

Riccardo Bresciani

Probabilistic Program Verification
in the Style of the
Unifying Theories of Programming

Ph.D. 2013

School of Computer Science and Statistics
Trinity College Dublin

Submitted on October 29th, 2012

This thesis has not been submitted as an exercise for a degree at this or any other university. It is entirely the candidate's own work. The candidate agrees that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Riccardo Bresciani

To my wife, Chiara.

Summary

We present a novel framework to reason on programs based on probability distributions on the state space of a program: they are functions from program states to real numbers in the range $[0..1]$, which can be used to represent the probability of a program being in that state.

Such framework can be used to provide an elegant semantics in the style of *UTP* to a variety of programming languages using both probabilistic and nondeterministic constructs: the use of probability distributions allows us to give programs a semantics which is based on homogeneous relations.

The behaviour of probabilistic nondeterministic programs is treated algebraically via this framework, and as a result it is straightforward to derive algebraic expressions for the probability of some properties to hold for a given program.

Moreover our framework unifies all of the different kinds of choice under a single “generic choice” construct, and the usual choice constructs (disjunction, conditional choice, probabilistic choice, and nondeterministic choice) can be viewed as some of its specific instances. Later on we will discuss also other possible specific instances (namely conditional probabilistic choice, switching probabilistic choice, conditional nondeterministic choice, nondeterministic probabilistic choice, and fair nondeterministic choice).

The use of probability allows us to introduce the notion of probabilistic refinement, which generalises the traditional one: this is important in view of formal verification of probabilistic properties of programs via refinement-based techniques.

Acknowledgements

I wish to thank Andrew Butterfield and all of the guys from the FMG, without whom this work would not have been possible.

Obviously my friends and family have played a big part in this. Not long ago it took me a while to write up the acknowledgements for my master thesis, and now I would probably repeat the same things to the same people: for this reason allow me to simply condense my gratitude in a big thanks to you all.

The present work has emanated from research supported by Science Foundation Ireland grant 08/RFP/CMS1277 and, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero — the Irish Software Engineering Research Centre.

Contents

1	Introduction	1
1.1	Our approach	4
1.1.1	Key contributions	4
1.1.2	Organization of this thesis	5
2	Background and related work	7
2.1	Kozen's framework	7
2.2	<i>pGCL</i>	11
2.3	<i>pCSP</i>	17
2.4	<i>UTP</i>	18
2.4.1	Theory of Designs	19
2.4.2	Probabilistic <i>UTP</i>	21
3	A framework to deal with probability distributions over the state space	23
3.1	States and distributions, informally	23
3.2	Definitions	26
3.3	Programs	27
3.3.1	Deterministic programs	28
3.4	Nondeterminism	30
3.4.1	A generic choice construct	32
3.4.2	Program structure	34
3.5	Healthiness conditions	34
3.6	The program lattice	35
3.7	Refinement	36
3.7.1	Probabilistic refinement	38
3.8	Summary	38
4	<i>pGCL</i>	41
4.1	Interaction of probabilistic and nondeterministic choice	44
5	A probabilistic theory of designs	47
5.1	Healthiness conditions	49
5.2	Recasting total correctness	50
5.3	Link with the standard model	51
5.3.1	Weakening the link	51
5.4	Considerations on a <i>pCSP</i> theory	52
5.4.1	R1	53
5.4.2	R2	54
5.4.3	R3	54
5.4.4	CSP1 and CSP2	55

6 Conclusion	57
A States and distributions	59
A.1 Variables, types and expressions	59
A.2 States	59
A.2.1 Evaluation of an expression	60
A.2.2 Abstract states	62
A.2.3 Assignments	63
A.3 Distributions	64
A.3.1 Operations on distributions	66
A.3.2 Specific types of distributions	67
A.3.3 A simpler notation	68
A.3.4 The <i>remap</i> operator	68
B Distributions as vectors	71
B.1 Operations on vectors	71
B.1.1 The set \mathcal{D}_p	73
B.2 Programs as matrices	73
B.2.1 Interpretation of the columns of the program matrix	74
B.2.2 Random Variables and <i>pGCL</i> Expectations	75
B.2.3 Interpretation of the rows of the program matrix	75
B.2.4 Probability of an event	76
B.3 Deterministic Programs	77
B.3.1 Some considerations on loops	78
B.3.2 Healthiness conditions	79
B.4 Nondeterministic choice	83
C Proofs	91
D Other case studies	109
D.1 Monty Hall	109
D.2 Rabin's choice coordination algorithm	134
D.3 Protocol verification	140
D.3.1 The Dolev-Yao Model	141
D.3.2 A strategy to evaluate the probability of successful attacks by means of standard protocol verifiers	144
D.3.3 An example: using ProVerif to verify the Yahalom protocol	145
D.3.4 Protocol runs as predicates	148
D.3.5 An example: key-guessing on the Yahalom protocol	150
D.3.6 Towards a <i>UTP</i> -style protocol verification technique	151
E Notation	153
F Mathematical Background	157
F.1 General Notions	157
F.2 Vector spaces	157
F.2.1 The vector space \mathbb{R}^N	160
F.2.2 The vector space $\mathbb{R}^{N \times N}$	160

F.3	Boolean algebra	160
F.4	Measure Theory	161
F.5	Probability Theory	161

CHAPTER 1

Introduction

Formal verification is now mainstream in computer science: the task of establishing if a given program behaves according to its specification is now a routine step, because of its higher reliability compared to tests, coupled with time- and money-saving possibilities deriving from a development approach based on formal methods.

The research community has played an important role in the development of current formal methods, and its effort has not stopped. As a result of all the different approaches adopted towards formal verification, nowadays we have a variety of available techniques: the advantage is that for each of the many verification scenarios a particular technique may prove more efficient than others.

On the other hand the disadvantage is interoperability of these techniques, which would be quite a desirable feature — it is absolutely standard to use several different techniques towards the verification of different parts of the same system.

Being able to use different models together is the aim of the *Unifying Theories of Programming (UTP)*, which rely on predicate logic to give a semantics to different languages, so that unification can happen at the level of the common underlying semantics.

The focus of our work is *UTP*, in particular we want to come up with an approach that allows us to treat probabilistic programs in an analogous way as standard nondeterministic programs are treated in *UTP*.

There are several reasons for wanting probability in the picture.

A model including probability can offer a more precise description of a system, allowing the verification procedure to assert that a given property *is verified with probability* p rather than simply asserting that the property *may be verified*.

An example is a system with two alternative behaviours A and B : if we knew nothing more than this, the only way we would be able to formalise is by merely saying that the system may show behaviours A and B *but* we cannot make any other kind of forecasts on the actual observable behaviour — we could write this using Dijkstra's nondeterministic choice operator \sqcap as $A \sqcap B$.

What if we had also some kind of statistical characterization of the system, for example that A and B happen randomly half of the time? It would be an unnecessary and detrimental waste of information not to take this into account and model the system again with nondeterministic choice: a more desirable option is something saying that both A and B happen with probability 0.5 — we could write something in the style of $A \text{ }_{0.5} \oplus B$.

Sometimes a good statistical behaviour is what makes things usable. Actually, that is always the real-world case: any appliance or machine that we use is not guaranteed to work every time we want it to, it is just very likely that it will work.

Statistical observations are what draws a line between good appliances and bad ones: from

an observer's perspective the functional difference between a Trabant and a BMW is based on statistics, one car being probabilistically less reliable than the other.

If we want to describe what happens when the driver turns the ignition key, simply saying that the engine may (or may not) start running is not a good enough description, a probabilistic information saying how likely it is that the car turns on is highly desirable.

We obviously have some expectations when we compare a Trabant with a BMW, as we assume that the components in one car are more reliable than the ones in the other car: we know that if we assemble correctly reliable components we will obtain a reliable car.

We have no numerical idea about the failure rate for the components used in each car, but an engineer does: from his perspective the observations on the behaviour of the car can be made at a lower level, directly on the parts, and this would allow him to infer the probability of some behaviour (assuming that there are no other design flaws).

This is another reason why it is interesting to talk about probability: the rules for deriving the probability of composite events from those of the single events are well established and understood. As a result the overall probability that a certain property holds in a system can be inferred bottom-up, starting from the probability of the relevant events.

An analogous perspective is that of the interaction between software and hardware: when we model the behaviour of a program the implicit underlying assumption is that the hardware is working properly. It would be interesting to integrate in the model also some information concerning possible hardware failures, which are to be characterised statistically.

An example is the use of flash memories: the physical principles they are based on are quite brutal (informally speaking, electrons are kicked through a barrier to store information, which can be then erased by a strong current flow that resets the whole memory block) so failures are of usual occurrence.

Having a framework which is capable to handle probabilistic information would be of great value here, for example each write operation to a flash memory could be rendered as:

$$\text{write}(x) \triangleq \text{successful_write}(x) \text{ }_p\oplus \text{write_error}$$

for an appropriate p , which can vary depending on number of previous writes and time.

From a different point of view, sometimes probability is the very reason why things do work. Miller-Rabin primality test is an example, and plenty of other examples of interest can be found in Computer Science, but also in everyday's situations we rely on this: imperfect systems can be made more reliable through redundancy, both in terms of physical duplication of components and/or repetition of measures and experiments.

There are several pitfalls when dealing with probability, as sometimes the solution to some problems is quite confusing and counter-intuitive. Here are a few examples.

Think of a man who is the father of two children:

- what is the probability that both of them are girls?
- knowing that one of them is a girl, what is the probability that both of them are girls?
- knowing that the older is a girl, what is the probability that both of them are girls?

The difference between the first question and the following two is apparent, but it may be not

so obvious what the difference between the last two. Assuming that there is a 50% probability for each child to be male or female, the answers are:

- $1/4$;
- $1/3$;
- $1/2$.

The middle one is probably the most surprising answer.

This example highlights the subjective component of probability, as it varies depending on how well we know a situation we are talking about.

Another example is the (in)famous Monty Hall game: the setting is a TV show, where a participant in front of three closed doors is given the chance to win a car if he guesses the door, which the car has been hidden behind. Behind the doors there are two goats and a car, so the probability to choose the “right” door is $1/3$.

But what if Monty Hall (the host) opens one of the remaining doors, thus revealing where one of the goat has been hidden, and offers the participant the possibility to change his mind and switch his choice to the other closed door? Should he do it?

The answer is yes, because in this way he will double his chances to win, jumping to a nice $2/3$. This might sound a bit surprising at a first glance.

When trying to model a situation involving probability, the first and most important issue is the decomposition of such situation into events, distinguishing atomic events from composite ones. Atomic events are mutually independent, whereas composite events are a combination/union of atomic events.

For example if we pick a random natural number in the range $[1..10]$ we can say that:

- the probability of $x = 2$ is $1/10$;
- the probability of x being even is $1/2$;
- the probability of $x = 2$ and x being even is $1/10 \cdot 1/2 = 1/20$.

Or is it?!

The answer $1/20$ is wrong, because we have considered the two events “ $x = 2$ ” and “ x being even” as independent: in fact “ x being even” is a composite event which can be rewritten in this case as “ $x = 2$ or $x = 4$ or $x = 6$ or $x = 8$ or $x = 10$ ” — after this observation it is clear that x being even is always true when $x = 2$ (i.e. the conditional probability of x being even when $x = 2$ is 1), so the correct answer is $1/10$.

