UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SLICING OF PROBABILISTIC PROGRAMS BASED ON SPECIFICATIONS

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS DE LA INGENIERÍA, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MARCELO ANDRÉS NAVARRO PACHECO

PROFESOR GUÍA:
Federico Olmedo Berón

MIEMBROS DE LA COMISIÓN:
Johan Fabry
Matías Toro Ipinza
Pedro D'Argenio

Este trabajo ha sido parcialmente financiado por:
FONDECYT No. 11181208.

SANTIAGO DE CHILE
2024

RESUMEN DE LA TESIS PARA OPTAR AL GRADO DE:
MAGÍSTER EN CIENCIAS DE LA INGENIERÍA, MENCIÓN COMPUTACIÓN
Y RESUMEN DE LA MEMORIA PARA OPTAR AL TÍTULO DE:
INGENIERO CIVIL EN COMPUTACIÓN
POR: MARCELO ANDRÉS NAVARRO PACHECO
FECHA: 2024
PROF. GUÍA: FEDERICO OLMEDO BERÓN

REBANADO DE PROGRAMAS PROBABILÍSTICOS BASADO EN ESPECIFICACIONES

El objetivo del *slicing* de programas es, dado un programa y un criterio de *slicing*, por ejemplo, un conjunto de variables, identificar y eliminar los fragmentos del programa que no afecten su comportamiento, con respecto a dicho criterio de *slicing*. El resultado se conoce como un *slice* del programa original. Las aplicaciones principales del *slicing* de programas incluyen testing, comprensión de programas, depuración de programas y extracción de componentes reutilizables.

Se han propuesto varios enfoques desde la introducción del *slicing* de programas, centrados principalmente en el análisis de dependencias de datos y control en diferentes tipos de programas. Sin embargo, estas técnicas solo brindan un soporte parcial para la aleatorización.

En programación probabilística, avanzar en técnicas de *slicing* ofrece varias ventajas. Además de las aplicaciones mencionadas anteriormente, conduce a un mejor rendimiento en el proceso de inferencia —determinar una representación explícita de la distribución de probabilidad inducida implícitamente por un programa—. También ayudaría a la comprensión de estos programas, que son significativamente más complejos que los programas deterministas y que a menudo requiere un conocimiento profundo de la teoría de la probabilidad.

Debido a las razones anteriores, en este estudio hemos desarrollado la primera técnica de slicing para programas probabilísticos basada en especificaciones. Mostramos que cuando los programas probabilísticos están acompañados de sus especificaciones en la forma de pre- y post-condición, podemos aprovechar esta información semántica para producir *slices*, que preservan las especificaciones, estrictamente más precisos que los *slices* producidos por técnicas convencionales basadas en dependencia de datos y control.

La técnica desarrollada es sensible a las propiedades de terminación de los programas, permitiendo preservar tanto la corrección parcial como total de los programas respecto a sus especificaciones. Es modular, con un principio de razonamiento local, y demostramos formalmente su correctitud.

Como ingredientes técnicos fundamentales de nuestra técnica, diseñamos generadores de condiciones de verificación para establecer la corrección parcial y total de los programas probabilísticos y demostramos su correctitud, resultado que es de interés por sí mismos y pueden ser explotados en otros estudios independientes.

En el aspecto práctico, demostramos la aplicabilidad de nuestro enfoque mediante algunos ejemplos ilustrativos y un estudio de caso del campo de la modelación probabilística. También describimos un algoritmo para computar los *slices* más pequeños dentro del conjunto de *slices* derivados por nuestra técnica.

Abstract

The goal of *program slicing* is, given a program and a slicing criterion, for example, a set of variables, to identify and safely remove program fragments that do not affect the program behavior, with respect to the slicing criterion. The outcome is known as a *slice* of the original program. Primary applications of program slicing include testing, program comprehension, program debugging, and the extraction of reusable components.

Since the introduction of program slicing, several approaches have been proposed. These techniques primarily focus on data and control dependencies and have been applied to diverse program types. However, these techniques only offer partial support for randomization.

In the domain of probabilistic programming, making progress in more advanced slicing techniques offers several advantages. In addition to the benefits mentioned earlier, it leads to improved performance in inference calculations —to determine an explicit representation of the probability distribution implicitly represented by a program—. It would also aid in the comprehension of these programs, which are significantly more complex than deterministic programs, often requiring in-depth knowledge of probability theory.

Motivated by these reasons, in this study, we have developed the first slicing technique for probabilistic programs based on specifications. We show that when probabilistic programs are accompanied by their specifications in the form of pre- and post-condition, we can exploit this semantic information to produce specification-preserving slices strictly more precise than slices yielded by conventional techniques based on data/control dependency.

The developed technique is termination-sensitive, allowing to preserve the partial as well as the total correctness of probabilistic programs with respect to their specifications. It is modular, featuring a local reasoning principle, and is formally proved correct.

As fundamental technical ingredients of our technique, we design and prove sound verification condition generators for establishing the partial and total correctness of probabilistic programs, which are of interest on their own and can be exploited elsewhere for other purposes.

On the practical side, we demonstrate the applicability of our approach by means of a few illustrative examples and a case study from the probabilistic modelling field. We also describe an algorithm for computing least slices among the space of slices derived by our technique.

A la memoria de José Camacho.

Agradecimientos

Batallar contra una enfermedad de salud mental es un desafío complejo, pero tuve la fortuna de contar con personas extraordinarias durante este proceso. En primer lugar, agradezco a Ivette, mi psicóloga, por su sabiduría y empatía durante estos últimos dos años, que fueron fundamentales para poder culminar este trabajo.

También quiero expresar mi gratitud a Federico, cuyo apoyo y justa presión hicieron que este camino fuera mucho más llevadero. Su constante muestra de cariño y comprensión quedará grabada en mi corazón para siempre.

Mis compañeros del tenis de mesa, en especial mi profesor, merecen todo mi agradecimiento. Sus valiosos recuerdos y enseñanzas me dieron fuerzas en los momentos más difíciles y me inspiraron a retomar este hermoso deporte, que tanto significa para mí.

A mis amigos de la universidad, especialmente a los del pasillo, que fueron un pilar fundamental en este recorrido, les agradezco profundamente. Su cálida compañía y apoyo incondicional hicieron más llevaderos los grandes desafíos dentro de la facultad.

También agradezco a los cabros, mis amigos de la vida, quienes me acompañaron durante noches completas de trabajo, escritura o programación. Siempre había un momento de escape y relajo junto a ustedes, incluso en los momentos de mayor estrés.

A mi familia, que, a pesar de la distancia y las dificultades, me ha brindado todas las condiciones necesarias para aprovechar las oportunidades que la vida me ha dado, les agradezco de corazón.

Por supuesto, a Francisca, mi compañera de vida, le agradezco profundamente. Gracias por sostenerme en los momentos más frágiles, por darme fuerzas cuando más las necesité y por caminar a mi lado en cada paso de este recorrido. Espero poder acompañarte muchos años más. Extiendo también este agradecimiento a su familia, que me ha hecho sentir parte de ella. La vida es mucho más sencilla y bella junto a todos ustedes.

Mi más sincero agradecimiento a cada uno de ustedes, por ser parte de este viaje y por iluminar mi camino.

Table of Content

List of Figures

Chapter 1

Introduction

Since its introduction by Weiser [1], *program slicing* has been recognized for its wide range of applications in the process of software development. The basic idea behind program slicing is, given a program and a set of variables of interest (the slicing criterion), to identify the program fragments that can be safely removed without affecting the program behavior, with respect to the said set of variables; the "subset" program so obtained is known as a *slice* of the original program. Among others, primary applications of slicing include testing, program understanding, program debugging and extraction of reusable components [2].

Different approaches have been proposed to compute program slices [3]. However, two shared—and sometimes conflicting— requirements of these approaches are efficiency and precision. On the one hand, one is interested in computing slices as fast as possible, and on the other hand, in computing the least slices. Besides efficiency and precision, another fundamental aspect of slice approaches lies in the subset of *language features* that they support. Current approaches can be applied, for instance, to programs with procedural abstractions, unstructured control flow, composite data types and pointers, and concurrency primitives [3]. However, a fundamental language feature that is only partially supported is *randomization*.

At the programming language level, randomization is typically supported by some form of *probabilistic choice*. For instance, a probabilistic program can flip a (fair or biased) coin and depending on the observed outcome, continues its execution in one way or another.

Probabilistic programs have proved useful in a wealth of different domains. They are central in the field of machine learning due to their compelling properties for representing probabilistic models [4, 5]. They are the cornerstone of modern cryptography—modern public-key encryption schemes[1] are by nature probabilistic [6]. They lie at the heart of quantum computing—quantum programs are inherently probabilistic due to the random outcomes of quantum measurements [7]. Finally, they are the key ingredient for implementing randomized algorithms [8].

In the past years, the field of probabilistic programming has seen a resurgence, in partic-

[1] By "public-key encryption schemes" we here mean public-key encryption schemes understood as a whole, comprising all the public key generation, encryption and decryption phases.

ular, due to the emergence of new probabilistic modeling applications [9]. A wealth of new probabilistic programming systems has been developed, which conveniently allow representing probabilistic models as programs, and querying them, e.g., to determine the probability of a given event or the expected value of a given random variable. To enable this, probabilistic programming systems implement some form of *inference*, building an explicit representation of the probability distribution implicitly encoded by a program.

Notoriously, three distinguished features of probabilistic programs make the problem of slicing even more crucial for this class of programs. First, despite usually consisting in a few lines of code, probabilistic programs may present a complex and intricate behavior, which is hard to grasp for an average programmer, even when knowledgeable in probability theory. For example, the termination analysis of probabilistic programs is full of subtleties [10]. Second, the process of inference is known to be computationally highly expensive [11], which turns the problem of computing slices as small as possible even more critical for this class of programs. Third, the development of probabilistic programming systems is a daunting task, and several bugs have been recently discovered in many of them [12]. Any tool aiding program understanding is thus vital.

Hur et al. gave a first step toward supporting slicing for probabilistic programs, extended with conditioning [13]. In a later work, Amtoft and Banerjee introduced probabilistic control-flow graphs, which allows a direct adaptation of standard machinery from the slicing literature to the case of probabilistic programs [14, 15]. Both works adopt the classical slicing criteria where slicing is performed with respect to a set of program variables of interest, typically the variables influencing the program output. Said otherwise, these works aims at identifying those program fragments that do not have a true influence on the value of the program output variables (or other set of variables, at other execution point).

There exist, however, more precise slicing techniques based on program assertions instead of program variables. The idea here is to identify those program fragments that contribute to establishing a given program assertion, instead of fixing the variables' value [16, 17]. The main benefit of this approach is that it produces smaller slices, provided there exists a (functional) specification of the program in the form of a pre- and post-condition. As advocated by the *design-by-contract* methodology to software development [18], pre- and post-conditions specify program behavior by constraining the set of final states (post-condition) that are reachable from a given set of initial states (pre-condition). However, slicing techniques based on specifications have so far been restricted to deterministic programs, and it is an open problem whether they can be applied to probabilistic programs as well.

The main contribution of this work is to give a positive answer to the above problem. Concretely, given a probabilistic program together with its specification, we show how it can be sliced in order to *preserve the specification*. To illustrate this, consider the program below, accompanied by its specification

$$\{\mathit{if}\ y^2 \leq 0.5\ \mathit{then}\ \tfrac{1}{2}\ \mathit{else}\ 0\} \quad x \coloneqq 1.5 - y^2;\ \{x \coloneqq x-1\}\ [1/2]\ \{x \coloneqq x-2\} \quad \{x \geq 0\}^{\downarrow}$$

The program starts by assigning the value of $1.5 - y^2$ to variable x, then flips a fair coin and depending on the observed outcome, it decrements x by either 1 or 2. The specification says that, upon termination, the program establishes post-condition $x \geq 0$ with probability at least $1/2$ provided that initially $y^2 \leq 0.5$, and with probability at least 0, otherwise. In the

same way that for deterministic programs pre-conditions provide only *sufficient conditions* for establishing post-conditions, for probabilistic programs pre-conditions provide only *lower bounds for the probability* of establishing post-conditions.

To slice this program, we can apply existing techniques for probabilistic programs, by selecting x as the output variable of interest whose value we want to preserve (since x is the only program variable mentioned in the post-condition). However, the conventional dataflow analysis carried out by these techniques will say that the only valid slice of the program is the very same program.

On the contrary, our slicing technique implements a more precise analysis that captures quantitative relations between program variables, concluding that the proper subprogram

$$x \coloneqq 1.5 - y^2$$

is a valid slice that preserves the original program specification.[2] In effect, it is the *least* slice preserving the specification.

Besides yielding more precise slices, specification-based slicing opens the windows to further applications [19]. A prominent example is software reuse by *specification specialization* (weakening). Suppose a probabilistic program is known to establish a given post-condition e.g., with a minimal probability $1/2$. Now, if we are to use the program in a new context where it suffices to establish the post-condition with probability only $1/4$, we can slice the original program w.r.t. this weakened specification to yield to a potentially simpler and more efficient program which can safely be used in this context.

At the technical level, our slicing technique works by propagating post-conditions backward using (a variant of) the greatest pre-expectation transformer [20]—the probabilistic counterpart of Dijkstra's weakest pre-condition transformer [21]. This endows programs with an axiomatic semantics, expressed in terms of a verification condition generator (VCGen) that yields quantitative proof obligations.

In particular, we design (and prove sound) VCGens for both the partial (allowing divergence) and the total (requiring termination) correctness of probabilistic programs, making our slicing technique *termination-sensitive*. To handle iteration, we assume that program loops are annotated with invariants. To reason about (probabilistic) termination, we assume that loop annotations also include (probabilistic) variants.

Another appealing property of our slicing technique is its *modularity*: It yields valid slices of a program from valid slices of its subprograms. Most importantly, this involves only *local reasoning*. This is crucial for keeping the slice computation tractable.

In this regard, besides developing the theoretical foundations of our slicing technique, we also exhibit an algorithm for computing program slices. Interestingly, the algorithm computes the *least* slice that can be derived from our slicing technique, according to a proper notion of slice size, using, as main ingredient, a shortest-path algorithm.

Finally, we illustrate the application of our technique through some examples, showing

[2] Formally, we also rely on the assumption that y is a real-valued variable, and therefore it always holds that $y^2 \geq 0$.

that it yields strictly more precise slices than existing techniques.

Contributions of the thesis.

To summarize, the main contributions of this work are as follows:

- We develop the first slicing technique based on specifications for probabilistic (imperative) programs (Chapters 4 and 5). The slicing technique is termination-sensitive, modular featuring local reasoning principles, and formally proved correct.
- We demonstrate the applicability of the technique by means of illustrative examples. These comprise a set of small —yet instructive— programs while developing the theory (Sections 4.2 and 5.4) as well as a case study from the probabilistic modelling field (Chapter 6). All these examples confirm that our technique yields strictly more precise slices than other existing techniques, provided programs are accompanied by their specifications and annotated with loop invariants.
- We show that an adaption of the slicing algorithm introduced by Barros et al. [17] can be used for computing minimal slices of probabilistic programs, among the space of slices derived by our technique (Chapter 7).
- As a fundamental ingredient of our technique, we design and prove sound VCGens for establishing both the partial and the total correctness of probabilistic programs (Chapter 3 and Section 5.3). These VCGens are of self-interest and can be exploited elsewhere for other purposes, such as program verification.

Organization of the work.

The remainder of the work is organized as follows. Chapter 2 introduces the probabilistic imperative language used for describing programs and lays out their specification model. Chapter 3 presents the VCGen for characterizing the partial correctness of programs. Chapter 4 develops the specification-based slicing technique for preserving the partial correctness of programs and Chapter 5 extends this technique to the case of total correctness. Chapter 6 applies the slicing technique to a probabilistic model from the literature. Chapter 7 presents an algorithm for computing slices. Chapter 8 discusses some design decisions and limitations behind the slicing technique. Finally, Chapter 9 overviews the related work and Chapter 10 concludes.

Chapter 2

Foundations: Programming and Specification Model

In this chapter we introduce the programming language used for describing probabilistic programs and lay out their functional specification model. While not new, this provides the basic background for understanding the problem we address and fixes the programming model we adopt for our development.

2.1. Programming Language

To describe probabilistic programs we adopt a simple imperative language extended with probabilistic choices, dubbed pWhile. A *program* is a non-empty sequence of instructions, where an *instruction* is either a no-op, an assignment, a conditional branching, a probabilistic choice or a guarded loop, annotated with its invariant. Formally, it is given by grammar:

$$
\begin{array}{lll}
\mathcal{I} ::= & \mathsf{skip} & \text{no–op} \\
\mid & \mathcal{V} := \mathcal{E} & \text{assignment} \\
\mid & \mathsf{if}\,(\mathcal{E})\,\mathsf{then}\,\{\mathcal{C}\}\,\mathsf{else}\,\{\mathcal{C}\} & \text{conditional branching} \\
\mid & \{\mathcal{C}\}\,[p]\,\{\mathcal{C}\} & \text{probabilisitic choice} \\
\mid & \mathsf{while}\,(\mathcal{E})\;[\mathcal{A}]\;\mathsf{do}\,\{\mathcal{C}\} & \text{guarded loop} \\
 & & \\
\mathcal{C} ::= & \mathcal{I} & \text{single instruction} \\
\mid & \mathcal{I};\,\mathcal{C} & \text{sequential composition}
\end{array}
$$

Note that the set of programs, denoted by $\mathcal{C}$, and the set of instructions, denoted by $\mathcal{I}$, are defined mutually recursively. In the definition, we assume a set $\mathcal{V}$ of *variables* and a set $\mathcal{E}$ of *expressions* over program variables. Finally, we use $\mathcal{A}$ to denote the set of program *assertions*, in particular, loop invariants.

No-op's, assignments, conditional branchings and guarded loops are standard. However,

we assume that guarded loops are annotated with invariants so that they can be given an axiomatic semantics based on VCGens. Finally, instruction $\{c_1\}\ [p]\ \{c_2\}$ represents a *probabilistic choice*: it behaves like c_1 with probability p and like c_2 with the complementary probability $1 - p$.

As usual, a program *state* is mapping from variables to values; we use $\mathcal{S}$ to denote the set of program states. Given a state $s \in \mathcal{S}$ and a variable $x \in \mathcal{V}$, we write $s\,[x/v]$ for the state that is obtained from s, by updating the value of x to v. Finally, we assume the presence of an *interpretation function* $[\![\mathcal{E}]\!]$ for expressions, mapping program states to values.

Notational convention. Since sequential composition is associative, we omit parentheses in programs consisting of three or more instructions. In general, we write $c = i_1; i_2; \ldots; i_n$ to denote a program that consists in a sequence of n instructions.

2.2. Program Specifications

The (functional) specification of programs is given by a pair of *pre-* and *post-condition*, which are interpreted on the program initial and final states, respectively. Intuitively, the pre-condition provides a sufficient condition for establishing the post-condition. However, the precise interpretation of this varies depending on the program nature. If the program at stake is deterministic, each initial state either establishes or not the post-condition upon program termination. Therefore, *i*) pre-conditions are *qualitative*, that is, predicates over (initial) program states, and *ii*) an initial state satisfying the pre-condition is guaranteed to establish the post-condition, whereas nothing is guaranteed about an initial state violating the pre-condition. On the other hand, if the program at stake is probabilistic, each initial state establishes the post-condition *with a certain probability*. Thus, *i*) pre-conditions become *qualitative*, mapping each (initial) state to a probability in the interval $[0, 1]$, and *ii*) the probabilities reported by such pre-conditions represent *lower bounds* for the probability that the program establishes the post-condition. For example, specification

$$\{\mathit{if}\ y \geq 0\ \mathit{then}\ \tfrac{1}{2}\ \mathit{else}\ 0\} \quad \{x \coloneqq y\}\ [1/2]\ \{x \coloneqq x + 1\} \quad \{x \geq 0\}^{\downarrow}$$

says that program $\{x \coloneqq y\}[1/2]\{x \coloneqq x + 1\}$ establishes post-condition $x \geq 0$ with probability at least $\frac{1}{2}$ from an initial state where $y \geq 0$, and with probability at least 0 from an initial state where $y < 0$. (Note that the pre-condition is not tight, as it dismiss the case where the right branch of the probabilistic choice establishes the post-condition.)

In fact, both pre-conditions as well as post-conditions become quantitative in the probabilistic case: pre-conditions for the reason argued above and post-conditions because in the presence of sequential composition, say $c_1; c_2$, the established pre-condition of c_2 behaves as the post-condition of c_1, requiring thus a uniform treatment between pre- and post-conditions. Thus, pre- and post-conditions are both functions of type $\mathbb{E} \triangleq \mathcal{S} \to [0, 1]$, known as *expectations*, mapping program states to probabilities. Therefore, in the rest of the presentation we usually refer to the pre- and post-condition of a probabilisitic program as its *pre-* and *post-expectation*, respectively.

To accomodate this generalization, we lift predicates (in particular, post-conditions) to

expectations in a standard manner, taking their *characteristic function*, which maps states satisfying the predicate to 1, and states violating the predicate to 0. In terms of notation, if G is a Boolean expression over program variables encoding a predicate, we use $[G]$ to denote its characteristic function. For example, the above (informal) specification is formally written as

$$\{\tfrac{1}{2}\,[y \geq 0]\} \quad \{x \coloneqq y\}\ [1/2]\ \{x \coloneqq x+1\} \quad \{[x \geq 0]\}^{\downarrow}\ .$$

As already hinted, this pre-expectation is not "tight" or the most precise, as it says that from an initial state where $y < 0$, the program terminates in a final state where $x \geq 0$ with probability at least 0. Even though being (trivially) valid, there is room for significant improvement on this bound. In general, if f and f' are two valid pre-expectations for a probabilistic program specification, and $f \leq f'$ (where the "$\leq$" should be understood pointwise), we usually prefer f' over f. Said otherwise, we are typically interested in the *greatest* pre-expectation. In fact, *greatest* pre-expectations are the probabilistic counterpart of *weakest* pre-conditions. That is, while predicates are ordered by relation "$\Rightarrow$", expectations are ordered by relation "$\leq$". To better highlight this analogy at the notation level, in the rest of the presentation we use symbol $\Rrightarrow$ to denote the pointwise relation "$\leq$" over expectations:

Definition 2.1 (Entailment relation $\Rrightarrow$ between expectations) *For a pair of expectations $f, f' \colon \mathbb{E}$, we let*

$$f \Rrightarrow f' \quad = \quad \forall s \in \mathcal{S}.\ f(s) \leq f'(s)\ .$$

Importantly, this induces a *consistent* extension from the deterministic to the probabilistic case: if P, P' are predicates and $[P]\,, [P']$ denote their respective characteristic functions, then $P \Rightarrow P'$ if and only if $[P] \Rrightarrow [P']$.

Now that we have presented an intuitive approximation to the notion of specification for probabilistic programs, we proceed to define it formally. Like the specification of deterministic programs, that of probabilistic programs comes also in two flavors, differentiating on whether they account for the possibility of divergence, or not. The kind of specifications that we have presented so far corresponds to *total correctness* specifications, since the reported probabilities refer to the probability of *terminating and* establishing the post-condition. On the other hand, *partial correctness* specifications refer to the probability of *either* terminating and establishing the post-condition *or* diverging.

Formally, total and partial correctness are defined in terms of the respective expectation transformers

$$\mathsf{wp}\,[\cdot] : \mathbb{E} \to \mathbb{E} \qquad \text{and} \qquad \mathsf{wlp}\,[\cdot] : \mathbb{E} \to \mathbb{E}\ ,$$

which generalize Dijkstra's *weakest pre-condition* and *weakest liberal pre-condition* transformers [22] from the deterministic to the probabilistic case [20, 23, 24].

