Q \text{ else } R) \triangleright S = \text{if } b \text{ then } Q \triangleright S \text{ else } R \triangleright S \\
\text{PC-7} & \quad (Q \land R) \triangleright S \Rightarrow Q \triangleright S \land (R \triangleright S) \\
\text{PC-8} & \quad (Q \triangleright R) \triangleright (S \triangleright T) = (Q \triangleright S) \triangleright (R \triangleright T) \\
\text{PC-9} & \quad \mathcal{X} \circ P' = \mathcal{R}_x \triangleright \mathcal{X} \circ P' = \mathcal{S}_x = \mathcal{X} \circ P' = \mathcal{R}_x p \oplus \mathcal{S}_x \\
\text{PC-10} & \quad \mathcal{P}'(U) \geq u \triangleright \mathcal{P}'(V) \geq v = \mathcal{P}'(U) \geq pu \land \mathcal{P}'(V) \geq (1 - p)u \\
& \quad \land \mathcal{P}'(U \lor V) \geq pu + (1 - p)v \\
\text{PC-11} & \quad \mathcal{E}' \mathcal{X} = c \triangleright \mathcal{E}' \mathcal{X} = d \Rightarrow \mathcal{E}' \mathcal{X} = pc + (1 - p)d
\end{align*}

It may be surprising that probability composition is not idempotent (PC-4 does not hold if " \iff " is replaced by " \Rightarrow "). This will be discussed in Section 3.5, where we will establish a class of specifications for which idempotence of probabilistic composition does hold.

Laws PC-10 and PC-11 can be used to eliminate the existential quantifier in the notation. Law PC-10 says that if each alternative of a probabilistic composition describes a probability density function of some given function $\mathcal{X}$ over the state space, the probabilistic composition describes the probabilistic sum of the two probability density functions. In its application, we may not be interested in what the function $\mathcal{X}$ is. The main point is that $\mathcal{X}$ partitions the possible poststates into disjoint events, and for each alternative of the probabilistic composition a probability density function assigns a probability to each partition. The law states that the resultant probability of each partition is the weighted average of the probabilities assigned to the partition in the two alternatives (with weights $p$ and $1 - p$ respectively).
Example 3.11. For natural variable $x$:

$$\begin{align*}
\mathcal{P}'(x' = 0) &= \frac{1}{4} \land \mathcal{P}'(x' \geq 3) = \frac{1}{2} \\
\mathcal{P}'(x' = 0) &= (\frac{1}{3}) + (\frac{3}{5}) \land \mathcal{P}'(x' \geq 3) = (\frac{1}{3}) + (\frac{3}{5}) \\
\mathcal{P}'(x' = 0) &= \frac{1}{2} \land \mathcal{P}'(x' \geq 3) = \frac{1}{4}
\end{align*}$$

In each alternative, a probability is assigned to each of three partitions of the possible poststates: $\langle x' = 0 \rangle$, $\langle x' \geq 3 \rangle$ and $\langle x' = 1 \lor x' = 2 \rangle$; law PC-10 can therefore be applied. (The probability assigned to $\langle x' = 1 \lor x' = 2 \rangle$ is not explicit, but it is implied by the probability axioms.) To see this, we may define the function $X$ to be $\lambda x' \cdot$ if $x' = 0$ then 0 else if $x' \geq 3$ then 1 else 2, but here $X$ merely serves to assign a “label” (0, 1, 2) to each partition, and is rather redundant.

Law PC-10 can only be applied when the partitions with non-zero probabilities are the same in both alternatives and are disjoint (partitions with zero probability can be grouped in any way). Otherwise, to satisfy this requirement we can divide the poststates into further partitions with the use of existential variables for unknown probabilities. We illustrate this in the following example.

Example 3.12. We eliminate the existential quantifier for Ex. 3.10. Law PC-10 cannot be directly applied because the partitions are not the same in both alternatives unless $x = 3$. For $x \neq 3$, we divide the poststates into three disjoint partitions: $\langle x' = 3 \rangle$, $\langle x' = x \rangle$ and $\langle x' \neq 3 \land x' \neq x \rangle$; both alternatives then have the same partitions, and law PC-10 can be applied.

$$\begin{align*}
\text{if } x = 3 \\
\text{then (} & \mathcal{P}'(x' = x) = \frac{4}{5} \land \mathcal{P}'(x' = 3) = \frac{1}{2} \land \mathcal{P}'(x' = 3) = 1 \\
\text{else (} & \mathcal{P}'(x' = x) = \frac{4}{5} \land \mathcal{P}'(x' = 3) = \frac{9}{10} \land \mathcal{P}'(x' = 3) = 1 \\
\text{)}& \exists u \cdot 0 \leq u \leq \frac{1}{5} \land (\mathcal{P}'(x' = x) = \frac{4}{5} \land \mathcal{P}'(x' = 3) = u) \land \mathcal{P}'(x' = 3) = 1 \\
& \exists u \cdot 0 \leq u \leq \frac{1}{2} \land \mathcal{P}'(x' = x) = \frac{2}{5} \land \mathcal{P}'(x' = 3) = (1 + u)/2 \\
& \mathcal{P}'(x' = x) = \frac{2}{5} \land \mathcal{P}'(x' = 3) = \frac{3}{5} \\
& \mathcal{P}'(x' = x) = \frac{2}{5} \land \mathcal{P}'(x' = 3) \geq \frac{1}{2} \land \mathcal{P}'(x' = 3) = \frac{2}{5}
\end{align*}$$

So we have

$$\begin{align*}
\mathcal{P}'(x' = x) &= \frac{4}{5} \land \mathcal{P}'(x' = 3) = \frac{9}{10} \land \mathcal{P}'(x' = x) = \frac{2}{5} \land \mathcal{P}'(x' = 3) \geq \frac{1}{2}
\end{align*}$$

The proof of law PC-10 is presented in Section C.1. With the use of existential variables for unknown probabilities, law PC-11 can be proved from law PC-10, as shown in Section C.2. Although we do not provide the laws for the cases when "≥" in either alternative is replaced with "=", such laws can be derived similarly.
The specification $R_0 \, r_0 \ldots \, r_{n-1} \, Y \, R_n$

We have defined the notation for probabilistic composition of two alternatives. For probabilistic composition of any finite constant number of alternatives, the notation can be extended as follows:

$$R_0 \, r_0 \ldots \, r_{n-1} \, Y \, R_n =$$

$$\exists R_0, \ldots, R_n \cdot \quad r_0 > 0 \Rightarrow \text{(substitute } R_0 \text{ for } P' \text{ in } \langle R_0 \rangle_a)$$

$$\land \ldots$$

$$\land r_{n-1} > 0 \Rightarrow \text{(substitute } R_{n-1} \text{ for } P' \text{ in } \langle R_{n-1} \rangle_a)$$

$$\land r_0 + \ldots + r_{n-1} < 1 \Rightarrow \text{(substitute } R_n \text{ for } P' \text{ in } \langle R_n \rangle_a)$$

$$\land P' = R_0 \, r_0 \oplus \ldots \, r_{n-1} \oplus R_n$$

where

$$R_0 \, r_0 \oplus \ldots \, r_{n-1} \oplus R_n = r_0 \times R_0 + \ldots + r_{n-1} \times R_{n-1} + (1-r_0-\ldots-r_{n-1}) \times R_n$$

In the notation (called $r_0, \ldots, r_{n-1}$-probabilistic composition), for $0 \leq i \leq n-1$, $r_i$ is a probability-valued expression of the prestate, so that $r_0 + \ldots + r_{n-1} \leq 1$ (if this is violated for a prestate, the specification is unsatisfiable for the prestate). The specification says that the computation makes a probabilistic choice between $n+1$ alternatives, so that the behavior according to $R_0$ is selected with probability $r_0$, $R_1$ with probability $r_1$, and so on, down to $R_n$ which is selected with probability $1-r_0-\ldots-r_{n-1}$.

The above definition is equivalent to the following (this holds even if $r_0 = r_1 = 0$):

$$R_0 \, r_0 \Y R_1 \, r_1 \Y R_2 \, r_2 \Y \ldots \, r_{n-1} \Y R_n =$$

$$(R_0 \, r_0/(r_0+r_1) \Y R_1) \, r_0+r_1 \Y R_2 \, r_2 \Y \ldots \, r_{n-1} \Y R_n$$

Here a probabilistic composition of $n+1$ alternatives is reduced to a probabilistic composition of $n$ alternatives (where the first alternative is a probabilistic composition of $R_0$ and $R_1$); thus a probabilistic composition of any finite constant number of alternatives can be defined recursively in terms of probabilistic compositions of two alternatives.

3.4.3 The specification $R \, Y \, S$ ($R$ or $S$)

$$R \, Y \, S = \exists p : 0, \ldots, 1 \cdot R \, p \Y S$$

The above notation is called convex composition or nondeterministic composition. If $R$ and $S$ are implementable specifications, it can be satisfied by a computation that makes a nondeterministic choice between the behavior according to $R$ and the behavior according to $S$. The choice may be made during an execution; it may be probabilistic with any unknown probability, and may vary depending on the prestate.
The programming notation \( R \) or \( S \) is identical to \( R \land S \), except for precedence. Note that the definition of \( R \) or \( S \) here is different from the standard model (which is simply disjunction). This is because for nondeterministic choice, the behavior may be chosen probabilistically. Disjunction of two probabilistic specifications says that given any prestate, the computed poststate distribution must satisfy either specification, but a probabilistic choice between the behavior according to the two specifications does not necessarily satisfy either specification; so disjunction is too strong for our definition (see Section 3.5.3).

Example 3.13. For integer variable \( x \):

\[
\mathcal{P}'(x' = x) = \frac{4}{5} \land x' = 3
\]

\[
= \exists p : 0, 1 \cdot \mathcal{P}'(x' = x) = \frac{4}{5} \land x' = 3
\]

With a reasoning similar to Ex. 3.12, this can be further simplified:

\[
= \exists p : 0, 1 \cdot \begin{cases} 
  \text{if } x = 3 \text{ then } \mathcal{P}'(x' = 3) = 1 - \frac{1}{5}p \\
  \text{else } \mathcal{P}'(x' = x) = \frac{4}{5}p \land \mathcal{P}'(x' = 3) \geq 1 - p 
\end{cases}
\]

\[
= \begin{cases} 
  \text{if } x = 3 \text{ then } \mathcal{P}'(x' = 3) \geq \frac{4}{5} \text{ else } \frac{4}{5} \geq \mathcal{P}'(x' = x) \geq \frac{4}{5}(1 - \mathcal{P}'(x' = 3)) 
\end{cases}
\]

\[
\diamond
\]

Laws for \( R \land S \)

In the following, \( Q, R, \) and \( S \) are probabilistic specifications; \( U \) and \( V \) are standard specifications; \( b \) is a boolean expression of the prestate; \( p \) is a probability-valued expression of the prestate.

\[
\begin{align*}
\text{NC-1} & \quad R \land S = S \land R \\
\text{NC-2} & \quad (Q \land R) \land S = Q \land (R \land S) \\
\text{NC-3} & \quad R \land \bot = R \\
\text{NC-4} & \quad R \land T \\
\text{NC-5} & \quad R \land Q \iff R \lor Q \\
\text{NC-6} & \quad R \land R \iff R \\
\text{NC-7} & \quad (Q \lor R) \land S = (Q \land S) \lor (R \land S) \\
\text{NC-8} & \quad (\text{if } b \text{ then } Q \text{ else } R) \land S = \text{if } b \text{ then } Q \land S \text{ else } R \land S \\
\text{NC-9} & \quad (Q \land R) \land S \Rightarrow (Q \land S) \land (R \land S) \\
\text{NC-10} & \quad (Q \land R)_p \land S \Rightarrow (Q \land S)_p \land (R \land S) \\
\text{NC-11} & \quad (Q \land R) \land S \Rightarrow (Q \land S) \land (R \land S) \\
\text{NC-12} & \quad U \land V = (U \lor V)_a
\end{align*}
\]

Like probabilistic composition, convex composition does not have the idempotence property (NC-6 does not hold if "\( \iff \)" is replaced by "\( \Rightarrow \)"), but in Section 3.5 we will discuss a class of specifications for which the property holds (in fact, as we will see, that class of specifications is defined to have this property).
Law NC-12 relates the definitions of nondeterministic composition in the standard and probabilistic models; it says that if both specifications in a nondeterministic (convex) composition are standard, the resultant is equivalent to the general specification resulting from standard nondeterministic composition (disjunction) of the two standard specifications. Therefore, the definition of nondeterministic composition in the probabilistic model is an extension of its standard definition.

3.4.4 The specification \( Q \triangleright \lambda \varphi \cdot S \)

In Section 3.4.2 probabilistic composition is defined for any finite constant number of alternatives, but the notation cannot be used directly for a countably infinite number of alternatives (e.g. assign \( x \) a natural number so that \( x \) has a geometric probability distribution), or when the number of alternatives may vary depending on the prestate (e.g. assign \( y \) a random integer in \( 0, \ldots, x \)). It is also inconvenient to use when the number of alternatives is large, but the alternatives are similar (e.g. assign \( x \) a random integer in \( 0, \ldots, 100 \)), since formally one has to list all the alternatives and then combine them with the operator \( \triangleright \). Here we invent a notation for probabilistic composition which can be used in these situations.

\[
Q \triangleright \lambda \varphi \cdot S = \exists F : \Delta Q \rightarrow D \rightarrow 0, \ldots, 1 \cdot \\
(\forall \varphi : \Delta Q \cdot Q \varphi > 0 \Rightarrow (\text{substitute } F \varphi \text{ for } P' \text{ in } (S)_a)) \\
\wedge P' = \Sigma \varphi : \Delta Q \cdot Q \varphi \times F \varphi
\]

\( Q \) is a probability distribution, the choice distribution, which may depend on the prestate (if \( Q \) violates the probability axioms for a prestate, the specification is unsatisfiable for the prestate). \( S \) is a specification with an additional variable, \( \varphi \), the choice variable, which is formally introduced by \( \lambda \varphi \cdot S \) to distinguish it from other state variables. When the context is clear, we may omit \( \lambda \varphi \cdot \) and just write \( Q \triangleright S \). Note that from the definition, \( \varphi \) is local to the expression \( Q \triangleright \lambda \varphi \cdot S \). We call the notation probabilistic composition of variable alternatives. As we will see, it is more general than \( R_p \triangleright S \), but the concept is basically the same.

Given any value \( \varphi_0 \) of \( \varphi \), \( S \) with \( \varphi \) substituted by \( \varphi_0 \) (i.e. \( (\lambda \varphi \cdot S)\varphi_0 \)) is a specification. For each value \( \varphi_0 \) of \( \varphi \), \( S \) (with \( \varphi \) substituted by \( \varphi_0 \)) represents an alternative of probabilistic choice, and the behavior according to it is selected with probability \( Q \varphi_0 \).

In the notation, \( F \) is a function that maps each value of \( \varphi \) to a poststate distribution of the corresponding alternative. The use of \( F \) is similar to the use of the existential variables \( R \) and \( S \) in the definition of \( R_p \triangleright S \); the existential quantification of \( F \) serves as (nested) existential quantifications of any number of probability distributions. For
each alternative corresponding to $\varphi$, any occurrence of $\mathcal{P}'$ in $(S)_a$ is renamed to $\mathcal{F}\varphi$ (unless $Q\varphi = 0$, in which case the alternative is never chosen), so that the resultant poststate distribution for the specification can be constructed, in a manner similar to the construction of $\mathcal{P}'$ from $\mathcal{R}$ and $S$ in $R \triangleright S$.

Since $\lambda\varphi \cdot S$ is merely a function, any variable name can be used in place of $\varphi$ as the choice variable. If $\varphi$ is a list, we may use variables to refer to the items in $\varphi$, as in the following example.

**Example 3.14.** For integer variables $x$ and $y$:

$$[2 \ast (1,6)] \to 1/36 \triangleright \lambda [i,j] \cdot x' = x + i \land y' = y + j$$

$$= \forall [i,j] : [2 \ast (1,6)] \cdot \mathcal{P}'(x' = x + i \land y' = y + j) = 1/36$$

This specification says that the behavior is probabilistically chosen as if two fair dice are rolled, and the numbers on the dice are added to $x$ and $y$ respectively. Note that $i$ and $j$ are choice variables local to the specification.

The choice distribution may depend on the prestate, as in the following example.

**Example 3.15.** For natural variable $x$:

$$0,..x \rightarrow 1/x \triangleright \lambda i \cdot x' = i$$

$$= \forall i : 0,..x \cdot \mathcal{P}'(x' = i) = 1/x$$

This specification says that $x$ is assigned a random natural in $0,..x$, with probability $1/x$ for each number. Note that $0,..x \rightarrow 1/x$ is a probability distribution only for positive $x$, so the specification is unsatisfiable for the prestate $x = 0$.

**Laws for $Q \triangleright \lambda \varphi \cdot S$**

In the following, $R$ and $S$ are probabilistic specifications with an additional choice variable $\varphi$ (except for PV-7), and $\mathcal{P}$ and $Q$ are choice distributions; $\mathcal{F}$ is an expression of the prestate whose value is a function from $\Delta Q$ to poststate distributions; $\mathcal{X}$ is an expression of the prestate whose value is a function from $\Delta Q$ to poststates; $p$ is a probability-valued expression of the prestate (excluding $\varphi$); $b$ is a boolean expression of the prestate (excluding $\varphi$).

\begin{align*}
PV-1 & \quad \mathcal{P} \triangleright Q \triangleright \lambda \varphi \cdot S \Rightarrow (\mathcal{P} \triangleright \lambda \varphi \cdot S)_{\mathcal{R}} (Q \triangleright \lambda \varphi \cdot S) \\
PV-2 & \quad \mathcal{P} \triangleright \lambda \varphi \cdot R \lor S \leftarrow (\mathcal{P} \triangleright \lambda \varphi \cdot R) \lor (\mathcal{P} \triangleright \lambda \varphi \cdot S) \\
PV-3 & \quad \mathcal{P} \triangleright \lambda \varphi \cdot \text{if } b \text{ then } R \text{ else } S \leftarrow \text{if } b \text{ then } \mathcal{P} \triangleright \lambda \varphi \cdot R \text{ else } \mathcal{P} \triangleright \lambda \varphi \cdot S \\
PV-4 & \quad \mathcal{P} \triangleright \lambda \varphi \cdot R \land S \Rightarrow (\mathcal{P} \triangleright \lambda \varphi \cdot R) \land (\mathcal{P} \triangleright \lambda \varphi \cdot S) \\
PV-5 & \quad \mathcal{P} \triangleright \lambda \varphi \cdot R_{\mathcal{R}} S \leftarrow (\mathcal{P} \triangleright \lambda \varphi \cdot R)_{\mathcal{R}} (\mathcal{P} \triangleright \lambda \varphi \cdot S) \\
PV-6 & \quad \mathcal{P} \triangleright \lambda \varphi \cdot R_{\mathcal{Y}} S \leftarrow (\mathcal{P} \triangleright \lambda \varphi \cdot R)_{\mathcal{Y}} (\mathcal{P} \triangleright \lambda \varphi \cdot S) \\
PV-7 & \quad R_{\mathcal{R}} S = 0 \rightarrow p \mid 1 \rightarrow 1 - p \triangleright 0 \rightarrow R \mid 1 \rightarrow S
\end{align*}
**PV-8**  \( Q \triangleright \lambda \varphi \cdot P' = \mathcal{F}\varphi = P' = \Sigma \varphi : \Delta Q \cdot Q\varphi \times \mathcal{F}\varphi \)

**PV-9**  \( Q \triangleright \lambda \varphi \cdot \sigma' = \mathcal{X}\varphi = P' = \mathcal{X}oQ \)

Law **PV-7** states that probabilistic composition of two alternatives can be expressed in terms of probabilistic composition of variable alternatives. The proof is presented in Section C.3; it illustrates how the existential quantification of \( \mathcal{F} \) in the definition of the notation serves as (nested) existential quantifications of any number of probability distributions. A similar law can also be derived for probabilistic composition of three or more alternatives; the notation is therefore more general than probabilistic composition of any finite constant number of alternatives as defined in Section 3.4.2.

Law **PV-8** merely states that if the specification \( S \) in \( Q \triangleright \lambda \varphi \cdot S \) is deterministic (each value of \( \varphi \) and the prestate corresponds to exactly one poststate distribution), the existential quantification of \( \mathcal{F} \) can be eliminated (by the one-point law) and the resultant specification is also deterministic. Law **PV-9** further simplifies the notation when \( S \) is both deterministic and standard (so it is an assignment to the poststate depending on the values of \( \varphi \) and the prestate); its application is illustrated in the following example.

**Example 3.16.** For natural variable \( x \):

\[
[2*(1,.,6)] \rightarrow 1/36 \triangleright \lambda[j;k] \cdot x := \max j k
= P' = (\lambda[j;k] \cdot \max j k) \circ ([2*(1,.,6)] \rightarrow 1/36)
= \forall i: \text{nat} \cdot P'(x' = i) = \Sigma[j;k]:(\forall i: [2*(1,.,6)] \cdot \max j k = i) \cdot 1/36
= \forall i: 1,.,6 \cdot P'(x' = i) = (2i - 1)/36
\]

\[\diamond\]

### 3.4.5 The specification \( R \cdot S \)

\[
R \cdot S = \exists P'' \cdot (\text{substitute } P'' \text{ for } P' \text{ in } \langle R \rangle_a) \\
\land P'' \triangleright \lambda \sigma'' \cdot (\text{substitute } \sigma'' \text{ for } \sigma \text{ in } S) \\
= \exists P'' \cdot (\text{substitute } P'' \text{ for } P' \text{ in } \langle R \rangle_a) \\
\land \exists \mathcal{F} : D_\sigma \rightarrow D_\sigma \rightarrow 0,.,1 \cdot \\
(\forall \sigma'' \cdot P''(\sigma'' > 0 \Rightarrow (\text{substitute } \sigma'' ; \mathcal{F}\sigma'' \text{ for } \sigma ; P' \text{ in } \langle S \rangle_a))) \\
\land P' = \Sigma \sigma'' \cdot P''(\sigma'' \times \mathcal{F}\sigma'')
\]

As in the standard model, *dependent composition* specifies that the computation behaves according to \( R \), then behaves according to \( S \), with the poststate from \( R \) serving as the prestate for \( S \). In the standard model, dependent composition is defined simply as the existential quantification of the intermediate state \( \sigma'' \), which is the poststate for \( R \) and the prestate for \( S \). In the probabilistic model, the intermediate state cannot be obtained directly from the specification \( R \), but is rather the outcome of the probability experiment described by the poststate distribution for \( \langle R \rangle_a \). We therefore use an existential quantification of the intermediate state distribution \( P'' \), which is the poststate distribution for \( \langle R \rangle_a \). (For clarity we may also change any occurrence of \( \sigma' \) to \( \sigma'' \) in any
event expression that appears in \( \langle R \rangle_\sigma \), and the local variable of any event expression in \( \langle R \rangle_\sigma \) is then assumed to be \( \sigma'' \). The intermediate state \( \sigma'' \), which is the prestate for \( S \), is chosen probabilistically according to \( P'' \). The behavior according to \( S \) then depends on the intermediate state chosen: for each intermediate state \( \sigma'' \), \( S \) with \( \sigma \) substituted by \( \sigma'' \) can be viewed as an alternative of probabilistic choice, whose probability of being selected is \( P'' \sigma'' \). Dependent composition can therefore be defined as probabilistic composition of variable alternatives (Section 3.4.4), with the intermediate state distribution (the poststate distribution for \( R \)) as the choice distribution, and the intermediate state (the prestate for \( S \)) as the choice variable.

**Example 3.17.** For integer variable \( x \):

\[
\mathcal{P}'(x' = 1) \geq 4/5. \text{ ok } \forall x \colon x := x + 1
\]

We simplify \( \mathcal{P}'' \triangleright \lambda x'' \cdot x' = x \text{ } \forall x' = x + 1 \) first.

\[
\triangleright \lambda x'' \cdot x' = x'' \text{ } \forall x' = x'' + 1
\]

\[
\Rightarrow \lambda x'' \cdot \exists p \cdot \mathcal{P}'(x'' + 1) = 1 - p
\]

\[
= \exists \mathcal{F} \cdot (\forall x'' \cdot \mathcal{P}'(x'' + 1) = 1 - p)
\]

Use the distributive and Skolem laws for existential variable \( p \):

\[
= \exists f \cdot (\forall x'' \cdot \mathcal{P}'(x'' + 1) = 1 - f(x''))
\]

From the construction of \( \mathcal{F} \), \( \mathcal{F}x''x' \) is equal to \( f(x'') \) if \( x'' = x' \), \( 1 - f(x'') \) if \( x'' = x' - 1 \), and 0 otherwise.

So we have

\[
\mathcal{P}'(x' = 1) \geq 4/5. \text{ ok } \forall x := x + 1
\]

The proof of this step is given below.

\[
= \mathcal{P}' \lor \mathcal{P}'' \geq 4/5
\]

We prove the mentioned step by double implication. The " \( \Rightarrow \) " part is straightforward. For the " \( \Leftarrow \) " part, we assume \( \mathcal{P}' \lor \mathcal{P}'' \geq 4/5 \) and let

\[
\mathcal{P}'' = \lambda x'' \text{ if } x'' = 1 \text{ then } \mathcal{P}' \lor \mathcal{P}'' \text{ else if } x'' < 1 \text{ then } \mathcal{P}'(x'' + 1)
\]

\[
f = \lambda x'' \text{ if } x'' = 1 \text{ then } \mathcal{P}' \lor \mathcal{P}'' \text{ else if } x'' < 1 \text{ then } 0
\]

It can be proved that \( \mathcal{P}' \lor \mathcal{P}'' \geq 4/5 \) and \( \mathcal{P}' = \lambda x' \cdot \mathcal{P}''x' \times f(x') + \mathcal{P}''(x' - 1) \times (1 - f(x' - 1)) \), and the " \( \Leftarrow \) " part follows immediately from the generalization law.
Since the definition of dependent composition for the standard model has been replaced by the definition for the probabilistic model, we introduce a notation $R \cdot S$ for dependent composition as defined in the standard model (Section A.3.5), where $R$ and $S$ must be standard specifications.