Sometimes two events appear to be independent, but in reality there is a (possibly hidden) relation. A trivial example is a simple program that picks a random number for x (say either 0 or 1) and then assigns the new value of x to y .

Clearly the probability of $x = y$ is 1, but would we be able to tell it if we were simply given *separately* the probability distribution for each value for each variable, like:

$$\mathcal{P}(x = 0) = 1/2 \quad \mathcal{P}(x = 1) = 1/2 \quad \mathcal{P}(y = 0) = 1/2 \quad \mathcal{P}(y = 1) = 1/2.$$

In this case our answer would be probably that the probability is

$$\mathcal{P}(x = 0) \cdot \mathcal{P}(y = 0) + \mathcal{P}(x = 1) \cdot \mathcal{P}(y = 1) = 1/2.$$

This is because we lost the “entanglement” between the variables that was created by the program.

1.1 Our approach

We want to give a brief overview of our approach, in order to give the reader an intuition without getting bogged down in details — which will be presented extensively in §3.

With the last example in mind, it is clear that if we want to give a probabilistic description of a program, we cannot deal with variables separately, but rather treat them as bundled into a single entity, which we call “state”.

Although the concept of state is very general, it can be thought of as a snapshot of the current memory content (what is in the RAM, in the processor registries, in the hard drive, and so on).

Each state can be assigned a probability, which corresponds to the probability of the program being in that state: in this way we create a *probability mass distribution*, or shortly a *probability distribution*, on the state space of a program, so that we can reason on the probability distribution (and its evolution) to understand the program behaviour.

Lumping variables together poses some serious challenges to track the evolution of a system: when a state evolves to another state, the associated probability (or a part thereof) has to be “transferred” to the new state. This is nontrivial in the case of several states evolving into the same state, so that the associated probabilities (or a part thereof) have to be summed together and “transferred” to the new state.

This is quite a common occurrence, for example this happens when there is an assignment operation.

This framework has interesting algebraic properties, as the set of probability distributions is a precise part of a vector space made of more general distributions (*i.e.* those distributions that map states to real numbers, but that are not necessary a probability mass distribution), which is isomorphic to \mathbb{R}^n , with n equal to the number of states.

Programs can consequently be seen as *distribution transformers*: they take an initial distribution (*before-distribution*) and transform it into a final distribution (*after-distribution*) that accounts for the changes made by the program.

In the case of deterministic programs, the corresponding space has interesting properties as well, as it is isomorphic to a portion of $\mathbb{R}^n \times \mathbb{R}^n$.

Nondeterminism arises when a program is entitled to choose internally between different alternative behaviours: as a result a single before-distribution can evolve to different possible after-distributions, all of which are equally valid and no forecasts on the actual outcome is possible.

As a result a nondeterministic program relates a single before-distribution to a set of after-distributions.

This allows us to see programs as predicates in the style of *UTP*, which are based on homogeneous relations among distributions: we are going to give a predicate semantics to a set of common constructs, and use this to reason on programs with the rules of predicate logic.

1.1.1 Key contributions

The key contributions of this work are:

- a novel framework to reason on programs based on probability distributions on the state space of a program: they are functions from program states to real numbers in the range $[0..1]$, which can be used to represent the probability of a program being in that state;
- such framework can be used to provide an elegant semantics in the style of *UTP* to a variety of programming languages using both probabilistic and nondeterministic constructs: the use of probability distributions allows us to give programs a semantics which is based on homogeneous relations. For this reason we believe that we took important steps towards...

... the so-far-unachieved goal of unifying probabilism with other programming constructs in the style of Unifying Theories of Programming.

Chen and Sanders [CS09]

- such framework allows us to treat algebraically the behaviour of probabilistic nondeterministic programs, and as a result it is straightforward to derive algebraic expressions for the probability of some properties to hold for a given program;
- moreover our framework unifies all of the different kinds of choice under a single “generic choice” construct, and the usual choice constructs (disjunction, conditional choice, probabilistic choice, and nondeterministic choice) can be viewed as some of its specific instances. Later on we will discuss also other possible specific instances, namely:
 - conditional probabilistic choice;
 - switching probabilistic choice;
 - conditional nondeterministic choice;
 - nondeterministic probabilistic choice;
 - fair nondeterministic choice.
- the use of probability allows us to introduce the notion of probabilistic refinement, which generalises the traditional one: this is important in view of formal verification of probabilistic properties of programs via refinement-based techniques.

1.1.2 Organization of this thesis

We are going to present the core background material which constitutes the foundations and main references for this thesis in §2.

Chapter 3 and Appendices A and B are dedicated to a detailed presentation of our framework, and some case studies are presented in Chapters 4 and 5 and Appendix D; some of the material

from these chapters has been previously published as part of the work emanated from this research [BB09; BB11; BPB11; BB12a; BB12b; BB12c].

We conclude in §6 and include other appendices on mathematical background, proofs and notation, which the reader can refer to when necessary.

CHAPTER 2

Background and related work

In this chapter we are going to present the background material relevant to this thesis and the related work; we assume familiarity with all of the underlying mathematics (linear algebra, measure theory and probability theory), which is anyway briefly presented in Appendix F.

The topic of probability in computer science has been addressed within different scopes in a variety of different ways, including the Dempster-Shafer belief theory [Dem68; Sha76; Jøs01; Koh03], Bayesian networks [Pea88; FHM90], probabilistic argumentation [Hae⁺01], logical/relational Markov models [DK03; JKB07], and probabilistic powerdomain techniques [JP89; Jon90].

Our approach to probability builds on higher-level work relying on Markov models and probabilistic powerdomains, and in particular the main references are Dexter Kozen’s framework [Koz81; Koz85] and the *pGCL* framework [MM04]; such an approach yields a *UTP*-style framework where nondeterminism and probabilism coexist.

Kozen’s framework, *pGCL* and *UTP* are our three main reference areas: although the notation used in the different references varies, we will try to uniform it for the sake of understandability — refer to Appendix E for the notation used.

2.1 Kozen’s framework

In the early 1980s Dexter Kozen proposed a formalism to reason about probabilistic programs [Koz81; Koz85], with an approach which is very different from conventional logic:

Unfortunately, almost all of our logical apparatus belongs to the nondeterministic form. The usual logical connectives and the existential quantifier are clearly nondeterministic in nature. We must therefore be prepared to depart radically from conventional logic in order to accommodate probability in a satisfactory way.

Kozen [Koz85]

Dijkstra’s nondeterministic choice is therefore left out in Kozen’s approach, and replaced by probabilistic choice: as we will see later on, this has profound implications.

The motivation for Kozen’s work was providing a common framework to unify the two mainstream approaches of the late 1970s, *i.e.* the *distributional approach* and the *randomized approach*, and to analyse probabilistic programs, which had been previously analysed exclusively by *ad hoc* methods.

The distributional approach sees a program as being deterministic, and probabilism emerges from a probability distribution on the input; the randomized approach allows a program to

take probabilistic steps, but the input is fixed. Yao proved the equivalence of these approaches. [Yao77]

The roots of Kozen's approach go down to the theory of linear operators in Banach spaces: a probabilistic program is in fact interpreted as a continuous linear operator on a Banach space of (probability) distributions.

Kozen deals with *probabilistic while programs* in [Koz81], which act over the variables v_1, v_2, \dots, v_n (all of the same type \mathcal{W} for the sake of simplicity) and use the following constructs:

- *assignment*: $v_i := e(v_1, v_2, \dots, v_n)$, where the expression e , which is a function of the program variables, is evaluated in the current state and the resulting value is assigned to v_i ;
- *random assignment*: $v_i := \text{random}$, where *random* is a function returning a random variate from some random variable of the appropriate type¹;
- *sequential composition*: $A ; B$, which executes the program B after A has terminated;
- *conditional choice*: $A \triangleleft c \triangleright B$, which executes A or B depending on the evaluation of the condition c , which is a boolean expression²;
- (*while*) *loop*: $c * A$, which executes the body of the loop A as long as c holds true.

In semantics 1 of [Koz81] program variables are seen as random variables on the probability space $(\mathcal{S}, \Sigma_{\mathcal{S}}, \mu_{\mathcal{S}})$, all of which have the same value space $(\mathcal{W}, \Sigma_{\mathcal{W}})$, where:

- $\Sigma_{\mathcal{S}} = \{\alpha_1, \alpha_2, \dots\} \subseteq \wp \mathcal{S}$ and $\Sigma_{\mathcal{W}} \subseteq \wp \mathcal{W}$ are σ -algebras defined on the state space \mathcal{S} and on the variable type \mathcal{W} ;
- $\mu_{\mathcal{S}} : \Sigma_{\mathcal{S}} \rightarrow [0..1]$ is a probability measure on the measurable space $(\mathcal{S}, \Sigma_{\mathcal{S}})$.

We have that the functional composition of the probability measure $\mu_{\mathcal{S}}$ after the random variable v_i defines a probability measure on $(\mathcal{W}, \Sigma_{\mathcal{W}})$:

$$\mu_{\mathcal{W}} \triangleq \mu_{\mathcal{S}} \circ v_i^{-1}.$$

The *random vector* $\underline{v} : (\mathcal{S}, \Sigma_{\mathcal{S}}, \mu_{\mathcal{S}}) \rightarrow (\mathcal{W}, \Sigma_{\mathcal{W}})$, where $\mathcal{W} = \mathcal{W}^n$ and $\Sigma_{\mathcal{W}} = \{\underline{\beta}_1, \underline{\beta}_2, \dots\} \subseteq \wp \mathcal{W}$, is a vectorial function whose i -th component is the i -th random variable; we can show that \underline{v} induces an isomorphism between the measurable spaces $(\mathcal{S}, \Sigma_{\mathcal{S}})$ and $(\mathcal{W}, \Sigma_{\mathcal{W}})$.

Similarly as above, the functional composition of the probability measure $\mu_{\mathcal{S}}$ after the random vector \underline{v} defines the joint distribution for input variables:

$$\mu_{\mathcal{W}} \triangleq \mu_{\mathcal{S}} \circ \underline{v}^{-1}.$$

¹Kozen's view of things in semantics 1 of [Koz81] is actually based on an infinite stack of random numbers, that serves as a random generator such that "each time $v_i := \text{random}$ is executed, the next random number is popped from the stack and assigned to v_i ". The presentation of semantics 1 here is amended in order to avoid this complication: it is possible to remove this by choosing to identify random vectors with the same distribution, according to Kozen's observation at the end of the presentation of this semantics.

²A boolean expression c will evaluate to *true* or *false* when \underline{v} is mapped to the elements of a subset $\underline{\beta}_c$ of \mathcal{W} or to $\underline{\beta}_{\bar{c}} = \mathcal{W} \setminus \underline{\beta}_c$, respectively.