Definition 2.2 (Program specification) *We say that a* pWhile *program c satisfies the* total correctness specification *given by pre-expectation f and post-expectation g, written $\models \{f\}\ c\ \{g\}^{\downarrow}$, iff*

$$f \ \Rrightarrow\ \mathsf{wp}\,[c]\,(g)\ .$$

Likewise, we say that a pWhile *program c satisfies the* partial correctness specification *given*

$$
\begin{array}{lcl}
\mathsf{w(l)p}\,[\mathsf{skip}]\,(g) & = & g \\
\mathsf{w(l)p}\,[x \coloneqq E]\,(g) & = & g\,[x/E] \\
\mathsf{w(l)p}\,[\mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}]\,(g) & = & [G]\cdot \mathsf{w(l)p}\,[c_1]\,(g) + [\neg G]\cdot \mathsf{w(l)p}\,[c_2]\,(g) \\
\mathsf{w(l)p}\,[\{c_1\}\,[p]\,\{c_2\}]\,(g) & = & p\cdot \mathsf{w(l)p}\,[c_1]\,(g) + (1-p)\cdot \mathsf{w(l)p}\,[c_2]\,(g) \\
\mathsf{w(l)p}\,[c_1; c_2]\,(g) & = & \mathsf{w(l)p}\,[c_1]\,(\mathsf{w(l)p}\,[c_2]\,(g)) \\
\\
\mathsf{wp}\,[\mathsf{while}\,(G)\,\mathsf{do}\,\{c\}]\,(g) & = & \mu f.\ [\neg G]\cdot g + [G]\cdot \mathsf{wp}\,[c]\,(f) \\
\mathsf{wlp}\,[\mathsf{while}\,(G)\,\mathsf{do}\,\{c\}]\,(g) & = & \nu f.\ [\neg G]\cdot g + [G]\cdot \mathsf{wlp}\,[c]\,(f)
\end{array}
$$

Figure 2.1: Expectation transformer wlp and wp.
$\mu f.\, F(f)$ (resp. $\nu f.\, F(f)$) represents the least (resp. greatest) fixed point of expectation transformer F w.r.t. the entailment order $\Rrightarrow$.

by pre-expectation f and post-expectation g, written $\models \{f\}\ c\ \{g\}^{\circlearrowleft}$, iff

$$f \ \Rrightarrow\ \mathsf{wlp}\,[c]\,(g)\ .$$

Transformers wp and wlp were originally introduced by Kozen [23, 24] and then further extended by McIver and Morgan [20]. They are defined by induction on the program structure, as shown in Figure 2.1. For all language constructs other than loops, both transformers follow the same rules. Let us briefly explain them. $\mathsf{w(l)p}\,[\mathsf{skip}]$ behaves as the identity since skip has no effect. The pre-expectation of an assignment is obtained by updating the program state and then applying the post–expectation, i.e. $\mathsf{w(l)p}\,[x \coloneqq E]$ takes post–expectation g to pre–expectation $g\,[x/E] = \lambda s.\, g(s\,[x/[\![E]\!](s)])$. $\mathsf{w(l)p}\,[\mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}]$ behaves either as $\mathsf{w(l)p}\,[c_1]$ or $\mathsf{w(l)p}\,[c_2]$ according to the evaluation of G. $\mathsf{w(l)p}\,[\{c_1\}\,[p]\,\{c_2\}]$ is obtained as a convex combination of $\mathsf{w(l)p}\,[c_1]$ and $\mathsf{w(l)p}\,[c_2]$, weighted according to p. $\mathsf{w(l)p}\,[c_1; c_2]$ is obtained as the functional composition of $\mathsf{w(l)p}\,[c_1]$ and $\mathsf{w(l)p}\,[c_2]$. Finally, $\mathsf{w(l)p}\,[\mathsf{while}\,(G)\,\mathsf{do}\,\{c\}]$ is defined using standard fixed point techniques, the only difference being the limit fixed point considered: wp takes the least and wlp takes the greatest (according to the $\Rrightarrow$ order between expectations). Observe that, as expected, the definition of $\mathsf{w(l)p}$ over loops dismisses annotated loop invariants (and we thus omit them in Figure 2.1).

We now illustrate the application of wp by means of an example.

Example 2.1 *Consider the program c_1 below that starts by assigning $1.5 - y^2$ to x, and then randomly decrements x, by either 1 or 2.*

$$c_1 \colon \quad x \coloneqq 1.5 - y^2;\ \{x \coloneqq x - 1\}\ [1/2]\ \{x \coloneqq x - 2\}$$

To obtain the probability that the program establishes post-condition $x \geq 0$, we proceed as

follows:

$$
\begin{aligned}
&\mathsf{wp}\,[c_1]\,([x \geq 0]) \\
&\quad = \ \mathsf{wp}\left[x \coloneqq 1.5 - y^2\right]\left(\mathsf{wp}\,[\{x \coloneqq x-1\}\ [1/2]\ \{x \coloneqq x-2\}]\,([x \geq 0])\right) \\
&\quad = \ \mathsf{wp}\left[x \coloneqq 1.5 - y^2\right]\left(\tfrac{1}{2}\,\mathsf{wp}\,[x \coloneqq x-1]\,([x \geq 0]) + \tfrac{1}{2}\,\mathsf{wp}\,[x \coloneqq x-2]\,([x \geq 0])\right) \\
&\quad = \ \mathsf{wp}\left[x \coloneqq 1.5 - y^2\right]\left(\tfrac{1}{2}\ [x-1 \geq 0] + \tfrac{1}{2}\ [x-2 \geq 0]\right) \\
&\quad = \ \tfrac{1}{2}\ \left[(1.5-y^2)-1 \geq 0\right] + \tfrac{1}{2}\ \left[(1.5-y^2)-2 \geq 0\right] \\
&\quad = \ \tfrac{1}{2}\ \underbrace{\left[y^2 \leq -0.5\right]}_{=0} + \tfrac{1}{2}\ \left[y^2 \leq 0.5\right] \\
&\quad = \ \tfrac{1}{2}\ \left[y^2 \leq 0.5\right]
\end{aligned}
$$

We can then conclude that the program establishes the post-condition $x \geq 0$ with (exact) probability $\frac{1}{2}$, when executed from an initial state where $y^2 \leq 0.5$, and with (exact) probability 0, otherwise. △

2.3. Summary

In this chapter we have introduced the programming language used for describing probabilistic programs and have laid out their functional specification model.

The programming language is a simple imperative language extended with a probabilistic choice operator that given a probability p and a pair of programs executes either of them with probability p and $1-p$, respectively.

On the other hand, a funcional specification of a probabilistic program is given by a pair of a pre- and a post-condition. Unlike deterministic programs, where pre- and post-conditions map states to either 0 or 1, in probabilistic programs, pre- and post-conditions map states to a probability in the interval $[0, 1]$. Such pre-conditions represent lower bounds for the probability that programs establish the post-conditions.

Finally, we present transformer $\mathsf{w(l)p}$, which is an adaptation of Dijkstra's weakest precondition for probabilistic programs. This adaptation enables us to propagate the post-condition backward along the program to obtain a valid pre-condition, specifically, the weakest pre-condition for the programs. We use this transformer to determine when a program satisfies a given specification.

Chapter 3

Verification Condition Generator

In this chapter we present the VCGen (Definition 3.1) that will serve as the axiomatic semantic of programs for slicing purposes. We prove it sound (Lemma 3.2) and establish other subsidiary properties (Lemmas 3.3 and 3.4) required for proving the correctness of our slicing approach. While the soundness of the VCGen is not "explicitly" used in our development, it legitimates the notion of slicing based on specifications (Definition 4.1) that we adopt.

Our ultimate goal here is to design a slicing technique that is specification-preserving: Given a program with its purported specification, we would like to synthesize a "subset" of the program that still complies with the specification. A fundamental requirement for the practical adoption of this—and any other—slicing technique is that it is amenable to automation. However, determining whether a program complies with a given specification is known to be an undecidable problem (primarily because of the undecidability of entailment in first-order logic).

To address this limitation, we draw on a well-known tool from the program verification community: Verification Condition Generators (VCGens). A VCGen is a tool that given a program *annotated with loop invariants*, together with its purported specification, generates a set of proof obligations, also known as *verification conditions*, such that their validity entails the program correctness w.r.t. the specification. The key point here is that these so-generated verification conditions can be typically discharged by automated theorem provers such as SMT Solvers.

The classical approach for designing VCGens leverages predicate transformers, or in the case of probabilistic programs, expectation transformers. The transformer wpre that we use for designing our VCGen (see Figure 3.1) is an adaptation of the transformers $\mathsf{w(l)p}$ from Figure 2.1, deviating from them in the case of loops to support the automatization enabled by annotated invariants. More specifically, for any instruction different from a loop, wpre behaves like $\mathsf{w(l)p}$ transforming a post-expectation into the greatest (i.e. the most precise) pre-expectation establishing the post-expectation. For a loop, it simply returns the annotated loop invariant. The intuition behind this latter rule is that the VCGen will generate the necessary proof obligations to ensure that the annotated invariant (*inv* in Figure 3.1) is a valid pre-expectation—though possibly not the greatest—w.r.t. the given post-expectation (g in Figure 3.1).

$$
\begin{array}{lcl}
\mathsf{wpre}\,[\mathsf{skip}]\,(g) & = & g \\
\mathsf{wpre}\,[x \coloneqq E]\,(g) & = & g\,[x/E] \\
\mathsf{wpre}\,[\mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}]\,(g) & = & [G] \cdot \mathsf{wpre}\,[c_1]\,(g) + [\neg G] \cdot \mathsf{wpre}\,[c_2]\,(g) \\
\mathsf{wpre}\,[\{c_1\}\,[p]\,\{c_2\}]\,(g) & = & p \cdot \mathsf{wpre}\,[c_1]\,(g) + (1-p) \cdot \mathsf{wpre}\,[c_2]\,(g) \\
\mathsf{wpre}\,[c_1; c_2]\,(g) & = & \mathsf{wpre}\,[c_1]\,(\mathsf{wpre}\,[c_2]\,(g)) \\
\mathsf{wpre}\,[\mathsf{while}\,(G)\;[inv]\;\mathsf{do}\,\{c\}]\,(g) & = & inv
\end{array}
$$

Figure 3.1: Expectation transformer wpre used for defining the VCGen.

The transformer wpre satisfies appealing algebraic properties, which include monotonicity and linearity:

Lemma 3.1 (Basic properties of transformer wpre) *For any* pWhile *program c, any two expectations $f, f' \colon \mathbb{E}$ and any probability $p \in [0,1]$, it holds:*

$$
\begin{array}{ll}
\text{Monotonicity:} & f \Rrightarrow f' \quad \Longrightarrow \quad \mathsf{wpre}\,[c]\,(f) \Rrightarrow \mathsf{wpre}\,[c]\,(f') \\
\text{Linearity:} & \mathsf{wpre}\,[c]\,(pf + (1-p)f') \;=\; p \cdot \mathsf{wpre}\,[c]\,(f) + (1-p) \cdot \mathsf{wpre}\,[c]\,(f')
\end{array}
$$

Proof. Both proofs proceed by induction on the program structure. For the case of loops, the results are immediate since the transformer is constant (always yielding the annotated loop invariant). For the remaining cases, the proofs follow the same arguments as for transformer wp; see, e.g., [25]. □

Having introduced the expectation transformer wpre, we are now in a position to define the VCGen for probabilistic programs. For making the presentation more incremental, in this chapter we introduce the VCGen for establishing partial correctness specifications only, and defer the treatment of total correctness to Chapter 5.

The VCGen takes a pWhile program c, a pre-expectation f and a post-expectation g, and returns a set $\mathrm{VCG}\,[c]\,(f,g)$ of verification conditions such that their validity entails that c adheres to the specification given by f and g. The returned verification conditions are entailment claims between expectations, i.e. claims of the form $f' \Rrightarrow g'$ (which generalize the entailment between predicates returned by VCGens for deterministic programs).

Definition 3.1 (VCGen for partial correctness) *The set of verification conditions* $\mathrm{VCG}\,[c]\,(f,g)$ *for the partial correctness of a* pWhile *program c w.r.t. pre-expectation f and post-expectation g is defined as:*

$$\mathrm{VCG}\,[c]\,(f,g) \;=\; \{f \Rrightarrow \mathsf{wpre}\,[c]\,(g)\} \cup \mathsf{vc}\,[c]\,(g) \;\;,$$

where $\mathsf{vc}\,[c]\,(g)$ *is defined in Figure 3.2, by induction on the structure of c.*

To extract the verification conditions, $\mathrm{VCG}\,[c]\,(f,g)$ proceeds roughly as follows. First, it leverages transformer wpre to compute a valid pre-expectation $\mathsf{wpre}\,[c]\,(g)$ that establishes

$$
\begin{array}{lcl}
\mathsf{vc}\,[\mathsf{skip}]\,(g) & = & \emptyset \\
\mathsf{vc}\,[x \coloneqq E]\,(g) & = & \emptyset \\
\mathsf{vc}\,[\mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}]\,(g) & = & \mathsf{vc}\,[c_1]\,(g)\ \cup\ \mathsf{vc}\,[c_2]\,(g) \\
\mathsf{vc}\,[\{c_1\}\ [p]\ \{c_2\}]\,(g) & = & \mathsf{vc}\,[c_1]\,(g)\ \cup\ \mathsf{vc}\,[c_2]\,(g) \\
\mathsf{vc}\,[c_1; c_2]\,(g) & = & \mathsf{vc}\,[c_1]\,(\mathsf{wpre}\,[c_2]\,(g))\ \cup\ \mathsf{vc}\,[c_2]\,(g) \\
\mathsf{vc}\,[\mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c\}]\,(g) & = & \{[G] \cdot inv \Rrightarrow \mathsf{wpre}\,[c]\,(inv)\,, \\
 & & \quad [\neg G] \cdot inv \Rrightarrow g\}\ \cup\ \mathsf{vc}\,[c]\,(inv)
\end{array}
$$

Figure 3.2: Verification condition generator for partial correctness of pWhile programs.

the declared post-expectation g, and then verifies that the declared pre-expectation f entails the so-computed pre-expectation. This generates verification condition $f \Rrightarrow \mathsf{wpre}\,[c]\,(g)$.

However, while computing the pre-expectation $\mathsf{wpre}\,[c]\,(g)$, the VCGen makes two assumptions about c loops that must be accounted for: On the one hand, that the annotated invariants are indeed invariants, that is, that they are preserved by the body of the respective loops. On the other hand, that the invariants are strong enough as to establish the expectations that should hold upon exit of the loops (as computed by transformer wpre). The verification conditions accounting for these assumptions are captured by $\mathsf{vc}\,[c]\,(g)$ (see Figure 3.2).

Rules defining $\mathsf{vc}\,[c]\,(g)$ are mostly self-explanatory. The most important rule is the one for loops, as these are the only instructions that generate verification conditions. Concretely, $\mathsf{vc}\,[\mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c\}]\,(g)$ extends the potential set of verification conditions induced by (the loops in) c with two additional verification conditions: *i)* $[G] \cdot inv \Rrightarrow \mathsf{wpre}\,[c]\,(inv)$, which ensures that inv is indeed a loop invariant, and *ii)* $[\neg G] \cdot inv \Rrightarrow g$, which ensures that upon loop exit, the invariant is strong enough as to establish post-expectation g. The remaining rules simply collect the verification conditions generated by loops. The only subtlety appears in the rule for sequential composition, where $\mathsf{vc}\,[c_1; c_2]\,(g)$ collects the verification conditions generated by c_1 applying $\mathsf{vc}\,[c_1]$ to $\mathsf{wpre}\,[c_2]\,(g)$ as this is the post-expectation of c_1 (yielded by wpre) when g is the post-expectation of $c_1; c_2$.

We next establish three relevant properties of the VCGen. First, the VCGen is *sound* meaning that the validity of the verification conditions $\mathrm{VCG}\,[c]\,(f, g)$ entail the validity of partial correctness specification $\{f\}\ c\ \{g\}^{\circlearrowleft}$.

Lemma 3.2 (Soundness of VCG) *For any* pWhile *program c and any two expectations $f, g \colon \mathbb{E}$,*

$$\models \mathrm{VCG}\,[c]\,(f, g) \quad \Longrightarrow \quad \models \{f\}\ c\ \{g\}^{\circlearrowleft}\ .$$

Proof. The result follows as an immediate corollary of the following property:

$$\models \mathsf{vc}\,[c]\,(g) \quad \Longrightarrow \quad \mathsf{wpre}\,[c]\,(g) \Rrightarrow \mathsf{wlp}\,[c]\,(g)\ ,$$

which can be established by induction on the structure of c. See Appendix A for details. $\square$

The remaining two properties are required to prove the correctness of the slicing techniques from Chapter 4 (Theorems 4.1 and 4.2). One property is the monotonicity of $\mathsf{vc}\,[c]$ and $\mathrm{VCG}\,[c]$:

Lemma 3.3 (Monotonicity of vc/VCG) *For any* pWhile *program c and any four expectations $f, f', g, g' : \mathbb{E}$,*

$$\begin{aligned} g \Rrightarrow g' \qquad &\Longrightarrow \qquad \models \mathsf{vc}\,[c]\,(g) \;\Rightarrow \models \mathsf{vc}\,[c]\,(g') \;, \\ f' \Rrightarrow f \;\wedge\; g \Rrightarrow g' \qquad &\Longrightarrow \qquad \models \mathrm{VCG}\,[c]\,(f,g) \;\Rightarrow \models \mathrm{VCG}\,[c]\,(f',g') \;. \end{aligned}$$

Proof. Both monotonicity proofs rely on the monotonicity of $\mathsf{wpre}\,[c]$ (Lemma 3.1). The monotonicity proof of $\mathsf{vc}\,[c]$ proceeds by routine induction on the structure of c (see Appendix A for details). The monotonicity proof of $\mathrm{VCG}\,[c]$ follows as an immediate corollary. $\square$

The last property is an alternative characterization of $\models \mathrm{VCG}\,[c]\,(f,g)$ for the case where c contains compound instructions featuring subprograms, that is, conditional branches, probabilistic choices or loops. To state the result we need variants of $\mathsf{wpre}\,[c]\,(g)$ and $\mathsf{vc}\,[c]\,(g)$ that act on c suffixes, and variants of $\mathrm{VCG}\,[c]\,(f,g)$ that act on c suffixes and prefixes. Assuming that $c = i_1; \ldots; i_n$, we then define:

$$\begin{aligned} \mathsf{wpre}_{\geq j}\,[c]\,(g) &= \begin{cases} \mathsf{wpre}\,[i_j; i_{j+1}; \ldots; i_n]\,(g) & \text{if } 1 \leq j \leq n \\ g & \text{if } j = n+1 \end{cases} \\ \mathsf{vc}_{\geq j}\,[c]\,(g) &= \begin{cases} \mathsf{vc}\,[i_j; i_{j+1}; \ldots; i_n]\,(g) & \text{if } 1 \leq j \leq n \\ \emptyset & \text{if } j = n+1 \end{cases} \\ \mathrm{VCG}_{\geq j}\,[c]\,(f,g) &= \begin{cases} \mathrm{VCG}\,[i_j; i_{j+1}; \ldots; i_n]\,(f,g) & \text{if } 1 \leq j \leq n \\ \emptyset & \text{if } j = n+1 \end{cases} \\ \mathrm{VCG}_{\leq j}\,[c]\,(f,g) &= \begin{cases} \mathrm{VCG}\,[i_1; \ldots; i_j]\,(f,g) & \text{if } 1 \leq j \leq n \\ \{f \Rrightarrow g\} & \text{if } j = 0 \end{cases} \end{aligned}$$

Lemma 3.4 (Alt. characterization of $\models$ VCG) *For any* pWhile *program $c = i_1; \ldots; i_n$ and any two expectations $f, g : \mathbb{E}$,*

1. *If $i_j = \mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}$ or $i_j = \{c_1\}\,[p]\,\{c_2\}$ for some $1 \leq j \leq n$, then*

$$\begin{aligned} \models \mathrm{VCG}\,[c]\,(f,g) \qquad \textit{iff} \qquad & \models \mathsf{vc}_{\geq j+1}\,[c]\,(g) \;\wedge \\ & \models \mathsf{vc}\,[c_1]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \;\wedge \\ & \models \mathsf{vc}\,[c_2]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \;\wedge \\ & \models \mathrm{VCG}_{\leq j-1}\,[c]\left(f, \mathsf{wpre}_{\geq j}\,[c]\,(g)\right) \;. \end{aligned}$$

2. If $i_j = \mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c'\}$ *for some* $1 \leq j \leq n$*, then*

$$\models \mathrm{VCG}\,[c]\,(f,g) \qquad \textit{iff} \qquad \begin{array}{l} \models \mathrm{VCG}_{\geq j+1}\,[c]\,([\neg G]\cdot inv, g) \quad \wedge \\ \models \mathrm{VCG}\,[c']\,([G]\cdot inv, inv) \quad \wedge \\ \models \mathrm{VCG}_{\leq j-1}\,[c]\,(f, inv) \ . \end{array}$$

Proof. It follows from the definition of $\models \mathrm{VCG}\,[c]\,(f,g)$, by splitting $c = i_1; \ldots; i_n$ into the prefix before i_j, i_j and the suffix after i_j. See Appendix A for details. □

3.1. Summary

In this chapter we have presented the axiomatic semantics that we use later on to prove the soundness of our slicing technique.

This semantics establishes that a program satisfies a specification when the invariants annotated in the program's loops, are strong enough as to establish the specification. This is formalized through a VCGen (Definition 3.1), which is defined in terms of an adaptation of wlp transformer, denoted by wpre (Figure 3.1).

Lastly, we show that the VCGen is sound with respect to the wlp (Lemma 3.2) and we prove some auxiliary properties that we require in the next chapter, among others, the monotonicity of VCGen (Lemma 3.3) and an alternative characterization of $\models$ VCG for conditional instructions, probabilisitic choices and loops (Lemma 3.4).

Chapter 4

The Slice Transformation

In this chapter we present the two fundamental results (Theorems 4.1 and 4.2) that allow identifying removable program fragments and underlie our slicing approach. We prove the theorems correct and show application examples (Examples 4.1 and 4.2). This pair of theorems form the cornerstone of our theoretical contribution.

4.1. Specification-based Slice

Roughly speaking, given a pWhile program c together with its specification, a *specification-based slice* is obtained by removing from c those fragments that do not contribute to establishing the specification. Thus, the notion of specification-based slice involves a syntactic and a semantic component that we formally define next.

The syntactic component is captured by the relation "*being-portion-of*" over programs, denoted by "$\preccurlyeq$". Informally, $c' \preccurlyeq c$ if c' is obtained from c by removing some instructions. The relation is formally defined by the set of rules in Figure 4.1. The first two rules say that we can obtain a portion of a program consisting in a sequence of n instructions by removing either all its instructions (resulting in skip) or a proper subsequence of contiguous instructions. The following four rules represent congruence rules for the sequential composition, conditional branching, probabilistic choice and loops. Finally, the last two rules encode the reflexivity and transitivity of the relation.[3]

As for the semantic component, we assume that the semantics of a program c is given by the verification condition generator $\mathrm{VCG}\,[c]$, or said otherwise, that a program c satisfies a specification given by, say pre-expectation f and post-expectation g, if and only if $\models \mathrm{VCG}\,[c]\,(f, g)$. The "if" direction refers to the *soundness* of the VCGen and was already established in Lemma 3.2. The "only if" direction refers to the *completeness* of the VCGen, that is, if a program satisfies a specification, then it is always possible to annotate the

[3] In view of the congruence rule for sequential composition and the rule stating that skip is a portion of any program, we could have discarded the rule that allows removing a proper subsequence of instructions of a program to obtain a portion thereof. However, we preferred to keep it because it yields cleaner program slices, e.g., $i_1; i_5$ instead of $i_1; \mathsf{skip}; \mathsf{skip}; \mathsf{skip}; i_5$, and also simplifies, to some degree, the proofs.

$$\frac{}{\mathsf{skip} \preccurlyeq i_1; \ldots; i_n} \qquad \frac{1 < j \leq k \leq n \text{ or } 1 \leq j \leq k < n}{i_1; \ldots; i_{j-1}; i_{k+1}; \ldots; i_n \preccurlyeq i_1; \ldots; i_j; \ldots; i_k; \ldots; i_n}$$

$$\frac{i'_j \preccurlyeq i_j \qquad 1 \leq j \leq n}{i_1; \ldots; i'_j; \ldots; i_n \preccurlyeq i_1; \ldots; i_j; \ldots; i_n}$$

$$\frac{c'_1 \preccurlyeq c_1 \qquad c'_2 \preccurlyeq c_2}{\mathsf{if}\ (G)\ \mathsf{then}\ \{c'_1\}\ \mathsf{else}\ \{c'_2\} \preccurlyeq \mathsf{if}\ (G)\ \mathsf{then}\ \{c_1\}\ \mathsf{else}\ \{c_2\}}$$

$$\frac{c'_1 \preccurlyeq c_1 \qquad c'_2 \preccurlyeq c_2}{\{c'_1\}\ [p]\ \{c'_2\} \preccurlyeq \{c_1\}\ [p]\ \{c_2\}}$$

$$\frac{c' \preccurlyeq c}{\mathsf{while}\ (G)\ [inv]\ \mathsf{do}\ \{c'\} \preccurlyeq \mathsf{while}\ (G)\ [inv]\ \mathsf{do}\ \{c\}}$$

$$\frac{}{c \preccurlyeq c} \qquad \frac{c_1 \preccurlyeq c_2 \qquad c_2 \preccurlyeq c_3}{c_1 \preccurlyeq c_3}$$

Figure 4.1: Relation "*is-portion-of*" over programs.

program with appropriate loop invariants such that the VCGen can establish the specification. In fact, one can prove that the exact semantics of a loop w.r.t. a post-expectation as given by transformer wlp is always a valid invariant, strong enough as to establish the post-expectation. For the rest of our development, we thus assume that programs are annotated with appropriate loop invariants as to establish the purported specification. (This is also a natural assumption for any other automated program verification task.)

Definition 4.1 (Program slicing based on partial correctness specification [17]) *We say that* pWhile *program c' is a* specification-based slice *of* pWhile *program c with respect to the partial correctness specification given by pre-expectation f and post-expectation g, written $\{f\}\ c' \preccurlyeq c\ \{g\}^{\circlearrowleft}$, iff*

1. *$c' \preccurlyeq c$, and*
2. *$\models \mathrm{VCG}\,[c]\,(f, g) \ \Rightarrow \models \mathrm{VCG}\,[c']\,(f, g)$*

Observe that it only makes sense to compute specification-based slices of programs that adhere to their specifications: If a program violates its declared specification, then any portion of the program (including, e.g., skip) is a vacuously valid specification-based slice.

4.2. Removing Instructions

We next present our two fundamental results for deriving specification-based slices of probabilistic programs. The first result allows removing top-level instructions of a program, and

the second result, nested instructions.

4.2.1. Removing top-level instructions

Given a program $c = i_1; \ldots; i_n$, the first result allows slicing away a contiguous subsequence of instructions. We thus begin introducing the function $\mathsf{remove}\colon \mathsf{pWhile} \to \mathsf{pWhile}$ that captures this program transformation:

$$\mathsf{remove}\,(j,k,c) \;=\; \begin{cases} \mathsf{skip} & \text{if } j = 1 \text{ and } k = n \\ i_1; \ldots; i_{j-1}; i_{k+1}; \ldots; i_n & \text{otherwise} \end{cases}$$

In words, $\mathsf{remove}\,(i,j,c)$ slices away from c from the j-th to k-th instructions, inclusive.