**Laws for $R \cdot S$**

As in the standard model, given a specification with dependent composition, we may want to simplify the expression so that we can talk about the result without having to refer to the intermediate state or state distribution. Unfortunately, since dependent composition involves a probabilistic composition of a possibly infinite number of alternates, it is generally much harder to simplify than in the standard model. Here we provide some laws that can help us to do this, as well as to prove other theorems.

In the following, $Q$, $R$, and $S$ are probabilistic specifications; $T$ is a probabilistic specification with an additional choice variable $\varphi$, and $Q$ is a choice distribution; $U$ and $V$ are standard specifications; $b$ is a boolean expression of the prestate; $p$, $u$, and $v$ are probability-valued expressions of the prestate; $e$ is an expression of the prestate in the domain of state variable $x$.

\[
\begin{align*}
\text{DC-1} & \quad \text{ok. } R = R \cdot \text{ok} = R \\
\text{DC-2} & \quad (Q \cdot R) \cdot S \Rightarrow Q \cdot (R \cdot S) \\
\text{DC-3} & \quad x := e. \ S = (\text{substitute } e \text{ for } x \text{ in } S) \\
\text{DC-4} & \quad Q \lor R \cdot S = (Q \cdot S) \lor (R \cdot S) \\
& \quad S \cdot Q \lor R \leftarrow (S \cdot Q) \lor (S \cdot R) \\
\text{DC-5} & \quad (\text{if } b \text{ then } Q \text{ else } R) \cdot S = \text{if } b \text{ then } (Q \cdot S) \text{ else } (R \cdot S) \\
\text{DC-6} & \quad Q \land R \cdot S \Rightarrow (Q \cdot S) \land (R \cdot S) \\
& \quad S \cdot Q \land R \leftarrow (S \cdot Q) \land (S \cdot R) \\
\text{DC-7} & \quad Q \cdot \lambda \varphi \cdot R \cdot S \Rightarrow (Q \cdot S) \cdot \lambda \varphi \cdot (R \cdot S) \\
& \quad S \cdot Q \cdot \lambda \varphi \cdot R \Rightarrow (S \cdot Q) \cdot \lambda \varphi \cdot (S \cdot R) \\
\text{DC-8} & \quad (Q \triangleright \lambda \varphi \cdot T) \cdot S \Rightarrow Q \triangleright \lambda \varphi \cdot (T \cdot S) \\
& \quad S \cdot (Q \triangleright \lambda \varphi \cdot T) \Rightarrow Q \triangleright \lambda \varphi \cdot (S \cdot T) \\
\text{DC-9} & \quad Q \cdot \gamma \cdot R \cdot S \Rightarrow (Q \cdot S) \cdot \gamma \cdot (R \cdot S) \\
\text{DC-10} & \quad U \cdot V = \langle U \cdot V \rangle \\
\text{DC-11} & \quad p'(U) \geq u \cdot p'(V) \geq v = p'(U \cdot V) \geq uv
\end{align*}
\]

A very desirable but lacked property for dependent composition is associativity (DC-2 does not hold if "$\Rightarrow$" is replaced by "$\equiv$"), Fortunately, the property holds for the

---

4Strictly speaking, $R \cdot S$ is semantically meaningful even if $R$ or $S$ mentions $p'$, since $p'$ may be considered a variable global to the expression. However, such use is confusing and should be avoided.
class of specifications we will discuss in Section 3.5, and in practice, we restrict our use of dependent composition to only that class of specifications. We define the order of association for dependent composition to be from left to right, so that \( Q \cdot R \cdot S \) means \( (Q \cdot R) \cdot S \) (though this concern is not necessary for practical purposes).

Law DC-10 says that if both specifications in a dependent composition are standard, the resultant is equivalent to the general specification resulting from standard dependent composition of the two specifications; thus the definition of dependent composition in the probabilistic model is an extension of its standard definition. Since we do not use the standard definition of dependent composition to combine probabilistic specifications (unlike the case for nondeterministic composition where disjunction can be used to combine probabilistic specifications), it is not necessary to distinguish between the standard and probabilistic cases, and the introduction of another notation for the standard case is not really necessary.\(^5\)

### 3.5 Convexity and nondeterminism

As can be seen in the notation laws, some desired properties are not included, such as idempotence of probabilistic composition. At first thought this is counter-intuitive, since one might argue that a computation which probabilistically chooses between the behavior according to specification \( S \) and the behavior according to the same \( S \) should behave according to \( S \). Looking into it more closely, we see that if \( S \) is a nondeterministic specification (Def. 3.1, also see Section 3.5.3), there is more than one way to behave according to it, and a probabilistic choice between two different computations that behave according to \( S \) may not satisfy \( S \). For example, let \( S \) be the specification \( \langle x' = 0 \rangle_a \lor \langle x' = 1 \rangle_a \). Consider the specification

\[
S \downarrow \neg S = (\langle x' = 0 \rangle_a \lor \langle x' = 1 \rangle_a) \downarrow \neg (\langle x' = 0 \rangle_a \lor \langle x' = 1 \rangle_a)
\]

To behave according to this specification, the computation must make a probabilistic choice (with probability \( 1/2 \)) between two computations both satisfying \( S \). There are two ways for a computation to behave according to \( S \): it may always set \( x \) to 0, or it may always set \( x \) to 1. If the two alternatives in the probabilistic composition are such

\(^5\)The only situation where a new notation for the standard case may be useful is when we want to write \( U \cdot V \) not as a probabilistic specification itself, but as an expression that involves \( \sigma' \), such as in sub-expressions of a general specification or in event expressions. For example, the specification \( P'(x := 0. x := x) = 1/2 \) is not the same as the specification \( P'(x := 0. x := x) = 1/2 \) since for the latter, \( x' = 0. x' = x \) is equivalent to \( \langle x' = 0 \rangle_a \), which does not involve \( \sigma' \). The former is equivalent to \( P'(x' = 0) = 1/2 \), while the latter is equivalent to \( \bot \). To avoid confusion, the notation \( U \cdot V \) should not be used in sub-expressions where \( \sigma' \) is expected to be a variable (such as in event expressions).
that the computation always sets \( x \) to 0 if the first alternative is chosen, and always sets \( x \) to 1 if the second alternative is chosen, the resultant computation no longer satisfies \( S \), since it neither always sets \( x \) to 0 nor always sets \( x \) to 1 (it sets \( x \) to each of 0 and 1 with probability 1/2).

Some of the lacked properties are so important that even though they do not hold for every specification, we want to identify a class of specifications that have these properties. As can be seen above, probabilistic composition does not have the idempotence property because a nondeterministic specification may be satisfied by two poststate distributions while not satisfied by some probabilistic sum of them. The property will hold for probabilistic specification \( S \) if \( S \) is satisfied by every probabilistic sum of any two poststate distributions satisfying \( S \), or formally

\[
\forall \mathcal{P}, \mathcal{Q}, \forall \mathcal{R} \cdot (\text{substitute } \mathcal{Q} \text{ for } \mathcal{P}' \text{ in } S) \land (\text{substitute } \mathcal{R} \text{ for } \mathcal{P}' \text{ in } S) \implies (\text{substitute } (\mathcal{Q} \oplus \mathcal{R}) \text{ for } \mathcal{P}' \text{ in } S)
\]

\[
\forall \mathcal{P}', \forall \mathcal{P} : 0, 1 \cdot \forall \mathcal{Q}, \forall \mathcal{R} \cdot \mathcal{P}' = \mathcal{Q} \oplus \mathcal{R} \land (\text{substitute } \mathcal{Q} \text{ for } \mathcal{P}' \text{ in } S) \land (\text{substitute } \mathcal{R} \text{ for } \mathcal{P}' \text{ in } S) \implies S
\]

We therefore have the following definition.

**Definition 3.5.** A specification \( S \) is convex for prestate \( \sigma \) if and only if

\[
\forall \mathcal{P}' \cdot \mathcal{P} : 0, 1 \cdot \forall \mathcal{Q}, \forall \mathcal{R} \cdot \mathcal{P}' = \mathcal{Q} \oplus \mathcal{R} \land (\text{substitute } \mathcal{Q} \text{ for } \mathcal{P}' \text{ in } S) \land (\text{substitute } \mathcal{R} \text{ for } \mathcal{P}' \text{ in } S) \implies S
\]

If \( S \) is convex for every prestate, we just say that \( S \) is convex. For specification \( S \), the convex closure of \( S \) is the strongest convex specification implied by \( S \), denoted \( \triangledown S \).

Given specification \( S \), \( \triangledown S \) can be obtained by repeating nondeterministic composition with \( S \) itself until the resultant specification is convex; however, as shown in the following example, a finite number of repetitions may not suffice.

**Example 3.18.** For integer variable \( x \), consider the specification

\[
S = \exists i \cdot i \geq 1 \land \mathcal{P}'(x' = i) = 1
\]

The convex closure of \( S \) is \( R \), but it cannot be obtained by any finite number of repetitions of nondeterministic composition with \( S \): the specification (say, \( R \)) resulting from a finite number of repetitions (say, \( n \)) says that a maximum of \( n \) positive integers can have non-zero probabilities. For the specification \( R \), \( R \) is not convex.

The specification \( R \) does not imply \( R \), and \( R \) is not convex.
In this case, an infinite number of repetitions of nondeterministic composition with $S$ is required to obtain $oS$.

### 3.5.1 Laws for convex specifications

Here we present some laws that do not hold for specifications in general, but hold for convex specifications. In the following, $Q$, $R$, and $S$ are probabilistic specifications; $T$ is a probabilistic specification with an additional choice variable $\varphi$, and $Q$ is a choice distribution; $U$ is a standard specification; $p$ is a probability-valued expression of the prestate.

**PC-4**  
$\circ R \circ R = \circ R$

**NC-6**  
$\circ R \circ R = \circ R$

**NC-10**  
$(Q \circ R) p \circ S = (Q \circ S) p \varphi (R \circ \circ S)$

**PV-1**  
$\mathcal{P} \circ Q \varphi \circ S = \mathcal{P} \varphi \circ Q \circ S$

**DC-2**  
$(\circ Q \circ R) \circ S = \circ Q \circ S$

**DC-7**  
$Q \circ R \circ S = (Q \circ S) p \varphi (R \circ \circ S)$

**DC-8**  
$(Q \circ \varphi \circ T) \circ S = Q \varphi \circ (T \circ \circ S)$

**DC-9**  
$Q \circ R \circ S = (Q \circ S) \varphi (R \circ \circ S)$

### 3.5.2 Test for convexity

Although Def. 3.5 can be used directly to prove whether a given specification is convex or not, here we provide some laws that can help us to do this more conveniently.

**Theorem 3.6.** Let $U$ be a standard specification; let $I$ be a probability-valued interval and $J$ be a real interval, whose either end may be included or excluded; let $p$ be a probability-valued expression of the prestate; let $b$ be a boolean expression of the prestate; let $\mathcal{X}$ be a random variable; let $\mathcal{Y}$ be a function with domain $D_\varphi$, and $\mathcal{P}_y$ be a probability distribution over the range of $\mathcal{Y}$; let $\circ R$ and $\circ S$ be convex specifications. All the following specifications are convex:

\[
\langle U \rangle_s \\
\mathcal{P}' \langle U \rangle : I \\
\mathcal{Y} \circ \mathcal{P}' = \mathcal{P}_y \\
\varepsilon \mathcal{X} : J
\]

---

It can be shown that the maximum number of repetitions required does not exceed the cardinality of the state space (which may be infinite), so we may define

\[
\circ S = \exists Q \cdot Q \varphi D_\varphi \rightarrow (S)_a
\]

where the existential variable $Q$ serves as any choice distribution over $D_\varphi$; each value of the choice variable corresponds to the same $S$. 
Chapter 3. The Probabilistic Model of Specifications

\[ \text{if } b \text{ then } \circ R \text{ else } \circ S \]
\[ \circ R \land \circ S \]
\[ \circ R \_\_ \_ \circ S \]
\[ \circ R \_\_ \_ \circ S \]
\[ \circ R \_\_ \_ \circ S \]

The proofs of these laws are straightforward from Def. 3.5 and the specification laws.

Example 3.19. The specification

\[ P'(x' \geq 0) = \frac{2}{3} \land P'(y' = 1 \lor y' = x) < \frac{1}{2}. \ 2 < E'(y) \leq \frac{7}{2} \ \lor (x' < x) \]

is convex. \( \diamond \)

3.5.3 Nondeterministic specifications

Since convex specifications have some very important properties, it is desirable that a specification be convex, so that it is easier to prove theorems about it with the use of laws that do not hold otherwise. Moreover, as we will see in Chapter 4, a convex specification allows more flexibility in refinement, since any nondeterminism in the specification can be resolved by probabilistic or nondeterministic choice; this cannot be said about non-convex specifications in general. Given a non-convex specification, very often the first step of program development is to refine the specification by a convex one.

However, there is no reason to restrict our language of specifications or implementable specifications to only those that are convex. As discussed in Section 3.2.1, specifications denote observable behavior, and in the probabilistic model, an observation of poststate distribution can be regarded as the “observed” relative frequency of each poststate over an infinite number of executions on the same prestate. A non-convex specification such as \( (x' = 0) \lor (x' = 1) \) is perfectly acceptable as a denotation of observable behavior, meaning that given the same prestate, the “observed” relative frequency must be such that \( x' \) is always 0 or always 1, and not any other combination of 0 and 1. This specification can clearly be implemented, for example, by a computation that always sets \( x \) to 0. Note that \( x' \) may not be the same for a different prestate; for example, the above specification can also be implemented by a computation that always sets \( x \) to 0 if \( x = 0 \), and always sets \( x \) to 1 otherwise.

On the other hand, since nondeterminism may be resolved during execution (run-time nondeterminism), it is meaningless to specify that a computation chooses between \( (x' = 0) \) and \( (x' = 1) \) “nondeterministically but not probabilistically”: if a computation nondeterministically sets \( x \) to either 0 or 1 each execution, there cannot be any guarantee that the “observed” relative frequency is such that \( x \) is always 0 or always 1. So unlike
the case in the standard model, the word "nondeterminism" has different meanings when applied to specifications and to computations. This can be reflected by the fact that in the standard model, both observations and nondeterministic choices are based on individual executions; in the probabilistic model, observations are based on repeated executions on the same input, while nondeterministic choices are based on individual executions.

If nondeterminism must be resolved before the program is executed, and it is resolved once and for all executions (this is called construction time nondeterminism), it may be meaningful to specify that the implementer or the compiler chooses nondeterministically between two specifications, but not a probabilistic composition of the two. However, construction time nondeterminism requires a different model, such as the lifted model [3], and is beyond the scope of this thesis.\footnote{Our conception of "nondeterministic specification" is called "underspecification" in some formalisms.}

\footnote{Even in the standard model, there is no way to specify that nondeterminism be resolved at construction time. For example, one cannot specify that the implementer nondeterministically chooses between the programs $x := 0$ and $x := 1$ by writing the specification $x := 0 \lor x := 1$, since the implementer may refine the specification by $\text{if } z = 0 \text{ then } x := 0 \text{ else } x := 1$, which is not exactly what we want since it is sensitive to the prestate but neither $x := 0$ nor $x := 1$ is. Construction time nondeterminism is stronger than runtime nondeterminism, but weaker than determinism.}
Chapter 4
Probabilistic programs

Having presented the basic model for probabilistic specifications, we are ready to look into probabilistic programs. In this chapter, we discuss the concept of refinement for probabilistic specifications, which is the heart of program development. Then we set forth the basic programming language for probabilistic programs. We also discuss the issue of timing, which includes termination; in particular, we inspect expected time, which is often regarded as the criterion of efficiency for probabilistic programs. Furthermore, we explore some possibilities to enrich our repertoire of programming notations. Lastly, we present some simple examples to illustrate how the extended theory can be applied in the development of probabilistic programs.

4.1 Refinement

The concept of refinement for the probabilistic model is essentially the same as the standard model (Section A.4). We have the following definition:

**Definition 4.1.** Probabilistic specification $P$ is **refined** by probabilistic specification $S$ if and only if every computation that satisfies $S$ also satisfies $P$:

$$\forall \sigma, \mathcal{P}' \cdot P \nvdash S$$

As we can see, the definition is the same as in the standard model: both are defined as implication, universally quantified over all variables of the (refinement) problem $P$ and the (refinement) solution $S$. Any laws that do not distinguish between the variables $\sigma'$ and $\mathcal{P}'$ apply to both the standard and the probabilistic model. As usual, we leave out the universal quantifiers in proofs, and write $P \nvdash S$. Here are some examples of refinement.

**Example 4.1.** For integer variables $x$ and $y$:

$$\mathcal{P}'(x' > x) = \frac{2}{3} \nvdash x' > x \land y' = y \frac{3}{5} \text{ ok}$$

$$\mathcal{P}'(x' = 2) = \frac{1}{2} \land (y' > x') \nvdash x := 2 \frac{4}{7} x := 3 \cdot y := x + 1$$

41
4.1.1 Refinement laws

Here we present some laws that can help us to prove theorems about refinement as well as to develop programs from specifications. In the following, we consider only probabilistic specifications; general specifications may be converted to probabilistic specifications (Section 3.3) for these laws to be applied (there is a simpler treatment for general specifications, discussed in the next section). Since the definition of refinement is the same as the standard model, some laws are imported from standard refinement (Section 4.4.1); however, laws concerning notations that are redefined (such as dependent composition) cannot be imported (though their proofs are straightforward).

Theorem 4.2. Refinement by steps

If $A \iff \text{if } b \text{ then } C \text{ else } D$ and $C \iff E$ and $D \iff F$ are theorems,
then $A \iff \text{if } b \text{ then } E \text{ else } F$ is a theorem.
If $A \iff B \_ E C$ and $B \iff D$ and $C \iff E$ are theorems,
then $A \iff D \_ E C$ is a theorem.
If $A \iff B \_ C$ and $B \iff D$ and $C \iff E$ are theorems,
then $A \iff D \_ E C$ is a theorem.
If $A \iff B$ and $B \iff C$ are theorems, then $A \iff C$ is a theorem.

Proof: The laws for probabilistic composition, convex composition and dependent composition follow from PC-6, NC-7 and DC-4 respectively; the others are imported from standard refinement.

Theorem 4.3. Refinement by parts

If $A \iff \text{if } b \text{ then } C \text{ else } D$ and $E \iff \text{if } b \text{ then } F \text{ else } G$ are theorems,
then $A \wedge E \iff \text{if } b \text{ then } C \wedge F \text{ else } D \wedge G$ is a theorem.
If $A \iff B \_ E C$ and $D \iff E \_ F C$ are theorems,
then $A \wedge D \iff B \wedge E \_ F C \wedge F$ is a theorem.
If $A \iff B \_ C$ and $D \iff E \_ F C$ are theorems,
then $A \wedge D \iff B \wedge E \_ C \wedge F$ is a theorem.
If $A \iff B$ and $C \iff D$ are theorems, then $A \wedge C \iff B \wedge D$ is a theorem.

Proof: The laws for probabilistic composition, convex composition and dependent composition follow from PC-8, NC-9 and DC-6 respectively; the others are imported from standard refinement.
Theorem 4.4. Refinement by cases

\( P \iff \text{if } b \text{ then } Q \text{ else } R \) is a theorem if and only if

\( P \iff b \land Q \) and \( P \iff \neg b \land Q \) are theorems.

If \( \circ P \iff Q \) and \( \circ P \iff R \) are theorems,

then \( \circ P \iff Q \uplus R \) is a theorem.

\( \circ P \iff Q \uplus R \) is a theorem if and only if

\( \circ P \iff Q \) and \( \circ P \iff R \) are theorems.

Proof: The law for conditional composition is imported from standard refinement. For nondeterministic composition, the "only if" part follows from NC-5. For probabilistic composition and for the "if" part of nondeterministic composition:

\[
(\circ P \iff Q) \land (\circ P \iff R) \Rightarrow (\circ P \iff Q \uplus R)
\]

4.1.2 Refinement for general specifications

Let \( P \) and \( S \) be general specifications. Semantically, they represent \( \langle R \rangle_a \) and \( \langle S \rangle_a \), so refinement of \( P \) by \( S \) is just \( \forall \sigma, \mathcal{P'} \cdot \langle R \rangle_a \iff \langle S \rangle_a \). However, as indicated in the following law, the conversion from general specifications to probabilistic specifications is not always necessary.

Theorem 4.5. Let \( P \) and \( S \) be general specifications; let \( p \) be a probability-valued expression. If

\[
\forall \sigma, \sigma', \mathcal{P'} \cdot P \iff S
\]

is a theorem, then

\[
\forall \sigma, \mathcal{P'} \cdot \langle P \rangle_a \geq p \iff \mathcal{P'} \langle S \rangle_a \geq p
\]

is a theorem; and in particular, for \( p = 1 \),

\[
\forall \sigma, \mathcal{P'} \cdot \langle P \rangle_a \iff \langle S \rangle_a
\]

is a theorem.

Proof: The proof is straightforward; from law PR-6 we have

\[
\forall \sigma' \cdot P \iff S \Rightarrow \mathcal{P'} \langle P \rangle \geq \mathcal{P'} \langle S \rangle
\]

\[
= \forall p \cdot \mathcal{P'} \langle P \rangle \geq p \iff \mathcal{P'} \langle S \rangle \geq p
\]

\[
\Rightarrow \langle P \rangle_a \iff \langle S \rangle_a
\]

As usual, we leave out the universal quantifiers in proofs of refinement, and just write \( P \iff S \). Since it implies the refinement of \( \langle P \rangle_a \) by \( \langle S \rangle_a \), we do not have to distinguish between general specifications and probabilistic specifications in proofs of refinement.
Note that the law stated in Thm. 4.5 is slightly more general, so that it can also be applied for refinement of probabilistic specifications of the type $\mathcal{P}'(P) \geq p$.

In its application, Thm. 4.5 can be used with the refinement laws stated in the previous section. For example, if a general specification is a conjunction of a standard part and a probabilistic part, using refinement by parts, we may separate a refinement proof into two parts. The proof of the standard part may make use of laws that relate standard specification notations to their probabilistic counterparts (such as laws NC-12 and DC-10), so that the proof can use the simpler standard notations.

**Example 4.2.** For integer variables $x$ and $y$, we prove the refinement

$$x' > x \land \mathcal{P}'(y' = x + 1) \geq \frac{1}{2} \iff x' \geq x \land y' = x. \text{ ok } \frac{1}{2}y \ y := y + 1. \ x := x + 1$$

Applying refinement by parts, it is sufficient to prove both

(a) $x' > x \iff x' \geq x. \text{ ok } \frac{1}{2}y \ y := y + 1. \ x := x + 1$

and

(b) $\mathcal{P}'(y' = x + 1) = \frac{1}{2} \iff y' = x. \text{ ok } \frac{1}{2}y \ y := y + 1. \ x := x + 1$

To prove (a), first we prove

$$x' = x \iff \text{ ok } \frac{1}{2}y \ y := y + 1$$

using refinement by cases: $(x' = x)_a$ is convex and is refined by both alternatives. Then we prove

$$x' > x \iff x' \geq x. \ x' = x. \ x := x + 1$$

applying law DC-10, which says that we can just use the standard definition of dependent composition:

$$x' \geq x. \ x' = x. \ x := x + 1 = (x' \geq x \cdot x' = x \cdot x := x + 1)_a \Rightarrow (x' > x)_a$$

(a) then follows from refinement by steps. To prove (b), applying laws DC-7, DC-10 and PC-10,

$$y' = x. \text{ ok } \frac{1}{2}y \ y := y + 1. \ x := x + 1$$

$$= (y' = x. \text{ ok } x := x + 1) \frac{1}{2}y (y' = x. \ y := y + 1. \ x := x + 1)$$

$$= (y' = x. \text{ ok } x := x + 1) \frac{1}{2}y (y' = x. \ y := y + 1. \ x := x + 1)$$

$$= y' = x \frac{1}{2}y \ y' = x + 1$$

$$= \mathcal{P}'(y' = x) = \frac{1}{2} \land \mathcal{P}'(y' = x + 1) = \frac{1}{2}$$

$$\Rightarrow \mathcal{P}'(y' = x + 1) = \frac{1}{2}$$

And this completes the proof. \(\diamondsuit\)
4.2 The programming language

The conception of programs which is discussed in Section A.5 applies to all paradigms of programming. The programming language in the probabilistic model is therefore a sublanguage of the specification language, as in the standard model. We take the following:

**Definition 4.6.** The probabilistic programming language

(a) ok is a program.

(b) If x is any space variable and e is an implemented expression of the prestate in the domain of x, then \( x := e \) is a program.

(c) If b is an implemented boolean expression of the prestate, and P and Q are programs, then if \( b \) then P else Q is a program.

(d) If p is an implemented probability-valued expression, and P and Q are programs, then \( P \ p-or \ Q \) is a program.

(e) If P and Q are programs, then \( P \ or \ Q \) is a program.

(f) If P and Q are programs, then \( P \ and \ Q \) is a program.

(g) If P is an implementable specification and a program S is provided such that \( P \implies S \) is a theorem, then \( P \) is a program.

As in the standard model, (e) is excluded if we require that a computation be deterministic. For clarity, or and \( p-or \) should be used as program connectives only; for sub-expressions of a specification to be refined, \( \gamma \) and \( p\gamma \) should be used instead.

The computational behavior for (a)-(f) is described in Section 3.4. For (g), refer to Section A.5.1; recursion is allowed, which is illustrated in the following example.

**Example 4.3.** For integer variable \( x \):

\[ x' \leq 0 \iff \text{if } x \leq 0 \text{ then } \text{ok} \text{ else (} x := 0 \text{ or (} x := x - 1 \text{. } x' \leq 0) \)\]

The problem is \( x' \leq 0 \), which is solved by the above refinement. (Note that \( x' \leq 0 \) means \( (x' \leq 0)_a \).) When the program is executed, it runs as follows. It first evaluates \( x \). If \( x \leq 0 \), the execution terminates; otherwise, it makes a probabilistic choice: with probability \( 1/3 \), it sets \( x \) to 0 and terminates; with probability \( 2/3 \), it decreases \( x \) by 1, then recursively executes the program again.