A program A can be seen as a *partial measurable linear function* $\mathcal{L}_A : \underline{\mathcal{W}} \rightarrow \underline{\mathcal{W}}$ on the value space, which accounts for the changes made by A to the configuration of variables; it is therefore possible to express the joint distribution for the output we obtain after running program A as:

$$\mu'_{\underline{\mathcal{W}}} \triangleq \mu_S \circ \underline{v}^{-1} \circ \mathcal{L}_A^{-1} = \mu_{\underline{\mathcal{W}}} \circ \mathcal{L}_A^{-1}.$$

In view of a slightly different semantics that appeared later in [Koz85] (and which is going to be presented below), it is useful to define now the probability measure

$$\mu'_S \triangleq \mu_S \circ \text{Inv}_A,$$

where $\text{Inv}_A \triangleq \underline{v}^{-1} \circ \mathcal{L}_A^{-1} \circ \underline{v}$: the function Inv_A on (S, Σ_S) corresponds to the function \mathcal{L}_A^{-1} on $(\underline{\mathcal{W}}, \Sigma_{\underline{\mathcal{W}}})$ under the isomorphism induced by \underline{v} , and this implies that

$$\mu'_{\underline{\mathcal{W}}} = \mu'_S \circ \underline{v}^{-1}.$$

Semantics 2 from [Koz81] sees a program A as a homeomorphism on the set of all possible joint distributions of the program variables (including all linear combinations), or equivalently as a homeomorphism \mathcal{H}_A on the set $M_{\underline{\mathcal{W}}}$ of all possible probability measures on the measurable space $(\underline{\mathcal{W}}, \Sigma_{\underline{\mathcal{W}}})$: therefore a program transforms a measure $\mu_{\underline{\mathcal{W}}}$ accounting for the initial variable distribution into a measure $\mu'_{\underline{\mathcal{W}}} = \mathcal{H}_A(\mu_{\underline{\mathcal{W}}})$ accounting for the final variable distribution after the execution of A — the notation A is used both for the program .

$(M_{\underline{\mathcal{W}}}, \|\cdot\|, \leq)$, where $\|\cdot\|$ is the total variation norm and \leq is the complete partial order induced by the positive cone $M_{\underline{\mathcal{W}}}^+$ of $M_{\underline{\mathcal{W}}}$, is a *conditionally complete Banach lattice*, where the internal operations are defined as follows:

$$\begin{aligned} (\mu_{\underline{\mathcal{W}},i} + \mu_{\underline{\mathcal{W}},j})(\underline{\beta}) &= \mu_{\underline{\mathcal{W}},i}(\underline{\beta}) + \mu_{\underline{\mathcal{W}},j}(\underline{\beta}) \\ (\alpha\mu_{\underline{\mathcal{W}}})(\underline{\beta}) &= \alpha(\mu_{\underline{\mathcal{W}}}(\underline{\beta})). \end{aligned}$$

The space \mathcal{P} of programs, with addition and scalar multiplication extended point-wise, forms also a Banach space together with the uniform norm, which is defined $\|\mathcal{H}_A\|_\infty \triangleq \sup_{\|\mu_{\underline{\mathcal{W}}}\| \leq 1} \{\|\mathcal{H}_A(\mu_{\underline{\mathcal{W}}})\|\}$.

The intuition behind this approach is as follows. The program variables v_1, \dots, v_n satisfy some joint distribution $\mu_{\mathcal{W}}$ on input. We will forget the variables themselves and concentrate on the distribution $\mu_{\mathcal{W}}$. We can think of $\mu_{\mathcal{W}}$ as a fluid mass distributed throughout \mathcal{W} . This mass is concentrated more densely in some areas than others, depending on which inputs are more likely to occur. Execution of a simple or random assignment redistributes the mass in \mathcal{W} . Conditional tests cause the mass to split apart, and the two sides of the conditional are executed on the two pieces. In the while loop, the mass goes around and around the loop; at each cycle, the part of the mass which occupies β_c breaks off and exits the loop, and the rest goes around again. Part of the mass may go around infinitely often. Thus, at any point in time, there are different pieces of the mass that occupy different parts of the program, and each piece is spread throughout the domain according to the simple and random assignments that have occurred in its history. Different pieces that have come to occupy the same parts of the program through different paths are accumulated. At certain points in time, parts of the mass find their way out of the program. The output distribution $\mathcal{H}_A(\mu_{\mathcal{W}})$ is the sum of all the pieces that eventually find their way out. Thus the probability that program A halts on input distribution μ is $\mathcal{H}_A(\mu_{\mathcal{W}})(\mathcal{W})$, the probability of the universal event \mathcal{W} upon output.

adapted from Kozen [Koz81]

Subprobability measures are all those positive ones whose norm does not exceed 1, which are those belonging to the set $\mathcal{P} \triangleq M_{\mathcal{W}}^+ \cap \mathcal{B}_0[1]$.

It shall be noted that, as probability measures are those with unitary norm, *viz.* belonging to the boundary $\partial\mathcal{B}_0(1)$ of the unit ball, the set of all positive probability measures is $\tilde{\mathcal{P}} \triangleq M_{\mathcal{W}}^+ \cap \partial\mathcal{B}_0[1]$.

A program A can therefore be seen as a function $\mathcal{H}_A : \mathcal{P} \rightarrow \mathcal{P}$, which maps a probability measure to a subprobability measure³. This function can be extended to be applicable on the whole $M_{\mathcal{W}}$: such extension is a $\|\cdot\|$ -bounded continuous linear transformation $M_{\mathcal{W}} \rightarrow M_{\mathcal{W}}$.

As mentioned above, the space \mathcal{P} is a Banach one: its subset \mathcal{P}^+ of monotone elements induces an order \sqsubseteq on \mathcal{P} — which is the point-wise lifting of the order \leq on measures.

The semantics for the program constructs is the following:

- in the case of the assignment $v_i := e(v_1, v_2, \dots, v_n)$, the corresponding transformation is:

$$\mathcal{H}_e(\mu_{\mathcal{W}}) = \mu \circ \mathcal{L}_e^{-1},$$

where $\mathcal{L}_e : \mathcal{W} \rightarrow \mathcal{W}$ is the function

$$\mathcal{L}_e(v_1, v_2, \dots, v_n) = (v_1, v_2, \dots, v_{i-1}, e(v_1, v_2, \dots, v_n), v_{i+1}, \dots, v_n);$$

³Nevertheless more in general we can see them as a homeomorphism on the set of subprobability measures, as when the function representing the program is applied to a subprobability measure it returns a subprobability measure whose norm is no larger than that of the function argument.

- the random assignment $v_i := \text{random}$

$$\mathcal{H}_{\text{random}}(\underline{\mu}_{\mathcal{W}})(\beta_1 \times \beta_2 \times \cdots \times \beta_n) = \underline{\mu}_{\mathcal{W}}(\beta_1 \times \beta_2 \times \cdots \times \beta_{i-1} \times \mathcal{W} \times \beta_{i+1} \times \cdots \times \beta_n) \rho(\beta_i);$$

where $\beta_1, \beta_2, \dots, \beta_n \in \Sigma_{\mathcal{W}}$ and ρ is the probability distribution for the random number generator — the random assignment alters the measure β_i used to have before its execution, as the distribution of the i -th variable changes causing $\rho(\beta_i)$ to be the new measure of β_i ;

- the sequential composition $A; B$ yields the functional composition $\mathcal{H}_B \circ \mathcal{H}_A$;
- the conditional choice $A \triangleleft c \triangleright B$ is:

$$\mathcal{H}_{\text{if}}(\underline{\mu}_{\mathcal{W}})(\underline{\beta}) = \mathcal{H}_A \circ \underline{\mu}_{\mathcal{W}}(\underline{\beta}_c \cap \underline{\beta}) + \mathcal{H}_B \circ \underline{\mu}_{\mathcal{W}}(\underline{\beta}_{\bar{c}} \cap \underline{\beta}),$$

where β_c and $\beta_{\bar{c}}$ are a partition of \mathcal{W} : in these sets the condition c evaluates to *true* and *false* respectively — it is therefore clear how the measure is transformed via \mathcal{H}_A on the part of $\underline{\beta}$ where c is true and via \mathcal{H}_B on the part of $\underline{\beta}$ where c is false;

- the loop $c * A$ can be interpreted using the *least fixed point* operator:

$$\mathcal{H}_{\text{while}}(\underline{\mu}_{\mathcal{W}})(\underline{\beta}) = \text{Lfp } \mathcal{X}(\underline{\mu}_{\mathcal{W}})(\underline{\beta}) \bullet (\mathcal{X} \circ \mathcal{H}_A \circ \underline{\mu}_{\mathcal{W}}(\underline{\beta}_c \cap \underline{\beta}) + \underline{\mu}_{\mathcal{W}}(\underline{\beta}_{\bar{c}} \cap \underline{\beta})),$$

where, in the right-hand side, a construct similar to the conditional choice is clearly recognisable: this is because the bracketed term was obtained by unfolding the loop once — the existence of the least fixed point is guaranteed by the fact that the space of programs \mathcal{P} is a Banach lattice.

These ideas lead to the presentation in [Koz85], where programs are seen as *Markov transitions* (or *measurable kernels*), which are functions $p : \mathcal{S} \times \Sigma_{\mathcal{S}} \rightarrow \mathbb{R}$ satisfying the properties:

1. $f_{\alpha'}(\sigma) \triangleq p(\sigma, \alpha')$ is a bounded measurable function $f_{\alpha'} : \mathcal{S} \rightarrow \mathbb{R}$ on the measurable space $(\mathcal{S}, \Sigma_{\mathcal{S}})$ — let F denote the space of all such functions;
2. $\mu_{\sigma}(\alpha') \triangleq p(\sigma, \alpha')$ is a finite measure $\mu_{\sigma} : \Sigma_{\mathcal{S}} \rightarrow \mathbb{R}$ on $(\mathcal{S}, \Sigma_{\mathcal{S}})$ — let M denote the space of all such functions.

The Markov transition $p(\sigma, \alpha')$ maps a pair, formed by a state and a set of states, to a real number: with an appropriate choice of p , we can use a Markov transition to express the probability $p_A(\sigma, \alpha')$ that a program A ends up in some state $\sigma' \in \alpha'$ when starting in state σ .

With this in mind it is easy to relate this semantics to the measure-transformer semantics of [Koz81], by expressing the relation of a measure on the set of after-states \mathcal{S}' to that on the set of before-states \mathcal{S} as:

$$\mu'_{\mathcal{S}}(\alpha') = \sum_{\sigma \in \mathcal{S}} p_A(\sigma, \alpha') \mu_{\mathcal{S}}(\{\sigma\}).$$

It is also possible to use this to express the expected value $\langle f \rangle$ of a function $f : \mathcal{S}' \rightarrow \mathbb{R}$ on after-states after running a program A from a before-state σ — therefore it is $\langle f \rangle : \mathcal{S} \rightarrow \mathbb{R}$:

$$\langle f \rangle(\sigma) = \sum_{\sigma' \in \mathcal{S}'} p_A(\sigma, \{\sigma'\}) f(\sigma').$$

If f is the characteristic function of a set of states α' , then $\langle f \rangle$ is the probability that $\sigma' \in \alpha'$; if f is the function describing the probability of an event happening when a program halts in a given after-state, then $\langle f \rangle$ is the probability that this event happens when terminating in a state belonging to α' .

A technique by Jones and Plotkin [JP89] can be used to build what they term the *probabilistic powerdomain of evaluations*: they introduce probability into a semantic domain, and thus the behaviour described by Kozen's framework can be reproduced in that setting [Jon90] — this is the basis for the probabilistic predicate-transformer model presented in §2.2 [MMS96; MM04].

2.2 pGCL

The choice operator is the key ingredient of probabilistic systems, and it can be instantiated in three different ways:

- *demonic choice*, that picks the “worst-case” scenario for that choice;
- *angelic choice*, that picks the “best-case” scenario for that choice;
- *probabilistic choice*, that picks one of the two options with a given probability.