The slicing criteria requires propagating the program post-expectation, say g, backward, along all its instructions, that is, calculating $\mathsf{wpre}_{\geq j}\,[c]\,(g)$ for all $j = 1, \ldots, n$. Then, if for some $1 \leq j \leq k \leq n$, $\mathsf{wpre}_{\geq j}\,[c]\,(g)$ happens to entail $\mathsf{wpre}_{\geq k+1}\,[c]\,(g)$, we can remove the subsequence of instructions from j-th to k-th.

Theorem 4.1 (Removing top-level instructions for partial correctness) *Let $c = i_1; \ldots; i_n$ be a* pWhile *program together with its respective pre- and post-expectation f and g. Moreover, let $1 \leq j \leq k \leq n$. If*

$$\mathsf{wpre}_{\geq j}\,[c]\,(g) \;\Rrightarrow\; \mathsf{wpre}_{\geq k+1}\,[c]\,(g)$$

then,

$$\{f\}\ \mathsf{remove}\,(j,k,c) \preccurlyeq c\ \{g\}^{\circlearrowleft}\ .$$

Proof. We show that $\models \mathrm{VCG}\,[c]\,(f,g)$ entails $\models \mathrm{VCG}\,[\mathsf{remove}\,(j,k,c)]\,(f,g)$:

$$\begin{aligned}
&\models \mathrm{VCG}\,[c]\,(f,g) \\
\Leftrightarrow\quad & \{\text{def of VCG}\} \\
&\models \{f \Rrightarrow \mathsf{wpre}\,[c]\,(g)\} \cup \mathsf{vc}\,[c]\,(g) \\
\Leftrightarrow\quad & \{c = i_1; \ldots; i_n\ \} \\
&\models \{f \Rrightarrow \mathsf{wpre}\,[i_1; \ldots; i_n]\,(g)\} \cup \mathsf{vc}\,[i_1; \ldots; i_n]\,(g) \\
\Leftrightarrow\quad & \{\text{def of } \mathsf{wpre} \text{ and } \mathsf{vc} \text{ for sequential composition}\} \\
&\models \{f \Rrightarrow \mathsf{wpre}\,[i_1; \ldots; i_{j-1}]\,\Big(\mathsf{wpre}_{\geq j}\,[c]\,(g)\Big)\} \\
&\quad \cup\ \mathsf{vc}\,[i_1; \ldots; i_{j-1}]\,\Big(\mathsf{wpre}_{\geq j}\,[c]\,(g)\Big) \cup \mathsf{vc}\,[i_j; \ldots; i_n]\,(g) \\
\Rightarrow\quad & \{\text{hypothesis, monotonicity of } \mathsf{vc} \text{ and } \mathsf{wpre}\} \\
&\models \{f \Rrightarrow \mathsf{wpre}\,[i_1; \ldots; i_{j-1}]\,\Big(\mathsf{wpre}_{\geq k+1}\,[c]\,(g)\Big)\} \\
&\quad \cup\ \mathsf{vc}\,[i_1; \ldots; i_{j-1}]\,\Big(\mathsf{wpre}_{\geq k+1}\,[c]\,(g)\Big) \cup \mathsf{vc}\,[i_j; \ldots; i_n]\,(g) \\
\Leftrightarrow\quad & \{\text{def of } \mathsf{vc} \text{ for sequential composition}\}
\end{aligned}$$

$$
\begin{aligned}
&\models \{f \Rrightarrow \mathsf{wpre}\,[i_1;\ldots;i_{j-1}]\left(\mathsf{wpre}_{\geq k+1}\,[c]\,(g)\right)\} \\
&\quad \cup\ \mathsf{vc}\,[i_1;\ldots;i_{j-1}]\left(\mathsf{wpre}_{\geq k+1}\,[c]\,(g)\right) \cup \mathsf{vc}\,[i_j;\ldots;i_k]\left(\mathsf{wpre}_{\geq k+1}\,[c]\,(g)\right) \\
&\quad \cup\ \mathsf{vc}\,[i_k;\ldots;i_n]\,(g) \\
\Leftrightarrow\quad & \{\text{associativity and def of } \mathsf{vc} \text{ for sequential composition}\} \\
&\models \{f \Rrightarrow \mathsf{wpre}\,[\mathsf{remove}\,(j,k,c)]\,(g)\} \cup \mathsf{vc}\,[\mathsf{remove}\,(j,k,c)]\,(g) \\
&\quad \cup\ \mathsf{vc}\,[i_j;\ldots;i_k]\left(\mathsf{wpre}_{\geq k+1}\,[c]\,(g)\right) \\
\Leftrightarrow\quad & \{\text{def of VCG}\} \\
&\models \mathrm{VCG}\,[\mathsf{remove}\,(j,k,c)]\,(f,g) \cup \mathsf{vc}\,[i_j;\ldots;i_k]\left(\mathsf{wpre}_{\geq k+1}\,[c]\,(g)\right) \\
\Rightarrow\quad & \{\text{weakening}\} \\
&\models \mathrm{VCG}\,[\mathsf{remove}\,(j,k,c)]\,(f,g)
\end{aligned}
$$

□

We next illustrate the application of Theorem 4.1 to slice the program from Example 2.1.

Example 4.1 *Consider the program c_1 from Example 2.1, with pre-expectation $f = \frac{1}{2}\,[y^2 \leq 0.5]$ and post-expectation $g = [x \geq 0]$. Below we display the program, along with the expectations $\mathsf{wpre}_{\geq j}\,[c_1]\,(g)$ (abbreviated $\mathsf{wpre}_{\geq j}$) for $j = 1, 2, 3$, that are obtained by propagating g backward.*

$$
\begin{aligned}
f: &\quad \backslash\backslash\ \tfrac{1}{2}\,[y^2 \leq 0.5] \\
\mathsf{wpre}_{\geq 1}: &\quad \backslash\backslash\ \tfrac{1}{2}\,[1.5 - y^2 \geq 1] + \tfrac{1}{2}\,[1.5 - y^2 \geq 2] \\
i_1: &\quad x \coloneqq 1.5 - y^2; \\
\mathsf{wpre}_{\geq 2}: &\quad \backslash\backslash\ \tfrac{1}{2}\,[x \geq 1] + \tfrac{1}{2}\,[x \geq 2] \\
i_2: &\quad \{x \coloneqq x - 1\}\ [1/2]\ \{x \coloneqq x - 2\} \\
\mathsf{wpre}_{\geq 3} = g: &\quad \backslash\backslash\ [x \geq 0]
\end{aligned}
$$

By doing a case analysis on the value that variable x can have in an arbitrary state s, taking $x \in (-\infty, 0)$, $x \in [0, 1)$, $x \in [1, 2)$ or $x \in [2, \infty)$, it is not hard to see that in all four cases, $\mathsf{wpre}_{\geq 2}\,[c_1]\,(g)\,(s) \leq \mathsf{wpre}_{\geq 3}\,[c_1]\,(g)\,(s)$. In other words,

$$\mathsf{wpre}_{\geq 2}\,[c_1]\,(g) \ \Rrightarrow\ \mathsf{wpre}_{\geq 3}\,[c_1]\,(g)\ ,$$

which in view of Theorem 4.1 allows us to slice away the probabilistic choice i_2 from c_1, while preserving its specification. △

4.2.2. Removing nested instructions

Given a program $c = i_1; \ldots; i_n$, Theorem 4.1 allows slicing away "top-level" instructions of c. For example, if for some $1 \leq j \leq n$, instruction i_j is a conditional branching, Theorem 4.1 allows slicing away the entire conditional branches. However, in some circumstances, we may obtain a valid slice by removing instructions from either of its branches, only. In general, this may be the case for any other *compound* instruction i_j of c such as a probabilistic choice or a loop. Next, we present a complementary result to Theorem 4.1 that enables this kind of slice.

To state the result, we need the notion of *local specification*. Intuitively, if c is a program e.g. with a conditional branching, then any specification of c induces a "local specification" on each of the two branches. In turn, if one of the banches contains e.g. a loop, the local specification of the branch induces a (deeper) local specification on the loop body. Notationwise, we write

$$c \vdash \langle f, g \rangle^{\circlearrowleft} \rightsquigarrow c' \vdash \langle f', g' \rangle^{\circlearrowleft}$$

to denote that specification $\{f\}\ c\ \{g\}^{\circlearrowleft}$ induces local specification $\{f'\}\ c'\ \{g'\}^{\circlearrowleft}$ on the subprogram c' of c. The relation $\rightsquigarrow$ is formally defined by the set of rules in Figure 4.2.

Let us briefly explain the rules. Assume that the specification of the program at hand $c = i_1; \ldots; i_n$ is given by pre-expectation f and post-expectation g. Furthermore, assume that its instruction i_j is compound. If i_j is a conditional branching, then the local specification induced on either of its branches is as follows: the post-expectation is obtained by propagating g backward along $c = i_1; \ldots; i_n$, until reaching i_j; the pre-expectation is obtained by further propagating the so-calculated post-expectation along the branch, and restricting the result to the branch respective guard (rules [$\rightsquigarrow$ ift] and [$\rightsquigarrow$ iff]). If i_j is a probabilistic choice, the local specification induced on either of its branches is defined similarly, except that pre-expectations are not guarded (rules [$\rightsquigarrow$ pl] and [$\rightsquigarrow$ pr]). If i_j is a loop, the local specification induced on its body is as follows: the post-expectation is the loop invariant, and the pre-expectation is the loop invariant, restricted to the loop guard (rule [$\rightsquigarrow$ while]). Finally, these definitions can be applied recursively, to yield the local specification of a subprogram at any depth level of the original program (rule [$\rightsquigarrow$ trans]).

The value of local specifications resides in that they allow a *modular* approach to slicing: If a program with its specification induces a local specification on a given subprogram, then slicing the subprogram w.r.t. the local specification yields a valid slice of the original program (w.r.t. to its original specification).

Theorem 4.2 (Removing nested instructions for partial correctness) *Let c be a* pWhile *program together with its respective pre- and post-expectation f and g, and let c' be a subprogram of c such that $c \vdash \langle f, g \rangle^{\circlearrowleft} \rightsquigarrow c' \vdash \langle f', g' \rangle^{\circlearrowleft}$. If*

$$\{f'\}\ c'' \preccurlyeq c'\ \{g'\}^{\circlearrowleft}\ ,$$

then

$$\{f\}\ c\,[c'/c'']\ \preccurlyeq c\ \{g\}^{\circlearrowleft}\ ,$$

where $c\,[c'/c'']$ denotes the program that is obtained from c by replacing c' with c''.

Proof. By induction on the derivation of $c \vdash \langle f, g \rangle^{\circlearrowleft} \rightsquigarrow c' \vdash \langle f', g' \rangle^{\circlearrowleft}$. See Appendix A for details. □

Theorem 4.2 embodies a local reasoning principle, which is crucial for the simplicity (and elegance) of the technique, and for keeping the computation of slices tractable.

We now illustrate the application of Theorem 4.2, and more broadly, the application of specification-based slicing for software reuse.

$$\frac{i_j = \mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}}{c \vdash \langle f, g\rangle^{\circlearrowleft} \rightsquigarrow c_1 \vdash \left\langle [G] \cdot \mathsf{wpre}\,[c_1]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right), \mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right\rangle^{\circlearrowleft}} \;[\rightsquigarrow \mathsf{ift}]$$

$$\frac{i_j = \mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}}{c \vdash \langle f, g\rangle^{\circlearrowleft} \rightsquigarrow c_2 \vdash \left\langle [\neg G] \cdot \mathsf{wpre}\,[c_2]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right), \mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right\rangle^{\circlearrowleft}} \;[\rightsquigarrow \mathsf{iff}]$$

$$\frac{i_j = \{c_1\}\,[p]\,\{c_2\}}{c \vdash \langle f, g\rangle^{\circlearrowleft} \rightsquigarrow c_1 \vdash \left\langle \mathsf{wpre}\,[c_1]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right), \mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right\rangle^{\circlearrowleft}} \;[\rightsquigarrow \mathsf{pl}]$$

$$\frac{i_j = \{c_1\}\,[p]\,\{c_2\}}{c \vdash \langle f, g\rangle^{\circlearrowleft} \rightsquigarrow c_2 \vdash \left\langle \mathsf{wpre}\,[c_2]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right), \mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right\rangle^{\circlearrowleft}} \;[\rightsquigarrow \mathsf{pr}]$$

$$\frac{i_j = \mathsf{while}\,(G)\;[inv]\;\mathsf{do}\,\{c'\}}{c \vdash \langle f, g\rangle^{\circlearrowleft} \rightsquigarrow c' \vdash \langle [G] \cdot inv, inv\rangle^{\circlearrowleft}} \;[\rightsquigarrow \mathsf{while}]$$

$$\frac{}{c \vdash \langle f, g\rangle^{\circlearrowleft} \rightsquigarrow c \vdash \langle f, g\rangle^{\circlearrowleft}} \;[\rightsquigarrow \mathsf{refl}]$$

$$\frac{c_1 \vdash \langle f_1, g_1\rangle^{\circlearrowleft} \rightsquigarrow c_2 \vdash \langle f_2, g_2\rangle^{\circlearrowleft} \qquad c_2 \vdash \langle f_2, g_2\rangle^{\circlearrowleft} \rightsquigarrow c_3 \vdash \langle f_3, g_3\rangle^{\circlearrowleft}}{c_1 \vdash \langle f_1, g_1\rangle^{\circlearrowleft} \rightsquigarrow c_3 \vdash \langle f_3, g_3\rangle^{\circlearrowleft}} \;[\rightsquigarrow \mathsf{trans}]$$

Figure 4.2: Relation of local specification inducement.
For the first five rules we assume that $c = i_1; \ldots; i_n$ and $1 \leq j \leq n$.

Example 4.2 *Consider the program below, that assigns to variable r a random integer uniformly distributed in the interval $[0, 16)$:*

$$\begin{aligned}
&\backslash\backslash\ \tfrac{1}{16}\,[0 \leq K < 16]\\
&\{b_0 \coloneqq 0\}\ [1/2]\ \{b_0 \coloneqq 1\};\\
&\{b_1 \coloneqq 0\}\ [1/2]\ \{b_1 \coloneqq 1\};\\
&\{b_2 \coloneqq 0\}\ [1/2]\ \{b_2 \coloneqq 1\};\\
&\{b_3 \coloneqq 0\}\ [1/2]\ \{b_3 \coloneqq 1\};\\
&r \coloneqq b_0 + 2b_1 + 4b_2 + 8b_3\\
&\backslash\backslash\ [r = K]
\end{aligned}$$

Intuitively, it encodes r as a four-bit binary number (we assume that $b_0, \ldots, b_3$ are $\{0,1\}$-valued variables) and randomly assigns a value to each bit. Since all four bits and uniformly and independently distributed, r takes each of the values $0, \ldots, 15$ with probability $(1/2)^4 = 1/16$. Formally, the program satisfies the specification given by pre-expectation $\frac{1}{16}\,[0 \leq K < 16]$ and post-expectation $[r = K]$.

Now assume we would like to reuse the program in another context where a random integer is required, but instead of requiring that the integer be uniformly distributed in the interval

$[0,16)$, *the context only requires that the integer be at least 8 with probability (at least)* $1/2$. *We can then slice the program with respect to this weaker specification, given by pre-expectation* $f = \frac{1}{2}$ *and post-expectation* $g = [r \geq 8]$. *Propagating the post-expectation backward along the program and calculating the local specification induced over the left branch of the last probabilistic choice yields the result below. Therein, for convenience, we use* s_n *as a shorthand for the partial sum* $\sum_{i=0}^{n} 2^i b_i$.

$$
\begin{aligned}
& \backslash\backslash\ f = \tfrac{1}{2} \\
& \backslash\backslash\ \mathsf{wpre}_{\geq 1} = \textstyle\sum_{j\in\{0,\ldots,15\}} \frac{1}{16}\,[j \geq 8] \\
i_1 :\quad & \{b_0 \coloneqq 0\}\ [1/2]\ \{b_0 \coloneqq 1\}; \\
& \backslash\backslash\ \mathsf{wpre}_{\geq 2} = \textstyle\sum_{j\in\{0,2,4,\ldots,12,14\}} \frac{1}{8}\,[s_0 + j \geq 8] \\
i_2 :\quad & \{b_1 \coloneqq 0\}\ [1/2]\ \{b_1 \coloneqq 1\}; \\
& \backslash\backslash\ \mathsf{wpre}_{\geq 3} = \textstyle\sum_{j\in\{0,4,8,12\}} \frac{1}{4}\,[s_1 + j \geq 8] \\
i_3 :\quad & \{b_2 \coloneqq 0\}\ [1/2]\ \{b_2 \coloneqq 1\}; \\
& \backslash\backslash\ \mathsf{wpre}_{\geq 4} = \textstyle\sum_{j\in\{0,8\}} \frac{1}{2}\,[s_2 + j \geq 8] \\
i_4 :\quad & \{\backslash\backslash\ f_3 \quad b_3 \coloneqq 0 \quad \backslash\backslash\ g_3\}\ [1/2]\ \{b_3 \coloneqq 1\}; \\
& \backslash\backslash\ \mathsf{wpre}_{\geq 5} = [s_3 \geq 8] \\
i_5 :\quad & r \coloneqq b_0 + 2b_1 + 4b_2 + 8b_3 \\
& \backslash\backslash\ \mathsf{wpre}_{\geq 6} = g = [r \geq 8]
\end{aligned}
$$

where $f_3 = [s_2 \geq 8]$ *and* $g_3 = [s_3 \geq 8]$. *Observe that since* $s_3 \geq s_2$, *it holds that* $f_3 \Rrightarrow g_3$ *and in view of Theorem 4.2, we can remove the left branch* ($b_3 \coloneqq 0$) *of the probabilistic choice initializing* b_3.

Furthermore, appealing to Theorem 4.1 we can remove instructions i_1 *through* i_3 *because* $\mathsf{wpre}_{\geq 1} \Rrightarrow \mathsf{wpre}_{\geq 4}$. *To see why, observe that*

$$
\begin{aligned}
\mathsf{wpre}_{\geq 1} &= \sum_{j\in\{0,\ldots,15\}} \tfrac{1}{16}\,[j \geq 8] \;=\; \tfrac{1}{16}\cdot 8 \;=\; \tfrac{1}{2} \\
\mathsf{wpre}_{\geq 4} &= \sum_{j\in\{0,8\}} \tfrac{1}{2}\,[s_2 + j \geq 8] \;=\; \tfrac{1}{2}\underbrace{[s_2 + 0 \geq 8]}_{=0} + \tfrac{1}{2}\underbrace{[s_2 + 8 \geq 8]}_{=1} \;=\; \tfrac{1}{2}
\end{aligned}
$$

In summary, we obtain the following program slice:

$$
\begin{aligned}
& \{\mathsf{skip}\}\ [1/2]\ \{b_3 \coloneqq 1\}; \\
& r \coloneqq b_0 + 2b_1 + 4b_2 + 8b_3
\end{aligned}
$$

Observe that slicing the program to preserve the value of variable of r *(as allowed by existing techniques) would be futile because there does not exist any proper such slice.* △

4.3. Summary

In this chapter we have presented the two fundamental results of this work , which allow identifying removable program fragments and underlie our slicing approach.

- The first slicing technique allows removing top-level instructions from programs.
- The second slicing technique allows removing nested instructions. For example, instructions in the body of a loop or in either branch of a conditional. To achieve this, it is necessary to exploit the notion of *local specification*, which , loosely speaking, represents the specification induced at a local level by a global specification at the top level of a program.

We prove that both techniques are correct for partial correctness specifications (Theorem 4.1 and 4.2) and show application examples (Examples 4.1 and 4.2). This pair of results form the cornerstone of our theoretical contribution.

Chapter 5

Total Correctness

In this chapter we adapt the slicing approach developed in the previous chapter to preserve the *total*—rather than *partial*—correctness of programs. In Sections 5.1-5.3 we develop the prerequisites for the adaptation and in Section 5.4 we present the two fundamental results for program slicing based on total correctness specifications (Theorems 5.1 and 5.2).

The slicing techniques developed in the previous chapter concern the *partial* correctness of programs: they guarantee that if a program satisfies a partial correctness specification, then so do the slices provided by Theorems 4.1 and 4.2. In other words, they aim at preserving (lower bounds for) the probability that the resulting program slices either terminate establishing the post-condition, or diverge. However, if the program at hand satisfies a given *total* correctness specification where preconditions refer to (lower bounds for) the probability of terminating *and* establishing the post-condition, we will certainly be interested in preserving the total correctness for program slices, too.

This is particularly desirable because when considering partial correctness, programs with loops admit trivial slices where loop bodies are simply removed. To see why, let us consider a program containing e.g. loop $\mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c\}$, together with its specification. From Figure 4.2, the local specification induced on the loop body c has pre-expectation $[G] \cdot inv$ and post-expectation inv. Thus, a trivial portion of the loop body c that preserves the local specification is skip, that is, $\{[G] \cdot inv\}\ \mathsf{skip} \preccurlyeq c\ \{inv\}^{\circlearrowleft}$. Therefore, in view of Theorem 4.2, removing the loop body c from the original program yields a valid slice thereof.

This may raise doubts about the value of slicing based on partial correctness specifications, as developed in the previous chapter. However, this type of slicing turns out very useful at the practical level for two reasons. First, it allows concluding that a program slice never —or only with *low probability*— terminates with an *incorrect* result (which is different from always —or with *high probability*— terminating with a *correct* result). Second, it allows for better "separation of concerns" and "tool synergy": one could slice a program w.r.t. its partial correctness specification using the results from the previous chapter, and exploit any other approach at hand to prove its termination.

5.1. Probabilistic Termination

The *de facto* notion of termination for probabilistic programs is that of *almost-sure termination* (AST), that is, termination with probability 1. Roughly speaking, we say that a pWhile program is *almost-sure terminating* from an initial state s if the probabilities of all its finite executions sum up to 1. Note that this does not prohibit the presence of infinite executions, but instead requires them to have an overall null probability. For example, the program

$$c \coloneqq 1;\ \mathsf{while}\,(c = 1)\,\mathsf{do}\,\{\{c \coloneqq 1\}\,[1/2]\,\{c \coloneqq 0\}\}$$

that simulates a geometric distribution by flipping a fair coin until observing the first heads (represented by 0) is almost-sure terminating: For all $n \geq 1$, the loop terminates after n iterations with probability $(1/2)^n$, thus the set of all its finite executions has probability $\sum_{n\geq 1}(1/2)^n = 1$. Note that besides these finite executions, the program also admits an infinite execution where all coin flips return tails (represented by 1). However, as required for almost-sure termination, this execution has probability $\lim_{n\to\infty}(1/2)^n = 0$.

5.2. Proving Termination via Variants

The traditional approach for establishing the total correctness of a program, either deterministic or probabilistic, consists in combining partial correctness with a termination argument. For example, for the case of a probabilistic program c, if we know on the one hand that it satisfies the partial correctness specification $\{f\}\ c\ \{g\}^{\circlearrowleft}$ and on the other hand, that it terminates almost-surely from any state satisfying, say predicate T (for termination), then we can conclude that it satisfies the total correctness specification $\{[T]\cdot f\}\ c\ \{g\}^{\downarrow}$.

Since loops are the only possible source of divergence in our language, let us focus on termination arguments for loops. For deterministic programs, loop termination is established through the presence of a so-called variant. Informally, a loop *variant* is an integer expression that decreases in each loop iteration and cannot decrease infinitely many times without before leaving the loop. For a VCGen for deterministic programs only, this would require adapting the verification conditions generated by loops from

$$\begin{aligned}\mathsf{vc}\,[\mathsf{while}\,(G)\ [I]\ \mathsf{do}\,\{c\}]\,(Q) \;=\; &\{G \wedge I \Rightarrow \mathsf{wpre}\,[c]\,(I)\ ,\\ &\neg G \wedge I \Rightarrow Q\}\ \cup\\ &\mathsf{vc}\,[c]\,(I)\end{aligned}$$

to

$$\begin{aligned}\mathsf{vc}^{\downarrow}\,[\mathsf{while}\,(G)\ [I, v, \mathtt{l}]\ \mathsf{do}\,\{c\}]\,(Q) \;=\; &\{G \wedge I \wedge v = v_0 \Rightarrow \mathsf{wpre}\,[c]\,(I \wedge v < v_0)\ ,\\ &G \wedge I \Rightarrow v \geq \mathtt{l}\,,\\ &\neg G \wedge I \Rightarrow Q\}\ \cup\\ &\mathsf{vc}^{\downarrow}\,[c]\,(I \wedge v < v_0)\end{aligned} \tag{5.1}$$

where v denotes the loop variant, $\mathtt{l}$ a lower bound thereof established by the loop invariant

and guard, and v_0 a fresh logical variable [17].[4]

McIver and Morgan [20, Lemma 7.5.1] showed how to generalize this variant-based termination argument to probabilistic loops. However, the generalization deviates from the argument for deterministic programs in two aspects. First, it does not require that the variant decreases with probability 1 in each loop iteration, but only with a fixed positive probability $\epsilon > 0$. Second, besides being bounded from below, the loop variant must be bounded also from above.