The proof of the above refinement is straightforward, using refinement by cases for both the conditional composition and the probabilistic composition:

\[ x \leq 0 \land \text{ok} \Rightarrow x' \leq 0 \]
\[ x > 0 \land x := 0 \Rightarrow x' \leq 0 \]
\[ x > 0 \land (x := x - 1 \land x' \leq 0) \Rightarrow x' \leq 0 \]
4.3 Recursive time

To talk about recursive time, we add a time variable to the state (see Section 3.2.2), and follow a procedure very much the same as in the standard model (Section A.6). We must modify the program the same way as in the standard model, and replace every recursive call to $P$ by $(t := t + 1. P)$ (note that $t := t + 1$ means $(t := t + 1)_a$); then we can include time in each refined specification of the implementation. We may simply use a conjunction $P \land T$ for specification $P$ and timing expression $T$ as in the standard model. As in the standard model, all refinements must hold with the inclusion of time.

**Example 4.4.** For Ex. 4.3, with the inclusion of time, the refinement becomes

$$x' \leq 0 \land T \iff \begin{cases} \text{if } x \leq 0 \text{ then ok} \\ \text{else } (x := 0 \frac{1}{3} - \text{or } (x := x - 1. t := t + 1. x' \leq 0 \land T)) \end{cases}$$

Using various expressions for $T$, there are many ways to talk about time. For example, we can specify a worst-execution time bound $T = t' \leq t + f\sigma$ (which means $(t' \leq t + f\sigma)_a$). We can specify an expected time bound $T = \mathcal{E}'(t' - t) \leq f\sigma$, as well as the exact expected time $T = \mathcal{E}'(t' - t) = f\sigma$ (if the computation is deterministic in the probability density function of the running time). We can also talk about the probability of each possible running time by specifying the probability density function of the random variable $t' - t$.

However, since $t'$ is included in $\sigma'$ which is the parameter of $\mathcal{P}'$, sometimes we cannot see the whole picture unless we specify the poststate distribution with time included, in which case $T$ may be strictly stronger than $P$ (so $T$ is just a replacement of $P$).

As in the standard model, suitable timing expressions must be provided for all refined specifications (see Section A.6.1). It is usually much harder to find a suitable timing expression in the probabilistic case than in the standard case.

In the following examples, we will find suitable timing expressions for Ex. 4.4 in three different ways.

**Example 4.5.** We find a worst-execution time bound for Ex. 4.4. From the program, we can see that the computation terminates immediately if $x \leq 0$, and since $x$ decreases by 1 whenever it loops, its execution time is at worst $x$. So we write

$$T = t' \leq t + \max 0 \ x$$

Applying refinement by parts, we only have to prove the part for $T$. Applying refinement by cases:

$$x \leq 0 \land \text{ok} \Rightarrow x \leq 0 \land t' = t \Rightarrow t' \leq t + \max 0 \ x$$
$$x > 0 \land x := 0 \Rightarrow x > 0 \land t' = t \Rightarrow t' \leq t + \max 0 \ x$$
$$x > 0 \land (x := x - 1. t := t + 1. t' \leq t + \max 0 \ x)$$
$$\implies x > 0 \land t' \leq t + 1 + \max 0 \ (x - 1)$$
$$\implies t' \leq t + \max 0 \ x$$
This means that no matter what probabilistic choices have been made, we will always obtain the result within $\max 0 \times$ recursive time.

**Example 4.6.** We find an expected time bound for Ex. 4.4. From the program, we see that in each call to $x \leq 0$, there is probability of at least $1/3$ that the execution terminates, and the recursive time is the total number of calls minus 1 (since the first call does not cost any time). From conventional probability theory, if we neglect the value of $x$, we see that the number of calls to $x \leq 0$ has a geometric distribution, so the expected number of calls is 3; and this number can only be smaller if we take into consideration the value of $x$, since there is an additional circumstance $x \leq 0$ that can result in termination. So we write

$$T = \mathcal{E}'(t' - t) \leq 2$$

Applying refinement by cases:

1. $x \leq 0 \land \text{ok} \Rightarrow t' = t \Rightarrow \mathcal{E}'(t' - t) \leq 2$
2. $x > 0 \land (x := 0 \forall y (x := x - 1 \land t := t + 1 \land \mathcal{E}'(t' - t) \leq 2))$

Let $x > 0 \land (x' = 0 \land t' = t \forall y \mathcal{E}'(t' - (t + 1)) \leq 2)$

$$\Rightarrow \mathcal{E}'(t' - t) = 0 \forall y \mathcal{E}'(t' - t) \leq 3$$

$$\Rightarrow \mathcal{E}'(t' - t) \leq 2$$

In the proof, we have used law PC-12, which says that the expected value of a random variable in a probabilistic composition is the weighted average of the expected values of that random variable in the two alternatives. The timing expression means that on average, we will obtain the result within 2 recursive time, even though sometimes the execution may take much longer than that.

**Example 4.7.** We find the probability density function of the running time for Ex. 4.4 (which is possible since the computation is deterministic). For $x > 0$, we see that the recursive time is $i$, where $0 \leq i < x$, if and only if the computation does not terminate before time $i$, and terminates by the probabilistic choice at time $i$; also, the recursive time is $x$ if and only if the computation does not terminate before time $x$, and terminates by the condition $x \leq 0$ at time $x$. (It has roughly a geometric distribution, except that the probability is 0 after time $x$.) So we write

$$T = \begin{cases} \text{if } x \leq 0 \text{ then } t' = t \\ \text{else } \mathcal{P}'(t' - t = x) = \left(\frac{2}{3}\right)^{x} \land \forall i:0,..x \cdot \mathcal{P}'(t' - t = i) = \left(\frac{1}{3}\right)\left(\frac{2}{3}\right)^{i} \end{cases}$$

To prove the refinement, we again apply refinement by cases. The case $x \leq 0$ is trivial. For the case $x > 0$, we first see that

1. $x > 0 \Rightarrow (x := x - 1 \land t := t + 1 \land T$)
2. $\text{if } x - 1 \leq 0 \text{ then } t' = t + 1$
3. $\text{else } \mathcal{P}'(t' - t = x) = \left(\frac{2}{3}\right)^{x - 1} \land \forall i:0,..x - 1 \cdot \mathcal{P}'(t' - t - 1 = i) = \left(\frac{1}{3}\right)\left(\frac{2}{3}\right)^{i}$
4. $\text{if } x = 1 \text{ then } \mathcal{P}'(t' - t = 1) = 1$
5. $\text{else } \mathcal{P}'(t' - t = x) = \left(\frac{2}{3}\right)^{x - 1} \land \forall i:1,..x \cdot \mathcal{P}'(t' - t = i) = \left(\frac{1}{3}\right)\left(\frac{2}{3}\right)^{i - 1}$
6. $\mathcal{P}'(t' - t = x) = \left(\frac{2}{3}\right)^{x - 1} \land \forall i:1,..x \cdot \mathcal{P}'(t' - t = i) = \left(\frac{1}{3}\right)\left(\frac{2}{3}\right)^{i - 1}$


Writing \( U \) for the last expression, we have

\[
  x > 0 \Rightarrow (x := 0, t := x - 1, t := t + 1, U) \\
  \Rightarrow P(t' - t = 0) = 1 \mathrel{\triangleleft} U \\
  = P(t' - t = 0) = \left(\frac{1}{3}\right)(1) + \left(\frac{2}{3}\right)(0) \\
  \land P(t' - t = x) = \left(\frac{1}{3}\right)(0) + \left(\frac{2}{3}\right)(\frac{2}{3})^{x-1} \\
  \land \forall i : 1, \ldots, x \cdot P(t' - t = i) = \left(\frac{1}{3}\right)(0) + \left(\frac{2}{3}\right)(\frac{2}{3})^{i-1} \\
  = P(t' - t = x) = \left(\frac{2}{3}\right)^x \land \forall i : 0, \ldots, x \cdot P(t' - t = i) = \left(\frac{2}{3}\right)i \\
  = T \quad)
\]

In the proof, we have used law \( \text{PC-10} \), which can be used since each of \( P(t' - t = 0) = 1 \) and \( U \) describes a probability density function of \( t' - t \). The timing expression tells the exact probability of each possible running time depending on the prestate. \( \diamond \)

Note that the timing expression in Ex. 4.7 implies both the worst-execution time bound in Ex. 4.5 and the expected time bound in Ex. 4.6.

### 4.3.1 Termination

As discussed in Section A.6.2, if a specification in the standard model does not mention time, it can be satisfied by a computation that never terminates, and no complaint can be made about the computation. The same can be said for the probabilistic model: if a probabilistic specification \( P \) does not mention time, it is trivially refined by \( (t := t + 1, P) \).

If termination must be ensured, it must be implied by the specification with the inclusion of time, and the timing expression must be established by the implementer if not provided by the specifier. However, as in the standard model, we cannot just write \( t < \infty \Rightarrow \langle t' < \infty \rangle_a \), as it is still refined by the trivial refinement. Here we will investigate the condition under which a timing expression can be used for termination.

In the standard model, termination can be guaranteed by including a time bound in a specification. A time bound in the standard model translates to the probabilistic model as a worst-execution time bound: \( \langle t' \leq t + f \sigma \rangle_a \) where \( f \) is a nonnegative finite function of the prestate. A worst-execution time bound says that the computation has probability zero of running longer than the time bound, and this guarantees termination. However, it is too strong to be considered a necessary condition for termination, since for example, it is often considered that the program

\[
x' = x \iff \text{ok} \mathrel{\frac{1}{2}} \text{ or } x' = x
\]

guarantees termination, even though a worst-execution time bound does not exist: for any finite time \( t_0 \), the probability \( P(t' - t \geq t_0) \) is non-zero (precisely, it is \( 1/2^{t_0} \)), yet the
limit of this probability is zero as $t_0$ goes to infinity (in other words, the probability of non-termination is zero). This is exactly our requirement for termination, as stated in the following definition.

**Definition 4.7.** Given a specification with time $T$ and a prestate $\sigma$, $T$ is said to guarantee eventual termination for $\sigma$ if and only if for $\sigma$ and any poststate distribution $P'$ satisfying $T$, the limit of the probability $P'(t' - t \geq t_0)$ is zero as $t_0$ goes to infinity ($P'(t' - t \geq t_0)$ converges to zero). Formally, this means that for any positive probability $p$, there exists a finite time $t_0$ such that the probability $P'(t' - t \geq t_0)$ is at most $p$:

$$\forall p: 0..1, \exists t_0: nat \cdot \forall P' \cdot P'(t' - t \geq t_0) \leq p \iff T$$

The application of Def. 4.7 is illustrated in the following two examples.

**Example 4.8.** The refinement

$$T \leftarrow \text{ok } \frac{1}{2} - \sigma (t := t + 1. T)$$

is a theorem when

$$T \equiv \forall t_0: nat \cdot P'(t' - t \geq t_0) = 1/2^{t_0}$$

For any $p > 0$, let $t_0 = \text{ceil}(-\log p)$, then for any $P'$,

$$T \Rightarrow P'(t' - t \geq t_0) = 1/2^{\text{ceil}(-\log p)} \leq p$$

By Def. 4.7, eventual termination is guaranteed for any prestate.

**Example 4.9.** We show by contradiction that the timing expression

$$T = t < \infty \Rightarrow (t' < \infty)_\sigma$$

does not guarantee eventual termination. Let $p = 1/2$. For any natural $t_0$, let $P'$ be such that it satisfies the expression $P'(t' - t = t_0) = 1$, then $T$ is satisfied, but $P'(t' - t \geq t_0) \leq 1/2$ is not; this contradicts Def. 4.7.

4.3.2 Expected time

The expected time is a very useful measure of time, and is often regarded as the criterion of efficiency for probabilistic programs. A probabilistic program is often considered efficient if its expected time is low, even though its worst-execution time may be unbounded. The rationale is that although the execution may actually run much longer than the expected time, the probability that it happens is small, so it is relatively unimportant as far as efficiency is concerned. We account for this in the following theorem; moreover, we also see that an expected time bound guarantees eventual termination.

**Theorem 4.8.** Let $T$ be a timing expression indicating an expected time bound,

$$T \Rightarrow E'(t' - t) \leq f\sigma$$

where $f\sigma$ is finite. Then for any $k \geq 1$,

$$T \Rightarrow P'(t' - t > k \times f\sigma) < 1/k$$

Moreover, eventual termination is guaranteed for prestate $\sigma$. 
Proof: For any \( k \geq 1 \), we see that when \( \mathcal{E}'(t'-t) \) is finite,

\[
\mathcal{E}'(t' \cdot t) = \sum_{t_0} \cdot t_0 \times \mathcal{P}'(t' - t = t_0)
\]

\[
> \sum_{t_0} (\sum_{t_0} > k \times \mathcal{E}'(t' - t)) \cdot t_0 \times \mathcal{P}'(t'-t = t_0)
\]

\[
\geq \sum_{t_0} (\sum_{t_0} > k \times \mathcal{E}'(t' - t)) \cdot k \times \mathcal{E}'(t'-t) \times \mathcal{P}'(t'-t = t_0)
\]

\[
= k \times \mathcal{E}'(t'-t) \times \mathcal{P}'(t'-t > k \times \mathcal{E}'(t'-t))
\]

So for \( \mathcal{E}'(t'-t) \leq f \sigma \),

\[
\mathcal{P}'(t'-t > k \times f \sigma) \leq \mathcal{P}'(t'-t > k \times \mathcal{E}'(t'-t)) < 1/k
\]

To see that eventual termination is guaranteed for prestate \( \sigma \), for any \( p > 0 \), let \( t_0 = \text{ceil}(1 + (1/p)f \sigma) \), then

\[
T \Rightarrow \mathcal{P}'(t'-t \geq t_0) \leq \mathcal{P}'(t'-t > (1/p)f \sigma) < p
\]

And the guarantee of eventual termination follows from Def. 4.7.

An example of expected time analysis has been given in Ex. 4.6. In that scenario, the probability of termination in each recursive call is at least some fixed probability \( p > 0 \) (\( p = 1/3 \) in that example), and the expected number of recursive calls (excluding the first call) is at most \( 1/p - 1 \), a constant. For that example, if we replace the terminating condition \( x \leq 0 \) by \( 1 \), the worst-execution time bound in Ex. 4.5 no longer holds, but the expected time bound still holds, and eventual termination is guaranteed. In the following three examples we examine some other scenarios; in each example a worst-execution time bound does not exist, but eventual termination is guaranteed by an expected time bound.

Example 4.10. For integer variable \( x \), consider the refinement

\[
T \leftarrow \text{if } x \leq 0 \text{ then ok}
\]

\[
\text{else (ok } \frac{2}{3} \text{-or } x := x - 1 . t := t + 1 . T
\]

We find the expected time for the refinement. Here we do not have a fixed probability of termination each loop, and termination relies solely on the terminating condition, which is \( x \leq 0 \). In each recursive call, \( x \) decreases by 1 with probability \( 1/3 \), and remains unchanged with probability \( 2/3 \); intuitively, we see that it takes 3 calls on average to decrease \( x \) by 1, and therefore \( 3x \) calls on average to decrease \( x \) by \( x \). So we guess the timing expression to be

\[
T = \mathcal{E}'(t' - t) = \max 0 (3x)
\]

Applying refinement by cases, laws DC-7 and PC-12:

\[
x \leq 0 \wedge \text{ok} \Rightarrow \mathcal{E}'(t' - t) = 0 = \max 0 (3x)
\]

\[
x > 0 \wedge ((\text{ok } \frac{2}{3} \wedge x := x - 1). t := t + 1. \mathcal{E}'(t' - t) = \max 0 (3x))
\]

\[
\Rightarrow x > 0 \wedge (\mathcal{E}'(t' - t) = 1 + \max 0 (3x) \wedge \mathcal{E}'(t' - t) = 1 + \max 0 (3(x - 1)))
\]

\[
\Rightarrow x > 0 \wedge \mathcal{E}'(t' - t) = \frac{2}{3}(1 + 3x) + \frac{1}{3}(3x - 2) = 3x
\]

\[
\Rightarrow \mathcal{E}'(t' - t) = \max 0 (3x)
\]
Ex. 4.10 illustrates the common scenario where one can establish an integer expression (sometimes called a variant) that is bounded below, so that in each recursive call the variant decreases with at least some fixed probability \( p > 0 \), and never increases. In the example, the variant may be \( x \), bounded below by 0, and the probability of decrease is (at least) \( 1/3 \). The earlier scenario where there is a probability of at least some fixed \( p > 0 \) of termination each loop can be regarded as a special case of this scenario, since we may invent a variant that takes only two values (say, 0 and 1), which has the value 1 when the loop starts; whenever the loop terminates by probability, the variant is set to 0. The variant is then bounded below by 0, and decreases (from 1 to 0) with probability at least \( p \) each loop.

Our second example illustrates a more general scenario.

**Example 4.11.** For integer variable \( x \) and constant natural \( N \), consider the refinement

\[
T \leftarrow \begin{cases} 
\text{if} \; \text{abs} \; x = N \; \text{then} \; \text{ok} \\
\text{else} \; (x := x - 1 \; \text{or} \; x := x + 1. \; t := t + 1. \; T)
\end{cases}
\]

We find the expected time for the refinement, supposing we are interested in the prestate \( x = 0 \) only. This example is called a one-dimensional random walk with two absorbing boundaries. The terminating condition is \( \text{abs} \; x = N \). In each recursive call, \( x \) either increases or decreases by 1 with equal probability of \( 1/2 \). We see that the expected value of \( x \) is unchanged each loop; however, this is no indication of non-termination. For termination, we argue informally as follows. Since \( x \) is always between \(-N \) and \( N \), at any time where \( x \geq 0 \), the loop will terminate if \( x \) increases consecutively until \( x \) has the value \( N \), which takes at most \( N \) calls; likewise, at any time where \( x \leq 0 \), the loop will terminate if \( x \) decreases consecutively until \( x \) has the value \(-N \), which takes at most \( N \) calls. So for every time interval of \( N \) calls, there is a probability of at least \( 1/2^N \) that the loop terminates within that interval. At worst, the number of time intervals (of \( N \) calls) follows a geometric distribution, and the expected time is therefore at most \( N \times 2^N \), which is exponential in \( N \).

In fact, standard random walk results [2] show that the expected time is much better than exponential: if the loop starts at \( x = 0 \), the expected time of execution is \( N^2 \). Here we discuss how this correlates with our theory. First, we see that the above refinement does not hold for the timing expression

\[
T = x = 0 \Rightarrow \mathcal{E}'(t' - t) = N^2
\]

Although we are only interested in the prestate \( x = 0 \), during the execution the loop may be called when \( x \) is any value between \(-N \) and \( N \), and suitable time bounds for these prestates must also be included in \( T \). The timing expression

\[
T = \text{abs} \; x \leq N \Rightarrow \mathcal{E}'(t' - t) \leq N^2
\]

is still not good for the refinement (despite the fact that it is actually correct for the execution); this is because the expression in the antecedent is too weak for the refinement
to hold, since it does not reflect the fact that the "remaining" expected time decreases as \( \text{abs } x \) gets closer to \( N \). Therefore, we must find a stronger timing expression with tighter time bounds for all prestates \( \text{abs } x \leq N \) (in addition to the prestate \( x = 0 \)), even though we are not really interested in them.

The expected time for any prestate \( \text{abs } x \leq N \) may be stated from standard random walk results for which the refinement can be proved, but in this example we illustrate how a suitable timing expression for expected time can be derived from scratch. We write

\[
T = \text{abs } x \leq N \Rightarrow \mathcal{E}'(t' - t) = fx
\]

and find a suitable \( f \) so that the refinement holds. Applying refinement by cases:

\[
\begin{align*}
\text{abs } x = N \land \text{ok} & \Rightarrow \text{abs } x = N \land \mathcal{E}'(t' - t) = 0 \\
\text{abs } x \neq N \land (x := x - 1 \land t := t + 1 \land \text{abs } x \leq N) & \Rightarrow \mathcal{E}'(t' - t) = f(x - 1) + 1 \\
\forall y (\text{abs } x + 1 \leq N & \Rightarrow \mathcal{E}'(t' - t) = f(x + 1) + 1) \\
\Rightarrow \text{abs } x \neq N \land (\text{abs } x < N & \Rightarrow \mathcal{E}'(t' - t) = \frac{1}{2}(f(x - 1) + f(x + 1)) + 1)
\end{align*}
\]

Both cases imply \( T \) if \( f \) is such that

\[
\begin{align*}
fN &= f(-N) = 0 \\
f_{x} &= \frac{1}{2}(f(x - 1) + f(x + 1)) + 1 \iff \text{abs } x < N
\end{align*}
\]

This is in correspondence with the difference equation for expected time in standard random walk analysis. There are various methods to solve for \( f \) (for this example, one method is that we start by adding all the expressions for \( fx \)'s together for all \( x \) so that all expressions except \( f(N - 1) \) and \( f(-N + 1) \) cancel out). Here we omit the details and just state the solution for \( f \) (which can be verified easily):

\[
fx = N^2 - x^2 \iff \text{abs } x \leq N
\]

So the required timing expression is

\[
T = \text{abs } x \leq N \Rightarrow \mathcal{E}'(t' - t) = N^2 - x^2
\]

In retrospect, it is not hard to guess the required timing expression from the beginning, assuming we know that the expected time for the prestate \( x = 0 \) is \( N^2 \). For any state \( x = k \) (\( \text{abs } k \leq N \)), since \( x \) must have reached (and passed) \( k \) before it gets to \( N \), the expected time to move from \( k \) to \( N \) is the expected time to move from 0 to \( N \) minus the expected time to move from 0 to \( k \), which is \( N^2 - k^2 \). (Of course, the refinement must be proved.) On the other hand, if we have no knowledge about random walk results but are content with a loose expected time bound, based on the informal argument above we may guess

\[
T = \text{abs } x \leq N \Rightarrow \mathcal{E}'(t' - t) \leq N \times 2^N - \text{abs } x \times 2^{\text{abs } x}
\]

for which the refinement can also be proved. \( \diamond \)
Ex. 4.11 illustrates the scenario where no strictly decreasing variant can be established as in the scenario of Ex. 4.10. Instead, one can establish a variant that is bounded below and above, so that in each recursive call the variant decreases with at least some fixed probability \( p > 0 \); it may increase, but not above the upper bound. In the example, the variant may be \( N - \text{abs} \ x \), bounded below by 0 and above by \( N \), and the probability of decrease is (at least) \( 1/2 \). We see (informally) that in this scenario, an expected time bound always exists: for every time interval of \( U - L \) calls (\( U \) is the upper bound and \( L \) is the lower bound), there is a probability of at least \( p^{U-L} \) that the loop terminates within that interval (if the variant only decreases throughout that interval), so the expected time is at most \( (U-L) \times (1/p)^{U-L} \).

In our third example, we cannot establish a variant as described above, yet the expected time can be proved.

**Example 4.12.** For natural variables \( x \) and \( y \), consider the refinement

\[
T \iff \begin{cases} \text{if } x = y^2 \text{ then (ok } \frac{1}{2} \text{-or } (y := y + 1. \ x := x + 1. \ t := t + 1. \ T)) \\
\text{else (} x := x + 1. \ t := t + 1. \ T) 
\end{cases}
\]

To find the expected time, we first examine the program (informally) to see what it says. We see that whenever the execution loops, \( x \) always increases by 1. If initially \( x > y^2 \), the loop never terminates. Otherwise, \( x \) increases recursively until the value of \( x \) equals \( y^2 \), when it has a probability of \( 1/2 \) to terminate; if termination fails, \( y \) increases by 1 (as well as \( x \)), and the loop continues until \( x \) reaches the next square number, when the probability to terminate is again \( 1/2 \); and so on. Therefore if initially \( x = y^2 \), the probability that the loop terminates immediately is \( 1/2 \), the probability that the loop terminates after exactly \( (y+1)^2 - y^2 = 2y+1 \) calls is \( 1/4 \), and so on, so that for any \( k \), the probability that the loop terminates after exactly \( (y+k)^2 - y^2 = 2yk+k^2 \) calls is \( 1/2^{k+1} \). It can be proved that the infinite summation series

\[
\sum_{k \in \text{nat}} (2yk + k^2) / 2^{k+1}
\]

converges and equals \( 2y + 3 \), which is the expected time if initially \( x = y^2 \); if initially \( x < y^2 \), it takes an additional \( y^2 - x \) time. So we write

\[
T \equiv (x > y^2 \Rightarrow t' = \infty) \land (x \leq y^2 \Rightarrow \mathcal{E'}(t' - t) = 2y + 3 + y^2 - x)
\]

To prove the refinement, we apply refinement by parts and refinement by cases:

\[
\begin{align*}
x & = y^2 \land (\text{ok } \frac{1}{2} \text{-or } (y := y + 1. \ x := x + 1. \ t := t + 1. \ x > y^2 \Rightarrow t' = \infty)) \\
\Rightarrow & \ x = y^2 \\
\Rightarrow & \ x > y^2 \Rightarrow t' = \infty
\end{align*}
\]

---

\(^1\)The variant-based rule for eventual termination of probabilistic loops is presented by Morgan [16] in the probabilistic predicate transformer semantics [17], and is shown to be sound and finitary complete. In this thesis, we have not presented that rule formally in the semantics of our model (which is required if we are to present a formal proof that an expected time bound exists whenever the rule can be applied). Our timing approach is more general in that it also works for many cases of terminating loops where no variant satisfying the requirements of the rule can be established (when the state space is infinite); moreover, more information (time) is obtained.
For Ex. 4.12, the reason that no variant can be established is as follows. As long as the loop fails to terminate, it takes longer and longer for x to move from a point where there is a positive probability of termination to the next such point (as y increases). For a variant, the required upper and lower bounds limit the time interval within which there must be a positive probability of termination, but such time interval grows without bound.

As we have discussed, eventual termination can be guaranteed by an expected time bound. On the other hand, it is not true that an expected time bound always exists when eventual termination is guaranteed. Since the expected time is the infinite summation series \( \Sigma_{t_0} \cdot \text{nat} \cdot t_0 \times \mathcal{P}'(t' - t = t_0) \), it is possible that the series does not converge even though \( \mathcal{P}'(t' - t \geq t_0) \) converges to zero. A notable example is the one-dimensional random walk with one absorbing boundary, which may be expressed in our model as follows:

\[
T \leftarrow \text{if } x = 0 \text{ then ok else } (x := x + 1.5 - \text{or } x := x - 1. t := t + 1. T)
\]

From standard random walk results, the computation is guaranteed to terminate eventually, but the expected time is infinite. (This relies on the fact that the probability of the probabilistic composition in the loop is exactly 1/2; if the probability is greater than 1/2 and initially \( x > 0 \), eventual termination is only probabilistic.) We could write a timing expression for the refinement that guarantees eventual termination (i.e. the expression satisfies Def. 4.7), but the timing expression and the proof for it would be rather complicated. Here we use a simpler example to illustrate this situation.