Probabilistic choice is a desirable feature in a language, as it is doubtless that a quantitative formal analysis offers great advantages compared to a qualitative one: the challenge is to find a computationally feasible way of dealing with this.

Interactions among demonic, angelic and probabilistic choices may be subtle. In fact a deterministic (although probabilistic) program is characterised by monotonicity, conjunctivity and disjunctivity:

Monotonicity $(P \Rightarrow Q) \Rightarrow (\mathcal{P}(P) \Rightarrow \mathcal{P}(Q))$

Conjunctivity $\mathcal{P}(P \wedge Q) \equiv (\mathcal{P}(P) \wedge \mathcal{P}(Q))$

Disjunctivity $\mathcal{P}(P \vee Q) \equiv (\mathcal{P}(P) \vee \mathcal{P}(Q))$

where P and Q are predicates and \mathcal{P} is a predicate transformer.

When introducing demonic choice we drop disjunctivity; if demonic choice and angelic choice coexist in the same program, we lose also conjunctivity and we remain only with monotonicity. [MM98]

When composing processes one must be careful about the issue of *duplication*, which in presence of probabilistic and nondeterministic choice may lead to incorrect results. [Mor⁺95]

An example is given by the idempotency of the demonic choice operator, which depends on its definition: if the demonic choice operator can distribute through probabilistic choice operators we can have the following behaviour[Mis00]:

$$(A \frac{1}{2} \oplus B) \sqcap (A \frac{1}{2} \oplus B) = A \frac{1}{4} \oplus ((A \sqcap B) \frac{1}{3} \oplus B)$$

The reason for this is that two instances of the same program containing a demonic choice are actually two different programs because of it, as every demonic choice is a unique element.

Another way of seeing this is that it is crucial to know when a choice is made, thus we have to be very careful when we distribute choice operators.

The main shortcoming of Kozen’s approach is that he chooses not to retain nondeterministic choice, which — although being undoubtedly a source of complication — turns out to be a necessary and desirable feature of a programming language:

Dijkstra’s demonic \sqcap was not so easily discarded, however. Far from being “an unnecessary and confusing complication,” it is the very basis of what is now known as refinement and abstraction of programs.

McIver and Morgan [MM04]

In fact refinement and abstraction are the core of formal techniques for software specification and development, and are necessary to derive an implementation from a given specification via the *refinement calculus*.

Before going further on, let us take a step back and present the concept of *guarded commands*, which was introduced by Dijkstra in the 1970s [Dij75; Dij76]: a *guard* is a condition that precedes a command and is evaluated before the command is executed — obviously this happens only in case the guard is true.

The *Guarded Command Language (GCL)* uses the following constructs:

- *abort* is the aborting program;
- *skip* is the program which does nothing and terminates;
- *assignment*: $v_i := e(v_1, v_2, \dots, v_n)$, where the expression e is evaluated in the current state and the resulting value is assigned to v_i ;
- *sequential composition*: $A; B$, which executes the program B after A has terminated;
- *conditional choice*: $A \triangleleft c \triangleright B$, which executes A or B depending on the evaluation of the condition c ;
- *nondeterministic choice*: $A \sqcap B$, which executes A or B nondeterministically, depending on the desired outcome — in the case of *demonic nondeterminism* the executed program is that leading to the less desirable outcome, the one leading to the most desirable outcome in the case of *angelic nondeterminism*;
- (*while*) *loop*: $c * A$, which executes the body of the loop A as long as c holds true.

In Dijkstra’s work, *GCL* is given a semantics via the so-called *weakest precondition*, which is a predicate $wp.A.Post$ that is true in those *initial states* that guarantee that the postcondition $Post$ will be reached after running A ⁴.

The work by Morgan, McIver *et al.* leads to a probabilistic version of *GCL*, namely *pGCL* [MM97; MM04; MM05].

Our simple programming language will be Dijkstra’s, but with \oplus added and — crucially — demonic choice \sqcap retained: we call it pGCL.

⁴It is possible to use a *Hoare triple* to express the same concept: $\{Pre\}A\{Post\}$.

Their approach to probabilistic systems is based on what they term *expectations*, which are used in place of standard predicates: informally, an expectation is a function that assigns a *weight* (a non-negative real number) to program states, thus describing how much each state is “worth” in relation to some desired outcome. This is nothing but a non-negative real-valued *random variable*⁵.

There is a natural way of embedding the usual boolean predicates in this approach, as an expectation corresponding to a predicate $Pred$ can be defined as a random variable $[Pred]$ that maps a state to 1 if it satisfies the predicate and to 0 otherwise.

Arithmetic operators and relations are extended point-wise to expectations, as is multiplication by a scalar: the space of all expectations over the state space \mathcal{S} is

$$\mathcal{E} = (\mathcal{S} \rightarrow \mathbb{R}^+, \leq);$$

functions modifying an expectation are referred to as *expectation transformers*.

$pGCL$ is given a semantics based on expectations, which generalises the concept of weakest precondition to that of *weakest pre-expectation*: for this reason this semantics is usually referred to as the *weakest pre-expectation semantics* — one expectation is weaker than another if for all states it returns at most the same weight (it is the \leq relation lifted point-wise).

A *pre-expectation* is an expectation whose domain is that of initial states, whereas a *post-expectation* is an expectation whose domain is that of final states; given a post-expectation $PostE$ and a program A , informally $wp.A.PostE$ is the weakest pre-expectation which describes the expected “worth” of each initial state: the operator wp can be thought of a function $wp: \mathcal{P} \rightarrow \mathcal{T}$ returning the expectation transformer corresponding to each program, where \mathcal{T} is the space of expectation transformers.

So we have that $PostE \in \mathcal{E}$ and $wp.A \in \mathcal{T}$, and therefore $wp.A.PostE \in \mathcal{E}$.

The syntax of $pGCL$ comprises the following constructs:

- *abort* is the aborting program;
- *skip* is the program which does nothing and terminates;
- *assignment*: $v_i := e(v_1, v_2, \dots, v_n)$, where the expression e is evaluated in the current state and the resulting value is assigned to v_i ;
- *sequential composition*: $A ; B$, which executes the program B after A has terminated;
- *probabilistic choice*: $A \oplus_p B$, which executes A with probability p and B with probability $(1 - p)$ — this is the novelty with respect to GCL ;
- *conditional choice*: $A \triangleleft c \triangleright B$, which executes A or B depending on the evaluation of the condition c — this is syntactic sugar for $A \oplus_{[c]} B$;
- *nondeterministic choice*: $A \sqcap B$, which executes A or B nondeterministically;

⁵Attention must be paid to the terminology, which may be utterly misleading: many people refer to the expected value of a random variable X as “expectation of X ”, but we will refrain from doing this to try to minimize confusion and use systematically “expected value of X ”.

$$\begin{aligned}
wp.abort.PostE &\triangleq 0 \\
wp.skip.PostE &\triangleq PostE \\
wp.(x := e).PostE &\triangleq PostE\{e/x\} \\
wp.(A ; B).PostE &\triangleq wp.A.(wp.B.PostE) \\
wp.(A \sqcap B).PostE &\triangleq \min\{wp.A.PostE, wp.B.PostE\} \\
wp.(A \text{ }_p\oplus B).PostE &\triangleq p \cdot wp.A.PostE + (1 - p) \cdot wp.B.PostE \\
wp.(c * A).PostE &\triangleq \text{lfp } X \bullet wp.((A ; X) \triangleleft c \triangleright skip)
\end{aligned}$$

Figure 2.1: wp -semantics of pGCL, adapted from [MM04, p. 26].

- (*while*) loop: $c * A$, which executes the body of the loop A as long as c holds true.

Given a post-expectation $PostE$, the weakest pre-expectation semantics corresponding to the constructs listed above is as follows:

- the weakest pre-expectation with respect to the aborting program is 0 regardless of $PostE$:

$$wp.abort.PostE = 0;$$

- the program $skip$ does not alter the weight of each state, so the weakest pre-expectation is unchanged and therefore it is still $PostE$:

$$wp.skip.PostE = PostE;$$

- in the case of assignment the weight of each state is changed according to the evaluation of the expression e :

$$wp.(x := e).PostE = PostE\{e/x\},$$

where the notation $PostE\{e/x\}$ denotes the expression describing $PostE$ with all free occurrences of x replaced by e . From this we can see that in some sense it is necessary to go backwards in order to give a meaning to the assignment construct, as $PostE$ needs to be “translated” in terms of the states we have before it;

- sequential composition is rendered by functional composition, as the weakest pre-expectation relative to $PostE$ with respect to B acts as the post-expectation when deriving the weakest pre-expectation with respect to A :

$$wp.(A ; B).PostE = wp.A.(wp.B.PostE);$$

- the weakest pre-expectation with respect to the probabilistic choice $A \text{ }_p\oplus B$ is a linear combination of the two alternative weakest pre-expectations with respect to A and B , where the coefficients p and $(1 - p)$ respectively:

$$wp.(A \text{ }_p\oplus B).PostE = p \cdot wp.A.PostE + (1 - p) \cdot wp.B.PostE$$

- the nondeterministic model underlying *pGCL* is the demonic one, and therefore nondeterministic choice picks in each case the option yielding the worst-case behaviour. This is rendered by taking the point-wise minimum between the two alternative weakest pre-expectations:

$$wp.(A \sqcap B).PostE = \min\{wp.A.PostE, wp.B.PostE\};$$

- in the case of the loop, the weakest pre-expectation can be determined via the least fix point operator in a standard way:

$$wp.(c * A).PostE = lfp X \bullet wp.((A; X) \triangleleft c \triangleright skip).$$

This is also shown in Figure 2.1.

Having retained nondeterminism, it is possible to define a sensible refinement relation using this semantics:

$$S \sqsubseteq A \triangleq \forall PostE \bullet wp.S.PostE \leq wp.A.PostE,$$

where A is some program and S is its specification.

In other words a program A refines a specification S if the minimum expected weight for each state after A has run is at least as much as we would get after S has run.

An alternative is the *probabilistic relational model* [HSM97; MM04], which sees a program as a relation from states to *up-, convex- and Cauchy-closed sets of probability distributions* δ over the state space — the characteristics of these sets correspond to some healthiness conditions on the probability distributions they contain, which will be discussed —; the space of all probability distributions is

$$\mathcal{D}_p = \{\delta : \mathcal{S} \rightarrow [0, 1] \mid \sum_{\sigma \in \mathcal{S}} \delta(\sigma) \leq 1\}.$$

It is possible to see programs as relations from probability distributions to sets of probability distributions via the *Kleisli composition* of programs [MM04, Chp. 5] — incidentally, this is similar to our approach to give *pGCL* a *UTP* semantics based on distributions.

From this perspective a probabilistic program is seen as a function that maps an initial state to a fixed final probability distribution over \mathcal{S} ; the space of all deterministic programs is

$$\mathcal{P}_D = (\mathcal{S} \rightarrow \mathcal{D}_p, \sqsubseteq).$$

Because of nondeterminism each initial state can be mapped to different final probability distributions: it is therefore possible to see a demonic probabilistic program as taking an initial state to a set of fixed final probability distributions.

Such a set cannot be any subset of \mathcal{D}_p , as the distributions it contains must comply with some healthiness criteria, as mentioned above: this results in the set being *up-, convex- and Cauchy-closed*.