Even though adapting Equation 5.1 to the probabilistic case by accounting for these deviations is rather straightforward, another change is also necessary. To see why, observe that the role of invariant I in Equation 5.1 is twofold: on the one hand, to establish the desired partial correctness of the loop (in particular, post-condition Q) and, on the other hand, to encode a set of states from which the loop is guaranteed to terminate (recall that "partial correctness plus termination implies total correctness"). However, for the case of probabilistic programs, these two roles must be decoupled because the invariant required to establish the partial correctness of the loop might be itself probabilistic, i.e. a *proper* expectation, while the almost-sure termination of the loop remains encoded by a set of states, that is, a *predicate* over states.

To reason about slices that preserve the total correctness of probabilistic programs, we thus annotate loops as

$$\mathsf{while}\,(G)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c\}\ ,$$

where inv is an expectation representing the loop invariant (like for the case of partial correctness), T is a predicate representing the sets of states from which the loop terminates almost surely, v is an integer-valued function over program states representing the loop variant, $\mathtt{l}$ and $\mathtt{u}$ are integers representing a lower and upper bound for the variant, respectively, and ϵ is a probability in the interval $(0, 1]$ with which the variant is guaranteed to decrease in each iteration.

5.3. Verification Condition Generator

In view of the above discussion, to define the VCGen for total correctness specifications we adapt transformers wpre and vc as follows, where for convenience we display adaptations in red:

$$\mathsf{wpre}^{\downarrow}\,[\mathsf{while}\,(G)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c\}]\,(g)\ =\ [T]\cdot inv \tag{5.2}$$

[4] Without lost of generality, $\mathtt{l}$ can be considered to be 0. We prefer to leave it as an additional parameter in order to avoid (the otherwise required) adaptations of the variant v.

$$
\begin{aligned}
&\mathsf{vc}^{\downarrow}\,[\mathsf{while}\,(G)\ [\mathit{inv}, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c\}]\,(g) \qquad (5.3)\\
&\quad = \Big\{[G \wedge T] \ \Rrightarrow\ \mathsf{wpre}^{\downarrow}\,[c]\,([T])\ ,\\
&\qquad \epsilon\,[G \wedge T \wedge v = v_0] \ \Rrightarrow\ \mathsf{wpre}^{\downarrow}\,[c]\,([v < v_0])\ ,\\
&\qquad [G \wedge T] \ \Rrightarrow\ [\mathtt{l} \leq v \leq \mathtt{u}]\ \} \cup\\
&\qquad \mathsf{vc}^{\downarrow}\,[c]\,([T]) \cup\\
&\qquad \mathsf{vc}^{\downarrow}\,[c]\,([v < v_0]) \cup\\
&\qquad \{[G] \cdot \mathit{inv} \ \Rrightarrow\ \mathsf{wpre}\,[c]\,(\mathit{inv})\ ,\\
&\qquad [\neg G] \cdot \mathit{inv} \ \Rrightarrow\ g\ \} \cup\\
&\qquad \mathsf{vc}\,[c]\,(\mathit{inv})
\end{aligned}
$$

For the remaining language constructs, $\mathsf{wpre}^{\downarrow}$ and $\mathsf{vc}^{\downarrow}$ follow the same rules as their respective counterparts for partial correctness wpre and vc (see Figures 3.1 and 3.2), $\mathsf{vc}^{\downarrow}$ making use of $\mathsf{wpre}^{\downarrow}$ instead of wpre.

In Equation 5.3, the validity of the (added) verification conditions in red entails that the loop terminates almost surely from T (and that T is a standard, i.e. non-probabilistic, loop invariant) [20, Lemma 7.5.1]. The validity of the remaining verification conditions (as generated also by vc) entails that *inv* is a valid *partial correctness* invariant, strong enough as to establish post-expectation g. Combining these two results, we can conclude that the loop satisfies the *total correctness* specification given by pre-expectation $[T] \cdot \mathit{inv}$ (as reflected by Equation 5.2) and post-expectation g [20, Lemma 2.4.1-Case 2].

With these adaptations in place, we can readily define the VCGen for total correctness, mimicking the definition of the VCGen for total correctness:

Definition 5.1 (VCGen for total correctness) *The set of verification conditions* $\mathrm{VCG}^{\downarrow}\,[c]\,(f, g)$ *for the total correctness of a* pWhile *program* c *w.r.t. pre-expectation* f *and post-expectation* g *is defined as follows:*

$$
\mathrm{VCG}^{\downarrow}\,[c]\,(f, g) \quad = \quad \{f \Rrightarrow \mathsf{wpre}^{\downarrow}\,[c]\,(g)\} \cup \mathsf{vc}^{\downarrow}\,[c]\,(g)\ .
$$

Having introduced $\mathrm{VCG}^{\downarrow}$, we can restate the definition of $\mathsf{vc}^{\downarrow}$ more succinctly:

$$
\begin{aligned}
&\mathsf{vc}^{\downarrow}\,[\mathsf{while}\,(G)\ [\mathit{inv}, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c\}]\,(g) \qquad (5.4)\\
&\quad = \mathrm{VCG}^{\downarrow}\,[c]\,([G \wedge T]\,, [T]) \cup\\
&\qquad \mathrm{VCG}^{\downarrow}\,[c]\,(\epsilon\,[G \wedge T \wedge v = v_0]\,, [v < v_0]) \cup\\
&\qquad \{[G \wedge T] \ \Rrightarrow\ [\mathtt{l} \leq v \leq \mathtt{u}]\} \cup\\
&\qquad \mathrm{VCG}^{\downarrow}\,[c]\,([G] \cdot \mathit{inv}, \mathit{inv}) \cup\\
&\qquad \{[\neg G] \cdot \mathit{inv} \ \Rrightarrow\ g\}
\end{aligned}
$$

The VCGen for total correctness obeys the same properties of soundness and monotonicity as the VCGen for partial correctness.

Lemma 5.1 (Soundness of $\mathrm{VCG}^{\downarrow}$) *For any* pWhile *program* c *and any two expectations* $f, g \colon \mathbb{E}$,

$$\models \mathrm{VCG}^{\downarrow}[c](f,g) \quad \Longrightarrow \quad \models \{f\}\ c\ \{g\}^{\downarrow}\ .$$

Lemma 5.2 (Monotonicity of $\mathsf{vc}^{\downarrow}/\mathrm{VCG}^{\downarrow}$) *For any* pWhile *program* c *and any four expectations* $f, f', g, g' : \mathbb{E}$,

$$\begin{aligned} g \Rrightarrow g' \quad &\Longrightarrow \quad \models \mathsf{vc}^{\downarrow}[c](g) \ \Rightarrow \models \mathsf{vc}^{\downarrow}[c](g')\ , \\ f' \Rrightarrow f \ \wedge\ g \Rrightarrow g' \quad &\Longrightarrow \quad \models \mathrm{VCG}^{\downarrow}[c](f,g) \ \Rightarrow \models \mathrm{VCG}^{\downarrow}[c](f',g')\ . \end{aligned}$$

As for the alternative characterization of $\mathrm{VCG}^{\downarrow}$, the case of loops requires the adaptations displayed in red.

Lemma 5.3 (Alt. characterization of $\models \mathrm{VCG}^{\downarrow}$) *For any* pWhile *program* $c = i_1; \ldots; i_n$ *and any two expectations* $f, g : \mathbb{E}$,

1. *If* $i_j = \mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}$ *or* $i_j = \{c_1\}\,[p]\,\{c_2\}$ *for some* $1 \le j \le n$, *then*

$$\begin{aligned} \models \mathrm{VCG}^{\downarrow}[c](f,g) \qquad \textit{iff} \qquad & \models \mathsf{vc}^{\downarrow}_{\geq j+1}[c](g) \ \wedge \\ & \models \mathsf{vc}^{\downarrow}[c_1]\left(\mathsf{wpre}^{\downarrow}_{\geq j+1}[c](g)\right) \ \wedge \\ & \models \mathsf{vc}^{\downarrow}[c_2]\left(\mathsf{wpre}^{\downarrow}_{\geq j+1}[c](g)\right) \ \wedge \\ & \models \mathrm{VCG}^{\downarrow}_{\leq j-1}[c]\left(f, \mathsf{wpre}^{\downarrow}_{\geq j}[c](g)\right)\ . \end{aligned}$$

2. *If* $i_j = \mathsf{while}\,(G)\ [\mathit{inv}, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c'\}$ *for some* $1 \le j \le n$, *then*

$$\begin{aligned} \models \mathrm{VCG}^{\downarrow}[c](f,g) \qquad \textit{iff} \qquad & \models \mathrm{VCG}^{\downarrow}_{\geq j+1}[c]([\neg G]\cdot \mathit{inv}, g) \ \wedge \\ & \models \mathrm{VCG}\,[c']([G]\cdot \mathit{inv}, \mathit{inv}) \ \wedge \\ & \models \mathrm{VCG}^{\downarrow}_{\leq j-1}[c](f, [T]\cdot \mathit{inv}) \ \wedge \\ & \models \mathrm{VCG}^{\downarrow}[c']([G \wedge T], [T]) \ \wedge \\ & \models \mathrm{VCG}^{\downarrow}[c'](\epsilon\,[G \wedge T \wedge v = v_0], [v < v_0]) \ \wedge \\ & \models \{[G \wedge T] \ \Rrightarrow\ [\mathtt{l} \le v \le \mathtt{u}]\} \end{aligned}$$

5.4. Removing instructions

Armed with VCGen for total correctness, we can readily adapt the notion of specification-based slice to preserve total correctness properties:

Definition 5.2 (Program slicing based on total correctness specifications [17]) *We say that* pWhile *program* c' *is a* specification-based slice *of* pWhile *program* c *with respect to the total*

correctness specification given by pre-expectation f and post-expectation g, written $\{f\}\ c' \preccurlyeq c\ \{g\}^{\downarrow}$, iff

1. $c' \preccurlyeq c$, *and*
2. $\models \mathrm{VCG}^{\downarrow}[c](f,g) \ \Rightarrow \models \mathrm{VCG}^{\downarrow}[c'](f,g)$

The slicing criteria from previous chapter carry over to the case of total correctness.

Theorem 5.1 (Removing top-level instructions for total correctness) *Let $c = i_1; \ldots; i_n$ be a* pWhile *program together with its respective pre- and post-expectation f and g. Moreover, let $1 \leq j \leq k \leq n$. If*

$$\mathsf{wpre}^{\downarrow}_{\geq j}[c](g) \ \Rrightarrow\ \mathsf{wpre}^{\downarrow}_{\geq k+1}[c](g)$$

then,

$$\{f\}\ \mathsf{remove}(j,k,c) \preccurlyeq c\ \{g\}^{\downarrow}\ .$$

The criterion for removing nested instructions requires adapting the notion of local specification induced by loops. To see why, observe that given a program containing a loop, to generate valid slices of the program from slices of the loop body, the latter must preserve not only the invariant (inv), but also the termination predicate (T) and the probability of variant decrement (ϵ). To account for these simultaneous requirements, the *total local specification* relation associates total correctness specifications to *sets* of specifications rather than single specifications. We write

$$c \vdash \langle f,g\rangle^{\downarrow} \rightsquigarrow\ c' \vdash \{\langle f_1,g_1\rangle, \ldots, \langle f_n,g_n\rangle\}$$

to denote that the total correctness specification $\langle f,g\rangle^{\downarrow}$ of c induces the local specifications $\langle f_1,g_1\rangle, \ldots, \langle f_n,g_n\rangle$ over subprogram c'. Each local specification $\langle f_i,g_i\rangle$ can refer to either partial ($\langle f_i,g_i\rangle^{\circlearrowleft}$) or total ($\langle f_i,g_i\rangle^{\downarrow}$) correctness.

The rule for loops now reads:

$$\frac{i_j = \mathsf{while}(G)\ [inv,T,v,\mathtt{l},\mathtt{u},\epsilon]\ \mathsf{do}\{c'\}}{c \vdash \langle f,g\rangle^{\downarrow} \rightsquigarrow\ c' \vdash \left\{\begin{array}{l}\langle [G]\cdot inv, inv\rangle^{\circlearrowleft},\ \langle [G\wedge T],[T]\rangle^{\downarrow},\\ \langle \epsilon\ [G\wedge T\wedge v=v_0],[v<v_0]\rangle^{\downarrow}\end{array}\right\}}\ [\rightsquigarrow \mathsf{while}^{\downarrow}]$$

The rule encoding transitivity also needs to be adjusted to account for the multiplicity of induced local specifications:

$$\frac{\begin{array}{c}c \vdash \langle f,g\rangle^{\downarrow} \rightsquigarrow\ c' \vdash \{\langle f_i,g_i\rangle\}_{i=1,\ldots,n}\\ c' \vdash \langle f_i,g_i\rangle \rightsquigarrow\ c'' \vdash \{\langle f_{i,j},g_{i,j}\rangle\}_{j=1,\ldots,m_i} \quad \forall i=1,\ldots,n\end{array}}{c \vdash \langle f,g\rangle^{\downarrow} \rightsquigarrow\ c'' \vdash \{\langle f_{i,j},g_{i,j}\rangle\}_{\substack{i=1,\ldots,n\\ j=1,\ldots,m_i}}}\ [\rightsquigarrow \mathtt{trans}^{\downarrow}]$$

The remaining rules mirror their counterpart from Figure 4.2.

Theorem 5.2 (Removing nested instructions for total correctness) *Let c be a* pWhile *program*

together with its respective pre- and post-expectation f and g, and let c' be a subprogram of c such that $c \vdash \langle f, g\rangle^{\downarrow} \rightsquigarrow c' \vdash \{\langle f_i, g_i\rangle\}_{i=1,\ldots,n}$. If c'' is a portion of c' ($c'' \preccurlyeq c'$) such that for all $i = 1, \ldots, n$,

$$\{f_i\}\ c''\ \preccurlyeq c'\ \{g_i\}\ ,$$

then

$$\{f\}\ c\,[c'/c'']\ \preccurlyeq c\ \{g\}^{\downarrow}\ ,$$

where $c\,[c'/c'']$ denotes the program that is obtained from c by replacing c' with c''.

In the theorem statement, the correctness type (partial or total) of each premise $\{f_i\}\ c''\ \preccurlyeq c'\ \{g_i\}$ is inherited by the type of the local specification $\langle f_i, g_i\rangle$ induced over c'. For example, if for some i, the induced local specification refers to total correctness ($\langle f_i, g_i\rangle^{\downarrow}$), then the corresponding premise refers accordingly to total correctness ($\{f_i\}\ c''\ \preccurlyeq c'\ \{g_i\}^{\downarrow}$).

We next illustrate the application of Theorems 5.1 and 5.2 to reason about program slicing that preserve total correctness specifications.

Example 5.1 *Alice and Bob repeatedly flip each a fair coin until observing a matching outcome, either both heads or both tails. However, Alice decides to "trick" Bob and switches the outcome of her coin, before comparing it to Bob's. The game can be encoded by program*

$$\begin{array}{l}
\backslash\backslash\ f = \frac{1}{2^K}\,[K > 0] \\
n \coloneqq 0; \\
a, b \coloneqq 0, 1; \\
\mathsf{while}\ (a \neq b)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do} \\
\quad n \coloneqq n + 1; \\
\quad \{a \coloneqq 0\}\ [1/2]\ \{a \coloneqq 1\}; \\
\quad a \coloneqq 1 - a; \\
\quad \{b \coloneqq 0\}\ [1/2]\ \{b \coloneqq 1\} \\
\backslash\backslash\ g = [n = K]
\end{array}$$

The program is instrumented with a variable n that tracks the required number of rounds until observing the first match. The program terminates after K loop iterations with probability $1/2^K$ provided $K > 0$ and with probability 0 otherwise, satisfying the annotated specification. The specification refers to total correctness. Indeed, we can prove that the loop terminates almost-surely from any initial state, exploiting the fact that $[a \neq b]$ is a loop variant, bounded by 0 and 1, which decrements with probability $1/2$ in each iteration. Together with the invariant required to establish the specification, these correspond to the following loop annotations:

$$\begin{array}{lcl}
inv & = & [a \neq b]\,[n < K]\,2^{n-K} + [a = b]\,[n = K] \\
T & = & \mathsf{true} \\
v & = & [a \neq b] \\
\mathtt{l} & = & 0 \\
\mathtt{u} & = & 1 \\
\epsilon & = & 1/2
\end{array}$$

By appealing to Theorem 5.2, we show that we can remove the assignment $a \coloneqq 1 - a$ from the loop body, while preserving the program total correctness specification. Said otherwise, switching the outcome of Alice coin has no effect on the number of rounds required until observing the first match. Formally, if we call c' the original loop body and c'' the slice of c' obtained by removing the assignment, the application of Theorem 5.2 requires proving:

$$\{[a \neq b] \cdot inv\}\ c'' \preccurlyeq c'\ \{inv\}^{\circlearrowleft} \tag{5.5}$$

$$\{[a \neq b \wedge \mathsf{true}]\}\ c'' \preccurlyeq c'\ \{[\mathsf{true}]\}^{\downarrow} \tag{5.6}$$

$$\{\epsilon\ [a \neq b \wedge \mathsf{true} \wedge [a \neq b] = v_0]\}\ c'' \preccurlyeq c'\ \{[[a \neq b] < v_0]\}^{\downarrow} \tag{5.7}$$

To this end, we apply Theorem 4.1 (to (5.5)*) and Theorem 5.1 (to* (5.6) *and* (5.7)*), propagating the respective post-expectations backward along c', till traversing the assignment to be removed:*

$$\begin{array}{l}
\backslash\backslash\ \tfrac{1}{2}\,[a \neq 1]\,[n < K]\,2^{n-K} + \tfrac{1}{2}\,[a = 1]\,[n = K]\ + \\
\backslash\backslash\ \tfrac{1}{2}\,[a \neq 0]\,[n < K]\,2^{n-K} + \tfrac{1}{2}\,[a = 0]\,[n = K] \\
a \coloneqq 1 - a; \\
\backslash\backslash\ \tfrac{1}{2}\,[a \neq 0]\,[n < K]\,2^{n-K} + \tfrac{1}{2}\,[a = 0]\,[n = K]\ + \\
\backslash\backslash\ \tfrac{1}{2}\,[a \neq 1]\,[n < K]\,2^{n-K} + \tfrac{1}{2}\,[a = 1]\,[n = K] \\
\{b \coloneqq 0\}\ [1/2]\ \{b \coloneqq 1\} \\
\backslash\backslash\ inv
\end{array}$$

$\backslash\backslash\ [\mathsf{true}]$	$\backslash\backslash\ \tfrac{1}{2}\,[[a \neq 1] < v_0] + \tfrac{1}{2}\,[[a \neq 0] < v_0]$
$a \coloneqq 1 - a;$	$a \coloneqq 1 - a;$
$\backslash\backslash\ [\mathsf{true}]$	$\backslash\backslash\ \tfrac{1}{2}\,[[a \neq 0] < v_0] + \tfrac{1}{2}\,[[a \neq 1] < v_0]$
$\{b \coloneqq 0\}\ [1/2]\ \{b \coloneqq 1\}$	$\{b \coloneqq 0\}\ [1/2]\ \{b \coloneqq 1\}$
$\backslash\backslash\ [\mathsf{true}]$	$\backslash\backslash\ [[a \neq b] < v_0]$

In all three cases, we see that the expectation above the assignment entails (in fact coincides with) the expectation below it, allowing us to safely remove it while preserving the (overall) program specification.

Finally, observe that, similarly to the previous examples, applying an analysis based on data/control dependencies to produce a slice that preserves the value of n would be fruitless as the program admits no such proper slice. △

5.5. Summary

We have adapted the slicing approach developed in the previous chapter to preserve the *total*—rather than *partial*—correctness of programs. To achieve this, programs are not only annotated with invariants but also annotated with a termination witness. In order to stablish almost-sure termination, which indicates termination with probability 1, we leverage the almost-sure termination proof rule developed by McIver and Morgan.

Similarly to the preceding chapter, we develop two slicing techniques: one for removing top-level instructions and another for instructions in nested programs. We prove that both techniques are correct for total correctness specifications (Theorem 5.1 and 5.2), and provide an application example (Example 5.1).

Chapter 6

Case Study

We now showcase the applicability of our technique to the field of probabilistic modelling, in particular, aiding in model understanding and model simplification.

Since their introduction in the 80's, graphical models —in particular, Bayesian networks— have been the *de facto* formalism for encoding probabilistic models due to their accessibility and simplicity. For example, Figure 6.1 shows a Bayesian network by Lauritzen and Spiegelhalter [26] modeling (a quantitative version of) the following fictitious knowledge related to different lung diseases (tuberculosis, lung cancer and bronchitis) and factors (visit to Asia and smoking):

> Shortness-of-breath (dyspnoea) may be due to tuberculosis, lung cancer or bronchitis, or none of them, or more than one of them. A recent visit to Asia increases the chances of tuberculosis, while smoking is known to be a risk factor for both lung cancer and bronchitis. The results of a single chest X-ray do not discriminate between lung cancer and tuberculosis, as neither does the presence or absence of dyspnoea.

The network topology encodes the dependencies among the involved random variables. In particular, random variables can be distributed either independently of the reminding random variables, like a (visit to Asia) or conditionally on a subset of them, like d (dyspnea). The probability distribution of the random variables is specified by probability distribution tables (in Figure 6.1 depicted on the right of the node encoding the random variable). The network together with the probability distribution tables uniquely determines the joint distribution of all random variables.

With the emergence of probabilistic programming systems in the past years, probabilistic programs have become a more convenient formalism for encoding such probabilistic models for several reasons [27]. First, probabilistic programs provide additional abstractions not provided by Bayesian networks. For example, while Bayesian networks are inherently acyclic, probabilistic programs allow encoding recursive models. Also, probabilistic programs enable a more compositional approach to modelling due to the presence of functional abstractions. Second, modern probabilistic programming systems include state of the art inference algo-

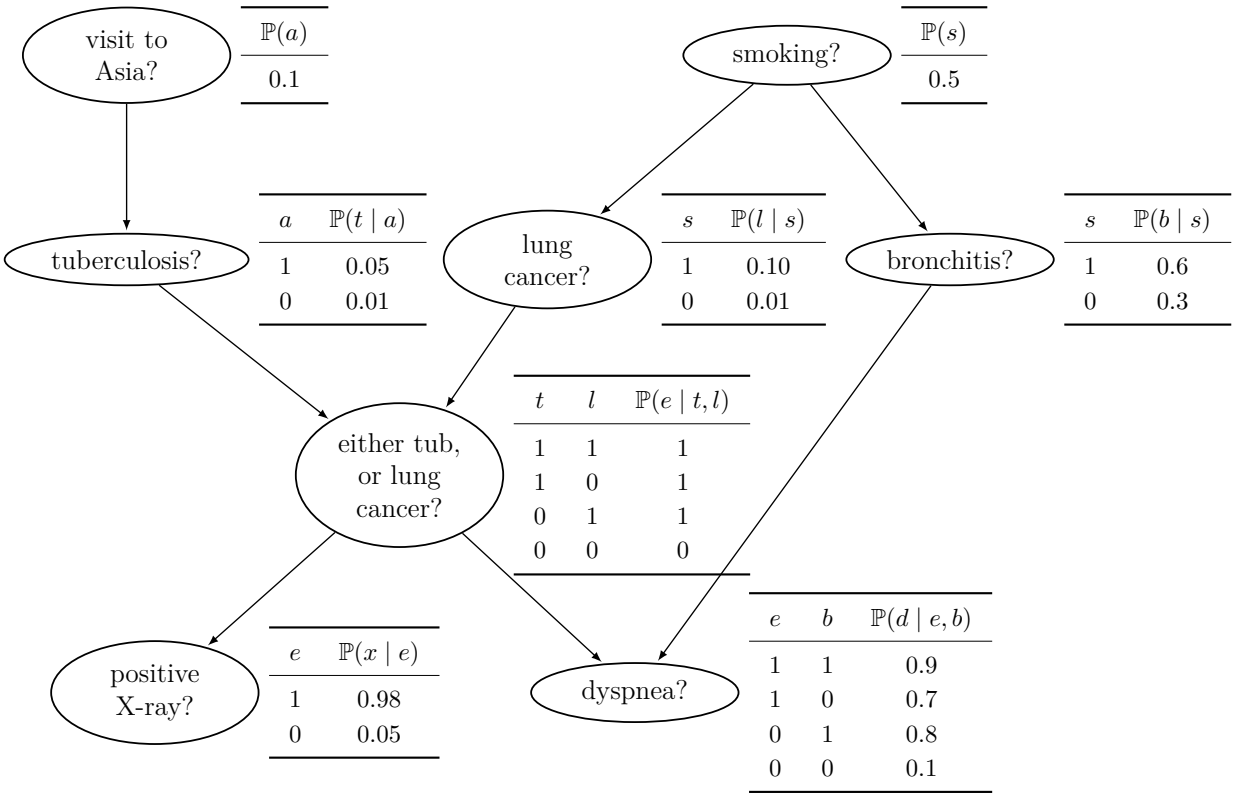

Figure 6.1: Bayesian network describing a fictional knowledge related to different lung diseases and factors [26].

rithms, which significantly improve inference time. Altogether, these features allow a faster and more convenient prototyping than Bayesian networks.

Through the use of traditional programming language abstractions, probabilistic programs can encode intricate models, involving multiple random variables which may be highly coupled. However, given such a complex model, we might be interested in a *partial view* thereof. For example, while the program in Figure 6.2 encodes the entire model described by the Bayesian network from Figure 6.1[5], we may be interested only in the probability that the X-ray of a patient turns out positive, e.g., because we want to compute this probability or because we want to identify the *fragment* of the model that defines this probability.