**Example 4.13.** For natural variables \( x \) and \( y \), consider the refinement

\[
T \leftarrow \text{if } x = 2^y \text{ then } (\text{ok 1/2-or } (y := y + 1. x := x + 1. t := t + 1. T)) \\
\quad \quad \text{else } (x := x + 1. t := t + 1. T)
\]
This example is similar to Ex. 4.12, except that the only time there is a positive probability of termination is when \( x \) is a power of 2. Following the (informal) argument in Ex. 4.12, if initially \( x = 2^y \), for any \( k \), the probability that the loop terminates after exactly \( 2^{y+k} - 2^y = 2^y(2^k - 1) \) calls is \( 1/2^{k+1} \). However, the infinite summation series
\[
\sum k : \text{nat} \cdot 2^y(2^k - 1)/2^{k+1}
\]
does not converge, and we therefore guess that the expected time is infinite.

To see that eventual termination is guaranteed for any prestate satisfying \( x \leq 2^y \), we use the following timing expression, which specifies the probability density function of the running time:
\[
T = x \leq 2^y \Rightarrow \forall k : \text{nat} \cdot \mathcal{P}'(t' - t = 2^{y+k} - x) = 1/2^{k+1}
\]
To prove the refinement, we apply refinement by cases: the case \( x = 2^y \) is straightforward from law \( \text{PC-10} \), and the case \( x \neq 2^y \) is trivial. For eventual termination, for any \( p > 0 \), let \( t_0 = 2^{\text{ceil}(y - \log p)} - x \), then for any \( \mathcal{P}' \),
\[
T \land x \leq 2^y \Rightarrow \mathcal{P}'(t' - t \geq t_0) = \sum k : \text{ceil}(-\log p) \cdot 1/2^{k+1} = 1/2^{\text{ceil}(\log p)} \leq p
\]
By Def. 4.7, eventual termination is guaranteed for \( x \leq 2^y \). From this timing expression, we can prove
\[
T \land x \leq 2^y \Rightarrow \mathcal{E}'(t' - t) = \sum k : \text{nat} \cdot (2^{y+k} - x)/2^{k+1} = \infty
\]
So the expected time is indeed infinite.

\[ \diamond \]

### 4.3.3 Correctness of expected time analysis

Consider the refinement
\[
T \leftarrow \text{ok } 1/2 \\lor (x := x + 1. t := t + 1. T)
\]
The expected time of the computation is 1, and as desired, the refinement is a theorem when
\[
T = \mathcal{E}'(t' - t) = 1
\]
However, the refinement is also a theorem when
\[
T = \mathcal{E}'(t' - t) = 2^x + 1
\]
but the expected time is not correct of the computation. As a matter of fact, the expected time analysis discussed in the previous section is not universally sound. To account for this, we inspect the recursive refinement as follows:
\[
T \leftarrow \text{ok } 1/2 \\lor (x := x + 1. t := t + 1. T)
\]
\[
\leftarrow \text{ok } 1/2 \\lor (x := x + 1. t := t + 1. (\text{ok } 1/2 \\lor (x := x + 1. t := t + 1. T)))
\]
\[
\leftarrow \text{ok } 1/2 \\lor ((x := x + 1. t := t + 1) 1/2 \\lor (x := x + 2. t := t + 2. T))
\]
\[
\leftarrow \text{ok } 1/2 \\lor (x := x + 1. t := t + 1) 1/4 \lor (x := x + 2. t := t + 2. T)
\]
\[
\leftarrow \ldots
\]
\[
\leftarrow \text{ok } 1/2 \\lor (x := x + 1. t := t + 1) 1/8 \lor (x := x + 2. t := t + 2) 1/8 \lor \ldots
\]
\[
(1/2)^{t_0} \\lor (x := x + t_0. t := t + t_0. T)
\]
For any natural \( t_0 \), if the computation has not terminated after \( t_0 \) recursive calls, \( x \) has increased by \( t_0 \), and the loop continues. The expected time of the computation can be seen as

\[
(\Sigma i: 0, \ldots, t_0 \cdot i \times (1/2^{i+1})) + \alpha_{t_0} \times (1/2^{t_0})
\]

where \( \alpha_{t_0} \) is the expected time of the computation \( x := x + t_0 \cdot t := t + t_0 \cdot T \); here \( T \) can be viewed as the remaining computation after \( t_0 \) recursive calls (given that the computation has not terminated), with the poststate after the first \( t_0 \) calls supplied as the prestate of the remaining computation. If we take the limit of the above expression as \( t_0 \) grows without bound, the expected time of the computation is

\[
(\Sigma i: \text{nat} \cdot i \times (1/2^{i+1})) + (\lim \ t_0 \cdot \alpha_{t_0} \times (1/2^{t_0}))\infty
\]

The summation part is the “real” expected time of the computation, and the other part must therefore be equal to zero. If we write \( T = \mathcal{E}'(t' - t) = f x \), we have

\[
x := x + t_0 \cdot t := t + t_0 \cdot T = \mathcal{E}'(t' - t) = f(x + t_0) + t_0
\]

and so \( \alpha_{t_0} = f(x + t_0) + t_0 \). Since \( (\lim \ t_0 \cdot \alpha_{t_0} \times (1/2^{t_0}))_{\infty} = 0 \), our only concern is with the choice of \( f \) in the timing expression \( T \): not only must it be such that the refinement holds, it must also satisfy

\[
(\lim \ t_0 \cdot f(x + t_0) \times (1/2^{t_0}))\infty = 0
\]

As we can see, if \( f x = 1 \), this is satisfied, so the “real” expected time of the computation is 1; if \( f x = 2^x + 1 \), this is not satisfied, and we see that this value of \( f x \) is exactly greater than the “real” expected time.

In the general case, for a recursive computation with finite expected time, the expected time of the computation is

\[
(\Sigma i: 0, \ldots, t_0 \cdot i \times \mathcal{P}'(t' - t = i)) + \alpha_{t_0} \times \mathcal{P}'(t' - t \geq t_0)
\]

where \( \alpha_{t_0} \) is the expected time of the computation provided it does not terminate in the first \( t_0 \) recursive calls. As \( t_0 \) grows without bound, the summation part is the “real” expected time of the computation, and so we must have

\[
(\lim \ t_0 \cdot \alpha_{t_0} \times \mathcal{P}'(t' - t \geq t_0))\infty = 0
\]

Writing \( f\sigma \) for the expected time \( (f\sigma \) does not mention \( t \) since we assume that the program is not time-dependent), and \( f_{t_0}\sigma \) for the expected time of the remaining computation after \( t_0 \) recursive calls \( (\sigma \) is the prestate of the whole computation), we have

\[
\alpha_{t_0} = f_{t_0}\sigma + t_0
\]
Obviously, we have $f_0 \sigma = f \sigma$; to obtain $f_{t_0} \sigma$, we may apply refinement by steps $t_0$ times as illustrated in the above example (where $f_{t_0} x = f(x + t_0)$).

It can be shown that if the expected time is finite, $(\lim t_0 \cdot t_0 \times P'(t' - t \geq t_0)) \infty = 0$. Our only concern is therefore with $f_{t_0}$: for timing expression $T = E'(t' - t) = f \sigma$, not only must it be such that the refinement holds, but in additional to that, it must satisfy the constraint

$$(\lim t_0 \cdot f_{t_0} \sigma \times P'(t' - t \geq t_0)) \infty = 0$$

If the value of this limit is greater than zero, $f \sigma$ is greater than the "real" expected time by exactly this value.

The limitation discussed above is a concern only if we are interested in the exact expected time of the computation. It is not a concern if we just want to prove an expected time bound (which suffices for eventual termination): by implementability, $f \sigma$ (and hence $f_{t_0} \sigma$) must be nonnegative for any $\sigma$, so any expected time bound that can be proved for the refinement can only be looser than the "real" expected time (though it may be too loose that one may want to find a tighter bound).

Given a timing expression $T = E'(t' - t) = f \sigma$ for which the refinement holds, we may want to verify that $f \sigma$ is the "real" expected time of the computation. We may expand the refinement (using refinement by steps as illustrated above) to find the expression for $f_{t_0} \times P'(t' - t \geq t_0)$ and establish its limit as $t_0$ approaches infinity; if this limit is non-zero, it must be subtracted from $f \sigma$ to obtain the "real" expected time. However, in many situations the verification is quite obvious. If the computation has a worst-execution time bound, the limitation never applies (since $t_0$ is bounded). If $f$ is bounded over all possible states of the computation, so is $f_{t_0}$, and the constraint stated above is satisfied; this is always the case when the state space (excluding time) is finite or bounded in terms of the prestate. For example, in Ex. 4.10, the state variable $x$ is bounded by 0 and its initial value (it either decreases by 1 or remains unchanged in each recursive call and never decreases below 0), so $f = \lambda x \cdot \text{max} 0 3x$ is bounded by $3x$ (for the initial value of $x$) over all possible states of the computation. In Ex. 4.11, $f = \lambda x \cdot N^2 - x^2$ is bounded by $N^2$ for any $x$ satisfying $\text{abs} x \leq N^2$ (the only possible states of the computation if it starts with $x = 0$).

---

The proof is briefly as follows. If the limit is greater than zero, there exists constants $c > 0$ and $k > 0$ such that for any $i \geq k$, $i \times P'(t' - t \geq i) \geq c$, so

$$\Sigma i : k, \ldots \infty \cdot i \times P'(t' - t = i) = (k - 1)P'(t' - t \geq k) + \Sigma i : k, \ldots \infty \cdot P'(t' - t \geq i) \geq \Sigma i : k, \ldots \infty \cdot c/i$$

Since the last summation diverges (because of the divergence of the infinite harmonic series), the expected time must be infinite.
Sometimes it may be easier to prove the refinement for the probability density function of the running time, and then derive the "real" expected time from it. For example, in Ex. 4.12, we can prove the refinement for

\[ T = x \leq y^2 \Rightarrow \forall k: \text{nat} \cdot \mathcal{P}(t' - t = y + k^2 - x) = 1/2^{k+1} \]

and the "real" expected time can be derived from \( T \); in this case no verification is necessary. However, in other situations such as Ex. 4.11, the probability density function of the running time may be hard to find; it may be easier to prove the expected time directly by refinement, even though we may have to verify its validity.

Because of the limitation discussed above, the timing expression \( T' = \mathcal{E}(t' - t) = \infty \) should never be used to prove that the expected time of a computation is infinite, since even if the refinement holds the expected time may be finite. To prove that the expected time is infinite, we must prove the refinement for some other timing expression that implies infinite expected time—an expression for which the limitation does not apply, such as the probability density function of the running time, as in Ex. 4.13.

Although we only discuss expected time in this section, a similar limitation also applies for the expected value of any random variable (which may or may not mention time), as well as for some other situations. If we do not take this into consideration, the computation resulting from recursive refinement of a specification may not satisfy the specification. Intuitively, this limitation may apply for an eventually terminating computation because a computation may behave chaotically as time approaches infinity, in the sense that it may satisfy a specification that is never satisfied at finite time (this is true even for the standard model); and even though such behavior occurs with probability approaching zero, the computation may satisfy the specification if the (increasing) effect of the chaotic behavior on the specification subdues its (decreasing) probability of occurrence, as time approaches infinity. On the other hand, this limitation is never a concern for a computation that has a worst-execution time bound.

It would be interesting to investigate the precise condition under which recursive refinement is not "correct" of a specification. The issue discussed in this section could be analyzed more formally and in further depth, but we leave this to future work.

### 4.4 Expanding the programming language

In this section we explore a few possibilities to extend the basic programming language in Def. 4.6: variable declaration, random number generators and the for-loop. There are other possibilities, such as independent composition and communication; but we leave these to future work.
4.4.1 Variable declaration

In the standard model, local variable declaration is defined simply as existential quantification of the new variable in the prestate and the poststate (Section A.7.1). In the probabilistic model, this is invalid since the poststate is not a variable of a probabilistic specification (though the prestate can be treated the same way). Following a discussion in Section 3.2.2, we add a variable to the state within the scope of the declaration; the new variable (in the poststate) is contained in the (list) variable of $P'$ within the scope of the declaration, but not in the (list) variable of $P'$ outside the scope of the declaration.

\[
\text{var } x : D_x \cdot S = \exists x : D_x \cdot \exists S : D_{\sigma^+}[D_x] \to 0,1.
\]

\[
P' = (\lambda \sigma' + [x'] \cdot \sigma') \circ S \land (\text{substitute } S \text{ for } P' \text{ in } \langle S \rangle_a)
\]

\[
= \exists x : D_x \cdot \exists S : D_{\sigma^+}[D_x] \to 0,1.
\]

\[
(\forall \sigma'_0 : D_{\sigma^+} \cdot P' \sigma'_0 = S(\sigma = \sigma'_0)) \land (\text{substitute } S \text{ for } P' \text{ in } \langle S \rangle_a)
\]

We add a variable $x$ to the state within the scope of the declaration, so the (list) variable of $P'$ in general specification $S$ is $\sigma^+ + [x']$. Since $x$ is not contained in the state outside the scope of the declaration, the (list) variable of $P'$ in var $x : D_x \cdot S$ is $\sigma$ only. Writing $\mathcal{Y}$ for the function $\lambda \sigma^+ + [x'] \cdot \sigma'$, for any $\sigma'_0$, $(\mathcal{Y} \circ P') \sigma'_0$ in $S$ is the the probability of the event that the value of $\sigma'$ in the poststate $\sigma^+ + [x']$ is $\sigma'_0$; this equals the probability $P' \sigma'_0$ in var $x : D_x \cdot S$. The poststate distribution $P'$ of $S$ is therefore renamed to $S$, existentially quantified, and the poststate distribution $P'$ for var $x : D_x \cdot S$ is the probability density function $\mathcal{Y} \circ S$.

**Example 4.14.** For integer variable $x$:

\[
\text{var } y : \text{int} \cdot y := 0 \text{ or } y := 1, x := x + y + 1
\]

\[
= \text{var } y : \text{int} \cdot P'(y = 0 \land x' = x) = P'(y = 1 \land x' = x + 1) = \frac{1}{6}
\]

\[
\land P'(y = 0 \land x' = x + 1) = P'(y = 1 \land x' = x + 2) = \frac{1}{3}
\]

\[
= \exists y : \text{int} \cdot \exists S : [\text{int}; \text{int}] \to 0,1.
\]

\[
P' = (\lambda [x'; y] \cdot x') \circ S
\]

\[
\land S(y' = 0 \land x' = x) = S(y' = 1 \land x' = x + 1) = \frac{1}{6}
\]

\[
\land S(y' = 0 \land x' = x + 1) = S(y' = 1 \land x' = x + 2) = \frac{1}{3}
\]

\[
= P'(x' = x) = \frac{1}{6} \land P'(x' = x + 1) = \frac{1}{2} \land P'(x' = x + 2) = \frac{1}{3}
\]

We can declare several variables in one declaration using a list variable; each item in the list is conveniently referred to by a name. If the (list) variable declared is $\varsigma$, the local state within the scope of the declaration is $\sigma^+ \varsigma$. For example,

\[
\text{var } [x; y] : [\text{int}; \text{nat}] \cdot S = \exists [x; y] : [\text{int}; \text{nat}] \cdot \exists S : D_{\sigma^+}[\text{int}; \text{nat}] \to 0,1.
\]

\[
P' = (\lambda \sigma^+ + [x'; y'] \cdot \sigma') \circ S \land (\text{substitute } S \text{ for } P' \text{ in } \langle S \rangle_a)
\]

declares two local variables $x$ (an integer) and $y$ (a natural) in $S$. 
Here we state two laws for variable declaration. In the following, x and y are local variables in probabilistic specification $S$; $\zeta$ is a local list variable in standard specification $U$.

**VD-1**

$\text{var}[x;y]:[D_x;D_y]\cdot S = \text{var }x:D_x\cdot \text{var }y:D_y\cdot S
= \text{var }y:D_y\cdot \text{var }x:D_x\cdot S$

**VD-2**

$\text{var }\zeta:D_\zeta\cdot U = \langle \exists \zeta',\zeta':D_\zeta\cdot U \rangle_a$

Law **VD-1** says that declaring two variables in one declaration is the same as declaring them in two declarations, one at a time. (This is not as trivial as in the standard model; though, it follows easily from law **PF-1**, which says that the probability density function of $Z$ with respect to the probability density function of $X$ is equal to the probability density function of the function composition $Z \cdot X$.) Law **VD-2** says that if the specification in which a local variable is declared is standard, the resultant is equivalent to the general specification resulting from the standard definition of variable declaration.

### 4.4.2 Random number generators

So far we have introduced one programming notation for probabilistic choice, $R\ p\text{-or }S$. The other specification notation for probabilistic choice, $Q \triangleright \lambda \varphi \cdot S$, is not included in the programming language because for arbitrary $Q$ it is hard to implement on a computer. On the other hand, many languages do not include a syntax for probabilistic composition as defined in our model; instead, they implement probabilistic behavior with the use of *random number generators*.³ We shall assume that we have at our disposal an implemented random number generator $\text{rand}$: for positive integer $n$, a call to $\text{rand }n$ returns a natural number in $0,..,n$, with equal probability for each number. Each number returned by $\text{rand }n$ is independent of the numbers returned on previous calls. Here we see how $\text{rand}$ can be incorporated in our programming language.

In essence, we relate each call to $\text{rand }n$ with an expression of the type $Q \triangleright \lambda i \cdot S$, where $Q$ is the *uniform distribution* $0,..,n \rightarrow 1/n$, so that $Q$ gives a uniform probability $1/n$ for each number in $0,..,n$. We first consider the assignment of a random number to a state variable:

\[
x := \text{rand }n = (0,..,n \rightarrow 1/n) \triangleright \lambda i \cdot x := i
\]

³In practice, probabilistic behavior is adequately implemented with the use of *pseudorandom number generators*. Numbers generated by a pseudorandom number generator are deterministic (thus not truly random), but they "look" statistically random in that the procedure makes any relationship among the numbers unnoticeable by most applications. Further discussion of this issue is beyond the scope of this thesis.
More generally, a call to \( \text{rand} \, n \) can appear in any implemented number expression to be evaluated by the computation; more than one call can appear in one number expression. In our programming language, the "implemented number expression" refers to the expression \( e \) in \( x := e \), and any number expression that may appear in the boolean expression \( b \) in if \( b \) then \( R \) else \( S \). For each call to \( \text{rand} \, n \), we declare a local variable (one for each call) and assign \( \text{rand} \, n \) to it, then use dependent composition with the remaining program, where each call in the implemented number expression is replaced by the corresponding local variable. This is illustrated in the following example.

**Example 4.15.** For integer variable \( x \):

\[
\begin{align*}
\text{if } \text{rand} \, 2 = 0 \text{ then } x &:= 2 \times \text{rand} \, 6 \text{ else } x := \text{rand} \, 6 + \text{rand} \, 6 \\
= \quad & \text{var } l : \text{nat} \cdot l := \text{rand} \, 2 . \\
& \text{if } l = 0 \text{ then } (\text{var } m : \text{nat} \cdot m := \text{rand} \, 6 . \, x := 2 \times m) \\
& \text{else } (\text{var } m, n : \text{nat} \cdot m := \text{rand} \, 6 . \, n := \text{rand} \, 6 . \, x := m + n) \\
= \quad & \text{var } l : \text{nat} \cdot (0, \ldots, 2 \rightarrow 1/2 \triangleright \lambda i . l := i) . \\
& \text{if } l = 0 \text{ then } (\text{var } m : \text{nat} \cdot (0, \ldots, 6 \rightarrow 1/6 \triangleright \lambda i . m := i) . \, x := 2 \times m) \\
& \text{else } (\text{var } m, n : \text{nat} \cdot (0, \ldots, 6 \rightarrow 1/6 \triangleright \lambda i . m := i) . \\
& \quad (0, \ldots, 6 \rightarrow 1/6 \triangleright \lambda i . n := i) . \, x := m + n) \\
= \quad & \text{var } l : \text{nat} \cdot l := 0 \frac{1}{2} \triangleright l := 1 . \\
& \text{if } l = 0 \text{ then } (\text{var } m : \text{nat} \cdot \forall i : 0, \ldots, 6 . \, \mathcal{P}'(m' = i \land x' = 2i \land l' = l) = 1/6) \\
& \text{else } (\text{var } m, n : \text{nat} \cdot \forall i, j : 0, \ldots, 6 . \, \mathcal{P}'(m' = i \land n' = j \land x' = i + j \land l' = l) = 1/36) \\
= \quad & \text{var } l : \text{nat} \cdot \quad (l := 0 . \, \forall i : 0, \ldots, 6 . \, \mathcal{P}'(x' = 2i \land l' = l) = 1/6) \\
& \quad \frac{1}{2} \triangleright (l := 1 . \, \forall i : 0, \ldots, 11 . \, \mathcal{P}'(x' = i \land l' = l) = (6 - \text{abs}(5 - i))/36) \\
= \quad & (\forall i : 0, \ldots, 6 . \, \mathcal{P}'(x' = 2i) = 1/6) \frac{1}{2} \triangleright (\forall i : 0, \ldots, 11 . \, \mathcal{P}'(x' = i) = (6 - \text{abs}(5 - i))/36) \\
= \quad & \forall i : 0, \ldots, 11 . \, \mathcal{P}'(x' = i) = \text{if even } i \text{ then } (12 - \text{abs}(5 - i))/72 \\
& \text{else } (6 - \text{abs}(5 - i))/72 \\
\end{align*}
\]

Note that the expression \( \text{rand} \, 6 + \text{rand} \, 6 \) is not the same as the expression \( 2 \times \text{rand} \, 6 \) (for the former two random numbers are added, while for the latter a random number is multiplied by the number 2). Although \( \text{rand} \) appears as a function (and \( \text{rand} \, n \) appears as function application), it is in fact not a function; and \( x := \text{rand} \, n \) is not an assignment, but rather a probabilistic composition of assignments. To avoid inconsistency, we must get rid of \( \text{rand} \) expressions before applying any theory.

Given specification \( 0, \ldots, n \rightarrow 1/n \triangleright \lambda i . S \), it can be implemented by the program

\[
\text{var } i : \text{nat} \cdot i := \text{rand} \, n . \, S
\]

where \( S \) is an implemented specification (refinement by steps).

If \( p \)-or is not included in the programming language, \( R \not\forall S \) can be implemented by a random number generator provided that \( p \) is rational and we can find natural expressions.
c and d such that \( p = \frac{c}{d} \):

\[
R_{\frac{c}{d}} S \equiv \text{if } \text{rand } d < c \text{ then } R \text{ else } S
\]

Though not practically useful, we can also implement \text{rand } n in terms of \( R_{\frac{p}{q}} S \); see Section 4.5.1.

The running time of \text{rand } n depends on the implementation of the random number generator. For algorithmic analysis, we may be more interested in the numbers of calls to \text{rand } n during the computation. We can simply add a state variable \( r \), which represents the number of calls to \text{rand } n. For each call to \text{rand } n, we insert \( r := r + 1 \) correspondingly in the program (by dependent composition, just before the call); we can then talk about \( r' - r \) in a specification (the same way we talk about recursive time; we can talk about expected number of calls and worst-execution number of calls).

### 4.4.3 For loop

We have introduced two loop constructs in the standard model: the while-loop and the for-loop. The while-loop can be used in exactly the same way as in the standard model, in either account mentioned in Section A.7.2. (The only dissimilarity is that for the second account, the universal variable \( \sigma' \) in the induction axiom is replaced by \( \mathcal{P}' \).) For the for-loop, we take a slightly different approach than that presented in Section A.7.3; this account can be used for the standard model as well.

We again use the syntax

\[
\text{for } i := m;..n \text{ do } S
\]

where \( i \) is a fresh name, \( m \) and \( n \) are integer expressions such that \( m \leq n \), and \( S \) is a program; \( i \) is a local constant in \( S \). Again, we do not equate the for-loop to a specification, but instead show how it is used in refinement. Let \( A: \text{int} \rightarrow \text{bool} \), so that \( Ai \) is a specification. Then we can use the refinement

\[
P \leftarrow \text{for } i := m;\ldots n \text{ do } S
\]

if the following are theorems:

\[
\begin{align*}
\text{ok } & \Rightarrow \text{Am} \\
\forall i: m,..n \cdot (Ai \cdot S) & \Rightarrow A(i+1) \\
An & \Rightarrow P
\end{align*}
\]

Alternatively, let \( Z: \text{int} \rightarrow \text{bool} \), so that \( Zi \) is a specification. Then we can use the refinement

\[
P \leftarrow \text{for } i := m;\ldots n \text{ do } S
\]
if the following are theorems:

\[ P \Leftarrow Z m \]
\[ \forall i : m, \ldots n \cdot Z i \Leftarrow (S \cdot Z(i+1)) \]
\[ Z n \Leftarrow \text{ok} \]

This account of the \texttt{for}-loop is better than that presented in Section A.7.3 in that there is no restriction for the type of specifications that can be used for \( P \). For example, the account in Section A.7.3 is inadequate for the specification \( x' = x \times 2^n \), and we have to introduce a new variable for \( 2^n \). For this account, we can use the refinement

\[ x' = x \times 2^n \Leftarrow \text{for } i := 0 ; \ldots n \text{ do } x := 2 \times x \]

if we let

\[ A i = x' = x \times 2^i \]

or alternatively

\[ Z i = x' = x \times 2^{n-i} \]

This account also allows us to talk about recursive time of the \texttt{for}-loop more succinctly: we can simply replace \( S \) by \( (t := t+1 \cdot S) \), then include time in \( A i \) or \( Z i \).

An example of a probabilistic program using a \texttt{for}-loop is given in Section 4.5.2.

### 4.5 Examples of program development

In this section we present three examples to illustrate the development of probabilistic programs.