The space of all demonic probabilistic programs is therefore

$$\mathcal{P} = (\mathcal{S} \rightarrow \mathcal{H}, \sqsubseteq),$$

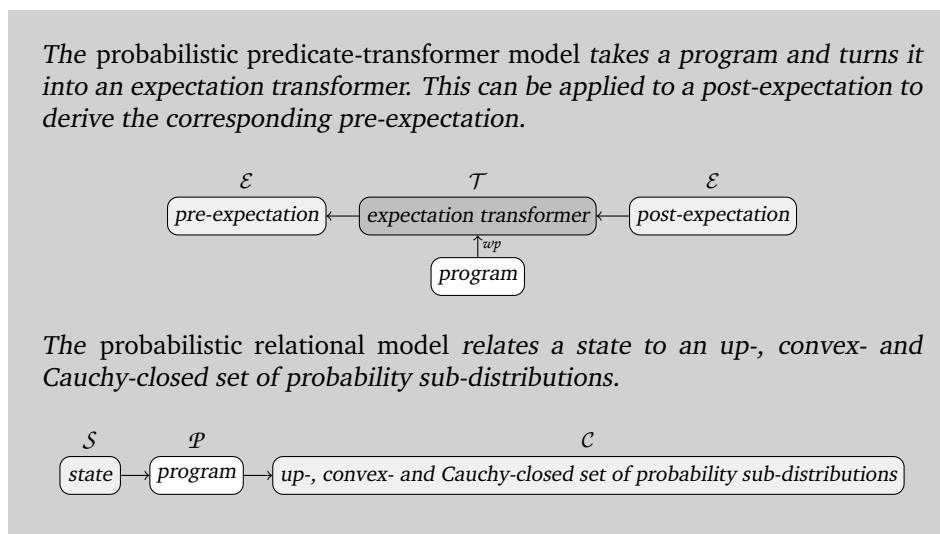


Figure 2.2: The two semantic models of pGCL from [MM04].

where $\mathcal{H} \subseteq \wp\mathcal{D}_p$ is the set of all up-, convex- and Cauchy-closed sets of probability distributions; these three set properties descend from healthiness conditions that are satisfied only by those distributions that result from sensible probabilistic programs:

Probabilistic programs are now modelled as the set of functions from initial state in \mathcal{S} to sets of final distributions over \mathcal{S} , where the result sets are restricted by so-called healthiness conditions characterising viable probabilistic behaviour, motivated in detail elsewhere [MM04]. By doing so the semantics accounts for specific features of probabilistic programs. In this case we impose up-closure (the inclusion of all \sqsubseteq -dominating distributions), convex closure (the inclusion of all convex combinations of distributions), and Cauchy closure (the inclusion of all limits of distributions according to the standard Cauchy metric on real-valued functions [MMS96]). Thus, by construction, viable computations are those in which miracles dominate (refine) all other behaviours (implied by up-closure), nondeterministic choice is refined by probabilistic choice (implied by convex closure), and classic limiting behaviour of probabilistic events (such as so-called “zero-one laws”) is also accounted for (implied by Cauchy closure). A further bonus is that (as usual) program refinement is simply defined as reverse set inclusion. We observe that probabilistic properties are preserved with increase in this order.

adapted from McIver, Cohen, and Morgan [MCM06]

The visual synthesis of the semantic models is presented in Figure 2.2.

Some work by Hehner [Heh04; Heh11] revisits what has been done on pGCL, with a focus on predicative semantics.

To conclude this brief presentation of *pGCL*, here is a representative sample of laws about probabilistic programs, that it is possible to prove in this framework:

$$\begin{aligned}
A \sqcap B &\sqsubseteq A \text{ }_{p\oplus} \text{ } B \\
(A \sqcap B) \text{ }_{p\oplus} \text{ } C &= (A \text{ }_{p\oplus} \text{ } C) \sqcap (B \text{ }_{p\oplus} \text{ } C) \\
(A \sqcap C) \text{ }_{p\oplus} \text{ } (B \sqcap C) &\sqsubseteq (A \text{ }_{p\oplus} \text{ } B) \sqcap C \\
(A \sqcap B); C &= (A; C) \sqcap (B; C) \\
A; (B \sqcap C) &\sqsubseteq (A; B) \sqcap (A; C)
\end{aligned}$$

2.3 *pCSP*

On the side of process algebras, probabilistic CSP is obtained by adding probability to Hoare's CSP [Hoa85b].

In [Mor⁺96] we can find one of the possible definitions, where probability is defined in such a way that it distributes through all operators: this leads to a surprising behaviour in the demonic choice operator, which is not idempotent.

In this paper they define a refinement operator and discuss the ideas of an associated probabilistic refinement calculus, where an implementation satisfies a specification with null probability: this shows that it is not reasonable to expect an absolute specification in this setting, but it is wiser to have a sort of “timed” specification. This is in line with real-world systems, as they cannot possibly work forever (we simply have to wait long enough for their failure probability to raise), and for this reason we can specify a time limit for which a specification has to be satisfied.

A different presentation is given in [Mor04], where *pCSP* is built on top of probabilistic action systems written in *pGCL* and is linked back to the relational semantics of *pGCL*.

This view of the subject highlights how compositionality of probabilistic CSP is not straightforward, because of the introduction of probability: in a way probability splits the deterministic scenario into several possible different scenarios, and one has to take this into account when composing probabilistic programs.

They explain this using the metaphor of the colour of a child's eye, knowing the colour of the parents' — too much information has to be brought forward if we want accurate information, but simply a phenotypical description is unreliable and not sufficient, as what is enough is to know colour and whether the allele is predominant or recessive. This same kind of information is the one that has to be sought to have an accurate probabilistic compositionality: in fact if we observe an event, we would want to be able to identify the facts that have led to that event.

For example if we observe a failure (*i.e.* a composite event) during the run of a program, we want to track down the reasons of this failure and to identify what factors (*i.e.* base events) have been responsible for the happening.

2.4 *UTP*

The Unifying Theories of Programming (*UTP*) research activity seeks to bring models of a wide range of programming and specification languages under a single semantic framework in order to be able to reason formally about their integration [HH98; DS06; But10; Qin10].

Computing science is a new subject, and we have not yet achieved the unification of theories that should support a proper understanding of its structure. [...] we face the challenge of building a coherent structure for the intellectual discipline of computing science, and in particular for the theory of programming. Such a comprehensive theory must include a convincing approach to the study of the range of languages in which computer programs may be expressed. It must introduce basic concepts and properties which are common to the whole range of programming methods and languages. Then it must deal separately with the additions and variations which are particular to specific groups of related programming languages.

Hoare and He [HH98]

A success in this area has been the development of the *Circus* language [OCW09], which is a fusion of Z and CSP, with a UTP semantics, providing specifications using a “state-rich” process algebra along with a refinement calculus; recent extensions to *Circus* have included timed [SH03] and synchronous [GB09] variants. Recent interest in aspects of the POSIX filestore case study in the Verification Grand Challenge [FWB08] has led us to consider integrating probability into UTP, with a view to eventually having a probabilistic variant of *Circus*.

UTP follows the key principle that “programs are predicates” [Heh84; Hoa85a] and so does not distinguish between the syntax of some language and its semantics as alphabetised predicates; theories in UTP are expressed as second-order predicates⁶ over a pre-defined collection of free *observation variables*, referred to as the *alphabet* of the theory. The predicates are generally used to describe a relation between a before-state and an after-state, the latter typically characterised by dashed versions of the observation variables. For example, a program using two variables x and y might be characterised by having the set $\{x, x', y, y'\}$ as an alphabet, and the meaning of the assignment $x := y + 4$ would be described by the predicate

$$x' = y + 4 \wedge y' = y.$$

In effect UTP uses predicate calculus in a disciplined way to build up a relational calculus for reasoning about programs.

In addition to observations of the values of program variables, often we need to introduce observations of other aspects of program execution via so-called *auxiliary variables*. So, for example, in order to reason about total correctness, we need to introduce boolean observations that record the starting ($o\kappa$) and termination ($o\kappa'$) of a program, resulting in the above assignment having the following semantics:

$$o\kappa \Rightarrow o\kappa' \wedge x' = y + 4 \wedge y' = y$$

(if started, it will terminate, and the final value of x will equal the initial value of y plus four, with y unchanged).

As an example of a UTP theory using both observation and auxiliary variables, we have shown in Figure 2.3 the UTP semantics of a variant of Dijkstra’s *GCL* [Dij76] according to the so-called theory of “designs”, which characterises total correctness for imperative programs. κ is a

⁶ Most definitions are in fact first-order, but we need second-order in order to handle the notion of “healthiness”, and recursion.

$$\begin{aligned}
\text{abort} &\triangleq \text{true} \\
\text{skip} &\triangleq \text{ok} \Rightarrow \text{ok}' \wedge \underline{v}' = \underline{v} \\
x := e &\triangleq \text{ok} \wedge e \text{ is defined} \Rightarrow \text{ok}' \wedge x' = e \wedge \underline{v}' = \underline{v} \\
P_1; P_2 &\triangleq \exists \text{ok}_m, \underline{v}_m \bullet P_1[\text{ok}_m, \underline{v}_m / \text{ok}', \underline{v}'] \wedge P_2[\text{ok}_m, \underline{v}_m / \text{ok}, \underline{v}] \\
P_1 \triangleleft c \triangleright P_2 &\triangleq c \wedge P_1 \vee \neg c \wedge P_2 \\
P_1 \sqcap P_2 &\triangleq P_1 \vee P_2 \\
c * P &\triangleq \forall X \bullet (P; X) \triangleleft c \triangleright \text{skip}
\end{aligned}$$

Figure 2.3: UTP Design semantics of simplified GCL

program variable and \underline{v} is the list of all other program variables, and thus these are observation variables, and ok is an auxiliary variable.

A problem with allowing arbitrary predicate calculus statements to give semantics is that it is possible to write unhelpful predicates such as $\neg \text{ok} \Rightarrow \text{ok}'$, which describes a “program” that must terminate when not started. In order to avoid assertions that are either nonsense or infeasible, UTP adopts the notion of *healthiness conditions* which are monotonic idempotent predicate transformers whose fixpoints characterise sensible (healthy) predicates. Collections of healthy predicates typically form a sub-lattice of the original predicate lattice under the reverse implication ordering [HH98, Chapter 3].

Key in UTP is a general notion of program refinement as the universal closure of reverse implication⁷:

$$S \sqsubseteq P \triangleq [P \Rightarrow S]$$

Program P refines S if for all observations (free variables), S holds whenever P does.

The UTP framework also uses Galois connections to link different languages/theories with different alphabets [HH98, Chapter 4], and often these manifest themselves as further modes of refinement.

2.4.1 Theory of Designs

The theory of designs patches the relational theory, in the sense that predicates from the relational theory fail to satisfy the following equality:

$$\text{true}; \mathcal{P} = \text{true}$$

In fact according to the relational theory *true* is a left identity of the sequential composition operator:

$$\begin{aligned}
\text{true}; \mathcal{P} &\equiv \exists \underline{v}_m \bullet \text{true}\{\underline{v}_m / \underline{v}'\} \wedge \mathcal{P}\{\underline{v}_m / \underline{v}\} \\
&\equiv \exists \underline{v}_m \bullet \text{true} \wedge \mathcal{P}\{\underline{v}_m / \underline{v}\} \\
&\equiv \exists \underline{v}_m \bullet \mathcal{P}\{\underline{v}_m / \underline{v}\}
\end{aligned}$$

⁷Square brackets denote universal closure — [P] asserts that P is true for all values of its free variables.