Applying the slicing technique from Chapter 5, we can conclude that the program fragments from Figure 6.2 colored in red are extraneous to this probability and can thus be sliced away of the program. Formally, the resulting program represents a slice of the original program w.r.t. post-expectation $[x = 1]$, and any pre-expectation. The detailed derivation can be found in Appendix B.

In a similar way, if we are interested only in the probability that a patient suffers from both tuberculosis and lung cancer at the same time, we can slice the program w.r.t. post-expectation $[t = 1 \wedge l = 1]$. This allows a more aggressive slicing, as depicted in Figure 6.3. The detailed derivation is also found in Appendix B.

[5] The translation of Bayesian networks into probabilistic programs is rather straightforward; for a formal description, see, e.g., [28].

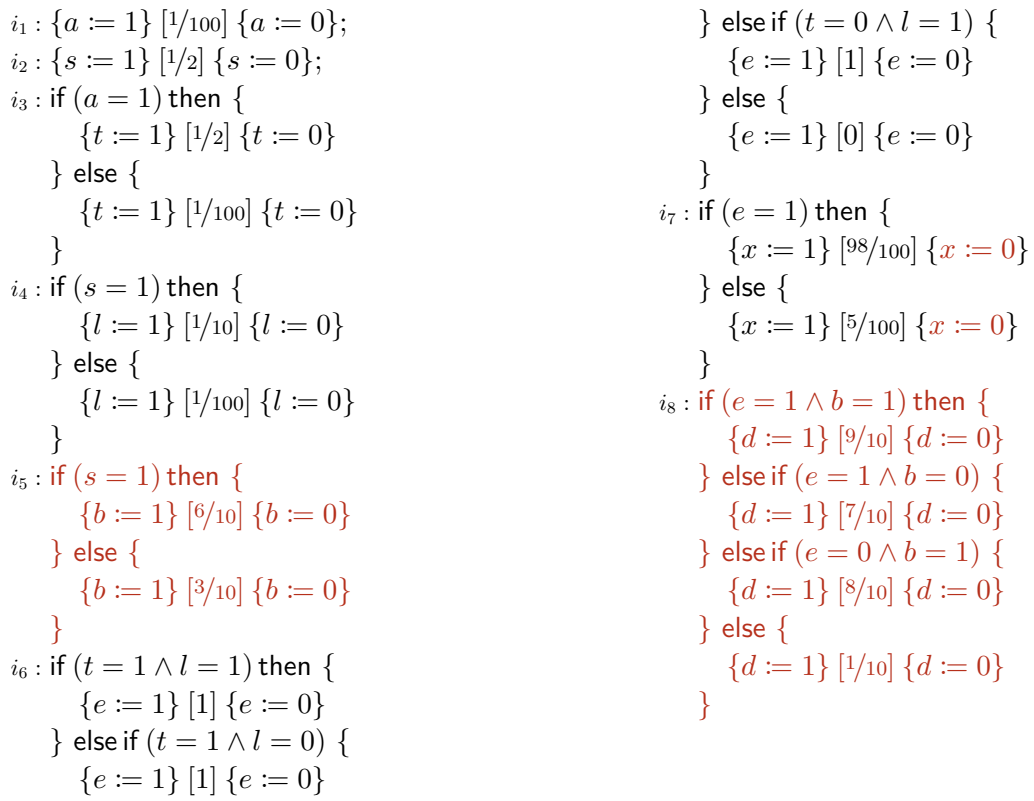

Figure 6.2: Probabilistic program describing a model that relates different lung diseases and factors. Code fragments in red can be sliced away when considering post-expectation $[x = 1]$.

Notoriously, in both cases we obtain more precise slices than the one yield by traditional slicing techniques based on data and control dependencies. Concretely, in the first case these techniques fail to identify assignment $x \coloneqq 0$ as a removable piece of code, and in the second case they fail to identify assignments $l \coloneqq 0$ and $b \coloneqq 0$ as such.

6.1. Summary

We have showcased the applicability of our technique in the field of probabilistic modelling. In particular, we consider a medical model associated with lung diseases (tuberculosis, lung cancer and bronchitis) and factors that are possibly related to these diseases (visit to Asia and smoking).

We apply the slicing techniques from the previous chapter twice, with two different post-conditions, the first case we managed to remove slightly more than 25% of the instructions and in the second case, with a more restrictive postcondition, we successfully eliminated over 50% of the instructions.

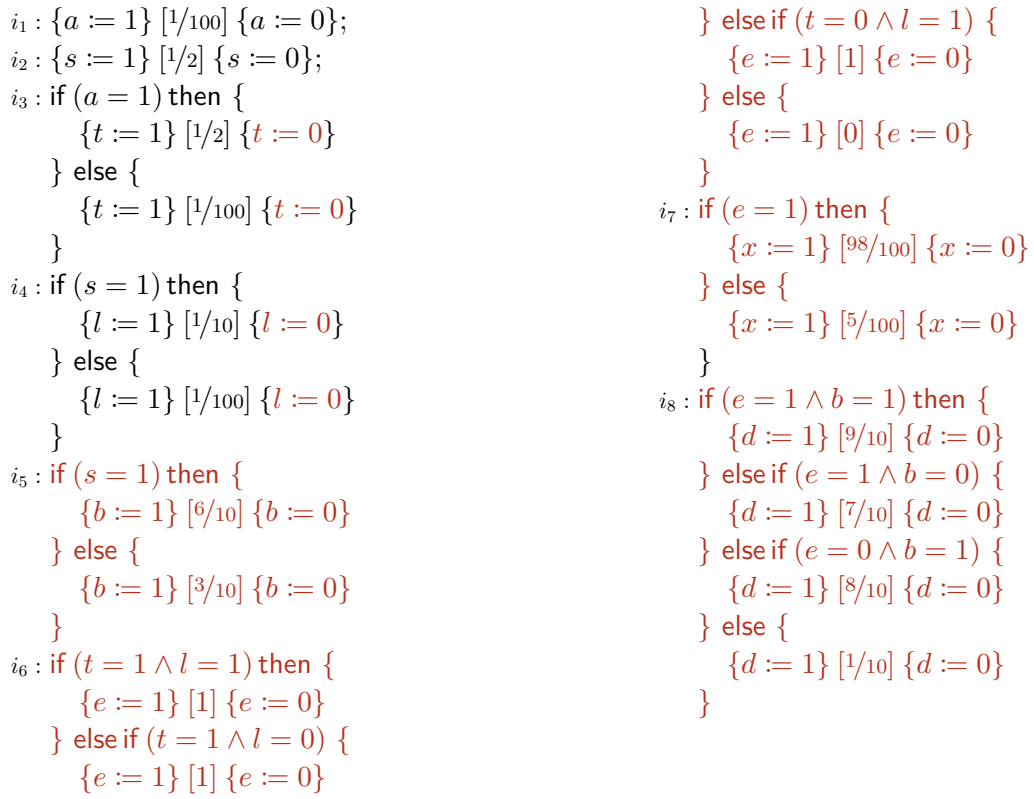

Figure 6.3: Probabilistic program describing a model that relates different lung diseases and factors. Code fragments in red can be sliced away when considering post-expectation $[t = 1 \wedge l = 1]$.

Chapter 7

Slicing Algorithm

In this chapter we present an algorithm for computing program slices. The algorithm is based on the construction of a *slice graph* (Definition 7.2) and returns the *least* slice (in a sense to be defined later) that can be derived by the application of Theorems 4.1/5.1 and 4.2/5.2. This algorithm provides a starting point for a (semi-)automated application of the slicing technique developed in Chapters 4 and 5.

Despite being a mild adaptation of the algorithm introduced by Barros et al. [17] for slicing deterministic programs—while we use only a backward propagation of post-conditions, Barros et al. combine backward propagation of post-conditions with forward propagation of pre-conditions (see Chapter 8 for a further discussion)—we prefer to include a full description of the adaptation here to make the presentation more self-contained.

The slice graph of a program is obtained by extending its control flow graph first with semantic information (assertions) in the labels, yielding an intermediate *labelled control flow graph*, and then with additional edges that "short-circuit" removable instructions. Determining the least program slice is then cast as a (generalization of a) weighted shortest path problem on the slice graph.

For convenience, we develop the algorithm for computing program slices that preserve partial correctness, and then discuss the necessary adaptations for total correctness.

Intuitively, the labelled control flow graph of a program $c = i_1; \ldots; i_n$ with respect to a post-expectation g associates to each edge (i_j, i_{j+1}) the expectation that is obtained by propagating (via transformer wpre) the post-expectation g backward, till traversing i_{j+1}. For example, the labelled control flow graph of the program from Example 4.2 w.r.t. post-expectation $[r \geq 8]$ is depicted in Figure 7.1 (thick edges are not part of the labelled control flow graph, but of the slice graph).

Formally, it is defined as follows:

Definition 7.1 (Labelled control flow graph; adapted from [17]) *Given a* pWhile *program* $c = i_1; \ldots; i_n$ *and a (post-) expectation* $g \colon \mathbb{E}$*, the* labelled control flow graph $\mathsf{LCFG}\,[c]\,(g)$ *is a directed acyclic graph, whose edges are labelled with expectations. To construct it, we make use of the auxiliary functions* $\mathit{in}\,(i_j)$ *and* $\mathit{out}\,(i_j)$ *that associate each instruction* i_j *of* c *with a*

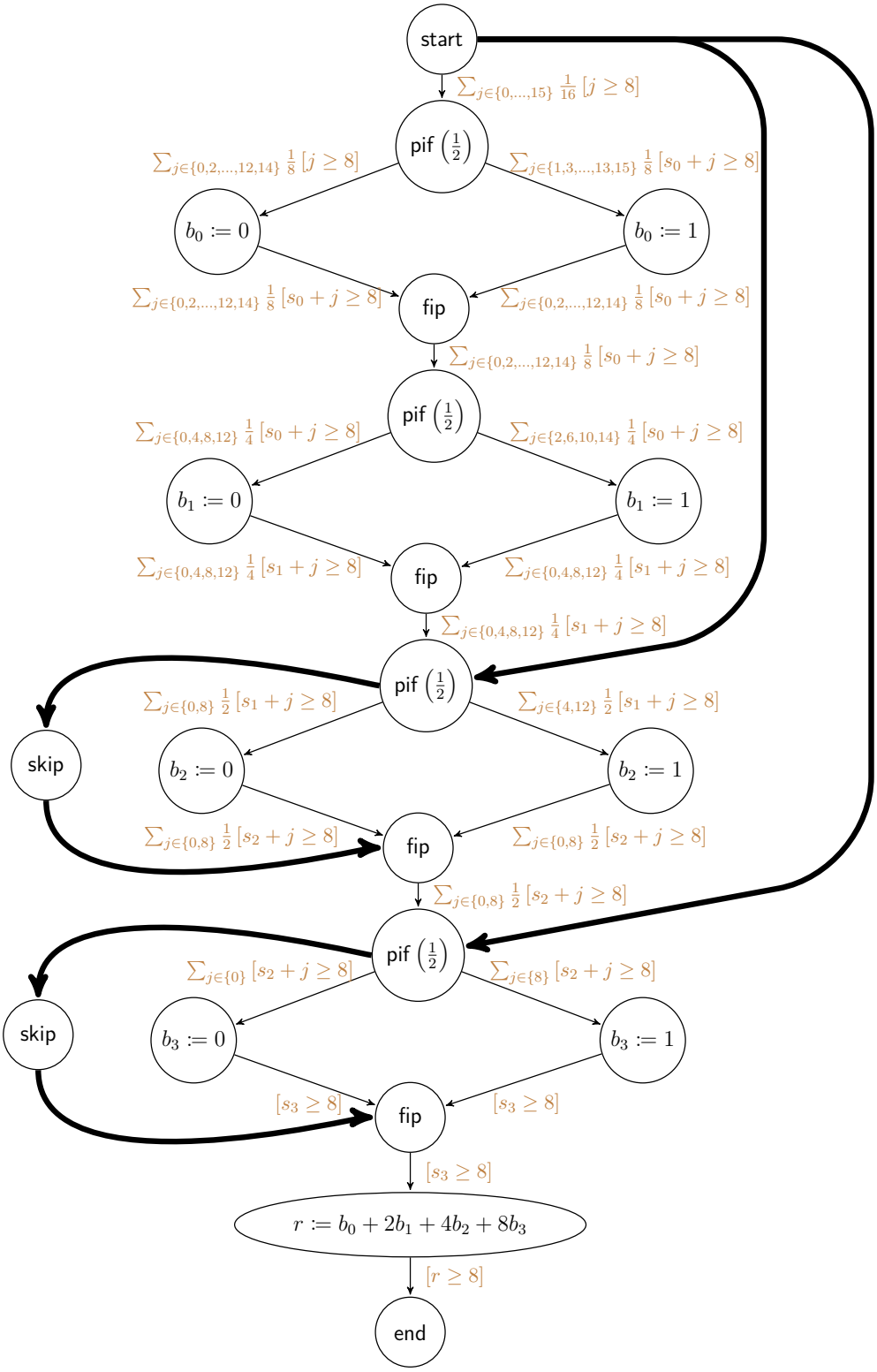

Figure 7.1: Excerpt of the slice graph associated to the program from Example 4.2.

respective input *and* output node *in* $\mathsf{LCFG}\,[c]\,(g)$. *The graph* $\mathsf{LCFG}\,[c]\,(g)$ *is then constructed as follows:*

1. *Each instruction i_j induces one (*skip *or assignments) or two (conditional branches, probabilistic choices or loops) nodes in* $\mathsf{LCFG}\,[c]\,(g)$*:*

   - *If* $i_j = \mathsf{skip}$ *or* $i_j = x \coloneqq E$*, then* i_j *is a node of* $\mathsf{LCFG}\,[c]\,(g)$*. Moreover, we define* $in\,(i_j) = out\,(i_j) = i_j$*.*
   - *If* $i_j = \mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}$*, then* $\mathsf{bif}\,(G)$ *and* fib *are nodes of* $\mathsf{LCFG}\,[c]\,(g)$*. Moreover, we define* $in\,(i_j) = \mathsf{bif}\,(G)$ *and* $out\,(i_j) = \mathsf{fib}$*.*
   - *If* $i_j = \{c_1\}\,[p]\,\{c_2\}$*, then* $\mathsf{pif}\,(p)$ *and* fip *are nodes of* $\mathsf{LCFG}\,[c]\,(g)$*. Moreover, we define* $in\,(i_j) = \mathsf{pif}\,(p)$ *and* $out\,(i_j) = \mathsf{fip}$*.*
   - *If* $i_j = \mathsf{while}\,(G)\,[inv]\,\mathsf{do}\,\{c'\}$*, then* $\mathsf{do}\,(G)$ *and* od *are nodes of* $\mathsf{LCFG}\,[c]\,(g)$*. Moreover, we define* $in\,(i_j) = \mathsf{do}\,(G)$ *and* $out\,(i_j) = \mathsf{od}$*.*

2. start *and* end *are two distinguished nodes of* $\mathsf{LCFG}\,[c]\,(g)$*.*

3. $(\mathsf{start}, in\,(i_1))$*,* $(in\,(i_j)\,, out\,(i_{j+1}))$ *and* $(out\,(i_n)\,, \mathsf{end})$ *are edges of* $\mathsf{LCFG}\,[c]\,(g)$ *for each* $j = 1, \ldots, n-1$*. The labels of these edges are defined as follows:*

$$\begin{array}{lcll} label\,(\mathsf{start}, in\,(i_1)) & = & \mathsf{wpre}_{\geq 1}\,[c]\,(g) & \\ label\,(out\,(i_j)\,, in\,(i_{j+1})) & = & \mathsf{wpre}_{\geq j+1}\,[c]\,(g) & \forall\, j = 1, \ldots, n-1 \\ label\,(out\,(i_n)\,, \mathsf{end}) & = & g & \end{array}$$

4. *If* $i_j = \mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}$ *or* $i_j = \{c_1\}\,[p]\,\{c_2\}$ *for some* $j = 1, \ldots, n$*, we recursively construct the labelled control flow graphs*

$$\mathsf{LCFG}\,[c_1]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \qquad \textit{and} \qquad \mathsf{LCFG}\,[c_2]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right)\ .$$

   This pair of graphs are incorporated into $\mathsf{LCFG}\,[c]\,(g)$ *by removing their* start *nodes and setting* $in\,(i_j)$ *as the origin of the dangling edges, and similarly removing their* end *nodes and setting* $out\,(i_j)$ *as the destination of the dangling edges.*

5. *If* $i_j = \mathsf{while}\,(G)\,[inv]\,\mathsf{do}\,\{c'\}$ *for some* $j = 1, \ldots, n$*, we recursively construct the labelled control flow graph*

$$\mathsf{LCFG}\,[c']\,(inv)\ .$$

   This graph is incorporated into $\mathsf{LCFG}\,[c]\,(g)$ *by removing its* start *node and setting* $in\,(i_j)$ *as the origin of the dangling edge, and similarly removing its* end *node and setting* $out\,(i_j)$ *as the destination of the dangling edge.*

Observe that the labelled control flow graph of a program can be constructed by first building the traditional control flow graph, and then traversing it backward to propagate the post-expectation.

The *slice graph* of a program is obtained by extending its labelled control flow graph with edges that short-circuit removable instructions, as identified by Theorems 4.1 and 4.2.

Definition 7.2 (Slice graph; adapted from [17]) *The* slice graph $\mathsf{SLG}\,[c]\,(f, g)$ *of a* pWhile *program* c *w.r.t. pre-expectation* f *and post-expectation* g *is obtained by extending its labelled*

control flow graph $\mathsf{LCFG}\,[c]\,(g)$ *with additional edges and* skip *nodes. Concretely, for each subprogram* $c' = i'_1, \ldots, i'_n$ *of* c *such that* $c \vdash \langle f, g\rangle^{\circlearrowleft} \rightsquigarrow c' \vdash \langle f', g'\rangle^{\circlearrowleft}$ *we proceed as follows:*

1. *If* $f' \Rrightarrow' g'$*, we add a new* skip *node, together with the pair of edges* $(in\,(c')\,, \mathsf{skip})$ *and* $(\mathsf{skip}, out\,(c'))$*;*
2. *For all* $j = 1, \ldots, n$*, if* $f' \Rrightarrow \mathsf{wpre}_{\geq j+1}\,[c']\,(g')$*, we add edge* $(in\,(c')\,, in\left(i'_{j+1}\right))$*;*
3. *For all* $j = 1, \ldots, n$*, if* $\mathsf{wpre}_{\geq j}\,[c']\,(g') \Rrightarrow g'$*, we add edge* $(out\left(i'_{j-1}\right), out\,(c'))$*;*
4. *For all* $j, k = 1, \ldots, n$ *such that* $j < k$*, if* $\mathsf{wpre}_{\geq j}\,[c']\,(g') \Rrightarrow \mathsf{wpre}_{\geq k}\,[c']\,(g')$*, we add edge* $(out\left(i'_{j-1}\right), in\,(i'_k))$*.*

Returning to the program from Example 4.2, the thick edges in Figure 7.1 represent a subset of the edges incorporated by the slice graph. An edge in the slice graph that is not depicted in the figure is, for example, the one short-circuiting the probabilistic choice assigning a value to b_0, only.

It is not hard to see that, by construction, all slices of a program that can be derived by (the repeated application of) Theorems 4.1 and 4.2 are represented in the slice graph. We are thus left to choose the *minimal* slice. In this regard, we define the size of a slice to be the number of atomic instructions in the subgraph representing the slice.

Slicing algorithm. For straight-line programs, i.e. programs free of conditional branches and probabilistic choices, the minimal slice can be computed by calculating the shortest path between the start and end vertices of the slice graph. However, for branching program, this will select a single branch. To address this problem, Barros et al. [17] suggests combining a weighted shortest path algorithm with graph rewriting as follows:

1. Assign weight 1 to every edge of the slice graph G.
2. For all branching instructions that do not contain any other branching instruction as subprogram,
   a) run a shortest path algorithm on each of the two branches and let $s = 1 + l + r$, where l and r are the lengths of the shortest paths of each of branch;
   b) replace the pair of branches with a single edge joining the origin $(\mathsf{bif}\,(G)\,/\mathsf{pif}\,(p))$ and the destination $(\mathsf{fib}/\mathsf{fip})$ of the branching instruction, and assign it weight s.
3. Go to step 2 if the resulting graph still contains any branching instruction with straight-line branches (observe that the step 2 above could have created new such instructions).

When applied to the slice graph of Figure 7.1, this algorithm can choose to keep the assignment of 0 to b_3 instead of replacing them with a skip (observe that this is consistent with our notion of slice with the least number of atomic instructions). However, this is not what one would expect in practice. To address this issue, in step 1 we can assign weight, e.g., $1/2$ (instead of 1) to all edges incident to skip vertices.

Finally, to compute slices that preserve the total correctness of programs, recall that programs containing while loops induce three local specifications on the loop body (number which can grow larger in the presence of nested loops). Therefore, the labels of control flow graphs must consist in tuples of expectations rather than single expectations. The entailment relation between expectations is naturally extended to tuples by taking the canonical lifting, e.g., $(f_1, f_2, f_3) \Rrightarrow (f_1', f_2', f_3')$ iff $f_1 \Rrightarrow f_1'$, $f_2 \Rrightarrow f_2'$ and $f_3 \Rrightarrow f_3'$. Observe that the labelling of the graph will not necessarily be uniform, since different subprograms may be labeled with tuples of different sizes.

7.1. Summary

In this chapter, we have presented an algorithm for computing program slices. This algorithm provides a starting point for a (semi-)automated application of the slicing technique developed in Chapters 4 and 5.

Given a program and a postcondition, the algorithm begins by creating a control flow graph of the program and then traverses it backward to propagate the post-condition, placing the postconditions of the propagation on the edges of the graph. The result is called a *labeled control flow graph*.

From the labeled control flow graph, we construct a *slice graph* by adding additional edges that short-circuit removable instructions, as defined in, as defined in Theorems 4.1/5.1.

Finally, we compute the minimal slice by applying a weighted shortest path algorithm to the slice graph between the start and end nodes. We provide an example of the result of this algorithm in Figure 7.1.

Chapter 8

Discussion

In this chapter we discuss some design decisions, extensions and limitations behind the developed slicing approach, pointing out some relevant directions of future work.

Termination of probabilistic programs.

The termination problem for probabilistic programs is significantly more challenging than for deterministic programs. For example, at the computational hardness level, while determining whether a deterministic program terminates on a given input is a semi-decidable problem (lying in the Σ_1^0-complete class of the arithmetical hierarchy), determining whether a probabilistic program almost-surely terminates on a given input is not (lying in the Π_2^0-complete class) [29, 30]. Moreover, deciding almost-sure termination of a probabilistic program *on a single input* is as hard as deciding termination of an ordinary (deterministic) program *on all inputs*.

Matching this intuition, while the variant-based termination argument for deterministic programs overviewed in Chapter 5 is complete, the probabilistic version by McIver and Morgan [20] internalized by our VCGen is not. For example, it is unable to establish the almost-sure termination of the program below, representing a 1-dimensional (one-side bounded) random walk:

$$\mathsf{while}\,(x \neq 0)\,\mathsf{do}\,\{\{x \coloneqq x-1\}\,[1/2]\,\{x \coloneqq x+1\}\}\ .$$

In a recent work [31], McIver et al. generalized the termination argument internalized by our VCGen, incrementing its expressivity so as to establish the termination, for example, of the above program. The new rule strengthens the original rule in three aspects: *i*) the variant need not be upper-bounded, *ii*) the variant may be real-valued, and *iii*) the variant decrement probability may vary across iterations.

Even though for the sake of presentation accessibility, in Chapter 5 we designed our VCGen for total correctness internalizing the original rule, the more recent version can be internalized following a similar approach. Nevertheless, note that even though the more recent rule is (strictly) more expressive than its original version, completeness remains an open problem.

An interesting direction for future research is investigating how our VCGen can internalize

other classes of termination argument, in particular, those based on the notion of supermartingales, as developed in several recent works [32–40].

Forward propagation of pre-conditions.

Specification-based slicing techniques require the combination of both backward propagation of post-conditions and forward propagation of pre-conditions to yield minimal slices [19, 41]. For example, consider program

$$\begin{aligned}\mathsf{if}\ (y > 0)\ \mathsf{then}\ \{x \coloneqq 100;\ x \coloneqq x + 50;\ x \coloneqq x - 100\}\\ \mathsf{else}\ \{x \coloneqq x - 150;\ x \coloneqq x - 100;\ x \coloneqq x + 100\}\ ,\end{aligned}$$

together with the specification given by pre-condition $y > 0$ and post-condition $x \geq 0$. The minimal slice that preserves the specification is:

$$\begin{aligned}\mathsf{if}\ (y > 0)\ \mathsf{then}\ \{x \coloneqq 100\}\\ \mathsf{else}\ \{\mathsf{skip}\}\end{aligned}$$

To obtain it, we can, for example, first propagate the pre-condition $y > 0$ forward (via the *strongest post-condition transformer* [21]), removing this way all the (dead) instructions in the else branch, and in the resulting program then propagate the post-condition $x \geq 0$ backward (via the *weakest pre-condition transformer* [21]) removing this way the last two instructions of the then branch. Any slicing based only on either kind of propagation will produce less precise slices.

In her PhD thesis [42], Jones proved that it is unfortunately not possible to define an analogue of the strongest post-condition transformer for expectations. To see why, consider program

$$\{x \coloneqq 0\}\ [1/2]\ \{x \coloneqq 1\}\ .$$

With respect to pre-expectation $1/2$, two valid post-expectations are $[x = 0]$ and $[x = 1]$. Thus, the strongest post-expectation must bound both $[x = 0]$ and $[x = 1]$ from below (recall Definition 2.1 of expectation entailment). The only common lower bound is the constantly null expectation [false], which is clearly not a valid post-expectation of the program.