#### 4.5.1 Random number assignment

We want to implement random number assignment (Section 4.4.2) using our basic programming language in Def. 4.6. (In practice \texttt{rand} is implemented but not \texttt{p-or}, but anyway, we just use it as an illustrative example.) For natural variable \( x \) and positive integer variable \( n \), the problem is

\[ P = (0, \ldots n \rightarrow 1/n) \triangleright \lambda i \cdot x' = i \iff \forall i : 0, \ldots n \cdot P'(x' = i) = 1/n \]

Here we present two solutions. In our first solution, we consider one number at a time: with probability \( 1/n \), we assign the number \( n - 1 \) to \( x \); with probability \( (n-1)/n \), we assign a random number in \( 0, \ldots n-1 \) to \( x \). The solution is as follows.

\[ P \Leftarrow x := n - 1 \frac{1}{n} \text{-or} (n := n - 1 \cdot P) \]
The refinement can be easily proved using law **PC-10**. For timing, consider the refinement
\[ T \Leftarrow x := n - 1 \quad \text{or} \quad (n := n - 1. \ t := t + 1. \ T) \]
We can easily prove the refinement for the worst-execution time bound
\[ T = \langle t' \leq t + n - 1 \rangle_a \]
using refinement by cases (in the proof, we have to consider the case \( n = 1 \) separately), and the expected time
\[ T = \mathcal{E}'(t' - t) = (n - 1)/2 \]
using law **PC-11**. Since we have proved a worst-execution time bound, we are convinced that \((n - 1)/2\) is the "real" expected time of the computation (see Section 4.3.3).

Our first solution is linear in both the worst-execution time and the expected time, and we can do a lot better. In our second solution, we test whether \( n \) is even or odd; in the odd case we make no improvement; in the even case we cut \( n \) in half, select a half randomly, and assign a random number in the selected half to \( x \). Precisely, we use \( x \) to keep track of the smallest number in the selected half, then add a random number in \( 0, \ldots, n/2 \) to \( x \). Let
\[ Q = \forall i:0,\ldots,n. \ P'(x' = x + i) = 1/n \]
which says that a random number in \( 0, \ldots, n \) is added to \( x \). The solution is as follows.
\[ P \Leftarrow x := 0. \ Q \]
\[ Q \Leftarrow \text{if even } n \text{ then even } n \Rightarrow Q \text{ else odd } n \Rightarrow Q \]
\[ \text{odd } n \Rightarrow Q \Leftarrow x := x + n - 1 \quad \text{or} \quad (n := n - 1. \ \text{even } n \Rightarrow Q) \]
\[ \text{even } n \Rightarrow Q \Leftarrow n := n/2. \ (\text{ok } \frac{1}{2} \quad \text{or} \quad x := x + n). \ Q \]
Each refinement can be easily proved (using law **PC-10** for the last two). For timing, consider the refinements
\[ T \Leftarrow x := 0. \ T \]
\[ T \Leftarrow \text{if even } n \text{ then even } n \Rightarrow T \text{ else odd } n \Rightarrow T \]
\[ \text{odd } n \Rightarrow T \Leftarrow x := x + n - 1 \quad \text{or} \quad (n := n - 1. \ \text{even } n \Rightarrow T) \]
\[ \text{even } n \Rightarrow T \Leftarrow n := n/2. \ (\text{ok } \frac{1}{2} \quad \text{or} \quad x := x + n). \ t := t + 1. \ T \]
We can easily prove the refinements for the worst-execution time bound
\[ T = \langle t' \leq t + \log n \rangle_a \]
and the expected time can be no worse. (The exact expected time is somewhat trickier to find, since it depends on whether \( n \) becomes odd any time during the execution; however, it can be proved that if \( n \) is a power of 2, the expected time is equal to the worst-execution time bound, which is logarithmic.)
4.5.2 Binomial experiment

A binomial experiment is an experiment consisting of \( n \) independent trials, each with probability of success \( p \) and probability of failure \( 1 - p \). We want to assign the total number of successes to natural variable \( x \), so that \( x' \) follows a binomial distribution. The problem is

\[
P = \forall k: 0, \ldots, n \cdot \mathcal{P}'(x' = k) = \binom{n}{k} p^k (1 - p)^{n-k}
\]

where \( \binom{n}{k} = n!/(k! \times (n-k)! \) and ! is the factorial function \( n! = \Pi_{i=1}^{n} i \). To solve the problem, we simply implement the binomial experiment, and we do this using a for-loop (Section 4.4.3):

\[
P \Leftarrow x := 0. \ x = 0 \Rightarrow P
\]

\[
x = 0 \Rightarrow P \Leftarrow \text{for } i := 0; \ldots, n \text{ do } x := x + 1 \ p\text{-or ok}
\]

And we must prove the refinements. The first one is trivial; to prove the refinement for the for-loop, let

\[
A_i = x = 0 \Rightarrow \forall k: 0, \ldots, i \cdot \mathcal{P}'(x' = k) = \binom{i}{k} p^k (1 - p)^{i-k}
\]

We see that

\[
A_0 = x = 0 \Rightarrow (x' = 0) \Leftarrow \text{ ok}
\]

\[
A_n = x = 0 \Rightarrow P
\]

So it remains to prove

\[
\forall i: 0, \ldots, n \cdot (A_i. \ x := x + 1 \ p \ Y \ \text{ ok}) \Rightarrow A(i+1)
\]

For \( i: 0, \ldots, n \), we see that

\[
(\forall k: 0, \ldots, i \cdot \mathcal{P}'(x' = k) = \binom{i}{k} p^k (1 - p)^{i-k}). \ x := x + 1 \ p \ Y \ \text{ ok}
\]

\[
= (\forall k: 0, \ldots, i \cdot \mathcal{P}'(x' = k + 1) = \binom{i}{k} p^k (1 - p)^{i-k})
\]

\[
p \ Y \ (\forall k: 0, \ldots, i \cdot \mathcal{P}'(x' = k) = \binom{i}{k} p^k (1 - p)^{i-k})
\]

\[
= (\forall k: 1, \ldots, i+1 \cdot \mathcal{P}'(x' = k) = \binom{i}{k} p^k (1 - p)^{i-k})
\]

\[
p \ Y \ (\forall k: 0, \ldots, i \cdot \mathcal{P}'(x' = k) = \binom{i}{k} p^k (1 - p)^{i-k})
\]

\[
= \mathcal{P}'(x' = 0) = \binom{i}{0} (1 - p)^{i+1} \land \mathcal{P}'(x' = i + 1) = \binom{i}{i} p^{i+1}
\]

\[
\land \forall k: 1, \ldots, i \cdot \mathcal{P}'(x' = k) = \binom{i}{k} p^k (1 - p)^{i-k}
\]

\[
= \mathcal{P}'(x' = 0) = (1 - p)^{i+1} \land \mathcal{P}'(x' = i + 1) = p^{i+1}
\]

\[
\land \forall k: 1, \ldots, i \cdot \mathcal{P}'(x' = k) = \binom{i+1}{k} p^k (1 - p)^{i+1-k}
\]

\[
= \forall k: 0, \ldots, i + 1 \cdot \mathcal{P}'(x' = k) = \binom{i+1}{k} p^k (1 - p)^{i+1-k}
\]
And hence
\[
A_i \cdot x := x + 1 \cdot p \land \text{ok} \\
= (x = 0 \Rightarrow \forall k:0,..,i \cdot \mathcal{P}'(x' = k) = \binom{i}{k} p^k (1-p)^{i-k}) \cdot x := x + 1 \cdot p \land \text{ok} \\
\Rightarrow x = 0 \Rightarrow (\forall k:0,..,i \cdot \mathcal{P}'(x' = k) = \binom{i}{k} p^k (1-p)^{i-k}) \cdot x := x + 1 \cdot p \land \text{ok} \\
= x = 0 \Rightarrow \forall k:0,..,i+1 \cdot \mathcal{P}'(x' = k) = \binom{i+1}{k} p^k (1-p)^{i+1-k} \\
= A(i+1)
\]

Alternatively, to prove the refinement for the for-loop, we may let
\[
Z_i = \forall k:0,..,(n-i) \cdot \mathcal{P}'(x' = x+k) = \binom{n-i}{k} p^k (1-p)^{n-i-k}
\]
The proof is similar and omitted. For timing, let \( A_i = t' = t+i \), we have
\[
t' = t + n \iff \text{for } i := 0;..;n \text{ do } (t := t + 1. x := x + 1 \cdot p - \text{or} \text{ ok})
\]
We can also solve the problem using recursive refinement. Let
\[
Q = \forall k:0,..,n \cdot \mathcal{P}'(x' = x+k) = \binom{n}{k} p^k (1-p)^{n-k}
\]
Then the solution, including time, is as follows:
\[
P \land t' = t+n \iff x := 0. Q \land t' = t+n \\
Q \land t' = t+n \iff \text{if } n = 0 \text{ then ok} \\
\quad \text{else } ((\text{ok p-or x := x+1}). n := n-1. t := t+1. Q \land t' = t+n)
\]

### 4.5.3 Random permutation

For \( 0 \leq k \leq n \), we want to assign \( k \) random numbers in \( 0,..n \) to a list \( L \), so that each number is assigned at most once. The number of ways of permuting \( k \) distinct numbers in \( 0,..n \) is \( n!/(n-k)! \). Let \( K \) be the bunch of all \( k \)-permutations of \( 0,..n \), that is,
\[
K = \{ M : k*(0,..n) \} \cdot \forall i,j:0,..,k \cdot i \neq j \Rightarrow Mi \neq Mj
\]
The problem is
\[
P = \forall M : K \cdot \mathcal{P}'(L' = M) = (n-k)!/n! \\
\]
First, we show an obvious solution: starting with \( L = [\text{nil}] \) (the empty list), we take a random number in \( 0,..n \) one at a time, and if it differs from all numbers in \( L \), it is catenated to \( L \). Let
\[
A = \forall i,j:0,..,\#L \cdot Li:0,..n \land Li \neq Lj
\]
which says that \( L \) is a \( \#L \)-permutation of \( 0,..n \). Let
\[
Q = \forall M : K \cdot M[0;..,\#L] = L \Rightarrow \mathcal{P}'(L' = M) = (n-(k-\#L))!/n!
\]
which says that a random permutation of \( k - \#L \) numbers (all not in \( L \)) is catenated to \( L \), so that \( L' \) is a \( k \)-permutation of \( 0, \ldots, n \). Our first refinement is

\[
P \Leftarrow L := [\text{nil}]. \ #L \leq k \land A \Rightarrow Q
\]

To solve \( \#L \leq k \land A \Rightarrow Q \), we introduce a local variable \( x \) for the assignment of a random number in \( 0, \ldots, n \), and to check that \( x \) differs from all numbers in \( L \), we introduce a local variable \( i \) to keep track of the index in \( L \) we have checked thus far. Let

\[
B = x : 0, \ldots, n \land i \leq \#L \land \neg x : L(0, \ldots, i)
\]

which says that \( x \) differs from all numbers checked thus far. Let

\[
R = \begin{cases} 
Q & \text{if } x : L(0, \ldots, \#L) \\
\forall M : K \cdot M[0, \ldots, \#L+1] = L^+[x] \Rightarrow P'(L' = M) = (n - (k - \#L - 1))!/n! & \text{else}
\end{cases}
\]

which says that if \( x \) is in \( L \), then the computation is according to \( Q \) (\( x \) is discarded and reselected); otherwise (\( x \) differs from all numbers in \( L \)), the computation appends \( x \) followed by a random permutation of \( k - \#L - 1 \) numbers (all not in \( L \)) to \( L \). The refinements are as follows.

\[
\begin{align*}
\#L \leq k \land A \Rightarrow Q & \Leftarrow \text{if } \#L = k \text{ then } \text{ok} \text{ else } \#L < k \land A \Rightarrow Q \\
\#L < k \land A \Rightarrow Q & \Leftarrow \text{var } x, i : \text{nat}; x := \text{rand } n. \ i := 0. \ \#L < k \land A \land B \Rightarrow R \\
\#L < k \land A \land B \Rightarrow R & \Leftarrow \text{if } i = \#L \text{ then } (L := L^+[x]. \ \#L \leq k \land A \Rightarrow Q) \\
& \quad \text{else if } L_i = x \text{ then } \#L < k \land A \Rightarrow Q \\
& \quad \text{else } (i := i + 1. \ \#L < k \land A \land B \Rightarrow R)
\end{align*}
\]

The proofs of all the refinements are straightforward. For timing, we put \( t := t + 1 \) before each of the three recursive calls in the refinement solution of \( \#L < k \land A \land B \Rightarrow R \), and we have to prove

\[
T \Leftarrow L := [\text{nil}]. \ #L \leq k \land A \Rightarrow U \\
\#L \leq k \land A \Rightarrow U & \Leftarrow \text{if } \#L = k \text{ then } \text{ok} \text{ else } \#L < k \land A \Rightarrow U \\
\#L < k \land A \Rightarrow U & \Leftarrow \text{var } x, i : \text{nat}; x := \text{rand } n. \ i := 0. \ \#L < k \land A \land B \Rightarrow V \\
\#L < k \land A \land B \Rightarrow V & \Leftarrow \\
& \quad \text{if } i = \#L \text{ then } (L := L^+[x]. \ t := t + 1. \ \#L \leq k \land A \Rightarrow U) \\
& \quad \text{else if } L_i = x \text{ then } (t := t + 1. \ \#L < k \land A \Rightarrow Q) \\
& \quad \text{else } (i := i + 1. \ t := t + 1. \ \#L < k \land A \land B \Rightarrow R)
\]

for suitable timing expressions \( T, U \) and \( V \). We see that no worst-execution time bound exists for the computation unless \( k \leq 1 \): if the random number \( x \) selected in the execution of \( x := \text{rand } n \) is already in \( L \), \( x \) is discarded and reselected, and this may go on for an arbitrarily long time.
For expected time analysis, we first find an expression for the expected time $e$ required to catenate a number to $L$. When $x := \text{rand } n$ is executed, the probability that $x$ is not in $L$ is $(n - \#L)/n$, in which case it takes $\#L + 1$ recursive time to search through $L$ and catenate $x$ to $L$; the probability that $x$ is in $L$ is $\#L/n$, in which case it takes $(\$j \cdot Lj = x) + 1$ recursive time to search $L$ up to the location of $x$ in $L$ (there is at most one location of $x$ in $L$), and since the probability is equal for each location, the expected time for searching $L$ is $(\#L + 1)/2$; then $x$ is reselected, and the expected time for reselecting $x$ is $e$. Solving the equation

$$e = \frac{n - \#L}{n} \times (\#L + 1) + \frac{\#L}{n} \times \left(\frac{\#L + 1}{2} + e\right)$$

we get

$$e = \#L + 1 + \frac{\#L(\#L + 1)}{2(n - \#L)}$$

The expected time of the computation is the summation of the expected time required to catenate each number to $L$. We therefore define

$$T = E'(t' - t) = \Sigma j : 0,..,k \cdot f j$$

$$U = E'(t' - t) = \Sigma j : \#L,..,k \cdot f j$$

$$V = \text{if } x : L(0,..,\#L) \text{ then } E'(t' - t) = (\$j \cdot Lj = x) + 1 - i + \sum j : \#L,..,k \cdot f j$$

$$\text{else } E'(t' - t) = \#L + 1 - i + \sum j : \#L + 1,..,k \cdot f j$$

where

$$f = \lambda j \cdot j + 1 + \frac{j(j + 1)}{2(n - j)}$$

All the refinements can be proved. We see that if $k$ is small compared to $n$, the expected time of the above solution is quadratic in $k$.

Now we present a second solution; the problem $P$ is the same as above. The solution first initializes $L$ to some $n$-permutation of $0,..,n$; then, we just select $k$ random numbers from $L$. Precisely, at each stage $L$ is divided into two parts: one containing the selected numbers, and the other the remaining numbers. Each time a random number from the remaining part is selected, it is swapped with the first number of the remaining part, and becomes the last number of the selected part. Since the remaining part is always distinct from the selected part, we are sure that the selected random number is not in the selected part; this saves a costly step which is required in our first solution. Let $N$ be the bunch of all $n$-permutations ($N$ is $K$ defined above with $k$ substituted by $n$). Let

$$Q = L' : N \land \forall M : K \cdot P'(L'[0;..,k] = M) = (n-k)!/n!$$
which says that $L'$ is a $n$-permutation, and the first $k$ numbers of $L'$ are random. To solve $P$, we simply solve $Q$ and then cut off the remaining part:

$$P \Leftarrow Q. \ L := L[0;..k]$$

To solve $Q$, we introduce a local variable $i$ for the index of the first number of the remaining part of $L$, and start with $L = [0;..n]$ (though any other $n$-permutation also works). Let

$$R \Leftarrow L' : N \land \forall M : K : M[0;..i] = L[0;..i] \Rightarrow P'(L'[0;..k] = M) = (n-(k-i))/n!$$

which says that $L'$ is a $n$-permutation, the first $i$ numbers of $L'$ are unchanged, and the next $k-i$ numbers are random. The refinements are as follows.

$$Q \Leftarrow \var i : nat :: i := 0. \ L := [0;..n]. \ i \leq k \land L : N \Rightarrow R$$

$$i \leq k \land L : N \Rightarrow R \Leftarrow \text{if } i = k \text{ then } \text{ok else } i < k \land L : N \Rightarrow R$$

$$i < k \land L : N \Rightarrow R \Leftarrow \var x : \text{int} :: x := \text{rand}(n-i)+i. \ L := x \rightarrow Li \mid i \rightarrow Lx \mid L. \ i := i + 1 \cdot i \leq k \land L : N \Rightarrow R$$

$$L := x \rightarrow Li \mid i \rightarrow Lx \mid L \Leftarrow \var y : nat :: y := Li. \ L := i \rightarrow Lx \mid L. \ L := x \rightarrow y \mid L$$

Each of these refinements can be proved. We can also prove that the recursive time of the program is always $k$, which is a lot better than the first solution. However, this assumes that the assignment $L := [0;..n]$ (or any other $n$-permutation) does not take any recursive time; in reality, a computer usually needs a recursive time of $n$ to do so. While this solution is a lot more efficient than the first solution when $k$ is comparable to $n$, the first solution is better when $k$ is much smaller than $n$ (when quadratic time in $k$ is better than linear time in $n$). Moreover, this solution requires a memory space of $n$ numbers for the list $L$, but we only ask for $k$ numbers; in the first solution, the memory space required for the list $L$ is just $k$ numbers.
Chapter 5

Conclusions

5.1 Summary

In this thesis, we have extended Hehner's theory of programming [5] to incorporate probabilistic behavior. The underlying semantics is the same as in Hehner's theory—specification are boolean expressions, refinement is implication, and programs are specifications. The only difference is the variables of interest: for the standard model, they are the prestate and the poststate; for the probabilistic model, they are the prestate and the poststate distribution. We have introduced a general form of specifications which allows standard specifications to be embedded in probabilistic specifications.

We have introduced some notations for probabilistic specifications, and have provided some laws for them. We have defined the probabilistic programming language, which is a sublanguage of the specification language, and have illustrated the development of probabilistic programs with some simple examples.

We have discussed the issues of nondeterminism and timing. Timing, which includes termination, is treated the same manner as in Hehner's theory: we simply add a time variable to the state. We have inspected several criteria of timing, in particular, expected time.

With the probabilistic model, we feel that a theoretical foundation for the practical development of probabilistic programs has been established.

5.2 Discussion

Specifications are boolean expressions, and programs are specifications. This formalism provides a satisfactory theoretical foundation for programming, which is practically useful for software engineers; this is strongly reasoned by Hehner [4]. In the framework proposed by Hoare and He [9] for unifying theories of programming, the same formalism is adopted.
It is proposed that this formalism can be used to model any paradigm of programming, and as it turns out, probability is no exception.

A source for our model is the relational model proposed by He et al. [3]. In He’s model, a (probabilistic) program is a mapping from prestates to sets of poststate distributions; semantically, such a mapping can be represented, and is best represented as a boolean expression. (The argument for this, for the standard model, is presented by Hehner [4].)

A marked difference between He’s relational model and our model is the way termination is treated: He’s model employs a “total correctness” semantics, while our model employs a “partial correctness” semantics with a time variable. An account of the benefits of partial correctness plus time over total correctness, for the standard model, is presented by Hehner [6, 4]. His conclusion partly reads as follows:

“A total correctness semantics is not worth its trouble. It is a considerable complication over a partial correctness semantics in order to gain one bit of information of dubious value. Partial correctness, with a time variable, provides more information at less cost.”

The same can be said for the probabilistic model. Partial correctness with a time variable allows us to talk about time in just any way we want; the expected time, the worst-execution time and the probability density function of the running time are just some of the more useful ways. Even for termination, it allows us to distinguish between two criteria: eventual termination and worst-execution termination (with a time bound). In the absence of a time variable, total correctness only provides us a means to talk about the former, but not the latter.

In our semantics, the time variable is treated on a par with any other variable; for the theoretical part, no extraordinary concern about termination is necessary. In a total correctness semantics, the concern about termination necessarily complicates the theory. In He’s model, to allow for non-termination an improper state is added to the state space; the improper state is exploited by theory in a manner different from any other state. For total correctness, it is considered that a program which computation terminates more often than another refines the other; to allow refinement between programs to be defined as set inclusion (which is implication if the programs are interpreted as boolean expressions), the up-closure property is imposed on programs (see Section 1.3). In our semantics, all these complications are not necessary, yet more information can be obtained, and often in a simpler manner: reasoning about timing (and therefore termination) can be done in the context of recursive refinement. This is not possible in a total correctness semantics, and we have to rely on a loop construct instead, such as the while-loop defined as a
least fixed-point (Section A.7.2).

If one is indeed only interested in whether the computation of a program eventually terminates or not, and not interested in efficiency or timing whatsoever, a variant-based rule for eventual termination of probabilistic while-loops, as mentioned informally in Section 4.3.2, may be useful. The expected time analysis is more general than the rule, but the rule may be simpler for termination arguments, though it cannot be used in the context of recursive refinement. Although the rule has not been presented formally in the thesis, a total correctness semantics is not a prerequisite to formalizing the rule; it can be formalized using the time variable.

In Morgan's probabilistic predicate transformer model [17], the semantics of a program is defined as a function from probabilistic postconditions to probabilistic preconditions (see Section 1.3). The probabilistic weakest precondition predicate transformer \( \wp \) maps program \( P \) and probabilistic postcondition \( \beta \) to the minimum expected value of \( \beta \). This provides a formalism for proving properties about probabilistic programs.

However, weakest precondition predicate transformers are not necessary for proving properties about probabilistic programs. Expected values of probabilistic postconditions can be calculated using simple logical implication, and with the use of specification laws, this does not require any more effort than calculating with \( \wp \). For the example in Section 1.3, we can calculate the probability of the event \( \langle x' \leq y' \rangle \) (the expected value of the random variable \( \eta(x' \leq y') \)) as follows.

\[
\begin{align*}
x &:= 1 \frac{1}{3} & y &:= 2.
\end{align*}
\Rightarrow
\begin{align*}
x &:= 1 \frac{1}{3} \text{--or } x := 2. \ P'(x' \leq y') &= \eta(x \leq 1) \frac{1}{2} - \text{or } P'(x' \leq y') = \eta(x \leq 2) \\
x &:= 1 \frac{1}{3} \text{--or } x := 2. \ P'(x' \leq y') &= \frac{1}{2} \eta(x \leq 1) + \frac{1}{2} \eta(x \leq 2)
\end{align*}
\]

The proof is no more complicated than the \( \wp \) calculations shown in Section 1.3. The expression \( \mathcal{P}'(x' \leq y') \) even works backwards through dependent composition, in a manner similar to \( \wp \) calculations. (This approach can be used in general for any expression which does not mention prestates.) While \( \wp \) calculations may be more elegant in style, calculations in the context of logical implication are clearer and more comprehensible, because the underlying model is much simpler.
5.3 Possibilities for further research

The work in this thesis allows many possibilities for further research. Here we briefly describe one such possibility: the homogeneous model. In the homogeneous model, the input to a computation is a probability distribution over prestates, or prestate distribution, denoted $\mathcal{P}$, which characterizes randomized input; the variables of a specification are then $\mathcal{P}$ and $\mathcal{P}'$. This homogeneous view has the benefit that dependent composition can be defined in the same manner as in the standard model, which is much simpler than the probabilistic definition:

$$R \cdot S = \exists \mathcal{P}' \cdot (\text{substitute } \mathcal{P}' \text{ for } \mathcal{P} \text{ in } R) \land (\text{substitute } \mathcal{P}' \text{ for } \mathcal{P} \text{ in } S)$$

A general specification $S$ (whose variables are $\sigma$, $\sigma'$, and $\mathcal{P}'$) can be converted to a homogeneous specification using probabilistic composition of variable alternatives:

$$\mathcal{P} \triangleright \lambda \sigma \cdot S$$

Note that $\mathcal{P}$ is a global variable, and $\sigma$ becomes a local variable in $S$. $\mathcal{P}$ may be a variable of $S$; we thus obtain the most general form of specifications, whose variables are the prestate $\sigma$, the prestate distribution $\mathcal{P}$, the poststate $\sigma'$, and the poststate distribution $\mathcal{P}'$. This allows us to write specifications such as

$$(\forall i : 0 \ldots 6 : \mathcal{P}(x = i) = 1/6) \Rightarrow \mathcal{P}'(y' = 0) = \mathcal{P}'(y' = 1) = 1/2 \land x' = x + 1$$

which says that if $x$ starts as a random number in $0, \ldots, 6$, then $y$ is set to either 0 or 1 with equal probability, and $x$ is increased by 1. This kind of specifications would prove very useful, especially for stepwise refinement in dependent composition. It allows more refinement than probabilistic specifications; for example, the above specification can be refined by

$$y := 0 \text{ or } y := 1. \ x := x + 1$$

and also by

$$\text{if } x < 3 \text{ then } y := 0 \text{ else } y := 1. \ x := x + 1$$

The second refinement takes advantage of the randomized input, so the computation does not have to make a probabilistic choice on its own.