Which reduces to *true* if $\underline{v} \in fv(\mathcal{P})$, or to \mathcal{P} otherwise.

This has disastrous consequences, as this enables us to show that a program can recover from a never-ending loop:

$$true * skip \equiv lfp X \bullet X \equiv \perp \equiv true$$

... which is surprising, to say the least.

The theory of designs uses an additional auxiliary variable ok (along with its dashed version ok') to record start (and termination) of a program.

A design (specification) is made of a precondition Pre that has to be met when the program starts, and if so the program establishes $Post$ upon termination, which is guaranteed:

$$ok \wedge Pre \Rightarrow ok' \wedge Post$$

for which we use the following shorthand:

$$Pre \vdash Post$$

The semantics of the assignment $x := y + 3$ in this theory is the following:

$$true \vdash x' = y + 3 \wedge y' = y$$

(if started, it will terminate, and the final value of x will equal the initial value of y plus three, with y unchanged).

The behaviour of *true* with respect to sequential composition is the desirable one, as now we have:

$$\begin{aligned} true; (Pre \vdash Post) &\equiv true; ok \wedge Pre \Rightarrow ok' \wedge Post \\ &\equiv \exists ok_m, \underline{v}_m \bullet true\{ok_m/ok'\}\{\underline{v}_m/\underline{v}'\} \wedge (ok_m \wedge Pre\{\underline{v}_m/\underline{v}\} \Rightarrow ok' \wedge Post) \\ &\equiv \exists ok_m, \underline{v}_m \bullet true \wedge (ok_m \wedge Pre\{\underline{v}_m/\underline{v}\} \Rightarrow ok' \wedge Post) \\ &\equiv \exists ok_m, \underline{v}_m \bullet ok_m \wedge Pre\{\underline{v}_m/\underline{v}\} \Rightarrow ok' \wedge Post \\ &\equiv true \end{aligned}$$

and therefore *true* is a left zero for sequential composition.

Designs form a lattice, whose bottom and top elements are respectively:

$$abort \triangleq false \vdash false \equiv false \vdash true$$

and

$$miracle \triangleq true \vdash false \equiv \neg ok$$

It should be noted that *miracle* is a (infeasible) program that cannot be started.

Valid designs are predicates R which comply with four healthiness conditions [HH98]. The first one (*unpredictability*, H1) excludes from observation all programs that have not started, and therefore restricts valid relations to those such that:

$$R = (ok \Rightarrow R)$$

All H1-healthy predicates satisfy the left zero and left unit laws:

$$true; R = true \quad \text{and} \quad skip; R = R$$

The second one (*possible termination*, H2) states that a valid relation cannot require nontermination:

$$R\{false/o\kappa'\} \Rightarrow R\{true/o\kappa'\}$$

The third one (*dischargeable assumptions*, H3) states that preconditions cannot use dashed variables. All H3-healthy predicates satisfy the right unit law:

$$R; skip = R$$

The fourth one (*feasibility or excluded miracle*, H4) requires the existence of final values for the dashed variables that satisfy the relation:

$$\exists o\kappa', \underline{v}' \bullet R = true$$

H4 excludes *miracle* from the valid designs, and this implies that all H4-healthy predicates satisfy the right zero law:

$$R; true = true$$

This condition cannot be expressed as an idempotent healthiness transformer, and does not preserve the predicate lattice structure. It serves solely to identify and/or eliminate predicates that characterise infeasible behaviour.

2.4.2 Probabilistic UTP

There has already been a certain amount of work looking at encoding probability in a *UTP* setting. He and Sanders have presented an approach unifying probabilistic choice with standard constructs [HS06], and this work provides an example of how the laws of *pGCL* could be captured in *UTP* as predicates about program equivalence and refinement. However only an axiomatic semantics was presented, and the laws were justified via a Galois connection to an expectation-based semantic model.

Sanders and Chen then explored an approach that decomposed demonic choice into a combination of pure probabilistic choice and a unary operator that accounted for demonic behaviour [CS09]. There they commented on the lack of a satisfactory *UTP* theory which could prove effective towards...

... the so-far-unachieved goal of unifying probabilism with other programming constructs in the style of *Unifying Theories of Programming*.

Chen and Sanders [CS09]

A probabilistic BPEL-like language has recently been described by He [He10] that gives a *UTP*-style semantics for a web-based business semantics language. This language is *GCL* with extra constructs to handle probabilistic choice and compensations and coordination operators, including exception handling. The *UTP* model that is developed does not relate before- and after-variables of the same type, but instead uses predicates to encode a relationship between

an initial state and a final probability distribution over states.

In relatively recent times a paper by Jun Sun *et al.* [SSL10] has described a probabilistic analysis of the likelihood of a program in a medical device satisfying a safety specification, given that random, but hopefully unlikely events, can prevent the correct behaviour, even if the program is the best one possible. Their probabilistic model checking directly corresponds to the probabilistic refinement we are going to present in §3.7.1.

What all the treatments above have in common is that the *UTP* predicates relate an initial program variable state (σ) to a final probability distribution (δ') over states, so the relation is not homogeneous. This complicates the definition of sequential composition (which has to involve some form of Kleisli composition) and also makes building links to homogeneous *UTP* theories more difficult.

What is still missing is a *UTP* theory that is defined in terms of predicates based on before/after relations over the *same* observation space.

Several *UTP* theories are based on homogeneous relations: this means that all of these theories have uniform definitions of many common language features, such as sequential composition. For example the collection of theories surrounding *Circus* are all uniform, so the development of a homogeneous probabilistic *UTP* theory would open way towards a reasonably easy development of a probabilistic theory of *Circus* .

We believe the ideal such theory would use observations that corresponded to program variables and to other aspects of behaviour such as termination, in a manner analogous to the *UTP* theory of designs: here we present a framework based on probability distributions over the set of possible states, relating a before-distribution (δ) to an after-one (δ'), effectively making use of one observation. Key contributions here are the facts that we provide a means by which reasoning can still be carried out at program variable level, and we have uncovered a generic notion of choice that subsumes probabilistic, demonic and conditional choices.

CHAPTER 3

A framework to deal with probability distributions over the state space

This chapter is dedicated to a quite detailed presentation of the framework we have developed: we have decided to privilege clarity of the exposition over pedantic details, which are therefore presented in the appendices along with many of the proofs for properties and theorems.

The fundamental reason why we felt the need of a different framework is that the existing ones do not integrate very well in the *UTP* framework, in the sense that dealing with probability is dealing with a great amount of information and complex constructs at a very low level.

In particular one of the key strengths of our framework is the use of homogeneous relations among distributions on the state space to model programs: in previous work the approach was to relate a single state to a distribution on the state space, which contains information on the probability of the different resulting states. The non-homogeneity of this relation makes sequential composition a non-trivial matter.

Also, in order not to get bogged down in unnecessary details, much of the complexity under is hidden under the hood, so that we can reason (quite) smoothly on probabilistic programs at a higher level.

These features together allow us to deal in a straightforward way with both nondeterministic and probabilistic choice: we deem this to be an important step towards the development of general probabilistic theories of a variety of languages already treatable in *UTP* in their non-probabilistic version, such as *CSP* or *Circus*, as we believe it helps overcome the current unsatisfactory approaches bringing probabilism and nondeterminism together [CS09].

Coherently with the *UTP* approach, programs are captured as predicates with a suitable alphabet.

Hehner and Hoare wrote that “programs are predicates” [Heh84; Hoa85a], we affirm that *probabilistic* programs are predicates too.

3.1 States and distributions, informally

Before presenting formally the foundations of our framework, we find it useful to give a general and intuitive overview, where we sacrifice formality and rigour in favour of a more relaxed introduction of the basic concepts: this should allow the reader to have an intuitive understanding of the key points, which we are going to introduce formally in the remainder of the chapter.

UTP predicates usually involve relations between variables from the predicate alphabet and the corresponding values: some people feel that a tempting approach may be to try a naïve (and quite straightforward) generalization of this standard situation by relating a variable to a pair containing its possible value and the corresponding probability.

In this case the idea is to deal with objects with the following shape¹:

$$\underline{\mathcal{V}} \rightarrow (\underline{\mathcal{W}} \rightarrow [0..1]),$$

where \mathcal{V} and \mathcal{W} are appropriate sets of program variables and corresponding values, respectively.

It should be quite evident that this is not a satisfactory approach. At the risk of stating the obvious, the reason is that this approach takes each variable *individually*, so it assumes the independence of the value assumed by each variable from that of any other — and this is clearly an assumption which is wrong in general.

To see this let us consider an example, where a program with variables x, y starts in a state where x and y are each independently initialized to 0 with probability $1/2$ and to 1 with probability $1/2$. This program consists simply in the assignment $x := y$ and so the situation after the program has run can be described as follows, with obvious meaning of the notation:

$$x \mapsto \begin{pmatrix} 0 \mapsto 1/2 \\ 1 \mapsto 1/2 \end{pmatrix}, y \mapsto \begin{pmatrix} 0 \mapsto 1/2 \\ 1 \mapsto 1/2 \end{pmatrix}.$$

The information contained in this description is incomplete, as it does not tell us anything about the relation between the variables; in this case it seems we are able to make predictions on the expected value of each variable taken separately² (e.g. the probability of $x = 1$ is $1/2$), but as soon as we try to reason on a more complex event, such as the probability of $x = y$, things go terribly wrong. If we crudely look at the numbers, the probability we are looking for is:

$$\mathcal{P}(x = y) = \mathcal{P}(x = 0) \cdot \mathcal{P}(y = 0) + \mathcal{P}(x = 1) \cdot \mathcal{P}(y = 1) = 1/2.$$

This is quite an upsetting “I-told-you-so” result, as all the program did was to assign the value of y to x , so we would have expected $\mathcal{P}(x = y) = 1$.

So, although such an easy generalization may (?) look appealing, this example clearly shows how this is not a viable approach, as it loses the *entanglement* among the variables.

A better approach should rather use objects with this other shape:

$$(\underline{\mathcal{V}} \rightarrow \underline{\mathcal{W}}) \rightarrow [0..1].$$

The example above with this different approach gives the following description:

$$\begin{pmatrix} x \mapsto 0 \\ y \mapsto 0 \end{pmatrix} \mapsto 1/2, \begin{pmatrix} x \mapsto 1 \\ y \mapsto 1 \end{pmatrix} \mapsto 1/2.$$

This approach assigns different probability to the different mappings $\sigma : \mathcal{V} \rightarrow \mathcal{W}$ that relate

¹We underline whenever we talk about vectors or sets of vectors: \underline{A} stands for a n -th dimensional vectorial space $A \times A \times \dots \times A$, for an appropriate n .

²But only because the program is that easy, in general we cannot even do this!

each variable to its corresponding value in the different situations — these are the different *program states* —, so the objects we are using have this shape:

$$\mathcal{S} \rightarrow [0..1],$$

where \mathcal{S} is the set of all program states (*state space*).

With the position $\sigma_{ij} \triangleq \{x \mapsto i, y \mapsto j\}$ we can rewrite the output of the example above as:

$$\sigma_{00} \mapsto 1/2, \sigma_{11} \mapsto 1/2.$$

This mapping from program states to probability is what we term *probability distribution*, a function $\delta : \mathcal{S} \rightarrow [0..1]$ which has the additional property that the sum of the probabilities of all program states in \mathcal{S} (the *weight* of the distribution, $\|\delta\|$) cannot exceed 1.