An important line of future work is then to investigate the design of a slicing approach that allows both the propagation of pre-conditions forward and of post-conditions backward. A promising starting point here is to consider program logics that instead of representing pre- and post-conditions as real-valued functions over states (like expectations), represent them as Boolean predicates over state distributions. While many such logics already exist [43–46], to the best of our knowledge, none provides an analogue of a strongest post-condition transformer.

Efficiency vs precision tradeoff.

When designing our slicing technique, we privileged computation efficiency over slice precision. This is particularly reflected by Theorems 4.2 and 5.2, which embody a modular approach to slicing based on *local reasoning principles*: We can slice a program that contains, e.g., a probabilistic choice by slicing either of its branches, and this slicing requires

only the local specification induced on the branch—no other contextual information (like the local specification of the other branch) is required.

This design decision trades computational efficiency for slice precision. To illustrate this, consider the program below, together with its specification:

$$\{\tfrac{1}{2}\} \quad \{x \coloneqq 1\} \; [3/4] \; \{x \coloneqq x + 1\} \quad \{[x = 1]\}^{\circlearrowleft} \; .$$

The right branch of the probabilistic choice can be removed yielding a valid specification-preserving slice. However, Theorem 4.2 fails to identify it as a removable fragment: The local specification induced on the right branch is

$$\{[x = 0]\} \quad x \coloneqq x + 1 \quad \{[x = 1]\}^{\circlearrowleft} \; ,$$

and clearly, $[x = 0] \not\Rightarrow [x = 1]$. Intuitively, the problem is that the information available to slice the right branch (its local specification) does not account for the fact that the left branch can by itself already establish the post-condition.

To improve precision, the slicing approach should incorporate a mutual dependence analysis between the two branches, which might become highly expensive as nesting level increases. We leave as future work exploring a better compromise between efficiency and precision.

Chapter 9

Related Work

There is a vast body of work on program slicing; we refere the reader to [3] for an overview of different slicing techniques and to [2] for an overview of different applications. Here we will focus only on specification-based slicing, slicing approaches for probabilistic programs and VCGens for establishing probabilistic program specifications.

Specification-based slicing.

The notion of slicing with respect to a pre- and post-condition of programs, i.e. specification–based slicing, was introduced by Comuzzi and Hart [16]. Since then, the approach has been extended and refined by several authors [17, 19, 41, 47]. While the original approach of Comuzzi and Hart [16] uses a backward reasoning (i.e. weakest pre-conditions) for constructing slices, Lee et al. [19, 41] combine a backward with a forward reasoning (the latter through strongest postconditions), sequentially. Da Cruz et al. [47] extend specification–based slicing to a contract–based setting, where slicing is simultaneously performed over a set of procedures. Finally, Barros et al. [17] show that a *simultaneous* (rather than a sequential) combination of forward and backward reasoning is necessary (and sufficient) to deliver optimal slices. All of these approaches are restricted to deterministic programs. Our approach for probabilistic programs is along the lines of Barros et al. [17] approach, but restricted to backward reasoning, only, due to the limitations laid out in Chapter 8.

Slicing of probabilistic programs.

Hur et al. [13] were the first to explore the problem of slicing for probabilistic programs. They observed that the classical slicing approach based on data and control dependences becomes unsound for probabilistic programs with conditioning, and showed how to extended it with a new class of dependence to recover soundness. In a later work, Amtoft and Banerjee [14, 15] introduce the notion of *probabilistic control-flow graphs*, which allows a direct adaptation of conventional slicing machinery such as data dependence, postdominators and relevant variables to the case of probabilistic programs. Both the approaches by Hur et al. [13] and Amtoft and Banerjee [14, 15] perform a conventional slicing with respect to a set of (output) variables of interest.

VCGen for probabilistic programs.

Hur et al. [48] present a formalization of the wlp expectation transformers in the HOL4 proof assistant for pGCL, an extension of our language pWhile with (demonic) non-determinism and failure. They define a VCGen for establishing the partial correctness of programs annotated with loops invariants in the same line as our VCGen from Chapter 3. However, since the VCGen is implemented in a Prolog interpreter instead of in the same proof assistant, they are unable to provide a mechanized proof of the VCGen soundness. On the contrary, we provide VCGens for establishing both the total and partial correctness of programs, together with their respective (paper-and-pencil) soundness proofs.

Cock [49, 50] develops another formalization of the wlp expectation transformers for pGCL in the Isabelle/HOL proof assistant. In contrast to Hur et al. [48] who adopt a deep embedding, Cock [49] adopts a shallow embedding to take advantage of the proof assistant mechanization. He also implements a VCGen, but it is limited to loop-free programs.

In contrast to ours, Hur et al.'s [48] and Cock's [49] approaches, which are based on $[0, 1]$-valued assertions over states—expectations—Barthe et al. [46] present a Hoare logic based on Boolean assertions over state distributions. Even though they do not introduce a VCGen itself, they provide all the ingredients to do so: a weakest pre-condition transformer for non-looping programs and syntactic conditions for discharging the premises of (a subset of) the loop proof rules. However, the problem of assertion entailments in this logic (as required for slice computation) seems to be harder than that of expectation entailment. Finally, Chadha et al. [44] provide a decidable Hoare logic also based on Boolean assertions over state distributions. However, the logic is limited to straight-line programs, only.

Chapter 10

Conclusion

We have made significant strides by developing the first slicing approach for probabilistic programs, leveraging the power of specifications. The slicing approach is based on the backward propagation of post-conditions, exhibiting remarkable properties such as termination-sensitivity and modularity via local reasoning principles.

As an essential component of our methodology, we develop and prove sound VCGens for establishing both the partial and the total correctness of probabilistic programs. These VCGens have inherent value and can be leveraged in various contexts for different purposes, such as program verification. For example, they can be employed to rigorously analyze and validate the behavior and properties of the program.

By applying our approach to a set of examples, we have shown that the main benefit of specification-based program slicing—increased precision—carries over the class of probabilistic programs. This is particularly interesting due to the recent resurgence of probabilistic programing and its intrinsic complexity—any tool aiding program understanding becomes vital.

In this context, our approach provides software engineers with an additional tool to improve their understanding of probabilistic program behavior. It empowers them to effectively address critical code segments, enhance program comprehension, conduct refactoring tasks, eliminate redundant code, and unlock a multitude of other benefits. By leveraging our approach, software engineers can navigate the intricacies of probabilistic programs with greater confidence and make informed decisions that result in more reliable and efficient software systems.

Furthermore, We demonstrate that by adapting the slicing algorithm originally proposed by Barros et al. [17] and applying Theorems 4.1/5.1, it is possible to compute minimal slices of probabilistic programs within the space of slices generated by our technique.

Finally, our study has identified several directions for future work along with their potential associated challenges. One such direction involves enhancing slice precision by considering the incorporation of forward propagation of pre-conditions, which poses difficulties in defining a strongest postcondition. Alternatively, exploring approaches that deviate from the local reasoning principle underlying modular slicing can also present challenges. Moreover,

establishing a mutual dependence between the branches of a probabilistic choice can add complexity to the task.

Furthermore, an interesting research avenue is the extension of the language with conditioning, which is a crucial element in probabilistic modeling, and incorporating termination arguments based on martingales.

Bibliography

[1] Weiser, M., "Program slicing," en Proceedings of the 5th International Conference on Software Engineering, ICSE'81, p. 439–449, IEEE Press, 1981.

[2] Xu, B., Qian, J., Zhang, X., Wu, Z., y Chen, L., "A brief survey of program slicing," ACM SIGSOFT Softw. Eng. Notes, vol. 30, no. 2, pp. 1–36, 2005.

[3] Tip, F., "A survey of program slicing techniques," J. Program. Lang., vol. 3, no. 3, 1995.

[4] Ghahramani, Z., "Probabilistic machine learning and artificial intelligence," Nature, vol. 521, no. 7553, pp. 452–459, 2015.

[5] Claret, G., Rajamani, S. K., Nori, A. V., Gordon, A. D., y Borgström, J., "Bayesian inference using data flow analysis," en Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 92–102, ACM, 2013.

[6] Goldwasser, S. y Micali, S., "Probabilistic encryption," J. Comput. Sys. Sci., vol. 28, no. 2, pp. 270–299, 1984.

[7] Sanders, J. W. y Zuliani, P., "Quantum programming," en Proceedings of the 5th International Conference on Mathematics of Program Construction, MPC'00, pp. 80–99, Springer, 2000.

[8] Motwani, R. y Raghavan, P., Randomized Algorithms. Cambridge University Press, 1995.

[9] van de Meent, J.-W., Paige, B., Yang, H., y Wood, F., "An introduction to probabilistic programming," 2021.

[10] Kaminski, B. L., Katoen, J., Matheja, C., y Olmedo, F., "Weakest precondition reasoning for expected runtimes of randomized algorithms," J. ACM, vol. 65, no. 5, pp. 30:1–30:68, 2018.

[11] Darwiche, A., Modeling and Reasoning with Bayesian Networks. Cambridge University Press, 2009.

[12] Dutta, S., Legunsen, O., Huang, Z., y Misailovic, S., "Testing probabilistic programming systems," en Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, p. 574–586, ACM, 2018.

[13] Hur, C.-K., Nori, A. V., Rajamani, S. K., y Samuel, S., "Slicing probabilistic programs," en Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, pp. 133–144, ACM, 2014.

[14] Amtoft, T. y Banerjee, A., "A theory of slicing for probabilistic control flow graphs," en Proceedings of the 19th International Conference on Foundations of Software Science and Computation Structures, FoSSaCS'16, pp. 180–196, Springer, 2016.

[15] Amtoft, T. y Banerjee, A., "A theory of slicing for imperative probabilistic programs," ACM Trans. Program. Lang. Syst., vol. 42, no. 2, 2020.

[16] Comuzzi, J. J. y Hart, J. M., "Program slicing using weakest preconditions," en Proceedings of the 3rd International Symposium of Formal Methods Europe, FME'96, pp. 557–575, Springer, 1996.

[17] Barros, J. B., Da Cruz, D., Henriques, P. R., y Pinto, J. S., "Assertion-based slicing and slice graphs," Formal Aspects Comput., vol. 24, no. 2, pp. 217–248, 2012.

[18] Meyer, B., "Applying "Design by Contract"," Computer, vol. 25, no. 10, p. 40–51, 1992.

[19] Lee, W. K., Chung, I. S., Yoon, G. S., y Kwon, Y. R., "Specification-based program slicing and its applications," J. Syst. Archit., vol. 47, no. 5, pp. 427–443, 2001.

[20] McIver, A. y Morgan, C., Abstraction, Refinement And Proof For Probabilistic Systems. Springer, 2004.

[21] Dijkstra, E. W. y Scholten, C. S., Predicate Calculus and Program Semantics. Springer, 1990.

[22] Dijkstra, E. W., A Discipline of Programming. Prentice Hall, 1976.

[23] Kozen, D., "A probabilistic PDL," J. Comput. Syst. Sci., vol. 30, no. 2, pp. 162 – 178, 1985.

[24] Kozen, D., "Semantics of probabilistic programs," J. Comput. Syst. Sci., vol. 22, no. 3, pp. 328–350, 1981.

[25] Kaminski, B. L., Advanced weakest precondition calculi for probabilistic programs. PhD thesis, RWTH Aachen University, 2019.

[26] Lauritzen, S. L. y Spiegelhalter, D. J., "Local computations with probabilities on graphical structures and their application to expert systems," Journal of the Royal Statistical Society: Series B (Methodological), vol. 50, no. 2, pp. 157–194, 1988.

[27] Gordon, A. D., Henzinger, T. A., Nori, A. V., y Rajamani, S. K., "Probabilistic programming," en Proceedings of the on Future of Software Engineering, FOSE 2014, pp. 167–181, ACM, 2014.

[28] Batz, K., Kaminski, B. L., Katoen, J., y Matheja, C., "How long, O bayesian network, will I sample thee? - A program analysis perspective on expected sampling times," en Proceedings of the 27th European Symposium on Programming, ESOP 2018, vol. 10801 de Lecture Notes in Computer Science, pp. 186–213, Springer, 2018.

[29] Kaminski, B. L., Katoen, J., y Matheja, C., "On the hardness of analyzing probabilistic programs," Acta Informatica, vol. 56, no. 3, pp. 255–285, 2019.

[30] Kaminski, B. L. y Katoen, J.-P., "On the hardness of almost-sure termination," en Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science, MFCS'15, pp. 307–318, Springer, 2015.

[31] McIver, A., Morgan, C., Kaminski, B. L., y Katoen, J., "A new proof rule for almost-sure termination," Proc. ACM Program. Lang., vol. 2, no. POPL, pp. 33:1–33:28, 2018.

[32] Chakarov, A. y Sankaranarayanan, S., "Probabilistic program analysis with martingales," en Proceedings of the 25th International Conference Computer Aided Verification, CAV'13, vol. 8044 de Lecture Notes in Computer Science, pp. 511–526, Springer, 2013.

[33] Fioriti, L. M. F. y Hermanns, H., "Probabilistic termination: Soundness, completeness, and compositionality," en Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'15, pp. 489–501, ACM, 2015.

[34] Chatterjee, K., Novotný, P., y Zikelic, D., "Stochastic invariants for probabilistic termination," en Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL'17, pp. 145–160, ACM, 2017.

[35] Chatterjee, K., Fu, H., y Goharshady, A. K., "Non-polynomial worst-case analysis of recursive programs," en Proceedings of the 29th International Conference on Computer Aided Verification, CAV'17, Part II, vol. 10427 de Lecture Notes in Computer Science, pp. 41–63, Springer, 2017.

[36] Chatterjee, K., Fu, H., Novotný, P., y Hasheminezhad, R., "Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs," ACM Trans. Program. Lang. Syst., vol. 40, no. 2, pp. 7:1–7:45, 2018.

[37] Huang, M., Fu, H., y Chatterjee, K., "New approaches for almost-sure termination of probabilistic programs," en Proceedings of the 16th Asian Symposium on Programming Languages and Systems, APLAS'18, vol. 11275 de Lecture Notes in Computer Science, pp. 181–201, Springer, 2018.

[38] Agrawal, S., Chatterjee, K., y Novotný, P., "Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs," Proc. ACM Program. Lang., vol. 2, no. POPL, pp. 34:1–34:32, 2018.

[39] Fu, H. y Chatterjee, K., "Termination of nondeterministic probabilistic programs," en Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI'19, vol. 11388 de Lecture Notes in Computer Science, pp. 468–490, Springer, 2019.

[40] Huang, M., Fu, H., Chatterjee, K., y Goharshady, A. K., "Modular verification for almost-sure termination of probabilistic programs," Proc. ACM Program. Lang., vol. 3, no. OOPSLA, pp. 129:1–129:29, 2019.

[41] Chung, I. S., Lee, W. K., Yoon, G. S., y Kwon, Y. R., "Program slicing based on specification," en Proceedings of the 16th ACM Symposium on Applied Computing, SAC'01, pp. 605–609, ACM, 2001.

[42] Jones, C., Probabilistic Non-determinism. PhD thesis, University of Edinburgh, 1989.

[43] den Hartog, J. I., "Verifying probabilistic programs using a Hoare like logic," en Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science, ASIAN'99, pp. 113–125, Springer, 1999.

[44] Chadha, R., Cruz-Filipe, L., Mateus, P., y Sernadas, A., "Reasoning about probabilistic sequential programs," Theor. Comput. Sci., vol. 379, no. 1-2, pp. 142–165, 2007.

[45] Rand, R. y Zdancewic, S., "VPHL: A verified partial-correctness logic for probabilistic programs," en The 31st Conference on the Mathematical Foundations of Programming

Semantics, MFPS'15, vol. 319 de Electronic Notes in Theoretical Computer Science, pp. 351–367, Elsevier, 2015.

[46] Barthe, G., Espitau, T., Gaboardi, M., Grégoire, B., Hsu, J., y Strub, P.-Y., "An assertion-based program logic for probabilistic programs," en Proceedings of the 27th European Symposium on Programming, ESOP'18, pp. 117–144, Springer, 2018.

[47] Da Cruz, D., Henriques, P. R., y Pinto, J. S., "Contract-based slicing," en Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, ISoLA'10, pp. 106–120, Springer, 2010.

[48] Hurd, J., McIver, A., y Morgan, C., "Probabilistic guarded commands mechanized in HOL," Theor. Comput. Sci., vol. 346, no. 1, pp. 96–112, 2005.

[49] Cock, D., "pGCL for Isabelle," Arch. Formal Proofs, vol. 2014, 2014.

[50] Cock, D., "Verifying probabilistic correctness in Isabelle with pGCL," en Proceedings of the 7th Conference on Systems Software Verification, SSV'12, vol. 102 de EPTCS, pp. 167–178, 2012.

[51] Wechler, W., Universal algebra for computer scientists, vol. 25. Springer, 2012.

Annexes

Annex A. Ommited proofs

Proof of Lemma 3.2. We prove that

$$\models \mathsf{vc}\,[c]\,(g) \quad \Longrightarrow \quad \mathsf{wpre}\,[c]\,(g) \Rrightarrow \mathsf{wlp}\,[c]\,(g)\ ,$$

by induction on the structure of c.

Case $c = \mathsf{skip}$

$$\begin{array}{ll} & \mathsf{wpre}\,[\mathsf{skip}]\,(g) \\ = & \quad \{\text{def of } \mathsf{wpre} \text{ for no-op}\} \\ & g \\ = & \quad \{\text{def of } \mathsf{wlp} \text{ for no-op}\} \\ & \mathsf{wlp}\,[\mathsf{skip}]\,(g) \end{array}$$

Case $c = x \coloneqq E$

$$\begin{array}{ll} & \mathsf{wpre}\,[x \coloneqq E]\,(g) \\ = & \quad \{\text{def of } \mathsf{wpre} \text{ for assignment}\} \\ & g\,[x/E] \\ = & \quad \{\text{def of } \mathsf{wlp} \text{ for assignment}\} \\ & \mathsf{wlp}\,[x \coloneqq E]\,(g) \end{array}$$

Case $c = \mathsf{if}\ (G)\ \mathsf{then}\ \{c_1\}\ \mathsf{else}\ \{c_2\}$

From the hypothesis and the definition of vc we have $\models \mathsf{vc}\,[c_1]\,(g)$ and $\models \mathsf{vc}\,[c_2]\,(g)$. Thus

$$\begin{array}{ll} & \mathsf{wpre}\,[\mathsf{if}\ (G)\ \mathsf{then}\ \{c_1\}\ \mathsf{else}\ \{c_2\}]\,(g) \\ = & \quad \{\text{def of } \mathsf{wpre} \text{ for conditional branching}\} \end{array}$$

$$
\begin{array}{ll}
& [G] \cdot \mathsf{wpre}\,[c_1]\,(g) + [\neg G] \cdot \mathsf{wpre}\,[c_2]\,(g) \\
\Rrightarrow & \quad \{\text{inductive hypothesis}\} \\
& [G] \cdot \mathsf{wlp}\,[c_1]\,(g) + [\neg G] \cdot \mathsf{wlp}\,[c_2]\,(g) \\
= & \quad \{\text{def of } \mathsf{wlp} \text{ for conditional branching}\} \\
& \mathsf{wlp}\,[\mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}]\,(g)
\end{array}
$$

Case $c = \{c_1\}\,[p]\,\{c_2\}$

From the hypothesis and the definition of vc we have $\models \mathsf{vc}\,[c_1]\,(g)$ and $\models \mathsf{vc}\,[c_2]\,(g)$. Thus

$$
\begin{array}{ll}
& \mathsf{wpre}\,[\{c_1\}\,[p]\,\{c_2\}]\,(g) \\
= & \quad \{\text{def of } \mathsf{wpre} \text{ for probabilisitic choice}\} \\
& p \cdot \mathsf{wpre}\,[c_1]\,(g) + (1-p) \cdot \mathsf{wpre}\,[c_2]\,(g) \\
\Rrightarrow & \quad \{\text{inductive hypothesis}\} \\
& p \cdot \mathsf{wlp}\,[c_1]\,(g) + (1-p) \cdot \mathsf{wlp}\,[c_2]\,(g) \\
= & \quad \{\text{def of } \mathsf{wlp} \text{ for probabilisitic choice}\} \\
& \mathsf{wlp}\,[\mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}]\,(g)
\end{array}
$$

Case $c = c_1\,;\,c_2$

From the hypothesis and the definition of vc we have $\models \mathsf{vc}\,[c_1]\,(\mathsf{wpre}\,[c_2]\,(g))$ and $\models \mathsf{vc}\,[c_2]\,(g)$. Thus

$$
\begin{array}{ll}
& \mathsf{wpre}\,[c_1\,;\,c_2]\,(g) \\
= & \quad \{\text{def of } \mathsf{wpre} \text{ for sequential composition}\} \\
& \mathsf{wpre}\,[c_1]\,(\mathsf{wpre}\,[c_2]\,(g)) \\
\Rrightarrow & \quad \{\text{inductive hypothesis}\} \\
& \mathsf{wlp}\,[c_1]\,(\mathsf{wpre}\,[c_2]\,(g)) \\
\Rrightarrow & \quad \{\text{inductive hypothesis}\} \\
& \mathsf{wlp}\,[c_1]\,(\mathsf{wlp}\,[c_2]\,(g)) \\
= & \quad \{\text{def of } \mathsf{wlp} \text{ for sequential composition}\} \\
& \mathsf{wlp}\,[c_1\,;\,c_2]\,(g)
\end{array}
$$

Case $c = \mathsf{while}\,(G)\;[inv]\;\mathsf{do}\,\{c\}$

From the hypothesis and the definition of vc we have

$$
\models \{[G] \cdot inv \Rrightarrow \mathsf{wpre}\,[c]\,(inv)\} \;\wedge \tag{1}
$$

$$
\models \{[\neg G] \cdot inv \Rrightarrow g\} \;\wedge \tag{2}
$$

$$
\models \mathsf{vc}\,[c]\,(inv) \tag{3}
$$

Thus

$$\begin{array}{ll} & \mathsf{wpre}\,[\mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c\}]\,(g) \\ = & \quad \{\text{def of } \mathsf{wpre} \text{ for guarded loop}\} \\ & inv \\ = & \quad \{\text{algebra}\} \\ & [\neg G] \cdot inv + [G] \cdot inv \\ = & \quad \{\text{idempotency}\} \\ & [\neg G] \cdot ([\neg G] \cdot inv) + [G] \cdot ([G] \cdot inv) \\ \Rrightarrow & \quad \{\text{using (1) and (2)}\} \\ & [\neg G] \cdot g + [G] \cdot \mathsf{wpre}\,[c]\,(inv) \\ \Rrightarrow & \quad \{\text{inductive hypothesis, using (3)}\} \\ & [\neg G] \cdot g + [G] \cdot \mathsf{wlp}\,[c]\,(inv) \end{array}$$

Let us define $H(h) = [\neg G] \cdot g + [G] \cdot \mathsf{wlp}\,[c]\,(h)$. From the above derivation we can conclude that

$$inv \Rrightarrow [\neg G] \cdot g + [G] \cdot \mathsf{wlp}\,[c]\,(inv) = H(inv)\ .$$

Now by Park's Theorem [51],

$$\begin{array}{ll} & inv \\ \Rrightarrow & \quad \{\text{Park's Theorem}\} \\ & \nu h.\, H(h) \\ = & \quad \{\text{def of } H \text{ and } \mathsf{wlp} \text{ for guarded loop}\} \\ & \mathsf{wlp}\,[\mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c\}]\,(g) \end{array}$$

Combining the results, we obtain

$$\mathsf{wpre}\,[\mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c\}]\,(g)\ =\ inv\ \Rrightarrow \mathsf{wlp}\,[\mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c\}]\,(g) \qquad \square$$

Proof of Lemma 3.3. Let $g \Rrightarrow g'$. We prove the monotonicity of $\mathsf{vc}\,[c]$ by induction on the structure of c (the monotonicity of $\mathrm{VCG}\,[c]$ follows as immediate corollary).

Case $c = \mathsf{skip}$

Vacuously true.

Case $c = x := E$

Vacuously true.

Case $c = \mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}$

$$
\begin{array}{ll}
& \models \mathsf{vc}\,[\mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}]\,(g) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{vc}\} \\
& \models \mathsf{vc}\,[c_1]\,(g) \cup \mathsf{vc}\,[c_2]\,(g) \\
\Rightarrow & \quad \{\text{inductive hypothesis}\} \\
& \models \mathsf{vc}\,[c_1]\,(g') \cup \mathsf{vc}\,[c_2]\,(g') \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{vc}\} \\
& \models \mathsf{vc}\,[\mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}]\,(g')
\end{array}
$$

Case $c = \{c_1\}\,[p]\,\{c_2\}$

Analogous to the previous case.