There are other possibilities for further research, such as concurrency, communication, and data transformation; for the standard model, they are discussed in Hehner's book, in [5, Chap. 8], [5, Chap. 9], and [5, p. 116–119], respectively. For any of these possibilities, the probabilistic definition should be an extension of the standard definition. In other words, when the specifications involved are standard, it should make no difference which definition we use, so that we can opt for the simpler standard definition.
Appendix A

The standard model

In this appendix, we present a summary of the standard model for specifications and programs. For details, refer to Hehner's book [5].

A.1 The model of computation

We use a very simple model of computation. Given an input, the computation delivers an output. The input and the output are described as states. A state is a list of any items. There are only two observable states: the initial state, or prestate, denoted by \( \sigma \); and the final state, or poststate, denoted by \( \sigma' \). The bunch of possible states is called the state space, denoted by \( \mathcal{D} \). The components of a state are referred to by state variables, such as \( x \) and \( y \). The domain of a state variable \( v \), denoted by \( D_v \), is the bunch of possible values of the state variable in the state space. The state variables and their domains must be explicitly stated.

A computation is deterministic for a prestate if given the prestate as input, only one poststate can be possibly delivered as output. A computation is nondeterministic for a prestate if there is more than one possible poststate which it may choose to deliver as output, with no indication which it will be. The theory presented here deals with runtime nondeterminism, that is, the choice may be made during an execution, and may vary depending on the prestate. (There are other concepts of nondeterminism, but it is outside the scope of this thesis to deal with them here.)

To talk about time, we simply add a time variable \( t \) to the state. The state \( \sigma = [t; x; y] \) thus consists of a time variable \( t \) and space variables \( x \) and \( y \). To allow for non-termination we take the domain of time to be a number system extended with \( \infty \).
A.2 Specifications

A specification is a denotation of computational behavior. The purpose of a specification is to distinguish observations that satisfy it from those that don't, and the observable quantities of a computation are the prestate and the poststate. A specification is therefore a boolean expression whose (global) variables are the prestate $\sigma$ and the poststate $\sigma'$. Conventionally, we use variables in a specification to refer to the values of state variables in the prestate and the poststate. To refer to the values of state variables in the prestate, we use the state variables themselves. To refer to the values of state variables in the poststate, we put a prime to the names of the variables, such as $x'$ and $y'$. For example, for a state with two components, we may write the prestate $\sigma$ as $[x; y]$, and the poststate $\sigma'$ as $[x'; y']$. If we never use $\sigma$ or $\sigma'$ directly in a specification, it is not necessary to state the order of the variables used to refer to either state.

A.2.1 Definitions

An implemented specification is a specification for which an implementation has been provided, so that the computer can execute it. For a computation to satisfy a specification, the given prestate and the computed poststate must make the specification true. We have an implementation when every computation on any given prestate satisfies the specification. Based on the number of satisfactory poststates for each prestate, we have the following definitions.

**Definition A.1.** For specification $S$,

- $S$ is **satisfiable** for prestate $\sigma$: $\exists \sigma' \cdot S$
- $S$ is **deterministic** for prestate $\sigma$: $\forall \sigma' \cdot S = 1$
- $S$ is **nondeterministic** for prestate $\sigma$: $\exists \sigma' \cdot S \geq 2$
- $S$ is **implementable**: $\forall \sigma \cdot \exists \sigma' \cdot S$

**Example A.1.** For integer variables $x$ and $y$, the specification

$$x' = x + 1$$

is implementable, and can be satisfied by a computation that increases $x$ by 1; it may set $y$ to any value; the specification is thus nondeterministic for every prestate. The specification

$$x \neq 3 \land x' = y' = 0$$

cannot be satisfied by any computation in which the initial value of $x$ is 3, hence it is unimplementable, yet it is deterministic for any other prestate.
A.2.2 Specifications with time

To talk about time, we use the variables $t$ (for initial time) and $t'$ (for final time). Time cannot decrease, which is reflected in the following definition.

Definition A.2. A specification with time $S$ is implementable if and only if

$$\forall \sigma \cdot \exists \sigma' \cdot S \land t' \geq t$$

Hehner's theory [5, p. 51-53] covers both real time and recursive time. In this thesis, we only consider recursive time. If a specification with time implies $t' \leq t + f\sigma$ for some nonnegative function $f$, we say that the specification has an upper time bound (or simply time bound) of $f\sigma$ for prestate $\sigma$. (If $f\sigma = \infty$ for prestate $\sigma$, the time bound is infinite for $\sigma$, which is the same as saying that the time is unbounded for $\sigma$.)

A.3 Specification notations

The specification language is not definitive or restrictive. Any boolean expression whose variables refer to the prestate and the poststate is allowed. When called for, any mathematical notation with a clear meaning can be invented on the spot. Here we highlight some basic specification notations; these notations can be found in Hehner's book [5, p. 38–42,82].

A.3.1 The specification ok

$$\text{ok} \equiv \sigma' = \sigma$$
$$\equiv x' = x \land y' = y \land \ldots$$

The ok notation specifies that the values of all variables in the poststate equal their corresponding values in the prestate. ok can be satisfied by a computation that does nothing.

A.3.2 The specification $x := e$

$$x := e \equiv \text{(substitute } e \text{ for } x \text{ in ok)}$$
$$\equiv x' = e \land y' = y \land \ldots$$

In the assignment notation, $x$ is any state variable, and $e$ is any expression of the prestate. If $e$ is an expression in the domain of $x$, $x := e$ can be satisfied by a computation that evaluates $e$, then assigns the evaluated value to $x$. Otherwise, if $e$ is not in the domain of $x$, $x := e$ is unsatisfiable.
A.3.3 The expression if b then R else S

\[ \text{if } b \text{ then } R \text{ else } S = b \land R \lor \neg b \land S \]

The above notation is called \textit{conditional composition}. \( b \), \( R \), and \( S \) can be any boolean expressions. If \( b \) is a boolean expression of the prestate, and \( R \) and \( S \) are implementable specifications, \( \text{if } b \text{ then } R \text{ else } S \) can be satisfied by a computation that evaluates \( b \), then behaves according to \( R \) if \( b \) is true, or according to \( S \) otherwise. The notation is not restricted to be used for specifications only; it may as well be used in any boolean expressions.

A.3.4 The specification \( R \text{ or } S \)

\[ R \text{ or } S = R \lor S \]

The above notation is called \textit{nondeterministic composition}, which is simply disjunction; the only difference between \( R \text{ or } S \) and \( R \lor S \) is precedence. If \( R \) and \( S \) are implementable specifications, \( R \text{ or } S \) can be satisfied by a computation that either behaves according to \( R \) or according to \( S \), but with no indication which it will be. The choice may be made during an execution, and may vary depending on the prestate.

If exactly one of \( R \) or \( S \) is unsatisfiable for a prestate, \( R \text{ or } S \) can be satisfied by a computation that behaves according to the other specification on that prestate. If both \( R \) and \( S \) are unsatisfiable for a prestate, \( R \text{ or } S \) is unsatisfiable for that prestate, thus unimplementable.

A.3.5 The specification \( R \cdot S \)

\[ R \cdot S = \exists \sigma'' \cdot (\text{substitute } \sigma'' \text{ for } \sigma' \text{ in } R) \land (\text{substitute } \sigma'' \text{ for } \sigma \text{ in } S) \]

The above notation is called \textit{dependent composition}, which characterizes sequential computation. If \( R \) and \( S \) are implementable specifications, \( (R \cdot S) \) can be satisfied by a computation that first behaves according to \( R \), then behaves according to \( S \), with the poststate of \( R \) serving as the prestate of \( S \).

A.4 Refinement

The concept of \textit{refinement} is very important to \textit{program development}, that is, the construction of a program whose computational behavior satisfies a given specification. Refinement is defined as follows.
Definition A.3. Specification \( P \) is refined by specification \( S \) if and only if every computation satisfying \( S \) also satisfies \( P \):

\[
\forall \sigma, \sigma'. \ P \iff S
\]

In other words, refinement is implication, universally quantified. We call \( P \) “the refinement problem” (or just “the problem”) and \( S \) “the refinement solution” (or just “the solution”). In practice, we leave out the universal quantifiers in proofs of refinement, and write \( P \iff S \). Here are some examples of refinement.

Example A.2. For integer variables \( x \) and \( y \):

\[
\begin{align*}
x' > x & \iff x' = x + 1 \land y' > y \\
x' = x + y \land y' = y & \iff x := x + y \\
x' < y' < x & \iff y := x - 1. \ x := y - 1
\end{align*}
\]

A.4.1 Refinement laws

Here we present some laws that not only can help us to prove theorems about refinement, but are also very useful in the development of programs from specifications (this will be discussed in Section A.5).

Theorem A.4. Refinement by steps

If \( A \iff \text{if } b \text{ then } C \text{ else } D \) and \( C \iff E \) and \( D \iff F \) are theorems,
then \( A \iff \text{if } b \text{ then } E \text{ else } F \) is a theorem.

If \( A \iff B \lor C \) and \( B \iff D \) and \( C \iff E \) are theorems,
then \( A \iff D \lor E \) is a theorem.

If \( A \iff B \land C \) and \( B \iff D \) and \( C \iff E \) are theorems,
then \( A \iff D \land E \) is a theorem.

If \( A \iff B \) and \( B \iff C \) are theorems, then \( A \iff C \) is a theorem.

Theorem A.5. Refinement by parts

If \( A \iff \text{if } b \text{ then } C \text{ else } D \) and \( E \iff \text{if } b \text{ then } F \text{ else } G \) are theorems,
then \( A \land E \iff \text{if } b \text{ then } C \land F \text{ else } D \land G \) is a theorem.

If \( A \iff B \lor C \) and \( D \iff E \lor F \) are theorems,
then \( A \land D \iff B \land E \lor C \land F \) is a theorem.

If \( A \iff B \land C \) and \( D \iff E \land F \) are theorems,
then \( A \land D \iff B \land E \land C \land F \) is a theorem.

If \( A \iff B \) and \( C \iff D \) are theorems, then \( A \land C \iff B \land D \) is a theorem.

Theorem A.6. Refinement by cases

\( P \iff \text{if } b \text{ then } Q \text{ else } R \) is a theorem if and only if
\( P \iff b \land Q \) and \( P \iff \neg b \land Q \) are theorems.

\( P \iff Q \) or \( R \) is a theorem if and only if \( P \iff Q \) and \( P \iff R \) are theorems.
A.5 Programs

To have an implementation of a specification, every computation that satisfies the implementation must satisfy the specification; thus an implementation of a specification is an implemented specification (the solution) that refines the given specification (the problem). The *programming language*, the language used to describe implementations, is a sublanguage of the specification language, and *program development* is the process of finding a refinement solution in the programming language. A *program* is an implemented specification; once we have an implementation for a specification, we treat the specification as a program. To execute the specification, just execute the implementation for it.

The freedom of notations exhibited by the specification language cannot be extended to the programming language. The programming language is used to describe implementations, which are to be compiled and executed by the computer. When a computer executes a program, explicit instructions that can be understood by the computer must be provided to it. Programming notations must therefore be selected to ensure compilability and executability, which may depend on the specific environment and the capability of the specific computer. For example, the implementation of recursion may require extra memory from the computer to be used as a stack (except for tail recursion which may be converted by the compiler to an iteration). In modern computers such memory is usually abundant for most applications, and as an abstraction we often assume unlimited availability of such memory. But in a primitive computer where stack memory is not available, the use of recursion may be restricted. Moreover, if we include time, there are many problems that cannot be realistically solved: for example, it is proven impossible to sort a list of integers in linear time, unless the (imaginary) computer has an infinite number of processors.

In our definition, implementability guarantees the existence of a function that maps each prestate to a poststate satisfying the specification; it does not guarantee the existence of a program for the specification (though it is a necessary condition). The feasibility of program development also depends on the programming language selected and the computing resources available. Sorting in linear time is implementable according to our definition, but it cannot be implemented using any programming language understood by any realistic computer—having a finite number of processors.

A.5.1 The programming language

Our basic programming language is as follows.
Definition A.7. The programming language

(a) ok is a program.

(b) If \( x \) is any space variable and \( e \) is an implemented expression of the prestate in the domain of \( x \), then \( x := e \) is a program.

(c) If \( b \) is an implemented boolean expression of the prestate, and \( P \) and \( Q \) are programs, then \( \text{if } b \text{ then } P \text{ else } Q \) is a program.

(d) If \( P \) and \( Q \) are programs, then \( P \text{ or } Q \) is a program.

(e) If \( P \) and \( Q \) are programs, then \( P \cdot Q \) is a program.

(f) If \( P \) is an implementable specification and a program \( S \) is provided such that \( P \leftarrow S \) is a theorem, then \( P \) is a program.

If we require that a computation be deterministic, (d) is excluded from the programming language. Although \( P \text{ or } Q \) is identical to \( P \lor Q \), for clarity \( P \text{ or } Q \) should be used only when \( \text{or} \) is meant to be a program connective for programs \( P \) and \( Q \); if the whole disjunction is a sub-expression in a specification to be refined, \( P \lor Q \) should be used instead.

We have described the computational behavior for (a)–(e) in Section A.3. For (f), "\( P \leftarrow S \) is a theorem" states that \( P \) can be satisfied by any computation that behaves according to \( S \). To execute \( P \), just execute \( S \). The refinement acts as a procedure declaration; \( P \) acts as the procedure name, and \( S \) as the procedure body; use of the name \( P \) acts as a call. Recursion is allowed; to determine whether \( S \) is a program, we can presume that \( P \) is a program. If \( S \) is a program under the presumption, then \( P \) is a program. Whenever \( P \) is encountered during the execution of \( S \), \( S \) is is executed again. The following example illustrates a recursive program.

Example A.3. For integer variable \( x \):

\[
x' \leq 0 \leftarrow \text{if } x \leq 0 \text{ then ok else } (x := x - 1. x' \leq 0)
\]

The problem is \( x' \leq 0 \), and we provide the above refinement (which is a theorem). We presume that \( x' \leq 0 \) is a program, so the solution is a program under the presumption. \( x' \leq 0 \) is now implemented, and is therefore also a program.

In the development of a program, the refinement laws given in Section A.4.1 can be very useful. Examples of program development can be found in Hehner's book [5, Chap. 4].

A.6 Recursive time

To talk about recursive time, we add a time variable to the state, and must modify the program as follows: every recursive call to \( P \) must be replaced with \( (t := t + 1. \ P) \),
indicating that the call to \( P \) costs one time unit. Not every call is considered recursive; the general rule of recursive time is that in every loop of calls, at least one call must be charged with at least one time unit. Except for recursive calls, the program must not use the assignment notation to change the value of the time variable \( t \). (Although we focus on recursive time in this thesis, real time can be treated similarly: each operation is charged with the execution time of the operation. For details, see Hehner’s book [5, p. 51–53].)

After making the above modification, we can include time in each refined specification of the implementation. To include time in specification \( P \), we may simply use a conjunction \( P \land T \), where \( T \) is a timing expression (a boolean expression that involves time). For example, we may include \( T \equiv t' = t + f(\sigma) \) to specify the exact execution time (if the computation is deterministic in the execution time), or \( T \equiv t' \leq t + f(\sigma) \) to specify a time bound, where \( f \) is a nonnegative function of the prestate. All refinements must hold with the inclusion of time.

**Example A.4.** Consider Ex. A.3, where we refine \( x' \leq 0 \) by recursion. With the inclusion of time, the refinement becomes

\[
x' \leq 0 \land T \iff \text{if } x \leq 0 \text{ then ok else } (x := x - 1. t := t + 1. x' \leq 0 \land T)
\]

which is still a theorem when time is not mentioned (i.e. \( T = \top \)), but it is also a theorem when \( T \equiv t' \leq t + x^2 \), which specifies that \( x^2 \) is a time bound (though it is not a tight one). If we change the specification to \( x' \leq 0 \land t' = t \), the refinement no longer holds. If we want the recursive time to be zero, recursion cannot be used; we may of course use the refinement \( x' \leq 0 \land t' = t \iff x := 0 \) instead.

\[\Diamond\]

### A.6.1 Timing expressions

In order to prove refinements with time, suitable timing expressions must be provided for all refined specifications. The timing expression for the initial specification may (or may not) be given by the specifier, but for each other specification arising from program development (refinement by steps), it must be established by the implementer.

Given a refinement without time, there are many ways to find a suitable timing expression \( T \). We may keep making reasonable guesses for \( T \) until we can prove the refinement. The guesses may be based on perception or knowledge pertaining to the program; if no such perception or knowledge is available, we may even try executing the program on various inputs to see what we get, though it may be difficult to analyze the observed data.

Alternatively, one may treat the refinement as a fixed-point equation and try to find a fixed point with recursive program construction ([5, Chap. 6]), but there is no guarantee that a fixed point can be found (even when it exists); moreover, since part of the refinement is already solved, some of the work is redundant. With the help of
refinement laws, we can approach this in a simpler way. If we are interested in a time bound, we can write $T = t' \leq t + f \sigma$, and apply refinement by parts for $T$; then we find a solution of $f$ that makes the refinement a theorem.

**Example A.5.** For Ex. A.4, we want to find the exact execution time of the program (which is possible since the computation is deterministic). We write $T = t' = t + f x$. Applying refinement by parts for $T$ followed by refinement by cases, we obtain

$$
t' = t + f x \iff x \leq 0 \land t' = t
\quad t' = t + f x \iff x > 0 \land t' = t + 1 + f(x - 1)
$$

These refinements hold if (and only if)

$$
f x = 0 \iff x \leq 0
\quad f x = 1 + f(x - 1) \iff x > 0
$$

which is good enough if we accept a recursive definition for the execution time; otherwise, we may try to find an expression for $f$ in terms of $x$, and get

$$f x = \max 0 x$$

The whole refinement, with the inclusion of time, is therefore

$$x' \leq 0 \land t' = t + \max 0 x \iff \begin{cases} 
\text{if } x \leq 0 \text{ then ok} \\
\text{else (}x := x - 1. \ t := t + 1. \ x' \leq 0 \land t' = t + \max 0 x\text{)}
\end{cases}$$

This approach can be especially useful when coupled with reasonable guesses, for example, we may guess $f$ to be a quadratic function with unknown constants in the general cases, and our job reduces to finding the constants and establishing $f$ for the special cases. If we are not interested in the values of these constants, we do not even have to find them; we only have to show that such constants exist (for example, by writing a system of equations that provably has a solution).

There is no general method for finding a suitable time bound that holds for a refinement; one may want to employ methods from other disciplines of mathematics and computer science. Very often, it is much harder to find a suitable timing expression than to prove a given one.

**A.6.2 Termination**

If a specification $P$ does not mention the time variable, we have the following trivial refinement:

$$P \iff t := t + 1. \ P$$

When the program is executed, the computation is an infinite loop and never terminates. However, it can be argued, albeit informally, that the computation does satisfy $P$: since
it runs forever, the only observation (other than the prestate) that can be made is that it is running (in other words, time is increasing), and since \( P \) does not mention time, \( P \) is satisfied by this observation. The fault here is in the specification \( P \) (for not mentioning time), not the computation. A computation can only be complained about if and only if an observation of it does not satisfy \( P \); since it runs forever, there is no observation on which a complaint can be based.

To guarantee termination, we want to include a timing expression that implies it. We can express termination as \( t < \infty \Rightarrow t' < \infty; \)\(^1\) however, if we use this as the timing expression (specifying a "finite but unbounded" time), it does not fare any better, since the trivial refinement

\[
P \wedge (t < \infty \Rightarrow t' < \infty) \iff t := t + 1. \ P \wedge (t < \infty \Rightarrow t' < \infty)
\]

still holds, which again reflects (informally) the fact that there is never a time when we can complain about the computation running forever. If we want to avoid non-termination, a (finite) time bound must be implied by the specification, as in \( P \wedge t' \leq t + f \sigma \), where \( f \) is a nonnegative finite function of the prestate. If \( f \) is not provided by the specifier, the implementer must establish one if termination must be ensured. (If we do not require termination to be ensured for every prestate, it is only necessary that \( f \) be finite over those prestates that termination is required.) We see that when a time bound is provided,

\[
P \wedge t' \leq t + f \sigma \iff t := t + 1. \ P \wedge t' \leq t + f \sigma
\]

is no longer a theorem for any finite \( f \), which reflects the fact that since the computation does not terminate, we have grounds to complain as soon as it runs longer than \( f \sigma \).

### A.7 Expanding the programming language

In addition to the basic programming language presented in Def. A.7, many other notations that resemble the features found in some popular languages are introduced in Hehner's book [5, Chap. 5]; also, independent (parallel) composition [5, Chap. 8] and communication [5, Chap. 9] are discussed in detail. Here we present the definitions of some of them in the standard model, those which we redefine in the thesis for our probabilistic model.

---

\(^1\)The antecedent \( t < \infty \) is included for the termination expression because we only require termination for a computation that starts at finite time; if a computation starts at time \( \infty \) (in other words, it never starts), termination is meaningless.
A.7.1 Variable declaration

\[ \text{var } x : D_x \cdot S = \exists x, x' : D_x \cdot S \]

This notation declares a local variable \( x \) which can be used in specification \( S \). Specification \( S \) is an expression of the global variables plus the variable \( x \) in the prestate and the poststate. Specification \( \text{var } x : D_x \cdot S \) is an expression of the global variables only.

We can declare several variables in one declaration using a list variable; each item in the list is conveniently referred to by a name. For example,

\[
\begin{align*}
\text{var } [x; y] : [\text{int}; \text{nat}] \cdot S &= \exists [x; y], [x'; y'] : [\text{int}; \text{nat}] \cdot S \\
&= \exists x, x' : \text{int} \cdot \exists y, y' : \text{nat} \cdot S \\
&= \text{var } x : \text{int} \cdot \text{var } y : \text{nat} \cdot S
\end{align*}
\]

declares two local variables \( x \) (an integer) and \( y \) (a natural) in \( S \). If the domains of the variables are identical, we can omit their order and hence the list; for example,

\[
\begin{align*}
\text{var } x, y, z : \text{int} \cdot S &= \text{var } [x; y; z] : [3 * \text{int}] \cdot S
\end{align*}
\]

declares three local integer variables \( x, y \) and \( z \) in \( S \).

A.7.2 While loop

The while-loop has the syntax

\[
\text{while } b \text{ do } S
\]

where \( b \) is a boolean expression of the prestate, and \( S \) is a program. The computation iteratively executes \( S \) until \( b \) evaluates to 1 (possibly before \( S \) is first executed). In Hehner's book, two accounts are provided. In the first account [5, p. 74], we do not equate the while-loop to a specification; instead, we consider

\[
W \Leftarrow \text{while } b \text{ do } S
\]

to be an abbreviation of

\[
W \Leftarrow \text{if } b \text{ then } (S \cdot W) \text{ else ok}
\]

In the second account [5, p. 100], the specification while \( b \) do \( S \) is defined by the following while-loop construction and induction axioms:

\[
\begin{align*}
\text{while } b \text{ do } S &\Rightarrow \text{if } b \text{ then } (S \cdot \text{while } b \text{ do } S) \text{ else ok} \\
\forall \sigma, \sigma' \cdot (W \Rightarrow \text{if } b \text{ then } (S \cdot W) \text{ else ok}) &\Rightarrow \forall \sigma, \sigma' \cdot (W \Rightarrow \text{while } b \text{ do } S)
\end{align*}
\]
These two axioms are equivalent to the following fixed-point construction and induction axioms:

\begin{align*}
\text{while } b \text{ do } S & = \text{if } b \text{ then } (S. \text{ while } b \text{ do } S) \text{ else ok} \\
\forall \sigma, \sigma'. (W = \text{if } b \text{ then } (S. W) \text{ else ok}) & \Rightarrow \forall \sigma, \sigma'. (W \Rightarrow \text{while } b \text{ do } S)
\end{align*}

In other words, while \( b \) do \( S \) is the weakest solution of the fixed-point equation (in unknown \( W \))

\[ W = \text{if } b \text{ then } (S. W) \text{ else ok} \]

The two accounts are not equivalent.

**A.7.3 For loop**

The for-loop has the syntax

\[
\text{for } i := m;..n \text{ do } S
\]

where \( i \) is a fresh name, \( m \) and \( n \) are integer expressions such that \( m \leq n \), and \( S \) is a program; \( i \) is a local constant in \( S \) (\( i' \) is not a variable of \( S \)). The computation starts with \( i = m \), and iteratively executes \( S \); it increases \( i \) by 1 after each iteration, and the loop continues up to but excluding \( i = n \). Hehner's book provides the following account [5, p. 79]. As in the first account of the while-loop, we do not equate the for-loop to a specification, but instead show how it is used in refinement. Let \( A : \text{int} \rightarrow \text{bool} \), so that \( Ai \) is an expression of the prestate only, and \( A'i \) is the corresponding expression of the poststate only. Then

\[
Am \Rightarrow A'n \iff \text{for } i := m;..n \text{ do } i : m;..n \land Ai \Rightarrow A'(i+1)
\]

To refine \( Am \Rightarrow A'n \), we can use a for-loop in refinement; then we need only to refine \( i : m;..n \land Ai \Rightarrow A'(i+1) \).

This account of the for-loop has a deficiency in that specifications of the type \( Am \Rightarrow A'n \) are often inadequate to describe relations between the prestate and the poststate, such as \( x' = x \times 2^n \); to refine this specification by a for-loop, one has to introduce a new variable for \( 2^n \). A better account of the for-loop is given in Section 4.4.3.
Appendix B

Axioms and laws

This appendix presents a list of axioms and laws for the notations introduced or redefined in this thesis. Axioms and laws for the notations adopted from Hehner's book are not included; a complete list of them can be found in the book [5, p. 229–243]. For reference, each axiom and law is given a label (some laws share the same label for resemblance).

When we state a law as implication ("\(\Rightarrow\)" or "\(\Leftarrow\)"), we also mean that its converse is not a theorem; in other words, there exists values for its variables such that the consequent is satisfied, but the antecedent is not. (If its converse is indeed a theorem, we would have stated the law as equality.) So for this appendix only, one may interpret the law \(A \Rightarrow B\) (or \(B \Leftarrow A\)) as

\[(A \Rightarrow B) \land (\exists V \cdot \neg(A \Leftarrow B))\]

where \(V\) are the variables of \(A\) and \(B\).

The proofs for a selected number of laws are presented in Appendix C.