If we define the distributions

$$\begin{aligned} \delta &\triangleq \{\sigma_{00} \mapsto 1/4, \sigma_{01} \mapsto 1/4, \sigma_{10} \mapsto 1/4, \sigma_{11} \mapsto 1/4\} \\ \delta' &\triangleq \{\sigma_{00} \mapsto 1/2, \sigma_{11} \mapsto 1/2\}, \end{aligned}$$

we can describe the full behaviour of the program in the example by saying that it has transformed the (before-)distribution δ into the (after-)distribution δ' — in *UTP* we usually use a dash to mark a variable, in order to refer to the new value v' it contains: the same convention is adopted here, where we use dashes in a similar way to denote *after-distributions* (δ') and *after-states* (σ').

This transformation has been done by changing the probability associated to each state: the assignment modifies the variable mapping so that each before-state σ becomes the after-state σ' , therefore probability associated to σ has to be “transferred” to σ' .

More in general, given an assignment $\underline{v} := \underline{e}$, where \underline{e} is a vector of expressions, if we perform this operation on every state of a distribution δ we obtain the distribution $\delta\{\underline{e}/\underline{v}\}$: the postfix operator $\{\underline{e}/\underline{v}\}$ modifies δ to reflect the modifications introduced by the assignment — the intuition behind this, roughly speaking, is that all states σ where the expression \underline{e} evaluates

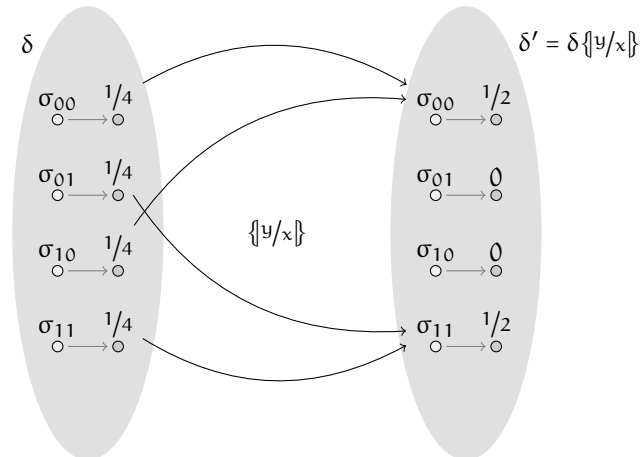


Figure 3.1: The assignment in the example.

to the same value $\underline{w} = \text{eval}_\sigma(\underline{e})$ are replaced by a single state $\sigma' = (\underline{v} \mapsto \underline{w})$ that maps to a probability that is the sum of the probabilities of the states it replaces.

$$\delta\{\underline{e}/\underline{v}\} \triangleq \{\sigma' \mapsto \sum_\sigma \delta(\sigma) \mid \sigma \in \text{dom}(\delta) \wedge \text{eval}_\sigma(\underline{e}) = \sigma'(\underline{v})\}.$$

Using the postfix notation, we have that the program in the example returns the after-distribution $\delta' = \delta\{\underline{v}/\underline{x}\}$, as shown in Figure 3.1.

It is also possible to operate on distributions by point-wise lifting in an obvious way operators such as addition, product and multiplication by a scalar number.

An interesting case is the one when we multiply a probability distribution by what we term a *weighting distribution*, which is a distribution π mapping states to real numbers in the interval $[0..1]$, *without* the constraint $\|\pi\| \leq 1$. The resulting probability distribution, noted $\delta\langle\pi\rangle$, has the property of being point-wise smaller than δ , and will have an important role when defining choice constructs:

$$\delta\langle\pi\rangle \triangleq \{\sigma \mapsto \pi(\sigma) \cdot \delta(\sigma) \mid \sigma \in \text{dom}(\delta)\}.$$

Another example is when we want to select the subset of a distribution δ , which comprises only states where a condition c (which is a boolean expression) is satisfied; for reasons that will become clear later on, we have chosen to overload the above notation and note this as $\delta\langle c \rangle$:

$$\delta\langle c \rangle \triangleq \{\sigma \mapsto \delta(\sigma) \mid \sigma \in \text{dom}(\delta) \text{ satisfies } c\}.$$

As the probability of a condition c to be true on a distribution δ can be calculated by adding up the probabilities relative to all states that satisfy such a condition, we can express this probability using the notation introduced so far as $\|\delta\langle c \rangle\|$.

This concludes our informal introduction of the foundational elements of our framework; the following §3.2 gives the main definitions, whereas Appendix A is dedicated to a more rigorous and pedantic presentation of the framework (as a result it is rather dense and filled with technicalities).

3.2 Definitions

A state σ is a map $\sigma : \underline{\mathcal{V}} \rightarrow \underline{\mathcal{W}}$ that maps each program variable to the corresponding value.

A distribution χ is a function $\chi : \mathcal{S} \rightarrow \mathbb{R}$ that assigns a *weight* to each state.

The weight of a distribution is defined as follows:

$$\|\chi\| \triangleq \sum_{\sigma \in \text{dom}(\chi)} \chi(\sigma)$$

Among distributions, there are two notable kinds:

- a *weighting distribution* π is a distribution such that $\text{img}(\pi) = [0..1]$;
- a *probability distribution* δ is a weighting distribution with the additional property that $\|\delta\| = 1$.

We can perform on distributions the following operations:

- *point-wise addition:*

$$(\chi_1 + \chi_2)(\sigma) \triangleq \chi_1(\sigma) + \chi_2(\sigma);$$

- *point-wise multiplication:*

$$(\chi_1 \circ \chi_2)(\sigma) \triangleq \chi_1(\sigma) \cdot \chi_2(\sigma);$$

- *multiplication by a scalar $a \in \mathbb{R}$:*

$$(a \cdot \chi)(\sigma) \triangleq a \cdot \chi(\sigma);$$

- *restriction through a condition:*

$$(\chi(c))(\sigma) \triangleq \begin{cases} \chi(\sigma) & \text{if } \text{eval}_\sigma(c) \text{ is } \textit{true} \\ 0 & \text{otherwise;} \end{cases}$$

- sometimes it is useful to see a point-wise multiplication as a *restriction through a distribution*:

$$\chi_1 \langle \chi_2 \rangle \triangleq \chi_1 \circ \chi_2;$$

- *remap:*

$$\delta \langle \underline{e}/\underline{v} \rangle \triangleq \left\{ \sigma' \mapsto \left\| \delta \langle \text{Inv}(\underline{v} := \underline{e}, \{\sigma'\}) \rangle \right\| \mid \text{alph}(\sigma') \in \text{alph}(\delta) \right\}$$

3.3 Programs

We see programs as predicates relating a before- and after-distribution pair: the *body* of the program is a distribution-transformer, which acts on an initial before-distribution δ and returns a final after-distribution δ' that accounts for the modifications it caused.

We capture this relation as a predicate $A(\delta, \delta')$, which is true if and only if δ' is a possible resulting distribution of program A when the initial distribution is δ . When it is clear from the context, we can simply write A .

In case of nondeterministic programs there are potentially many possible resulting distributions: we define the *program image* of δ as the set of all possible after-distributions δ' that can result from running the program from an initial distribution δ . Clearly in the case of a deterministic program, the corresponding program image is a singleton set: we will discuss deterministic program first (§3.3.1) and postpone the discussion of nondeterminism till §3.4

With a bit of notation overload, for the image of δ after a program A we would write the following:

$$A(\delta) \triangleq \{ \delta' \mid A(\delta, \delta') \}.$$

We can extend the notion of program image to the case of a set of probability distributions \mathcal{X} with the following definition:

$$A(\mathcal{X}) \triangleq \bigcup_{\delta \in \mathcal{X}} A(\delta).$$

In the case of a program A returning always the same program image regardless of the initial distribution, *viz.* when $\forall \delta_1, \delta_2 \bullet A(\delta_1) = A(\delta_2)$, we may simply write A instead of $A(\delta)$ or $A(\mathcal{X})$.

In order to help the reader remember this notation, we give the rationale behind it. For a program A , the possible notations involving its name are:

- $A(\delta, \delta')$, which is the predicate associated with the program. This is a function $A : \mathcal{D}_p \times \mathcal{D}_p \rightarrow \mathbb{B}$ that returns true if the before- and after-distributions passed as arguments are compatible: it is the relational view of “programs as predicates”;
- $A(\delta)$, which is the program image of δ . This is a function $A : \mathcal{D}_p \rightarrow \wp \mathcal{D}_p$ that returns the set of after-distributions compatible with δ as before-distribution: it is the functional view of “programs as distribution-transformers”;
- $A(\mathcal{X})$, which is the program image of \mathcal{X} . This is a function $A : \wp \mathcal{D}_p \rightarrow \wp \mathcal{D}_p$ that returns the set of after-distributions compatible with at least one of the elements of \mathcal{X} as before-distribution: it is again the functional view of “programs as distribution-transformers”, though they actually act on sets of distributions;
- A could stand either for $A(\delta, \delta')$, $A(\delta)$ or $A(\mathcal{X})$. In the first case it is the standard convention of *UTP* to omit input and output variables to make formulas more readable, whereas in the second and third cases it is the standard omission of the argument for constant functions. The context allows us to tell the difference between the two uses and no confusion should arise, as $A(\delta, \delta') \in \mathbb{B}$ while $A(\delta), A(\mathcal{X}) \in \wp \mathcal{D}_p$.

The evaluation of the weight of the program image restricted by a condition c returns information on the *probability* of the condition c being satisfied by program A when starting from the distribution δ :

$$\|A(\mathcal{X})\{c\}\| = \{\|\delta'\{c\}\| \mid \delta' \in A(\mathcal{X})\}.$$

This is a set of probabilities and the effective probability of c is nondeterministically chosen from here: it is therefore possible to extract information on the minimum and the maximum probability of c (a precise value in the case of a singleton set).

We refer to this set as the *weight* of the program A with respect to the condition c .

3.3.1 Deterministic programs

Initially we look at *deterministic programs*, where the relation from a before-distribution δ to the corresponding after-distribution δ' is injective, *viz.* for each δ there is one and only one corresponding δ' which is compatible with the possible outcome of a program:

$$\forall \delta \exists ! \delta' \bullet A(\delta, \delta').$$

We are now going to define a set of deterministic constructs, which can remind of those from *pGCL*, and give them a semantics based on the distributional framework introduced so far:

- the program *skip* does not produce any modification to the original distribution, therefore the after-distribution δ' equals the before-distribution δ :

$$\text{skip} \hat{=} \delta' = \delta;$$

- an assignment $\underline{v} := \underline{e}$ transforms the before-distribution δ by application of the corresponding remap operation, as described in §A.3.4:

$$\underline{v} := \underline{e} \triangleq \delta' = \delta \{\{\underline{e}/\underline{v}\}\};$$

- the sequential composition of two programs returns the after-distribution δ' output by the second program when it operates on the after-distribution δ_m resulting from the operation of the first program on the initial before-distribution δ :

$$A; B \triangleq \exists \delta_m \bullet A(\delta, \delta_m) \wedge B(\delta_m, \delta');$$

- the conditional choice between two programs depending on the evaluation of a condition effectively splits the before-distribution into two disjoint parts and operates on them according to the instructions of the two programs, before finally merging them together into a single after-distribution:

$$A \triangleleft c \triangleright B \triangleq \exists \delta_A, \delta_B \bullet A(\delta(c), \delta_A) \wedge B(\delta(-c), \delta_B) \wedge \delta' = \delta_A + \delta_B$$

- the probabilistic choice between two programs also splits the before-distribution into two parts, which are nothing but the original before-distribution scaled down by factors p and $(1 - p)$ respectively:

$$A \text{ }_p\oplus B \triangleq \exists \delta_A, \delta_B \bullet A(p \cdot \delta, \delta_A) \wedge B((1 - p) \cdot \delta, \delta_B) \wedge \delta' = \delta_A + \delta_B$$

- the loop construct has a standard definition based on Tarski's fixed point theorem [Tar55] and is, in particular, the weakest fixpoint, with respect to the refinement ordering discussed in the following §§3.6,3.7, of the function below:

$$c * A \triangleq \text{Ifp } X \bullet (A; X) \triangleleft c \triangleright \text{skip}$$

More on probabilistic choice

We want to make a few remarks on probabilistic choice.