Case $c = c_1; c_2$

$$
\begin{array}{ll}
& \models \mathsf{vc}\,[c_1; c_2]\,(g) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{vc}\} \\
& \models \mathsf{vc}\,[c_1]\,(\mathsf{wpre}\,[c_2]\,(g)) \cup \mathsf{vc}\,[c_2]\,(g) \\
\Rightarrow & \quad \{\text{inductive hypothesis and } \mathsf{wpre} \text{ monotonicity}\} \\
& \models \mathsf{vc}\,[c_1]\,(\mathsf{wpre}\,[c_2]\,(g')) \cup \mathsf{vc}\,[c_2]\,(g') \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{vc}\} \\
& \models \mathsf{vc}\,[c_1; c_2]\,(g')
\end{array}
$$

Case $c = \mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c\}$

$$
\begin{array}{ll}
& \models \mathsf{vc}\,[\mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c\}]\,(g) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{vc}\} \\
& \models \{[G] \cdot inv \Rrightarrow \mathsf{wpre}\,[c]\,(inv)\,, [\neg G] \cdot inv \Rrightarrow g\} \cup \mathsf{vc}\,[c]\,(inv) \\
\Rightarrow & \quad \{\text{transitivity of } \Rrightarrow\} \\
& \models \{[G] \cdot inv \Rrightarrow \mathsf{wpre}\,[c]\,(inv)\,, [\neg G] \cdot inv \Rrightarrow g'\} \cup \mathsf{vc}\,[c]\,(inv) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{vc}\} \\
& \models \mathsf{vc}\,[\mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c\}]\,(g')
\end{array}
$$

□

Proof of Lemma 3.4.

1. Let $i_j = \mathsf{if}\,(G)\,\mathsf{then}\,\{c_1\}\,\mathsf{else}\,\{c_2\}$ for some $1 \leq j \leq n$. Then

$$
\begin{aligned}
& \models \mathrm{VCG}\,[c]\,(f,g) \\
\Leftrightarrow \quad & \{\text{def of VCG}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}\,[c]\,(g)\} \cup \mathsf{vc}\,[c]\,(g) \\
\Leftrightarrow \quad & \{\text{def of vc for sequential composition}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}\,[c]\,(g)\} \cup \mathsf{vc}\,[i_1;\ldots;i_j]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \cup \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\
\Leftrightarrow \quad & \{\text{def of vc for sequential composition}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}\,[c]\,(g)\} \cup \mathsf{vc}\,[i_1;\ldots;i_{j-1}]\left(\mathsf{wpre}\,[i_j]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right)\right) \\
& \quad \cup \mathsf{vc}\,[i_j]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \cup \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\
\Leftrightarrow \quad & \{\text{def of vc for conditional branching}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}\,[c]\,(g)\} \cup \mathsf{vc}\,[i_1;\ldots;i_{j-1}]\left(\mathsf{wpre}\,[i_j]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right)\right) \\
& \quad \cup \mathsf{vc}\,[c_1]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \cup \mathsf{vc}\,[c_2]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \cup \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\
\Leftrightarrow \quad & \{\text{def of wpre for sequential composition}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}\,[i_1;\ldots;i_{j-1}]\left(\mathsf{wpre}_{\geq j}\,[c]\,(g)\right)\} \\
& \quad \cup \mathsf{vc}\,[i_1;\ldots;i_{j-1}]\left(\mathsf{wpre}_{\geq j}\,[c]\,(g)\right) \cup \mathsf{vc}\,[c_1]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \\
& \quad \cup \mathsf{vc}\,[c_2]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \cup \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\
\Leftrightarrow \quad & \{\text{def of } \mathrm{VCG}_{\leq k} \text{ and rearrange terms}\} \\
& \models \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\
& \quad \cup \mathsf{vc}\,[c_1]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \\
& \quad \cup \mathsf{vc}\,[c_2]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \\
& \quad \cup \mathrm{VCG}_{\leq j-1}\,[c]\left(f, \mathsf{wpre}_{\geq j}\,[c]\,(g)\right)
\end{aligned}
$$

The case where $i_j = \{c_1\}\,[p]\,\{c_2\}$ follows the same argument.

2. $i_j = \mathsf{while}\,(G)\;[inv]\;\mathsf{do}\,\{c'\}$ for some $1 \leq j \leq n$. Then

$$
\begin{aligned}
& \models \mathrm{VCG}\,[c]\,(f,g) \\
\Leftrightarrow \quad & \{\text{def of VCG}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}\,[c]\,(g)\} \cup \mathsf{vc}\,[c]\,(g) \\
\Leftrightarrow \quad & \{\text{def of wpre and vc for sequential composition}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}\,[i_1;\ldots;i_{j-1}]\left(\mathsf{wpre}\,[i_j]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right)\right)\} \\
& \quad \cup \mathsf{vc}\,[i_1;\ldots;i_j]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \cup \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\
\Leftrightarrow \quad & \{\text{def of wpre for guarded loop and def of vc for sequential composition}\}
\end{aligned}
$$

$$
\begin{array}{ll}
& \models \{f \Rrightarrow \mathsf{wpre}\,[i_1;\ldots;i_{j-1}]\,(inv)\} \\
& \quad \cup\ \mathsf{vc}\,[i_1;\ldots;i_{j-1}]\left(\mathsf{wpre}\,[i_j]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right)\right) \\
& \quad \cup\ \mathsf{vc}\,[i_j]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \cup \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{wpre} \text{ and } \mathsf{vc} \text{ for guarded loop}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}\,[i_1;\ldots;i_{j-1}]\,(inv)\} \cup \mathsf{vc}\,[i_1;\ldots;i_{j-1}]\,(inv) \\
& \quad \cup\ \left\{[G]\cdot inv \Rrightarrow \mathsf{wpre}\,[c']\,(inv)\,, [\neg G]\cdot inv \Rrightarrow \mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right\} \\
& \quad \cup\ \mathsf{vc}\,[c']\,(inv) \cup \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\
\Leftrightarrow & \quad \{\text{def of } \mathrm{VCG}_{\leq k} \text{ and algebra}\} \\
& \models \mathrm{VCG}_{\leq j-1}\,[c]\,(f, inv) \\
& \quad \cup\ \{[G]\cdot inv \Rrightarrow \mathsf{wpre}\,[c']\,(inv)\} \cup \mathsf{vc}\,[c']\,(inv) \\
& \quad \cup\ \left\{[\neg G]\cdot inv \Rrightarrow \mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right\} \cup \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\
\Leftrightarrow & \quad \{\text{def of VCG and rearrange terms}\} \\
& \models \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\
& \quad \cup\ \left\{[\neg G]\cdot inv \Rrightarrow \mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right\} \\
& \quad \cup\ \mathrm{VCG}\,[c']\,([G]\cdot inv, inv) \\
& \quad \cup\ \mathrm{VCG}_{\leq j-1}\,[c]\,(f, inv) \qquad \square
\end{array}
$$

Proof of Theorem 4.2. By induction on the derivation of the relation $c \vdash \langle f, g\rangle^{\circlearrowleft} \rightsquigarrow c' \vdash \langle f', g'\rangle^{\circlearrowleft}$ (see Figure 4.2).

Case [$\rightsquigarrow$ ift]

In this case we have

$$
\begin{array}{lcl}
c & = & i_1;\ldots;i_{j-1};\mathsf{if}\,(G)\,\mathsf{then}\,\{c'\}\,\mathsf{else}\,\{c_f\};i_{j+1};\ldots;i_n \\
f' & = & [G]\cdot\mathsf{wpre}\,[c']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \\
g' & = & \mathsf{wpre}_{\geq j+1}\,[c]\,(g)
\end{array}
$$

We must show that $c\,[c'/c''] \preccurlyeq c$ and $\models \mathrm{VCG}\,[c]\,(f,g) \implies \mathrm{VCG}\,[c\,[c'/c'']]\,(f,g)$. The first proof obligation is straightforward (see Figure 4.1). To establish the second proof obligation, we exploit Lemma 3.4 and the fact that $c\,[c'/c'']$ coincides with c in all but the j-th instruction. Therefore, assuming

$$
\begin{array}{lll}
 & \models \mathsf{vc}_{\geq j+1}\,[c]\,(g) & \\
\wedge & \models \mathsf{vc}\,[c']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) & (4) \\
\wedge & \models \mathsf{vc}\,[c_f]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) & \\
\wedge & \models \mathrm{VCG}_{\leq j-1}\,[c]\left(f, \mathsf{wpre}_{\geq j}\,[c]\,(g)\right) & (5)
\end{array}
$$

we have to conclude that

$$\begin{aligned} & \models \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\ \wedge \quad & \models \mathsf{vc}\,[c'']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) \qquad (6) \\ \wedge \quad & \models \mathsf{vc}\,[c_f]\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) \\ \wedge \quad & \models \mathrm{VCG}_{\leq j-1}\,[c]\,\Big(f, \mathsf{wpre}_{\geq j}\,[c\,[c'/c'']]\,(g)\Big) \qquad (7) \end{aligned}$$

We need to prove only Equations (6) and (7) since the other two are already part of the premises. Let us start with Equation (6). From hypothesis $\{f'\}\ c'' \preccurlyeq c'\ \{g'\}^{\circlearrowleft}$, we have:

$$\begin{aligned} & \models \Big\{[G]\cdot \mathsf{wpre}\,[c']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) \Rrightarrow \mathsf{wpre}\,[c']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big)\Big\} \\ & \quad \cup \mathsf{vc}\,[c']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) \\ \Rightarrow \ & \models \Big\{[G]\cdot \mathsf{wpre}\,[c']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) \Rrightarrow \mathsf{wpre}\,[c'']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big)\Big\} \\ & \quad \cup \mathsf{vc}\,[c'']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) \end{aligned} \qquad (8)$$

The premise of the above implication holds true from the fact that $[G]\cdot h \Rrightarrow h$ for any expectation h, and from Equation (4). Thus we can conclude, in particular,

$$\mathsf{vc}\,[c'']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big)\ ,$$

which amounts to Equation (6). Finally, we show that Equation (7) follows from Equation (5):

$$\begin{aligned} & \models \mathrm{VCG}_{\leq j-1}\,[c]\,\Big(f, \mathsf{wpre}_{\geq j}\,[c]\,(g)\Big) \\ \Leftrightarrow \quad & \{\text{def of } \mathsf{wpre} \text{ for sequential composition}\} \\ & \models \mathrm{VCG}_{\leq j-1}\,[c]\,\Big(f, \mathsf{wpre}\,[i_j]\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big)\Big) \\ \Leftrightarrow \quad & \{\text{def of } \mathsf{wpre} \text{ for conditional branching}\} \\ & \models \mathrm{VCG}_{\leq j-1}\,[c]\left(f, \begin{array}{l}[G]\cdot \mathsf{wpre}\,[c']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) + \\ [\neg G]\cdot \mathsf{wpre}\,[c_f]\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big)\end{array}\right) \\ \Leftrightarrow \quad & \{\text{idempotency}\} \\ & \models \mathrm{VCG}_{\leq j-1}\,[c]\left(f, \begin{array}{l}[G]\cdot[G]\cdot \mathsf{wpre}\,[c']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) + \\ [\neg G]\cdot \mathsf{wpre}\,[c_f]\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big)\end{array}\right) \\ \Rightarrow \quad & \{\text{by (8) and monotonicity of VCG}\} \\ & \models \mathrm{VCG}_{\leq j-1}\,[c]\left(f, \begin{array}{l}[G]\cdot \mathsf{wpre}\,[c'']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) + \\ [\neg G]\cdot \mathsf{wpre}\,[c_f]\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big)\end{array}\right) \\ \Leftrightarrow \quad & \{\text{def of } \mathsf{wpre} \text{ for conditional branching}\} \\ & \models \mathrm{VCG}_{\leq j-1}\,[c]\,\Big(f, \mathsf{wpre}\,[\mathsf{if}\ (G)\ \mathsf{then}\ \{c''\}\ \mathsf{else}\ \{c_f\}]\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big)\Big) \\ \Leftrightarrow \quad & \{\text{def of } c\,[c'/c'']\} \\ & \models \mathrm{VCG}_{\leq j-1}\,[c]\,\Big(f, \mathsf{wpre}_{\geq j}\,[c\,[c'/c'']]\,(g)\Big) \end{aligned}$$

Case [$\rightsquigarrow$ iff]

Analogous to case [$\rightsquigarrow$ ift].

Case [$\rightsquigarrow$ pl]

We proceed analogously to the case of rule [$\rightsquigarrow$ ift]. Now we have

$$\begin{array}{lcl} c & = & i_1; \ldots; i_{j-1}; \{c'\}\,[p]\,\{c_r\}; i_{j+1}; \ldots; i_n \\ f' & = & \mathsf{wpre}\,[c']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \\ g' & = & \mathsf{wpre}_{\geq j+1}\,[c]\,(g) \end{array}$$

Again, we must show that $c\,[c'/c'']\preccurlyeq c$ and $\models \mathrm{VCG}\,[c]\,(f,g) \implies \mathrm{VCG}\,[c\,[c'/c'']]\,(f,g)$. The first proof obligation is straightforward (see Figure 4.1). To establish the second proof obligation, we exploit Lemma 3.4 and the fact that $c\,[c'/c'']$ coincides with c in all but the j-th instruction. Therefore, assuming

$$\begin{array}{lll} & \models \mathsf{vc}_{\geq j+1}\,[c]\,(g) & \\ \wedge & \models \mathsf{vc}\,[c']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) & (9) \\ \wedge & \models \mathsf{vc}\,[c_r]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) & \\ \wedge & \models \mathrm{VCG}_{\leq j-1}\,[c]\left(f, \mathsf{wpre}_{\geq j}\,[c]\,(g)\right) & (10) \end{array}$$

we have to conclude that

$$\begin{array}{lll} & \models \mathsf{vc}_{\geq j+1}\,[c]\,(g) & \\ \wedge & \models \mathsf{vc}\,[c'']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) & (11) \\ \wedge & \models \mathsf{vc}\,[c_r]\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) & \\ \wedge & \models \mathrm{VCG}_{\leq j-1}\,[c]\left(f, \mathsf{wpre}_{\geq j}\,[c\,[c'/c'']]\,(g)\right) & (12) \end{array}$$

We need to prove only Equations (11) and (12) since the other two are already part of the premises. Let us start with Equation (11). From hypothesis $\{f'\}\ c''\preccurlyeq c'\ \{g'\}^{\circlearrowleft}$, we have:

$$\begin{array}{rl} & \models \left\{\mathsf{wpre}\,[c']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \Rrightarrow \mathsf{wpre}\,[c']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right)\right\} \\ & \quad \cup\, \mathsf{vc}\,[c']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \\ \Rightarrow & \models \left\{\mathsf{wpre}\,[c']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \Rrightarrow \mathsf{wpre}\,[c'']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right)\right\} \\ & \quad \cup\, \mathsf{vc}\,[c'']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right) \end{array} \qquad (13)$$

The premise of the above implication holds true from the fact that $h \Rrightarrow h$ for any expectation h, and from Equation (9). Thus we can conclude, in particular,

$$\mathsf{vc}\,[c'']\left(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\right)\ ,$$

which amounts to Equation (11). Finally, we show that Equation (12) follows from Equation (10):

$$
\begin{array}{ll}
& \models \mathrm{VCG}_{\leq j-1}\,[c]\,\Big(f, \mathsf{wpre}_{\geq j}\,[c]\,(g)\Big) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{wpre} \text{ for sequential composition}\} \\
& \models \mathrm{VCG}_{\leq j-1}\,[c]\,\Big(f, \mathsf{wpre}\,[i_j]\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big)\Big) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{wpre} \text{ for probabilistic choice}\} \\
& \models \mathrm{VCG}_{\leq j-1}\,[c]\left(f,\ \begin{array}{l} p \cdot \mathsf{wpre}\,[c']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) + \\ (1-p) \cdot \mathsf{wpre}\,[c_r]\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) \end{array}\right) \\
\Rightarrow & \quad \{\text{using (13) and monotonicity of VCG}\} \\
& \models \mathrm{VCG}_{\leq j-1}\,[c]\left(f,\ \begin{array}{l} p \cdot \mathsf{wpre}\,[c'']\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) + \\ (1-p) \cdot \mathsf{wpre}\,[c_r]\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big) \end{array}\right) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{wpre} \text{ for probabilistic choice}\} \\
& \models \mathrm{VCG}_{\leq j-1}\,[c]\,\Big(f, \mathsf{wpre}\,[\{c''\}\ [p]\ \{c_r\}]\,\Big(\mathsf{wpre}_{\geq j+1}\,[c]\,(g)\Big)\Big) \\
\Leftrightarrow & \quad \{\text{Use def of } c\,[c'/c'']\} \\
& \models \mathrm{VCG}_{\leq j-1}\,[c]\,\Big(f, \mathsf{wpre}_{\geq j}\,[c\,[c'/c'']]\,(g)\Big)
\end{array}
$$

Case [$\rightsquigarrow$ `pr`]

Analogous to case [$\rightsquigarrow$ `pl`]

Case [$\rightsquigarrow$ `while`]

In this case we have

$$
\begin{array}{lcl}
c & = & i_1; \ldots; i_{j-1}; \mathsf{while}\,(G)\ [inv]\ \mathsf{do}\,\{c'\}; i_{j+1}; \ldots; i_n \\
f' & = & [G] \cdot inv \\
g' & = & inv
\end{array}
$$

We must show that $c\,[c'/c''] \preccurlyeq c$ and $\models \mathrm{VCG}\,[c]\,(f,g) \implies \mathrm{VCG}\,[c\,[c'/c'']]\,(f,g)$. The first proof obligation is straightforward (see Figure 4.1). To establish the second proof obligation, we exploit Lemma 3.4 and the fact that $c\,[c'/c'']$ coincides with c in all but the j-th instruction. Therefore, assuming

$$
\begin{array}{lll}
 & \models \mathsf{vc}_{\geq j+1}\,[c]\,(g) & \\
\wedge & \models \{[\neg G] \cdot inv \Rrightarrow \mathsf{wpre}_{\geq j+1}\,[c]\,(g)\} & \\
\wedge & \models \mathrm{VCG}\,[c']\,([G] \cdot inv, inv) & (14) \\
\wedge & \models \mathrm{VCG}_{\leq j-1}\,[c]\,(f, inv) &
\end{array}
$$

we have to conclude that

$$\begin{aligned} & \models \mathsf{vc}_{\geq j+1}\,[c]\,(g) \\ \wedge \quad & \models \{[\neg G]\cdot inv \Rrightarrow \mathsf{wpre}_{\geq j+1}\,[c]\,(g)\} \\ \wedge \quad & \models \mathrm{VCG}\,[c'']\,([G]\cdot inv, inv) \\ \wedge \quad & \models \mathrm{VCG}_{\leq j-1}\,[c]\,(f, inv) \end{aligned} \tag{15}$$

We need to prove only Equation (14) since the other three are already part of the premises. From hypothesis $\{f'\}\ c'' \preccurlyeq c'\ \{g'\}^{\circlearrowleft}$, we have:

$$\models \mathrm{VCG}\,[c']\,([G]\cdot inv, inv) \ \Rightarrow\ \mathrm{VCG}\,[c'']\,([G]\cdot inv, inv)$$

The premise of the above implication holds true from Equation (14). Then we can conclude $\mathrm{VCG}\,[c'']\,([G]\cdot inv, inv)$ and (15) is proved.

Case [$\rightsquigarrow$ `refl`]

This case is inmediate because $f' = f$ and $g' = g$ and $c = c'$.

Case [$\rightsquigarrow$ `trans`]

Let c^* be a pWhile with its respective pre- and post-expectation f^* and g^* such that c' is a subprogram of c^*, $c \vdash \langle f, g\rangle^{\circlearrowleft} \rightsquigarrow c^* \vdash \langle f^*, g^*\rangle^{\circlearrowleft}$ and $c^* \vdash \langle f^*, g^*\rangle^{\circlearrowleft} \rightsquigarrow c' \vdash \langle f', g'\rangle^{\circlearrowleft}$. Let us apply the induction hypothesis to c^* and c', this gives $\{f'\}\ c''\ \preccurlyeq\ c'\ \{g'\}^{\circlearrowleft} \Rrightarrow \{f^*\}\ c^*\,[c'/c'']\ \preccurlyeq\ c^*\ \{g^*\}^{\circlearrowleft}$. We now apply this argument again $\{f^*\}\ c^*\,[c'/c'']\ \preccurlyeq\ c^*\ \{g^*\}^{\circlearrowleft} \Rrightarrow \{f\}\ c\,[c^*/c^*\,[c'/c'']]\ \preccurlyeq\ c\ \{g\}^{\circlearrowleft}$. This completes the proof. □

Proof of Lemma 5.1. We proceed to show that

$$\models \mathsf{vc}^{\downarrow}\,[c]\,(g) \quad \Longrightarrow \quad \mathsf{wpre}^{\downarrow}\,[c]\,(g) \Rrightarrow \mathsf{wp}\,[c]\,(g)$$

by induction on the structure of c. We only provide the case of loops as the remaining cases follow the same argument as for the counterpart VCG for partial correctness (see proof of Lemma 3.2). The proof of lemma follows as an immediate corollary of previous property.

Case $c = \mathsf{while}\,(G)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c'\}$

From the hypothesis
$\models \mathsf{vc}^{\downarrow}\,[\mathsf{while}\,(G)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c'\}]\,(g)$ we have

$$\models \left\{[G \wedge T] \Rrightarrow \mathsf{wpre}^{\downarrow}\,[c]\,([T])\right\}\ \wedge \tag{16}$$
$$\models \left\{\epsilon \cdot [G \wedge T \wedge v = v_0] \Rrightarrow \mathsf{wpre}^{\downarrow}\,[c]\,([v < v_0])\right\}\ \wedge \tag{17}$$
$$\models \{[G \wedge T] \Rrightarrow [\mathtt{l} \leq v \leq \mathtt{u}]\}\ \wedge$$
$$\models \mathsf{vc}^{\downarrow}\,[c]\,([T])\ \wedge \tag{18}$$
$$\models \mathsf{vc}^{\downarrow}\,[c]\,([v < v_0])\ \wedge \tag{19}$$
$$\models \{[G] \cdot inv \Rrightarrow \mathsf{wpre}\,[c]\,(inv)\}\ \wedge \tag{20}$$
$$\models \{[\neg G] \cdot inv \Rrightarrow g\}\ \wedge \tag{21}$$
$$\models \mathsf{vc}\,[c]\,(inv) \tag{22}$$

First, we begin proving that the loop terminates almost-surely from any state in T. To this end, we apply [20, Lemma 7.5.1], which requires proving that:

$$[G \wedge T] \Rrightarrow [\mathtt{l} \leq v \leq \mathtt{u}]\ \wedge$$
$$[G \wedge T] \Rrightarrow \mathsf{wp}\,[c]\,([T])\ \wedge \tag{23}$$
$$\epsilon \cdot [G \wedge T \wedge v = v_0] \Rrightarrow \mathsf{wpre}^{\downarrow}\,[c]\,([v < v_0]) \tag{24}$$

We need to prove only Equations (23) and (24) since the first equation is already part of the premises. To establish Equation (23), we apply inductive hypothesis on c and from premise (18), we conclude that $\mathsf{wpre}^{\downarrow}\,[c]\,([T]) \Rrightarrow \mathsf{wp}\,[c]\,([T])$, which together with premise (16) readily establishes Equation (23). To establish Equation (24), we follow the same argument (exploiting premises (19) and (17)).

In the proof of Lemma 3.2, we showed that

$$\models \mathsf{vc}\,[c]\,(g) \quad \Longrightarrow \quad \mathsf{wpre}\,[c]\,(g) \Rrightarrow \mathsf{wlp}\,[c]\,(g)\ .$$

Following the same argument as above, and exploiting premises (22) and (20), we conclude that

$$[G] \cdot inv \Rrightarrow \mathsf{wlp}\,[c]\,(inv)\ ,$$

which says that *inv* is a (weak) loop invariant. From this, and the fact that the loop terminates almost-surely from T, we can conclude the proof appealing to [20, Lemma 2.4.1-Case 2] as follows:

$$\begin{array}{ll} & \mathsf{wpre}^{\downarrow}\,[c]\,(g) \\ = & \quad \{\text{def of } \mathsf{wpre}^{\downarrow} \text{ for guarded loop}\} \\ & [T] \cdot inv \\ \Rrightarrow & \quad \{[20, \text{Lemma 2.4.1-Case 2}]\} \\ & \mathsf{wp}\,[\mathsf{while}\,(G)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c'\}]\,([\neg G] \cdot inv) \\ \Rrightarrow & \quad \{\text{use (21) and monotonicity of } \mathsf{wp}\} \end{array}$$

$$
\begin{array}{ll}
& \mathsf{wp}\,[\mathsf{while}\,(G)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c'\}]\,(g) \\
\Rrightarrow & \quad \{\text{def of } c\} \\
& \mathsf{wp}\,[c]\,(g)
\end{array}
$$

□

Proof of Lemma 5.2. The monotonicity proof of $\mathsf{vc}^{\downarrow}$ proceeds by induction on the program structure. We only provide the case of loops as the remaining cases follow the same argument as for the counterpart vc for partial correctness (see proof of Lemma 3.3).