B.1 Probability theory

B.1.1 Probability distributions and events

In the following, \(\mathcal{P}\) and \(\mathcal{Q}\) are probability distributions over sample space \(D\); \(B\) is a bunch; \(\langle a \rangle\) and \(\langle b \rangle\) are event expressions of variable \(\varphi\); \(p\) is a probability-valued expression.

\begin{align*}
\text{PR-0} & \quad (Probability\ axioms) \\
\mathcal{P}B & \geq 0 \\
\mathcal{P}D & = 1 \\
\mathcal{P}B & = \Sigma \varphi : B \cdot \mathcal{P}\varphi \\
\mathcal{P}B & = \mathcal{P}(B \upharpoonright D) \\
\text{EV-0} & \quad \mathcal{P}\langle b \rangle = \mathcal{P}(\varphi : D \cdot b)
\end{align*}
APPENDIX B. AXIOMS AND LAWS

PR-1 \[ 0 \leq \mathcal{P}(b) \leq 1 \]
PR-2 \[ \mathcal{P}(b) = \Sigma \varphi : (\xi \varphi : D \cdot b) \cdot \mathcal{P}\varphi \\
= \Sigma \varphi : D \cdot \text{if } b \text{ then } \mathcal{P}\varphi \text{ else } 0 \]
PR-3 \[ \mathcal{P}(T) = 1 \]
\[ \mathcal{P}(\perp) = 0 \]
PR-4 \[ \mathcal{P}(a) + \mathcal{P}(b) = \mathcal{P}(a \lor b) + \mathcal{P}(a \land b) \]
\[ \mathcal{P}(b) + \mathcal{P}(\neg b) = 1 \]
\[ \mathcal{P}(a \land b) + \mathcal{P}(a \land \neg b) = \mathcal{P}(a) \]
PR-5 \[ \forall \varphi : D \cdot a \Rightarrow b \Rightarrow \mathcal{P}(a) \leq \mathcal{P}(b) \]
PR-6 \[ \mathcal{P}(a \land b) = 1 \Rightarrow \mathcal{P}(a) = 1 \land \mathcal{P}(b) = 1 \]
\[ \mathcal{P}(a \lor b) = 0 \Rightarrow \mathcal{P}(a) = 0 \land \mathcal{P}(b) = 0 \]

B.1.2 Expected values and probability density functions

In the following, \( \mathcal{E} \) is the expectation function for probability distribution \( \mathcal{P} \) over sample space \( D \); \( \mathcal{Q} \) is a probability distribution over \( D \); \( \mathcal{X} \) and \( \mathcal{Y} \) are functions with domain \( D \), and \( \mathcal{Z} \) is a function with domain \( \mathcal{X}D \); \( c \) and \( d \) are real numbers; \( \langle b \rangle \) is an event expression of variable \( \varphi \).

EX-0 \[ \mathcal{E} \mathcal{X} = \varepsilon \mathcal{P} \mathcal{X} = \Sigma \varphi : D \cdot \mathcal{X}\varphi \times \mathcal{P}\varphi \]
EX-1 \[ \mathcal{E}(1) = 1 \]
EX-2 \[ \mathcal{E}(c \times \mathcal{X} + d \times \mathcal{Y}) = c \times \mathcal{E}\mathcal{X} + d \times \mathcal{E}\mathcal{Y} \]
EX-3 \[ \mathcal{E}(\eta b) = \mathcal{P}(b) \]
EX-4 \[ \mathcal{E} \mathcal{X} = \Sigma x : \mathcal{X}D \cdot x \times \mathcal{P}(\mathcal{X}\varphi = x) = \Sigma x : \mathcal{X}D \cdot x \times (\mathcal{X} \circ \mathcal{P})x \]
EX-5 \[ \varepsilon(\mathcal{P} p \oplus \mathcal{Q})\mathcal{X} = p \times \varepsilon \mathcal{P} \mathcal{X} + (1 - p) \times \varepsilon \mathcal{Q} \mathcal{X} \]
PF-0 \[ \mathcal{X}D : \Delta(\mathcal{X} \circ \mathcal{P}) \]
\[ (\mathcal{X} \circ \mathcal{P})x = \mathcal{P}(\mathcal{X}\varphi = x) \]
PF-1 \[ \mathcal{Z} \circ (\mathcal{X} \circ \mathcal{P}) = (\mathcal{Z} \mathcal{X}) \circ \mathcal{P} \]
PF-2 \[ \mathcal{X} \circ (\mathcal{P} p \oplus \mathcal{Q}) = (\mathcal{X} \circ \mathcal{P})p \oplus (\mathcal{X} \circ \mathcal{Q}) \]

B.2 Specification notations

B.2.1 Probabilistic composition (constant alternatives)

Two alternatives

In the following, \( Q, R, S, \) and \( T \) are probabilistic specifications; \( U \) and \( V \) are standard specifications; \( b \) is a boolean expression of the prestate; \( p, q, u, \) and \( v \) are probability-valued expressions of the prestate; \( c \) and \( d \) are real expressions of the prestate; \( \mathcal{X} \) is a
function with domain \( D_\sigma \); \( R_x \) and \( S_x \) are probability distributions over the range of \( X \).

**PC-0** \( R_p \gamma S = \exists R, S \cdot p > 0 \Rightarrow (\text{substitute } R \text{ for } P' \text{ in } R) \) 
\( \quad \wedge (p < 1 \Rightarrow (\text{substitute } S \text{ for } P' \text{ in } S)) \)  
\( \quad \wedge P' = R_p \bigoplus S \)  
\( R_p \bigoplus S = p \times R + (1 - p) \times S \)

**PC-1** \( R_p \gamma S = S_{1-p} \gamma R \)

**PC-2**  
\( ((Q \land R) \gamma S = Q \land R \gamma S) \Leftarrow u = p_q \land 1 - q = (1 - u)(1 - v) \)

**PC-3** \( R \gamma S = R \)  
\( R \delta \gamma S = S \)

**PC-4** \( R_p \gamma R \Leftarrow R \)  
\( \circ R_p \gamma \circ R = \circ R \)

**PC-5** \( R_p \gamma \perp = p = 0 \land R \)

**PC-6** \( (Q \lor R) \gamma S = (Q \gamma S) \lor (R \gamma S) \)

**PC-7** \( (\text{if } b \text{ then } Q \text{ else } R) \gamma S = \text{if } b \text{ then } Q \gamma S \text{ else } R \gamma S \)

**PC-8** \( (Q \land R) \gamma S \Rightarrow (Q \gamma S) \land (R \gamma S) \)

**PC-9** \( (Q \land R) \gamma S \Rightarrow (Q \land R) \gamma T \)  
\( \Rightarrow (Q \land R) \gamma S \Rightarrow (Q \land R) \gamma T \)

**PC-10** \( X \circ P' = R_x \gamma P \)  
\( X \circ P' = S_x \)  
\( \Rightarrow X \circ P' = R_x \bigoplus S_x \)

**PC-11** \( P'(U) \geq u \gamma P'(V) \geq v \Rightarrow P'(U) \geq pu \land P'(V) \geq (1 - p)v \)  
\( \land P'(U \lor V) \geq pu + (1 - p)v \)

**PC-12** \( \mathcal{E}' \mathcal{X} = c_p \gamma \mathcal{E}' \mathcal{X} = d \Rightarrow \mathcal{E}' \mathcal{X} = pc + (1 - p)d \)  
\( \mathcal{E}' \mathcal{X} \leq c_p \gamma \mathcal{E}' \mathcal{X} \leq d \Rightarrow \mathcal{E}' \mathcal{X} \leq pc + (1 - p)d \)

**Three or more alternatives**

In the following, for \( 0 \leq i \leq n \), \( R_i \) is a probabilistic specification; for \( 0 \leq i \leq n - 1 \), \( r_i \) is a probability-valued expression of the prestate, so that \( r_0 + \ldots + r_{n-1} \leq 1 \).

**P3-0** \( R_0 r_0 \gamma \ldots r_{n-1} \gamma R_n = \)  
\( \exists R_0, \ldots, R_n \cdot r_0 > 0 \Rightarrow (\text{substitute } R_0 \text{ for } P' \text{ in } R_0) \)  
\( \wedge \ldots \)  
\( \wedge r_{n-1} > 0 \Rightarrow (\text{substitute } R_{n-1} \text{ for } P' \text{ in } R_{n-1}) \)  
\( \wedge r_0 + \ldots + r_{n-1} < 1 \Rightarrow (\text{substitute } R_n \text{ for } P' \text{ in } R_n) \)  
\( \wedge P' = R_0 r_0 \bigoplus \ldots r_{n-1} \bigoplus R_n \)  
\( R_0 r_0 \ldots r_{n-1} \bigoplus R_n = r_0 \times R_0 + \ldots + r_{n-1} \times R_{n-1} + (1 - r_0 - \ldots - r_{n-1}) \times R_n \)

**P3-1** \( R_0 r_0 \gamma R_1 r_1 \gamma R_2 r_2 \gamma \ldots r_{n-1} \gamma R_n = \)  
\( (R_0 r_0 / (r_0 + r_1) \gamma R_1) r_0 + r_1 \gamma R_2 r_2 \gamma \ldots r_{n-1} \gamma R_n \)
B.2.2 Convex composition

In the following, $Q$, $R$, and $S$ are probabilistic specifications; $U$ and $V$ are standard specifications; $b$ is a boolean expression of the prestate; $p$ is a probability-valued expression of the prestate.

\[
\begin{align*}
\text{NC-0} & \quad R \uplus S = \exists p:0,1 \cdot R \triangleleft p \cdot S \\
\text{NC-1} & \quad R \uplus S = S \uplus R \\
\text{NC-2} & \quad (Q \uplus R) \uplus S = Q \uplus (R \uplus S) \\
\text{NC-3} & \quad R \uplus \bot = R \\
\text{NC-4} & \quad R \uplus T \\
\text{NC-5} & \quad R \uplus Q \iff R \uplus Q \\
\text{NC-6} & \quad R \uplus R \iff R \\
& \quad \circ R \uplus \circ R = \circ R \\
\text{NC-7} & \quad (Q \uplus R) \uplus S = (Q \uplus S) \uplus (R \uplus S) \\
\text{NC-8} & \quad (\text{if } b \text{ then } Q \text{ else } R) \uplus S = \text{if } b \text{ then } Q \uplus S \text{ else } R \uplus S \\
\text{NC-9} & \quad (Q \wedge R) \uplus S \Rightarrow (Q \uplus S) \wedge (R \uplus S) \\
\text{NC-10} & \quad (Q \wedge R) \circ p \uplus S \Rightarrow (Q \circ p \uplus S) \gamma (R \circ p \uplus S) \\
& \quad (Q \uplus R) \circ p \circ S = (Q \circ p \circ S) \gamma (R \circ p \circ S) \\
\text{NC-11} & \quad (Q \circ p \uplus R) \uplus S \Rightarrow (Q \uplus S) \circ p \gamma (R \uplus S) \\
\text{NC-12} & \quad U \uplus V = \langle U \uplus V \rangle_a
\end{align*}
\]

B.2.3 Probabilistic composition (variable alternatives)

In the following, $R$ and $S$ are probabilistic specifications with an additional choice variable $\varphi$ (except for PV-7), and $P$ and $Q$ are choice distributions; $\mathcal{F}$ is an expression of the prestate whose value is a function from $\Delta Q$ to poststate distributions; $\mathcal{X}$ is an expression of the prestate whose value is a function from $\Delta Q$ to poststates; $p$ is a probability-valued expression of the prestate (excluding $\varphi$); $b$ is a boolean expression of the prestate (excluding $\varphi$).

\[
\begin{align*}
\text{PV-0} & \quad Q \triangleright \lambda \varphi \cdot S = \exists F: \Delta Q \rightarrow D_\sigma \rightarrow 0,1 \cdot \\
& \quad \quad (\forall \varphi: \Delta Q \cdot Q\varphi > 0 \Rightarrow \text{(substitute } F\varphi \text{ for } P' \text{ in } S)) \\
& \quad \wedge P' = \Sigma \varphi: \Delta Q \cdot Q\varphi \times F\varphi \\
\text{PV-1} & \quad P \circ \oplus Q \triangleright \lambda \varphi \cdot S \Rightarrow (P \triangleright \lambda \varphi \cdot S) \circ p \gamma (Q \triangleright \lambda \varphi \cdot S) \\
& \quad P \circ \oplus Q \triangleright \lambda \varphi \cdot \circ S = P \triangleright \lambda \varphi \cdot \circ S \circ p \gamma Q \triangleright \lambda \varphi \cdot \circ S \\
\text{PV-2} & \quad P \triangleright \lambda \varphi \cdot R \uplus S \iff (P \triangleright \lambda \varphi \cdot R) \uplus (P \triangleright \lambda \varphi \cdot S) \\
\text{PV-3} & \quad P \triangleright \lambda \varphi \cdot \text{if } b \text{ then } R \text{ else } S = \text{if } b \text{ then } P \triangleright \lambda \varphi \cdot R \text{ else } P \triangleright \lambda \varphi \cdot S
\end{align*}
\]
APPENDIX B. AXIOMS AND LAWS

B.2.4 Dependent composition

In the following, $Q$, $R$, and $S$ are probabilistic specifications; $T$ is a probabilistic specification with an additional choice variable $\varphi$, and $Q$ is a choice distribution; $U$ and $V$ are standard specifications; $b$ is a boolean expression of the prestate; $p$, $u$, and $v$ are probability-valued expressions of the prestate; $e$ is an expression of the prestate in the domain of state variable $x$.

**DC-0** $R. S = \exists \mathcal{P}'' \cdot (\text{substitute } \mathcal{P}'' \text{ for } \mathcal{P}' \text{ in } R)$

$\land \mathcal{P}'' \triangleright \lambda \sigma'' \cdot (\text{substitute } \sigma'' \text{ for } \sigma \text{ in } S)$

$= \exists \mathcal{P}'' \cdot (\text{substitute } \mathcal{P}'' \text{ for } \mathcal{P}' \text{ in } \langle R \rangle_a)$

$\land \exists \mathcal{F} : D_\sigma \rightarrow D_\sigma \rightarrow 0 \cdot 1 \cdot$

$(\forall \sigma'' ; \mathcal{P}'' \sigma'' > 0 \Rightarrow (\text{substitute } \sigma'' ; \mathcal{F} \sigma'' \text{ for } \sigma ; \mathcal{P}' \text{ in } \langle S \rangle_a))$

$\land \mathcal{P}' = \Sigma \sigma'' \cdot \mathcal{P}'' \sigma'' \times \mathcal{F} \sigma''$

**DC-1** ok. $R = R$. ok $= R$

**DC-2** $(Q . R). S \Rightarrow Q . (R . S)$

$(\circ Q . \circ R). \circ S = \circ Q . (\circ R . \circ S)$

**DC-3** $x := e . S = (\text{substitute } e \text{ for } x \text{ in } S)$

**DC-4** $Q \lor R . S = (Q . S) \lor (R . S)$

$S . Q \lor R \leftarrow (S . Q) \lor (S . R)$

**DC-5** (if $b$ then $Q$ else $R$). $S = \text{if } b \text{ then } (Q . S) \text{ else } (R . S)$

**DC-6** $Q \land R . S \Rightarrow (Q . S) \land (R . S)$

$S . Q \land R \Rightarrow (S . Q) \land (S . R)$

**DC-7** $Q \models R . S \Rightarrow (Q . S) \models (R . S)$

$Q \models R . \circ S = (Q . \circ S) \models (R . \circ S)$

$S . Q \models R \Rightarrow (S . Q) \models (S . R)$

**DC-8** $(Q \triangleright \lambda \varphi . T). S \Rightarrow Q \triangleright \lambda \varphi . (T . S)$

$(Q \triangleright \lambda \varphi . T) . \circ S = Q \triangleright \lambda \varphi . (T . \circ S)$

$S . (Q \triangleright \lambda \varphi . T) \Rightarrow Q \triangleright \lambda \varphi . (S . T)$
DC-9 \[ Q \land R. \, S \Rightarrow (Q. \, S) \land (R. \, S) \]
\[ Q \land R. \circ S = (Q. \circ S) \land (R. \circ S) \]
\[ \circ S. \, Q \land R \iff (\circ S. \, Q) \land (\circ S. \, R) \]

DC-10 \[ U \cdot V = (U \cdot V)_a \]
DC-11 \[ P'(U) \geq u. \, P'(V) \geq v = P'(U \cdot V) \geq uv \]

B.2.5 Variable declaration

In the following, \( x \) (and for \( VD-1, \, y \)) is a local variable in probabilistic specification \( S \); \( \varsigma \) is a local list variable in standard specification \( U \).

VD-0 \[ \text{var } x: D_x \cdot S = \exists x: D_x \cdot \exists S: D_\sigma \cdot [D_x] \rightarrow 0,..,1 \cdot \]
\[ P' = (\lambda \sigma' + [x']. \cdot \sigma') \circ S \land (\text{substitute } S \text{ for } P' \text{ in } S) \]
\[ = \exists x: D_x \cdot \exists S: D_\sigma \cdot [D_x] \rightarrow 0,..,1 \cdot \]
\[ (\forall \sigma'_0: D_\sigma \cdot P' \sigma'_0 = S(\sigma' = \sigma'_0)) \land (\text{substitute } S \text{ for } P' \text{ in } S) \]

VD-1 \[ \text{var } [x; y]: [D_x; D_y] \cdot S = \text{var } x: D_x \cdot \text{var } y: D_y \cdot S \]
\[ = \text{var } y: D_y \cdot \text{var } x: D_x \cdot S \]

VD-2 \[ \text{var } \varsigma: D_\varsigma \cdot U = \langle \exists \varsigma, \varsigma': D_\varsigma \cdot U \rangle_a \]

B.2.6 Random number generators

In the following, \( x \) is a state variable; \( n \) is a positive integer expression of the prestate.

RN-0 \[ x := \text{rand } n = (0,..n \rightarrow 1/n) \triangleright \lambda i \cdot x := i \]

B.3 Refinement

In the following, all the capital letters are probabilistic specifications; \( b \) is a boolean expression of the prestate; \( p \) is a probability-valued expression of the prestate.

Refinement by steps

If \( A \iff \text{if } b \text{ then } C \text{ else } D \) and \( A \iff E \) and \( D \iff F \) are theorems,
then \( A \iff \text{if } b \text{ then } E \text{ else } F \) is a theorem.

If \( A \iff B \land C \) and \( B \iff D \) and \( C \iff E \) are theorems,
then \( A \iff D \land E \) is a theorem.

If \( A \iff B \lor C \) and \( B \iff D \) and \( C \iff E \) are theorems,
then \( A \iff D \lor E \) is a theorem.
If $A \iff B$, $C$ and $B \iff D$ and $C \iff E$ are theorems,
then $A \iff D$, $E$ is a theorem.

If $A \iff B$ and $B \iff C$ are theorems, then $A \iff C$ is a theorem.

Refinement by parts

If $A \iff \text{if } b \text{ then } C \text{ else } D$ and $E \iff \text{if } b \text{ then } F \text{ else } G$ are theorems,
then $A \land E \iff \text{if } b \text{ then } C \land F \text{ else } D \land G$ is a theorem.

If $A \iff B 
\text{p} \n C$ and $D \iff E 
\text{p} \n F$ are theorems,
then $A \land D \iff B \land E \ n C \land F$ is a theorem.

If $A \iff B \n C$ and $D \iff E \n F$ are theorems,
then $A \land D \iff B \land E \ n C \land F$ is a theorem.

If $A \iff B$ and $C \iff D$ are theorems, then $A \land C \iff B \land D$ is a theorem.

Refinement by cases

$P \iff \text{if } b \text{ then } Q \text{ else } R$ is a theorem if and only if
$P \iff b \land Q$ and $P \iff \neg b \land Q$ are theorems.

If $\circ P \iff Q$ and $\circ P \iff R$ are theorems,
then $\circ P \iff Q \n R$ is a theorem.

$\circ P \iff Q \n R$ is a theorem if and only if
$\circ P \iff Q$ and $\circ P \iff R$ are theorems.

B.4 Miscellaneous

In the following, $r$, $x$, and $y$ are reals; $n$ and $k$ are naturals.

$\eta = T \to 1 \mid \bot \to 0$

$r : x .. y \equiv x \leq r \leq y$

$r : x .. y \equiv x < r \leq y$

$r : x .. y \equiv x < r < y$

$n! = \Pi i : 1 .. , n \cdot i$

$\binom{n}{k} = n! / (k! \times (n-k)!)$
Appendix C

Proofs of selected laws

This appendix presents the proofs of some of the laws in Appendix B. Only a selected number of proofs are presented; the proofs of many of the other laws are either straightforward, or can be done in ways similar to those presented here.

C.1 Proof of PC-10

Here we present the proof of law PC-10, a very useful law for proving theorems regarding probabilistic composition.

PC-10 \( \mathcal{X} \circ \mathcal{P}' = \mathcal{R}_x \ p \mathcal{Y} \ \mathcal{X} \circ \mathcal{P}' = \mathcal{S}_x \Rightarrow \mathcal{X} \circ \mathcal{P}' = \mathcal{R}_x \ p \oplus \mathcal{S}_x \)

Proof: We prove equality by double implication. The cases for \( p = 0 \) and \( p = 1 \) are trivial, so we assume \( 0 < p < 1 \). First, for the \( \Rightarrow \) part:

\[
\mathcal{X} \circ \mathcal{P}' = \mathcal{R}_x \ p \mathcal{Y} \ \mathcal{X} \circ \mathcal{P}' = \mathcal{S}_x \\
= \exists \mathcal{R}, \mathcal{S} \cdot \mathcal{X} \circ \mathcal{R} = \mathcal{R}_x \land \mathcal{X} \circ \mathcal{S} = \mathcal{S}_x \land \mathcal{P}' = \mathcal{R}_p \oplus \mathcal{S} \\
\Rightarrow \exists \mathcal{R}, \mathcal{S} \cdot \mathcal{X} \circ \mathcal{R} = \mathcal{R}_x \land \mathcal{X} \circ \mathcal{S} = \mathcal{S}_x \\
\land \mathcal{X} \circ \mathcal{P}' = \mathcal{X} \circ (\mathcal{R}_p \oplus \mathcal{S}) = (\mathcal{X} \circ \mathcal{R})_p \oplus (\mathcal{X} \circ \mathcal{S}) \\
\Rightarrow \mathcal{X} \circ \mathcal{P}' = \mathcal{R}_x \ p \oplus \mathcal{S}_x
\]

For the \( \Leftarrow \) part, we assume the right side is satisfied, that is,

\( \mathcal{X} \circ \mathcal{P}' = \mathcal{R}_x \ p \oplus \mathcal{S}_x \)

and prove the existential quantification in the definition of probabilistic composition for the left side by constructing suitable probability distributions for \( \mathcal{R} \) and \( \mathcal{S} \). Let

\[
\mathcal{R} = \lambda \sigma' \cdot \left( \frac{\mathcal{R}_x(\mathcal{X} \sigma') \times \mathcal{P}' \sigma'}{(\mathcal{R}_x \ p \oplus \mathcal{S}_x)(\mathcal{X} \sigma')} \right) \\
\mathcal{S} = \lambda \sigma' \cdot \left( \frac{\mathcal{S}_x(\mathcal{X} \sigma') \times \mathcal{P}' \sigma'}{(\mathcal{R}_x \ p \oplus \mathcal{S}_x)(\mathcal{X} \sigma')} \right)
\]

First, we see obviously that

\( \mathcal{R}_x \ p \oplus \mathcal{S} = \mathcal{P}' \)
Then, for $R$

\[ X \circ R = \lambda x \cdot R(X \sigma' = x) \]
\[ = \lambda x \cdot (\Sigma \sigma' :(X \sigma' = x)) \cdot R \sigma' \]
\[ = \lambda x \cdot (\Sigma \sigma' :(X \sigma' = x)) \cdot \left( \frac{R_x x \times P' \sigma'}{(R_x p \oplus S_x)x} \right) \]
\[ = \lambda x \cdot \left( \frac{R_x x \times P(X \sigma' = x)}{(X \circ P')x} \right) \]
\[ = R_x \]

And similarly, for $S$

\[ X \circ S = S_x \]

In the proof for $R$, we have also verified that $R$ satisfy the probability axioms, since obviously $R \sigma' \geq 0$, and

\[ \Sigma R = \Sigma x \cdot R(X \sigma' = x) = \Sigma x \cdot R_x x = 1 \]

since $R_x$ is a given probability distribution over the range of $X$; same for $S$.

The implication of the left side by the right side follows immediately from the generalization law and the definition of probabilistic composition, and this completes the proof.

\[ \text{C.2 Proof of PC-11} \]

Law PC-10 cannot be applied when the two alternatives in a probabilistic composition assigns probabilities to different partitions of the poststates, but with the use of existential variables for unknown probabilities, we can divide the poststates into further partitions, so that the two alternatives assign probabilities to the same partitions. Here we show how existential variables can be used to derive law PC-11 from law PC-10. Many laws that we have not presented (for example, when \( \geq \) in either alternative is replaced with \( = \)) can be derived in a similar way.