Case $c = \mathsf{while}\,(G)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c'\}$

$$
\begin{array}{ll}
& \models \mathsf{vc}^{\downarrow}\,[\mathsf{while}\,(G)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c'\}]\,(g) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{vc}^{\downarrow} \text{ using (5.4) from chapter 5}\} \\
& \models \mathrm{VCG}^{\downarrow}\,[c']\,([G \wedge T]\,, [T])\ \cup \\
& \qquad \mathrm{VCG}^{\downarrow}\,[c']\,(\epsilon\ [G \wedge T \wedge v = v_0]\,, [v < v_0])\ \cup \\
& \qquad \{[G \wedge T]\ \Rrightarrow\ [\mathtt{l} \leq v \leq \mathtt{u}]\}\ \cup \\
& \qquad \mathrm{VCG}^{\downarrow}\,[c']\,([G] \cdot inv, inv)\ \cup \\
& \qquad \{[\neg G] \cdot inv\ \Rrightarrow\ g\} \\
\Rightarrow & \quad \{\text{hypothesis and transitivity of } \Rrightarrow\} \\
& \models \mathrm{VCG}^{\downarrow}\,[c']\,([G \wedge T]\,, [T])\ \cup \\
& \qquad \mathrm{VCG}^{\downarrow}\,[c']\,(\epsilon\ [G \wedge T \wedge v = v_0]\,, [v < v_0])\ \cup \\
& \qquad \{[G \wedge T]\ \Rrightarrow\ [\mathtt{l} \leq v \leq \mathtt{u}]\}\ \cup \\
& \qquad \mathrm{VCG}^{\downarrow}\,[c']\,([G] \cdot inv, inv)\ \cup \\
& \qquad \{[\neg G] \cdot inv\ \Rrightarrow\ g'\} \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{vc}^{\downarrow}\} \\
& \models \mathsf{vc}^{\downarrow}\,[\mathsf{while}\,(G)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c'\}]\,(g')
\end{array}
$$

The monotonicity proof of $\mathrm{VCG}^{\downarrow}$ follows as immediate corollary. □

Proof of Lemma 5.3. We give the proof only for the case where i_j is a loop since the other cases (where i_j is a conditional branching or a probabilistic choice) follow the same argument as the counterpart result for partial correctness (see proof of Lemma 3.4). Let $i_j = \mathsf{while}\,(G)\ [inv, T, v, \mathtt{l}, \mathtt{u}, \epsilon]\ \mathsf{do}\,\{c'\}$. Then

$$
\begin{array}{ll}
& \models \mathrm{VCG}^{\downarrow}\,[c]\,(f, g) \\
\Leftrightarrow & \quad \{\text{def of } \mathrm{VCG}^{\downarrow}\ \}
\end{array}
$$

$$
\begin{array}{ll}
& \models \{f \Rrightarrow \mathsf{wpre}^{\downarrow}[c](g)\} \cup \mathsf{vc}^{\downarrow}[c](g) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{vc}^{\downarrow} \text{ and } \mathsf{wpre}^{\downarrow} \text{ for sequential composition}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}^{\downarrow}[i_1;\ldots;i_{j-1}]\left(\mathsf{wpre}^{\downarrow}[i_j]\left(\mathsf{wpre}^{\downarrow}_{\geq j+1}[c](g)\right)\right)\} \\
& \cup\ \mathsf{vc}^{\downarrow}[i_1;\ldots;i_j]\left(\mathsf{wpre}^{\downarrow}_{\geq j+1}[c](g)\right) \cup \mathsf{vc}^{\downarrow}_{\geq j+1}[c](g) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{wpre}^{\downarrow} \text{ for guarded loop and def of } \mathsf{vc}^{\downarrow} \text{ for sequential composition}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}^{\downarrow}[i_1;\ldots;i_{j-1}]([T]\cdot inv)\} \cup \\
& \quad \mathsf{vc}^{\downarrow}[i_1;\ldots;i_{j-1}]\left(\mathsf{wpre}^{\downarrow}[i_j]\left(\mathsf{wpre}^{\downarrow}_{\geq j+1}[c](g)\right)\right) \cup \\
& \quad \mathsf{vc}^{\downarrow}[i_j]\left(\mathsf{wpre}^{\downarrow}_{\geq j+1}[c](g)\right) \cup \mathsf{vc}^{\downarrow}_{\geq j+1}[c](g) \\
\Leftrightarrow & \quad \{\text{def of } \mathsf{wpre}^{\downarrow} \text{ for guarded loop and}\} \\
& \models \{f \Rrightarrow \mathsf{wpre}^{\downarrow}[i_1;\ldots;i_{j-1}]([T]\cdot inv)\} \cup \\
& \quad \mathsf{vc}^{\downarrow}[i_1;\ldots;i_{j-1}]([T]\cdot inv) \cup \\
& \quad \mathsf{vc}^{\downarrow}[i_j]\left(\mathsf{wpre}^{\downarrow}_{\geq j+1}[c](g)\right) \cup \mathsf{vc}^{\downarrow}_{\geq j+1}[c](g) \\
\Leftrightarrow & \quad \{\text{def of VCG}^{\downarrow} \text{ and def of } \mathsf{vc}^{\downarrow} \text{ for guarded loop}\} \\
& \models \mathrm{VCG}^{\downarrow}_{\leq j-1}[c](f,[T]\cdot inv) \cup \\
& \quad \mathrm{VCG}^{\downarrow}[c']([G\wedge T],[T]) \cup \\
& \quad \mathrm{VCG}^{\downarrow}[c'](\epsilon\,[G\wedge T\wedge v=v_0],[v<v_0]) \cup \\
& \quad \{[G\wedge T] \Rrightarrow [\mathtt{l}\leq v\leq \mathtt{u}]\} \cup \\
& \quad \mathrm{VCG}^{\downarrow}[c']([G]\cdot inv, inv) \cup \\
& \quad \left\{[\neg G]\cdot inv \Rrightarrow \mathsf{wpre}^{\downarrow}_{\geq j+1}[c](g)\right\} \cup \\
& \quad \mathsf{vc}^{\downarrow}_{\geq j+1}[c](g) \\
\Leftrightarrow & \quad \{\text{def of VCG}^{\downarrow} \text{ and rearrange terms}\} \\
& \models \mathrm{VCG}^{\downarrow}_{\leq j+1}[c]([\neg G]\cdot inv, g) \cup \\
& \quad \mathrm{VCG}^{\downarrow}[c']([G]\cdot inv, inv) \cup \\
& \quad \mathrm{VCG}^{\downarrow}_{\leq j-1}[c](f,[T]\cdot inv) \cup \\
& \quad \mathrm{VCG}^{\downarrow}[c']([G\wedge T],[T]) \cup \\
& \quad \mathrm{VCG}^{\downarrow}[c'](\epsilon\,[G\wedge T\wedge v=v_0],[v<v_0]) \cup \\
& \quad \{[G\wedge T] \Rrightarrow [\mathtt{l}\leq v\leq \mathtt{u}]\}
\end{array}
$$

□

Proof of Theorem 5.1. It follows the same argument as the counterpart result for partial correctness (Theorem 4.1). □

Proof of Theorem 5.2. By induction on the derivation of the relation $c \vdash \langle f, g\rangle^{\downarrow} \rightsquigarrow c' \vdash \{\langle f_i, g_i\rangle\}_{i=1,\ldots,n}$ (see Figure 4.2). We only provide the case of loops and transitivity as the remaining cases follow the same argument of the counterpart removing nested instructions for partial correctness (see proof of Theorem 4.2).

Case [$\rightsquigarrow$ while$^{\downarrow}$]

In this case we have

$$c = i_1; \ldots; i_{j-1}; \mathsf{if}\ (G)\ \mathsf{then}\ \{c'\}\ \mathsf{else}\ \{c_f\}; i_{j+1}; \ldots; i_n$$

also we have

$$\{[G] \cdot inv\}\ c'' \preccurlyeq c'\ \{inv\}^{\circlearrowleft} \tag{25}$$
$$\{[G \wedge T]\}\ c'' \preccurlyeq c'\ \{[T]\}^{\downarrow} \tag{26}$$
$$\{\epsilon \cdot [G \wedge T \wedge v = v_0]\}\ c'' \preccurlyeq c'\ \{[v < v_0]\}^{\downarrow} \tag{27}$$

We must show that $c\,[c'/c''] \preccurlyeq c$ and $\models \mathrm{VCG}\,[c]\,(f, g) \implies \mathrm{VCG}\,[c\,[c'/c'']]\,(f, g)$. The first proof obligation is straightforward (see Figure 4.1). To establish the second proof obligation, we exploit Lemma 5.3 and the fact that $c\,[c'/c'']$ coincides with c in all but the j-th instruction. Therefore, assuming

$$\begin{aligned}
& \models \mathrm{VCG}^{\downarrow}_{\geq j+1}\,[c]\,([\neg G] \cdot inv, g) \\
\wedge\ & \models \mathrm{VCG}\,[c']\,([G] \cdot inv, inv) && (28)\\
\wedge\ & \models \mathrm{VCG}^{\downarrow}_{\leq j-1}\,[c]\,(f, [T] \cdot inv) \\
\wedge\ & \models \mathrm{VCG}^{\downarrow}\,[c']\,([G \wedge T]\,, [T]) && (29)\\
\wedge\ & \models \mathrm{VCG}^{\downarrow}\,[c']\,(\epsilon\ [G \wedge T \wedge v = v_0]\,, [v < v_0]) && (30)\\
\wedge\ & \models \{[G \wedge T]\ \Rrightarrow\ [\mathtt{l} \leq v \leq \mathtt{u}]\}
\end{aligned}$$

we have to conclude that

$$\begin{aligned}
& \models \mathrm{VCG}^{\downarrow}_{\geq j+1}\,[c]\,([\neg G] \cdot inv, g) \\
\wedge\ & \models \mathrm{VCG}\,[c'']\,([G] \cdot inv, inv) && (31)\\
\wedge\ & \models \mathrm{VCG}^{\downarrow}_{\leq j-1}\,[c]\,(f, [T] \cdot inv) \\
\wedge\ & \models \mathrm{VCG}^{\downarrow}\,[c'']\,([G \wedge T]\,, [T]) && (32)\\
\wedge\ & \models \mathrm{VCG}^{\downarrow}\,[c'']\,(\epsilon\ [G \wedge T \wedge v = v_0]\,, [v < v_0]) && (33)\\
\wedge\ & \models \{[G \wedge T]\ \Rrightarrow\ [\mathtt{l} \leq v \leq \mathtt{u}]\}
\end{aligned}$$

We need to prove only Equations (31), (32) and (33) since the others are already part of the premises. But each equation is straightforward since from Equations (25) and (28) we can conclude

$$\models \mathrm{VCG}\,[c'']\,([G] \cdot inv, inv)\,,$$

Also, from Equations (26) and (29) we get

$$\models \mathrm{VCG}^{\downarrow}\,[c'']\,([G \wedge T]\,, [T])\,.$$

Finally, from Equations (27) and (30) we obtain

$$\models \mathrm{VCG}^{\downarrow}\left[c''\right]\left(\epsilon \cdot \left[G \wedge T \wedge v = v_0\right], \left[v < v_0\right]\right).$$

Case [$\rightsquigarrow$ trans$^{\downarrow}$]

Let c^* be a pWhile with its respective sets of specifications $\{\langle f_i, g_i\rangle\}_{i=1,\ldots,n}$ such that $c' \preccurlyeq c^*$ and

$$c \vdash \langle f, g\rangle^{\downarrow} \rightsquigarrow c^* \vdash \{\langle f_i, g_i\rangle\}_{i=1,\ldots,n} \tag{34}$$

$$c^* \vdash \langle f_i, g_i\rangle \rightsquigarrow c' \vdash \{\langle f_{i,j}, g_{i,j}\rangle\}_{j=1,\ldots,m_i} \quad \forall i = 1, \ldots, n \tag{35}$$

We must show that $\{f\}\ c\left[c^*/c^*\left[c'/c''\right]\right] \preccurlyeq c\ \{g\}^{\downarrow}$. Let us consider a pair of arbitrary $i \in [1, \ldots, n]$ and $j \in [1, \ldots, m_i]$ and do a case analysis on the kind of local specification $\langle f_i, g_i\rangle$ refers to. If it refers to a total correctness specification, i.e. $c^* \vdash \langle f_i, g_i\rangle^{\downarrow}$, by inductive hypothesis, Equation (35) and the fact that $\{f_{i,j}\}\ c'' \preccurlyeq c'\ \{g_{i,j}\}$ we can conclude that

$$\{f_i\}\ c^*\left[c'/c''\right] \preccurlyeq c^*\ \{g_i\}^{\downarrow}.$$

If on the other hand, $\langle f_i, g_i\rangle$ refers to a partial correctness specification, i.e. $c^* \vdash \langle f_i, g_i\rangle^{\circlearrowleft}$, by Theorem 4.2, Equation (35) and the fact that $\{f_{i,j}\}\ c'' \preccurlyeq c'\ \{g_{i,j}\}$ we get

$$\{f_i\}\ c^*\left[c'/c''\right] \preccurlyeq c^*\ \{g_i\}^{\circlearrowleft}\ .$$

In either case, we obtain a specification-based slice of c^*. It follows that

$$\forall i = 1, \ldots, n, \quad \{f_i\}\ c^*\left[c'/c''\right] \preccurlyeq c^*\ \{g_i\}\ . \tag{36}$$

Finally, from Equations (34) and (36) and by inductive hypothesis, we can conclude that

$$\{f\}\ c\left[c^*/c^*\left[c'/c''\right]\right] \preccurlyeq c\ \{g\}^{\downarrow}$$ □

Annex B. Detailed analysis from Chapter 6

We slice the program from Figure 6.2 w.r.t. post-expectation $g = [x = 1]$ and an arbitrary pre-expectation f. To this end, we start by propagating post-expectation g backward along the program, as shown in Figure B1. Observe that $\mathsf{wpre}_{\geq 5} = \mathsf{wpre}_{\geq 6}$ and $\mathsf{wpre}_{\geq 8} = \mathsf{wpre}_{\geq 9}$. Thus, in view of Theorem 5.1 we can remove instructions i_5 and i_7.

Furthermore, let us consider the local specification induced over the right branch $\{x \coloneqq 0\}$ of the probabilistic choice in the *true* branch of the conditional branching i_7. To compute it, we first compute the local specification induced over the *true* branch of the conditional branching, obtaining post-expectation $[x = 1]$ and pre-expectation $\frac{98}{100}[e = 1]$. This induces itself local specification given by post-expectation $g_7 = [x = 1]$ and pre-expectation $f_7 = 0$ on the right branch of the probabilistic choice (here, we use 0 to denote the constant expectation $\lambda s.0$). Since $f_7 = 0$, it trivially holds that $f_7 \Rrightarrow g_7$ and an application of Theorem 5.1 together with a double application of Theorem 5.2 allows slicing away the whole content of right branch of the probabilistic choice, namely assignment $x \coloneqq 0$.

With a similar reasoning, we can also slice away the right branch of the probabilistic choice in the *false* branch of the conditional branching i_7. All the removable code above identified is colored in red in Figure B1.

Now we slice the same program, but this time w.r.t. post-expectation $g' = [t = 1 \wedge l = 1]$ (and an arbitrary pre-expectation f). Similarly, we propagate the post-expectation backward along the program, obtaining the result in Figure B2. Since $\mathsf{wpre}_{\geq 5} = \mathsf{wpre}_{\geq 9}$, Theorem 5.1 allows deleting the sequence of instructions $i_5; i_6; i_7; i_8$. Computing the local specifications of the right branches of the probabilistic choices in i_3 and i_4 yields:

$$\begin{array}{rcl} f_3 & = & \frac{1}{10}[0 = 1][s = 1] + \frac{1}{100}[0 = 1][s \neq 1] \;=\; 0 \\ g_3 & = & \frac{1}{10}[t = 1][s = 1] + \frac{1}{100}[t = 1][s \neq 1] \\ f_4 & = & [t = 1 \wedge 0 = 1] \;=\; 0 \\ g_4 & = & [t = 1 \wedge l = 1] \end{array}$$

Since $f_3 \Rrightarrow g_3$ and $f_4 \Rrightarrow g_4$, following a similar reasoning as before we can remove the right branches of the probabilistic choices in i_3 and i_4.

$\backslash\backslash\ f$

$\backslash\backslash\ \mathsf{wpre}_{\geq 1} = \frac{21623}{4\cdot 10^6} + \frac{99\cdot 219877}{2\cdot 10^8}$

$i_1 : \{a \coloneqq 1\}\ [1/100]\ \{a \coloneqq 0\};$

$\backslash\backslash\ \mathsf{wpre}_{\geq 2} = \frac{21623}{4\cdot 10^4}\,[a=1] + \frac{219877}{2\cdot 10^6}\,[a\neq 1]$

$i_2 : \{s \coloneqq 1\}\ [1/2]\ \{s \coloneqq 0\};$

$\backslash\backslash\ \mathsf{wpre}_{\geq 3} = f_{3,1}\,[a=1] + f_{3,2}\,[a\neq 1]$

$\backslash\backslash\ f_{3,1} = \frac{1123}{2\cdot 10^3}\,[s=1] + \frac{10393}{2\cdot 10^4}\,[s\neq 1]$

$\backslash\backslash\ f_{3,2} = \frac{15137}{10^5}\,[s=1] + \frac{68507}{10^6}\,[s\neq 1]$

$i_3 :$ if $(a=1)$ then $\{$
$\quad \{t \coloneqq 1\}\ [1/2]\ \{t \coloneqq 0\}$
$\}$ else $\{$
$\quad \{t \coloneqq 1\}\ [1/100]\ \{t \coloneqq 0\}$
$\}$

$\backslash\backslash\ \mathsf{wpre}_{\geq 4} = f_{4,1}\,[s=1] + f_{4,2}\,[s\neq 1]$

$\backslash\backslash\ f_{4,1} = \frac{98}{10^3} + \frac{9\cdot 98}{10^3}\,[t=1] + \frac{9\cdot 5}{10^3}\,[t\neq 1]$

$\backslash\backslash\ f_{4,2} = \frac{98}{10^4} + \frac{99\cdot 98}{10^4}\,[t=1] + \frac{99\cdot 5}{10^4}\,[t\neq 1]$

$i_4 :$ if $(s=1)$ then $\{$
$\quad \{l \coloneqq 1\}\ [1/10]\ \{l \coloneqq 0\}$
$\}$ else $\{$
$\quad \{l \coloneqq 1\}\ [1/100]\ \{l \coloneqq 0\}$
$\}$

$\backslash\backslash\ \mathsf{wpre}_{\geq 5} = \mathsf{wpre}_{\geq 6}$

$i_5 :$ if $(s=1)$ then $\{$
$\quad \{b \coloneqq 1\}\ [6/10]\ \{b \coloneqq 0\}$
$\}$ else $\{$
$\quad \{b \coloneqq 1\}\ [3/10]\ \{b \coloneqq 0\}$
$\}$

$\backslash\backslash\ \mathsf{wpre}_{\geq 6} = f_{6,1} +$

$\backslash\backslash\ [\neg(t=1\wedge l=1)]\ \big(f_{6,2} +$

$\backslash\backslash\ [\neg(t=1\wedge l\neq 1)]\ (f_{6,3} + f_{6,4})\big);$

$\backslash\backslash\ f_{6,1} = \frac{98}{100}\,[t=1\wedge l=1]$

$\backslash\backslash\ f_{6,2} = \frac{98}{100}\,[t=1\wedge l\neq 1]$

$\backslash\backslash\ f_{6,3} = \frac{98}{100}\,[t\neq 1\wedge l=1]$

$\backslash\backslash\ f_{6,4} = \frac{5}{100}\,[\neg(t\neq 1\wedge l=1)]$

$i_6 :$ if $(t=1\wedge l=1)$ then $\{$
$\quad \{e \coloneqq 1\}\ [1]\ \{e \coloneqq 0\}$
$\}$ else if $(t=1\wedge l\neq 1)$ $\{$
$\quad \{e \coloneqq 1\}\ [1]\ \{e \coloneqq 0\}$
$\}$ else if $(t\neq 1\wedge l=1)$ $\{$
$\quad \{e \coloneqq 1\}\ [1]\ \{e \coloneqq 0\}$
$\}$ else $\{$
$\quad \{e \coloneqq 1\}\ [0]\ \{e \coloneqq 0\}$
$\}$

$\backslash\backslash\ \mathsf{wpre}_{\geq 7} = \frac{98}{100}\,[e=1] + \frac{5}{100}\,[e\neq 1]$

$i_7 :$ if $(e=1)$ then $\{$
$\quad \{x \coloneqq 1\}\ [98/100]\ \{\backslash\backslash\ f_7\ x \coloneqq 0\ \backslash\backslash\ g_7\}$
$\}$ else $\{$
$\quad \{x \coloneqq 1\}\ [5/100]\ \{\backslash\backslash\ f_7\ x \coloneqq 0\ \backslash\backslash\ g_7\}$
$\}$

$\backslash\backslash\ \mathsf{wpre}_{\geq 8} = [x=1]$

$i_8 :$ if $(e=1\wedge b=1)$ then $\{$
$\quad \{d \coloneqq 1\}\ [9/10]\ \{d \coloneqq 0\}$
$\}$ else if $(e=1\wedge b\neq 1)$ $\{$
$\quad \{d \coloneqq 1\}\ [7/10]\ \{d \coloneqq 0\}$
$\}$ else if $(e\neq 1\wedge b=1)$ $\{$
$\quad \{d \coloneqq 1\}\ [8/10]\ \{d \coloneqq 0\}$
$\}$ else $\{$
$\quad \{d \coloneqq 1\}\ [1/10]\ \{d \coloneqq 0\}$
$\}$

$\backslash\backslash\ \mathsf{wpre}_{\geq 9} = g = [x=1]$

Figure B1: Program from Figure 6.2, annotated with the backward propagation of post-expectation $g = [x = 1]$ and the local specification induced over the right branches of the probabilistic choices in i_7. Code in red can be sliced away when considering post-expectation g.

$\backslash\backslash$ wpre$_{\geq 1} = \frac{11}{4\cdot 10^4} + \frac{99\cdot 11}{2\cdot 10^6}$

i_1 : $\{a \coloneqq 1\}\ [1/100]\ \{a \coloneqq 0\}$;

$\backslash\backslash$ wpre$_{\geq 2} = \frac{11}{4\cdot 10^2}\,[a = 1] + \frac{11}{2\cdot 10^4}\,[a \neq 1]$

i_2 : $\{s \coloneqq 1\}\ [1/2]\ \{s \coloneqq 0\}$;

$\backslash\backslash$ wpre$_{\geq 3} = f_{3,1}\,[a = 1] + f_{3,2}\,[a \neq 1]$

$\backslash\backslash$ $f_{3,1} = \frac{1}{2}(\frac{1}{10}\,[s = 1] + \frac{1}{100}\,[s \neq 1])$

$\backslash\backslash$ $f_{3,2} = \frac{1}{100}(\frac{1}{10}\,[s = 1] + \frac{1}{100}\,[s \neq 1])$

i_3 : if $(a = 1)$ then {
  $\{t \coloneqq 1\}\ [1/2]\ \{\backslash\backslash\ f_3\ t \coloneqq 0\ \backslash\backslash\ g_3\}$
} else {
  $\{t \coloneqq 1\}\ [1/100]\ \{\backslash\backslash\ f_3\ t \coloneqq 0\ \backslash\backslash\ g_3\}$
}

$\backslash\backslash$ wpre$_{\geq 4} = f_{4,1}\,[s = 1] + f_{4,2}\,[s \neq 1]$

$\backslash\backslash$ $f_{4,1} = \frac{1}{10}\,[t = 1]$

$\backslash\backslash$ $f_{4,2} = \frac{1}{100}\,[t = 1]$

i_4 : if $(s = 1)$ then {
  $\{l \coloneqq 1\}\ [1/10]\ \{\backslash\backslash\ f_4\ l \coloneqq 0\ \backslash\backslash\ g_4\}$
} else {
  $\{l \coloneqq 1\}\ [1/100]\ \{\backslash\backslash\ f_4\ l \coloneqq 0\ \backslash\backslash\ g_4\}$
}

$\backslash\backslash$ wpre$_{\geq 5} = [t = 1 \wedge l = 1]$

i_5 : if $(s = 1)$ then {
  $\{b \coloneqq 1\}\ [6/10]\ \{b \coloneqq 0\}$
} else {
  $\{b \coloneqq 1\}\ [3/10]\ \{b \coloneqq 0\}$
}

$\backslash\backslash$ wpre$_{\geq 6} = [t = 1 \wedge l = 1]$

i_6 : if $(t = 1 \wedge l = 1)$ then {
  $\{e \coloneqq 1\}\ [1]\ \{e \coloneqq 0\}$
} else if $(t = 1 \wedge l \neq 1)$ {
  $\{e \coloneqq 1\}\ [1]\ \{e \coloneqq 0\}$
} else if $(t \neq 1 \wedge l = 1)$ {
  $\{e \coloneqq 1\}\ [1]\ \{e \coloneqq 0\}$
} else {
  $\{e \coloneqq 1\}\ [0]\ \{e \coloneqq 0\}$
}

$\backslash\backslash$ wpre$_{\geq 7} = [t = 1 \wedge l = 1]$

i_7 : if $(e = 1)$ then {
  $\{x \coloneqq 1\}\ [98/100]\ \{x \coloneqq 0\}$
} else {
  $\{x \coloneqq 1\}\ [5/100]\ \{x \coloneqq 0\}$
}

$\backslash\backslash$ wpre$_{\geq 8} = [t = 1 \wedge l = 1]$

i_8 : if $(e = 1 \wedge b = 1)$ then {
  $\{d \coloneqq 1\}\ [9/10]\ \{d \coloneqq 0\}$
} else if $(e = 1 \wedge b \neq 1)$ {
  $\{d \coloneqq 1\}\ [7/10]\ \{d \coloneqq 0\}$
} else if $(e \neq 1 \wedge b = 1)$ {
  $\{d \coloneqq 1\}\ [8/10]\ \{d \coloneqq 0\}$
} else {
  $\{d \coloneqq 1\}\ [1/10]\ \{d \coloneqq 0\}$
}

$\backslash\backslash$ wpre$_{\geq 9} = g' = [t = 1 \wedge l = 1]$

Figure B2: Program from Figure 6.3, annotated with the backward propagation of post-expectation $g' = [t = 1 \wedge l = 1]$ and the local specification induced over the right branches of the probabilistic choices in i_3 and i_4. Code in red can be sliced away when considering post-expectation g'.