**PC-11** \( P'(U) \geq u \) \( P'(V) \geq v \) \( \implies \) \( P'(U) \geq pu \land P'(V) \geq (1 - p)v \) \land \( P'(U \lor V) \geq pu + (1 - p)v \)

**Proof:** The cases for \( p = 0 \) and \( p = 1 \) are trivial, so we assume \( 0 < p < 1 \). We divide the poststates into four disjoint partitions: \( U \land V \), \( U \land -V \), \( -U \land V \) and \( -U \land -V \). For each alternative, we introduce three existential variables for the probabilities of the first three partitions, which entail the probability of the fourth partition: for the left alternative, the four partitions have probabilities \( q_0, r_0, s_0 \) and \( 1 - q_0 - r_0 - s_0 \) respectively, where \( q_0 + r_0 \geq u \) (since \( P'(U) = P'(U \land V) + P'(U \land -V) \)); for the right alternative,
the probabilities are \( q_1, r_1, s_1 \) and \( 1 - q_1 - r_1 - s_1 \) respectively, where \( q_1 + s_1 \geq u \) (since \( \mathcal{P}'(V) = \mathcal{P}'(U \land V) + \mathcal{P}'(\neg U \land V) \)). We can then apply law \textbf{PC-10}.

\[
\mathcal{P}'(U) \geq u \quad \mathcal{P}'(V) \geq v
\]

\[
\implies \exists q_0, r_0, s_0, q_1, r_1, s_1 : 0, \ldots, 1 .
\]

\[
q_0 + r_0 + s_0 \leq 1 \land q_1 + r_1 + s_1 \leq 1 \land q_0 + r_0 \geq u \land q_1 + s_1 \geq v
\]

\[
\land \left( \mathcal{P}'(U \land V) = q_0 \land \mathcal{P}'(U \land \neg V) = r_0 \land \mathcal{P}'(\neg U \land V) = s_0 \right)
\]

\[
\land \left( \mathcal{P}'(U \land V) = q_1 \land \mathcal{P}'(U \land \neg V) = r_1 \land \mathcal{P}'(\neg U \land V) = s_1 \right)
\]

\[
\implies \exists q_0, r_0, s_0, q_1, r_1, s_1 : 0, \ldots, 1 .
\]

\[
q_0 + r_0 + s_0 \leq 1 \land q_1 + r_1 + s_1 \leq 1 \land q_0 + r_0 \geq u \land q_1 + s_1 \geq v
\]

\[
\land \mathcal{P}'(U \land V) = pq_0 + (1 - p)q_1
\]

\[
\land \mathcal{P}'(U \land \neg V) = pr_0 + (1 - p)r_1
\]

\[
\land \mathcal{P}'(\neg U \land V) = ps_0 + (1 - p)s_1
\]

Using law \textbf{PC-10}, we have eliminated the existential variables for probability distributions in the left side, but they are replaced by existential variables for probabilities, and these have to be eliminated too. We prove by double implication that the last expression is equal to the right side of law \textbf{PC-11}. The " \( \Rightarrow \) " part is easy; we work within the existential quantification:

\[
\mathcal{P}'(U) = \mathcal{P}'(U \land V) + \mathcal{P}'(U \land \neg V) \geq p(q_0 + r_0) \geq pu
\]

\[
\mathcal{P}'(V) = \mathcal{P}'(U \land V) + \mathcal{P}'(\neg U \land V) \geq (1 - p)(q_1 + s_1) \geq (1 - p)v
\]

\[
\mathcal{P}'(U \lor V) = \mathcal{P}'(U \land V) + \mathcal{P}'(U \land \neg V) + \mathcal{P}'(\neg U \land V)
\]

\[
\geq p(q_0 + r_0) + (1 - p)(q_1 + s_1) \geq pu + (1 - p)v
\]

The " \( \Leftarrow \) " part is the tricky part. We assume the right side is satisfied, that is,

\[
\mathcal{P}'(U) \geq pu
\]

\[
\mathcal{P}'(V) \geq (1 - p)v
\]

\[
\mathcal{P}'(U \lor V) \geq pu + (1 - p)v
\]

and find suitable expressions in \( 0, \ldots, 1 \) for all the existential variables in the left side, so that all conjuncts are satisfied. The conjuncts are

(a) \( q_0 + r_0 + s_0 \leq 1 \)

(b) \( q_1 + r_1 + s_1 \leq 1 \)

(c) \( q_0 + r_0 \geq u \)

(d) \( q_1 + s_1 \geq v \)

(e) \( \mathcal{P}'(U \land V) = pq_0 + (1 - p)q_1 \)

(f) \( \mathcal{P}'(U \land \neg V) = pr_0 + (1 - p)r_1 \)

(g) \( \mathcal{P}'(\neg U \land V) = ps_0 + (1 - p)s_1 \)
APPENDIX C. PROOFS OF SELECTED LAWS

For \( r_0, r_1, s_0 \) and \( s_1 \), let

\[
\begin{align*}
    r_0 &= \min 1 \left( \frac{\mathcal{P}'(U \land \neg V)}{p} \right) \\
    r_1 &= \max 0 \left( \frac{\mathcal{P}'(U \land \neg V) - p}{1 - p} \right) \\
    s_0 &= \max 0 \left( \frac{\mathcal{P}'(\neg U \land V) - (1 - p)}{p} \right) \\
    s_1 &= \min 1 \left( \frac{\mathcal{P}'(\neg U \land V)}{1 - p} \right)
\end{align*}
\]

We see obviously that all these expressions are in \( 0, 1 \). Before we write the expressions for \( q_0 \) and \( q_1 \), we see first that (f) and (g) are satisfied by these expressions. For (f), we look into the cases for \( \mathcal{P}'(U \land \neg V) \geq p \) and \( \mathcal{P}'(U \land \neg V) < p \):

if \( \mathcal{P}'(U \land \neg V) \geq p \)
then \(( p r_0 + (1 - p) r_1 = p + (\mathcal{P}'(U \land \neg V) - p) = \mathcal{P}'(U \land \neg V) ) \)
else \(( p r_0 + (1 - p) r_1 = \mathcal{P}'(U \land \neg V) + 0 = \mathcal{P}'(U \land \neg V) ) \)

The proof for (g) is similar to (f) by symmetry. For the expressions \( q_0 \) and \( q_1 \), we consider three distinct cases; rather than writing long expressions combining all the cases, for the sake of clarity we write the expressions separately for each case, and prove (a)-(e) accordingly. The cases are

(A) \( \mathcal{P}'(U \land \neg V) \geq pu \)
(B) \( \mathcal{P}'(U \land \neg V) < pu \land \mathcal{P}'(\neg U \land V) \geq (1 - p)v \)
(C) \( \mathcal{P}'(U \land \neg V) < pu \land \mathcal{P}'(\neg U \land V) < (1 - p)v \)

For case (A), let

\[
\begin{align*}
    q_0 &= \max 0 \left( \frac{\mathcal{P}'(U \land V) - (1 - p)(1 - s_1)}{p} \right) \\
    q_1 &= \min (1 - s_1) \left( \frac{\mathcal{P}'(U \land V)}{1 - p} \right)
\end{align*}
\]

We see obviously that \( q_0 \) and \( q_1 \) are in \( 0, 1 \). For (a), we look into the cases for \( \mathcal{P}'(\neg U \land V) \geq 1 - p \) and \( \mathcal{P}'(\neg U \land V) < 1 - p \):

if \( \mathcal{P}'(\neg U \land V) \geq 1 - p \)
then \(( q_0 + r_0 + s_0 \leq \frac{\mathcal{P}'(U \land V)}{p} + \frac{\mathcal{P}'(U \land \neg V)}{p} + \frac{\mathcal{P}'(\neg U \land V) - (1 - p)}{p} \leq 1 ) \)
else \(( q_0 + r_0 + s_0 \leq \max r_0 \left( \frac{\mathcal{P}'(U \land V) - (1 - p) + (1 - p)s_1 + r_0}{p} \right) \)
\leq \max 1 ( \frac{\mathcal{P}'(U \land V) - (1 - p) + \mathcal{P}'(\neg U \land V) + \mathcal{P}'(U \land \neg V)}{p} ) \)
\leq \max 1 1 = 1 \)

For (b), we look into the cases for \( \mathcal{P}'(U \land \neg V) \geq p \) and \( \mathcal{P}'(U \land \neg V) < p \):

if \( \mathcal{P}'(U \land \neg V) \geq p \)
then \(( q_1 + r_1 + s + 1 \leq \frac{\mathcal{P}'(U \land V)}{1 - p} + \frac{\mathcal{P}'(U \land \neg V) - p}{1 - p} + \frac{\mathcal{P}'(\neg U \land V)}{1 - p} \leq 1 ) \)
else \(( q_1 + r_1 + s + 1 \leq (1 - s_1) + s_1 = 1 ) \)
(c) follows from the assumption of case (A):
\[ q_0 + r_0 \geq 0 + \min 1 \left( \frac{\mathcal{P}'(U \land \lnot V)}{p} \right) \geq \min 1 \left( \frac{pu}{u} \right) = u \]

(d) follows from the first assumption of the right side:
\[
\begin{align*}
\text{if } \mathcal{P}'(-U \land V) &\geq 1 - p \\
\text{then } (q_1 + s_1 &= q_1 + 1 \geq v ) \\
\text{else } (q_1 + s_1 &= \min 1 \left( \frac{\mathcal{P}'(U \land V)}{1-p} + \frac{\mathcal{P}'(-U \land V)}{1-p} \right) \\
&= \min 1 \left( \frac{\mathcal{P}'(U)}{1-p} \right) \\
&\geq \min 1 \left( \frac{(1-p)u}{1-p} \right) = v )
\end{align*}
\]

For (e), we look into the cases for \( \mathcal{P}'(U \land V) \geq (1-p)(1-s_1) \) and \( \mathcal{P}'(U \land V) < (1-p)(1-s_1) \):
\[
\begin{align*}
\text{if } \mathcal{P}'(U \land V) &\geq (1-p)(1-s_1) \\
\text{then } (pq_0 + (1-p)q_1 &= (\mathcal{P}'(U \land V) - (1-p)(1-s_1)) + (1-p)(1-s_1) = \mathcal{P}'(U \land V) ) \\
\text{else } (pq_0 + (1-p)q_1 &= 0 + \mathcal{P}'(U \land V) = \mathcal{P}'(U \land V) )
\end{align*}
\]

This completes the proof for case (A).
Case (B) can be dealt with similarly, using expressions for \( q_0 \) and \( q_1 \) that are symmetric to the expressions for \( q_1 \) and \( q_0 \) respectively in case (A). Let
\[
\begin{align*}
q_0 &= \min 1 \left( 1 - r_0 \right) \left( \frac{\mathcal{P}'(U \land V)}{p} \right) \\
q_1 &= \max 0 \left( \frac{\mathcal{P}'(U \land V) - p(1-r_0)}{p} \right)
\end{align*}
\]

By symmetry, the proofs of (a), (b), (c), (d) and (e) in case (B) are similar to (b), (a), (d), (c) and (e) respectively in case (A): (c) follows from the second assumption of the right side, and (d) follows from the assumption of case (B).

For case (C), let
\[
\begin{align*}
q_0 &= \min 1 \left( k_0 + \max 0 \left( \frac{1-p(k_1-1)}{p} \right) - r_0 \\
q_1 &= \min 1 \left( k_1 + \max 0 \left( \frac{p}{1-p}(k_0-1) \right) - s_1 \\
\end{align*}
\]

where
\[
\begin{align*}
k_0 &= \left( \frac{u}{pu + (1-p)v} \right) \mathcal{P}'(U \lor V) \\
k_1 &= \left( \frac{v}{pu + (1-p)v} \right) \mathcal{P}'(U \lor V)
\end{align*}
\]

We have to prove that \( q_0 \) and \( q_1 \) are in \( 0, 1 \) and satisfy (a)-(e). From the assumption of case (C), we first see that
\[
\begin{align*}
r_0 &= \frac{\mathcal{P}'(U \land \lnot V)}{p} \leq u \\
r_1 &= 0 \\
s_0 &= 0 \\
s_1 &= \frac{\mathcal{P}'(-U \land V)}{1-p} \leq v
\end{align*}
\]
To prove that $q_0$ and $q_1$ are in $0..1$, since we will also prove (a) and (b), it suffices to prove $q_0 \geq 0$ and $q_1 \geq 0$, which follows from (c) and (d) since $r_0 \leq u$ and $s_1 \leq v$. From the third assumption of the right side, we have $k_0 \geq u$ and $k_1 \geq v$. So for (c) and $q_0 \geq 0$,

\[
q_0 + r_0 \geq (\min 1 k_0 - r_0) + r_0 \geq \min 1 u = u
\]

And similarly, (d) and $q_1 \geq 0$ by symmetry. For (a) and (b), we see that it cannot be the case that both $k_0 > 1$ and $k_1 > 1$, since

\[
k_0 > 1 \land k_1 > 1 \Rightarrow pk_0 + (1-p)k_1 > 1 = P(U \lor V) > 1 = \bot
\]

So we look into the cases for $k_1 > 1$ (which implies $k_0 \leq 0$) and $k_1 \leq 0$:

if $k_1 > 1$

then \[
q_0 + r_0 + s_0 = (k_0 + \frac{1-p}{p}(k_1 - 1) - r_0) + r_0 + 0
\]

\[
= pk_0 + (1-p)k_1 - (1-p)
\]

\[
= \frac{P(U \lor V) - (1-p)}{p}
\]

\[
\leq 1
\]

else \[
q_0 + r_0 + s_0 \leq (1+0 - r_0) + r_0 + 0 = 1
\]

(b) can be proved similarly by symmetry. And finally, for (e), we see that

\[
p(\min 1 k_0) + (1-p)\left(\max 0 \left(\frac{p}{1-p}(k_0 - 1)\right)\right) = pk_0
\]

and that

\[
p\left(\max 0 \left(\frac{1-p}{p}(k_1 - 1)\right)\right) + (1-p)(\min 1 k_1) = (1-p)k_1
\]

So

\[
pq_0 + (1-p)q_1
\]

\[
= pk_0 + (1-p)k_1 - pr_0 - (1-p)s_1
\]

\[
= P(U \lor V) - P(U \land \neg V) - P(\neg U \land V)
\]

\[
= P(U \land V)
\]

This completes the proof for case (C), and consequently the proof of the "$\iff$" part, and consequently the proof of law PC-11.

\[\square\]

C.3 Proof of PV-7

Here we present the proof of law PV-7, which illustrates how the existential quantification of $\mathcal{F}$ in the definition of the notation $Q \triangleright \lambda \varphi \cdot S$ serves as (nested) existential
quantifications of two probability distributions; the same reasoning applies for any number of probability distributions as well.

**PV-7** \( R_p \supset S = 0 \rightarrow p | 1 \rightarrow 1 - p \supset 0 \rightarrow R | 1 \rightarrow S \)

*Proof:* The proof is straightforward.

\[
0 \rightarrow p | 1 \rightarrow 1 - p \supset 0 \rightarrow R | 1 \rightarrow S = \exists \mathcal{F}: 0, 1 \rightarrow D_\sigma \rightarrow 0, 1, 1 \cdot
\]

\[
(\forall i: 0, 1 \cdot (0 \rightarrow p | 1 \rightarrow 1 - p)i > 0 \Rightarrow (\text{substitute } \mathcal{F}i \text{ for } \mathcal{P}' \text{ in } (0 \rightarrow R | 1 \rightarrow S)i))
\]

\[
\land \mathcal{P}' = \sum_{i=0}^{1} 1 \cdot Q_i \times \mathcal{F}i
\]

Expand universal quantification and summation over domain 0, 1:

\[
= \exists \mathcal{F}: 0, 1 \rightarrow D_\sigma \rightarrow 0, 1, 1 \cdot (p > 0 \Rightarrow (\text{substitute } \mathcal{F}0 \text{ for } \mathcal{P}' \text{ in } R))
\]

\[
\land (1 - p > 0 \Rightarrow (\text{substitute } \mathcal{F}1 \text{ for } \mathcal{P}' \text{ in } S))
\]

\[
\land \mathcal{P}' = p \times \mathcal{F}0 + (1 - p) \times \mathcal{F}1
\]

Use the one-point law (twice) to create existential quantifications of \( R \) and \( S \):

\[
= \exists \mathcal{F}: 0, 1 \rightarrow D_\sigma \rightarrow 0, 1 \cdot \exists \mathcal{R}, \mathcal{S} \cdot
\]

\[
\mathcal{F}0 = \mathcal{R} \land \mathcal{F}1 = \mathcal{S}
\]

\[
\land (p > 0 \Rightarrow (\text{substitute } \mathcal{R} \text{ for } \mathcal{P}' \text{ in } R))
\]

\[
\land (p < 1 \Rightarrow (\text{substitute } \mathcal{S} \text{ for } \mathcal{P}' \text{ in } S))
\]

\[
\land \mathcal{P}' = p \times \mathcal{R} + (1 - p)\mathcal{S}
\]

Since \((0 \rightarrow \mathcal{R} | 1 \rightarrow \mathcal{S})0 = \mathcal{R} \) and \((0 \rightarrow \mathcal{R} | 1 \rightarrow \mathcal{S})1 = \mathcal{S} \), \( \mathcal{F} \) can be eliminated by the generalization law:

\[
= \exists \mathcal{R}, \mathcal{S} \cdot (p > 0 \Rightarrow (\text{substitute } \mathcal{R} \text{ for } \mathcal{P}' \text{ in } R))
\]

\[
\land (p < 1 \Rightarrow (\text{substitute } \mathcal{S} \text{ for } \mathcal{P}' \text{ in } S))
\]

\[
\land \mathcal{P}' = p \times \mathcal{R} + (1 - p)\mathcal{S}
\]

\[
= R_p \supset S
\]

This completes the proof. \( \square \)

### C.4 Proof of PV-5

Here we present the proof of law **PV-5**, a general form of law **PC-9**. The proof makes use of the Skolem law to rearrange the existential and universal quantifiers in the definitions of the notations \( Q \supset \lambda \varphi \cdot S \) and \( R_p \supset S \). This technique can be used for proving many other laws regarding probabilistic composition of variable alternatives and dependent composition.

**PV-5** \( \mathcal{P} \supset \lambda \varphi \cdot R_p \supset S = (\mathcal{P} \supset \lambda \varphi \cdot R) \supset (\mathcal{P} \supset \lambda \varphi \cdot S) \)
Proof: The cases for $p = 0$ and $p = 1$ are trivial, so we assume $0 < p < 1$.

\[ P \triangleright \lambda \varphi \cdot R \forall S \]
\[ = \exists \mathcal{F}: \Delta P \rightarrow D_\sigma \rightarrow 0,.,1 \cdot \]
\[ (\forall \varphi: \Delta P \cdot P \varphi > 0 \Rightarrow \exists \mathcal{R}, S \cdot (\text{substitute } \mathcal{R} \text{ for } P' \text{ in } R)) \]
\[ \land (\text{substitute } \mathcal{S} \text{ for } P' \text{ in } S) \]
\[ \land \mathcal{F} \varphi = \mathcal{R}_p \oplus S \]
\[ \land P' = \Sigma \varphi: \Delta P \cdot P \varphi \times \mathcal{F} \varphi \]

Use the distributive and Skolem laws for existential variables $\mathcal{R}$ and $\mathcal{S}$:

\[ = \exists \mathcal{F}_R, \mathcal{F}_S: \Delta P \rightarrow D_\sigma \rightarrow 0,.,1 \cdot \]
\[ (\forall \varphi: \Delta P \cdot P \varphi > 0 \Rightarrow (\text{substitute } \mathcal{F}_R \varphi \text{ for } P' \text{ in } R)) \]
\[ \land (\text{substitute } \mathcal{F}_S \varphi \text{ for } P' \text{ in } S)) \]
\[ \land P' = \Sigma \varphi: \Delta P \cdot P \varphi \times (\mathcal{F}_R \varphi \oplus \mathcal{F}_S \varphi) \]
\[ = (\Sigma \varphi: \Delta P \cdot P \varphi \times \mathcal{F}_R \varphi) \oplus (\Sigma \varphi: \Delta P \cdot P \varphi \times \mathcal{F}_S \varphi) \]

Use the one-point law (twice) to create existential quantifications of $\mathcal{R}$ and $\mathcal{S}$, then distribute $\mathcal{F}_R$ and $\mathcal{F}_S$ inwards:

\[ = \exists \mathcal{R}, \mathcal{S} \cdot (\exists \mathcal{F}_R: \Delta P \rightarrow D_\sigma \rightarrow 0,.,1 \cdot \]
\[ (\forall \varphi: \Delta P \cdot P \varphi > 0 \Rightarrow (\text{substitute } \mathcal{F}_R \varphi \text{ for } P' \text{ in } R)) \]
\[ \land \mathcal{R} = \Sigma \varphi: \Delta P \cdot P \varphi \times \mathcal{F}_R \varphi \]
\[ \land (\exists \mathcal{F}_S: \Delta P \rightarrow D_\sigma \rightarrow 0,.,1 \cdot \]
\[ (\forall \varphi: \Delta P \cdot P \varphi > 0 \Rightarrow (\text{substitute } \mathcal{F}_S \varphi \text{ for } P' \text{ in } S)) \]
\[ \land \mathcal{S} = \Sigma \varphi: \Delta P \cdot P \varphi \times \mathcal{F}_S \varphi \]
\[ \land P' = \mathcal{R}_p \oplus \mathcal{S} \]
\[ = (P \triangleright \lambda \varphi \cdot R)_p \forall (P \triangleright \lambda \varphi \cdot S) \]

This completes the proof.
## Appendix D

### Notations and precedence

This appendix contains a glossary of all the notations used in this thesis. A table of precedence for all the notations is also included.

### D.1 Notations and names

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Notation</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\top$</td>
<td>true</td>
<td>$-$ minus</td>
</tr>
<tr>
<td>$\bot$</td>
<td>false</td>
<td>$\times$ times</td>
</tr>
<tr>
<td>$\neg$</td>
<td>not</td>
<td>$/$ divided by</td>
</tr>
<tr>
<td>$\land$</td>
<td>and</td>
<td>$,$ bunch union</td>
</tr>
<tr>
<td>$\lor$</td>
<td>or</td>
<td>$\ldots$ union from (incl) to (excl)</td>
</tr>
<tr>
<td>$\lhd$</td>
<td>convex (nondeterministic) composition</td>
<td>$\ldots$ union from (incl) to (incl)</td>
</tr>
<tr>
<td>$p\lhd$</td>
<td>$p$-probabilistic composition</td>
<td>$:'$ bunch intersection</td>
</tr>
<tr>
<td>$\Rightarrow$</td>
<td>implies</td>
<td>$;$ string catenation</td>
</tr>
<tr>
<td>$\iff$</td>
<td>implies</td>
<td>$\ldots$ catenation from (incl) to (excl)</td>
</tr>
<tr>
<td>$\Leftarrow$</td>
<td>follows from, is implied by</td>
<td>$:$ is in, bunch inclusion</td>
</tr>
<tr>
<td>$\iff$</td>
<td>follows from, is implied by</td>
<td>$:= $ assignment</td>
</tr>
<tr>
<td>$=$</td>
<td>equals, if and only if</td>
<td>$.$ dependent composition</td>
</tr>
<tr>
<td>$\equiv$</td>
<td>equals, if and only if</td>
<td>$\ldots$ union from (excl) to (excl)</td>
</tr>
<tr>
<td>$\neq$</td>
<td>differs from, exclusive or</td>
<td>$\ldots$ union from (excl) to (incl)</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>less than</td>
<td>$.$ variable introduction</td>
</tr>
<tr>
<td>$&gt;$</td>
<td>greater than</td>
<td>$!$ factorial</td>
</tr>
<tr>
<td>$\leq$</td>
<td>less than or equal to</td>
<td>$(\cdot)$ parentheses for grouping</td>
</tr>
<tr>
<td>$\geq$</td>
<td>greater than or equal to</td>
<td>$\binom{n}{k}$ binomial coefficient</td>
</tr>
<tr>
<td>$+$</td>
<td>plus</td>
<td>$[$ list brackets</td>
</tr>
<tr>
<td>$+$</td>
<td>list catenation</td>
<td>${}$ event expression</td>
</tr>
<tr>
<td>$p\oplus$</td>
<td>$p$-probabilistic sum</td>
<td>$\langle \rangle$ probability of event is 1</td>
</tr>
</tbody>
</table>
$\emptyset$ bunch size, cardinality
$\#$ list size, length
* repetition of a string
$\Delta$ domain of a function
$|$ selective union, otherwise
$\rightarrow$ function arrow
$\eta$ maps $T$ to $1$, $\bot$ to $0$
$\varepsilon$ $\varepsilon P\mathcal{X}$ is the expected value of $\mathcal{X}$ over $P$
$\circ$ $\mathcal{X} \circ P$ is the probability density function of $\mathcal{X}$ with respect to $P$
$\circ$ convex closure

for do for-loop
if then else conditional composition
ok the “empty” program
or nondeterministic composition

abs the absolute value function
bool the booleans
ceil the ceiling function
even the even function
lim for function $f = \lambda v:D \cdot b$, $(\lim f)x$ is the limit of $b$ as $v$ approaches $x$
log binary logarithm
int the integers

$\triangleright$ probabilistic composition of variable alternatives
$\lambda$ function, variable introduction
$\forall$ for all, universal quantifier
$\exists$ there exists, existential quantifier
$\Sigma$ sum of, summation quantifier
$\Pi$ product of, product quantifier
$\xi$ solutions of, solution quantifier
$\circ a b$ exponentiation

$p$-or $p$-probabilistic composition
rand random number generator
var local variable declaration

while do while-loop

max maximum (a binary function)
min minimum (a binary function)
nat the naturals
nil the empty string
null the empty bunch
odd the odd function
real the reals
$\times$nat the extended naturals
D.2 Precedence

0. $T \perp \text{ok } ( ) [ ] \langle \rangle$ numbers names
1. juxtaposition
2. exponentiation $! \# * \Delta \circ \circ$
3. $\times /$
4. $+ - + p \oplus$
5. ` ; ..`
6. `, .. `. .. ..`
7. $\rightarrow$
8. $|$`$
9. $= \neq < > \leq \geq :$
10. $\neg$
11. $\wedge$
12. $\lor p \gamma \gamma$
13. $\Rightarrow \Leftarrow$
14. $\Rightarrow \Rightarrow \Rightarrow \Rightarrow$ if then else for do while do or p-or
15. $\lambda \forall \exists \Sigma \Pi. \xi \lim. \varvar$
16. $\Rightarrow \Leftarrow$

Juxtaposition associates from left to right, so that $a b c$ means $(a b) c$. The infix operators $/ -$ associate from left to right. The infix operators $* \rightarrow$ associate from right to left. The infix operators $\times + + ; ' , | \wedge \lor \gamma$ are associative (they associate in both directions). The infix operator $.$ associates from left to right; it is associative when its operands are convex. The infix operators $p \gamma p \oplus$ do not associate; $(P p \gamma Q) \gamma R$, $P p \gamma (Q q \gamma R)$, and $P p \gamma Q q \gamma R$ are all different (see Section 3.4.2). On levels 9, 13, and 16 the operators are continuing; for example, $a = b = c$ neither associates to the left nor associates to the right, but means $a = b \land b = c$. On any one of these levels, a mixture of continuing operators can be used. For example, $a \leq b < c$ means $a \leq b \land b < c$. 
Bibliography