First of all it is worth noticing that, from the above definition of probabilistic choice, we have the following equivalence:

$$A \text{ }_p\oplus B \equiv B \text{ }_{(1-p)}\oplus A$$

In fact:

$$\begin{aligned} & A \text{ }_p\oplus B \\ \equiv & \quad [d:P:Ch:Prb] \text{ — §B.3} \\ & \exists \delta_A, \delta_B \bullet A(p \cdot \delta, \delta_A) \wedge B((1 - p)\delta, \delta_B) \wedge \delta' = \delta_A + \delta_B \\ \equiv & \quad \text{Arithmetic} \\ & \exists \delta_A, \delta_B \bullet A((1 - (1 - p)) \cdot \delta, \delta_A) \wedge B((1 - p)\delta, \delta_B) \wedge \delta' = \delta_A + \delta_B \\ \equiv & \quad [d:P:Ch:Prb] \\ & B \text{ }_{(1-p)}\oplus A \end{aligned}$$

This is a special case of Proof C.28.

Moreover we have the following property:

$$A \text{ }_p\oplus (B \text{ }_q\oplus C) \equiv (A \text{ }_r\oplus B) \text{ }_s\oplus C \wedge p = rs \wedge (1-s) = (1-p)(1-q)$$

In fact:

$$\begin{aligned} & A \text{ }_p\oplus (B \text{ }_q\oplus C) \\ \equiv & \quad [d:P:Ch:Prb] \text{ — §B.3} \\ & \exists \delta_A, \delta_{BC} \bullet A(p \cdot \delta, \delta_A) \wedge (B \text{ }_q\oplus C)((1-p) \cdot \delta, \delta_{BC}) \wedge \delta' = \delta_A + \delta_{BC} \\ \equiv & \quad [d:P:Ch:Prb] \wedge \delta_{BC} = \delta_B + \delta_C \text{ (One-point rule)} \\ & \exists \delta_A, \delta_B, \delta_C \bullet A(p \cdot \delta, \delta_A) \wedge B(q(1-p) \cdot \delta, \delta_B) \wedge C((1-q)(1-p) \cdot \delta, \delta_C) \wedge \delta' = \delta_A + \delta_B + \delta_C \\ \equiv & \quad (1-p)(1-q) = (1-s) \wedge p = rs \Rightarrow q(1-p) = (1-r)s \\ & \exists \delta_A, \delta_B, \delta_C \bullet A(rs \cdot \delta, \delta_A) \wedge B((1-r)s \cdot \delta, \delta_B) \wedge C((1-s) \cdot \delta, \delta_C) \wedge \delta' = \delta_A + \delta_B + \delta_C \\ \equiv & \quad [d:P:Ch:Prb] \wedge \delta_{AB} = \delta_A + \delta_B \text{ (One-point rule)} \\ & \exists \delta_{AB}, \delta_C \bullet (A \text{ }_r\oplus B)(s \cdot \delta, \delta_{AB}) \wedge C((1-s) \cdot \delta, \delta_C) \wedge \delta' = \delta_{AB} + \delta_C \\ \equiv & \quad [d:P:Ch:Prb] \\ & (A \text{ }_r\oplus B) \text{ }_s\oplus C \end{aligned}$$

A few words on the probability p , that parametrises this operator: this may be a number in the range $[0, 1]$ in the simplest setting, but in a more general case it is one of the possible values of a stochastic variable \mathcal{P} that follows a probability distribution, whose probability density function $f_{\mathcal{P}}$ has the property of being compact in the range $[0, 1]$:

$$\int_{-\infty}^{+\infty} f_{\mathcal{P}}(p) dp = \int_0^1 f_{\mathcal{P}}(p) dp = 1$$

The distribution of this stochastic variable need not depend on the program variables, but in an even more general case may depend on other parameters q_1, q_2, \dots, q_n :

$$\int_{-\infty}^{+\infty} f_{\mathcal{P}Q}(p, q_1, q_2, \dots, q_n) dp = \int_0^1 f_{\mathcal{P}Q}(p, q_1, q_2, \dots, q_n) dp = f_Q(q_1, q_2, \dots, q_n)$$

3.4 Nondeterminism

All deterministic programs presented in §3.3.1 are characterised by the fact that their program image for any before-distribution is always a singleton set, *viz.* any before-distribution is uniquely transformed into a precise after-distribution.

When we introduce nondeterminism this does not hold true anymore as for some before-distribution there are more than one possible after-distributions that can satisfy the predicate representing a nondeterministic program.

The most nondeterministic program is the aborting program, which poses (almost) no restrictions on the possible relations between before- and after-distributions:

$$\mathit{abort} \hat{=} \|\delta'\| \leq \|\delta\|,$$

so that the only restriction given by *abort* is that the distribution weight cannot be increased.

A different way for nondeterminism to arise is when a program has the possibility to choose internally between alternative execution paths:

$$A \sqcap B \triangleq \exists \pi, \delta_A, \delta_B \bullet A(\delta(\pi), \delta_A) \wedge B(\delta(\bar{\pi}), \delta_B) \wedge \delta' = \delta_A + \delta_B.$$

Nondeterministic choice allows picking any weighting distribution π to alter the weight of each state before applying the alternative programs and summing the results.

This definition is different than the one one might expect: according to the relational semantics of *pGCL* from [HSM97; MM04], which sees programs as relations from a state σ to a probability distribution, we have that³

$$(A \sqcap B).\sigma = \cup_{p \in [0..1]} (A \text{ }_p \oplus B).\sigma$$

If a demonic choice is performed on a state, the set of resulting distributions is that containing all possible distributions resulting from a probabilistic choice with probability p varying in the range $[0..1]$.

Seeing this, one could (reasonably?) expect the following definition for nondeterministic choice in our framework:

$$A \sqcap B \stackrel{?}{=} \exists p \bullet A \text{ }_p \oplus B$$

However this definition does not work. In particular, with the above definition, we can prove the following (which is most definitely not a law of *pGCL*):

$$(A \sqcap B); (C \text{ }_p \oplus D) = (C \text{ }_p \oplus D); (A \sqcap B) \quad (!?)$$

It describes a demonic choice that is both history-aware, and *prescient*, and this latter ability to look into the future is undesirable, and infeasible.

The key point to note is that the first statement is talking about the possible resulting distributions starting from one single state, whereas this last definition considers all possible starting states. As a result the set of after-distributions that satisfy this definition of demonic choice (for a given before-distribution) is strictly smaller than the set of after-distributions satisfying the first statement.

The solution that led to our definition is therefore that of taking a weighting distribution π , use it with its complementary distribution $\bar{\pi} = 1 - \pi$ to weight the distributions resulting from the left- and right-hand side respectively, and existentially quantify it: we obtain some after-distributions which are the result of composing programs where p is not constrained to be constant over all states, and these cases were all ruled out in the proposed definition by the single quantification of p valid for all states.

Usually we talk about demonic nondeterminism when we are expecting the worst-case behaviour, to model something that behaves as bad as it can for any desired outcome.

Our definition of nondeterministic choice *per se* has no such behaviour, but it will show up with the definition of refinement that we give in §3.7 or, more in general, whenever we explicitly choose to focus on the worst-case scenario: for this reason we refer to it as to the nondeterministic choice, rather than to the demonic choice.

The nondeterministic choice operator is idempotent according to the above definition, as cus-

³Here we are using the point notation for function application, as in [MM04].

tomy in $pGCL$ and UTP .

There are some frameworks where nondeterministic choice is not idempotent, for example the probabilistic CSP from [Mor⁺96]. This happens when on both sides we have the same program containing a probabilistic choice and this choice is resolved independently on each side *before* the nondeterministic choice is performed, then idempotency does not hold. Nonetheless idempotency would hold if the probabilistic choice is resolved *after* the nondeterministic choice is made — this is the behaviour that we can find in our framework.

We can reproduce the other behaviour if we run the program twice with probabilistic choice on local variables, and then we merge the outputs by means of a nondeterministic choice: this is a behaviour that has nothing to do with idempotency — we keep the actions of one program separate from the other's, so we are actually dealing with two *different* programs that share the same specification.

3.4.1 A generic choice construct

We can see how all choice constructs look quite similar, or at least they follow a common pattern. The reason is that all choice constructs can be seen as a specific instance of a generic choice construct:

$$choice(A, B, \mathcal{X}) \triangleq \exists \pi, \delta_A, \delta_B \bullet \pi \in \mathcal{X} \wedge A(\delta(\pi), \delta_A) \wedge B(\delta(\bar{\pi}), \delta_B) \wedge \delta' = \delta_A + \delta_B$$

where $\mathcal{X} \subseteq \mathcal{D}_w$ and \mathcal{D}_w is the set of all weighting distributions.

This construct covers conditional, probabilistic and nondeterministic choice (and more), in fact we have that:

- for $\mathcal{X} = \{\iota\langle c \rangle\}$ we have conditional choice:

$$A \triangleleft c \triangleright B = choice(A, B, \{\iota\langle c \rangle\})$$

- for $\mathcal{X} = \{p \cdot \iota\}$ we have probabilistic choice:

$$A \text{ }_p\oplus\text{ } B = choice(A, B, \{p \cdot \iota\})$$

- for $\mathcal{X} = \mathcal{D}_w$ we have nondeterministic choice:

$$A \sqcap B = choice(A, B, \mathcal{D}_w)$$

Moreover we can see the disjunction of two programs as another kind of choice, where $\mathcal{X} = \{\epsilon, \iota\}$:

$$A \vee B = choice(A, B, \{\epsilon, \iota\})$$

This is the “usual” definition of nondeterministic choice in standard UTP : we can see the difference with the definition of nondeterministic choice we have given, because the possible after-distribution after a disjunction are those obtained by running the two programs separately, whereas with nondeterministic choice we obtain after-distributions which are superpositions of those obtained by the disjunction.

Finally we can also use this generic construct to create new kinds of choices, other than the more traditional ones:

- for $X = \{p \cdot \iota(c)\}$ we have the *conditional probabilistic choice*, which behaves like A with probability p and like B with probability $(1 - p)$ in the case when c holds, but it behaves like B if c does not hold:

$$A \triangleleft_{p,c} \triangleright B = \mathit{choice}(A, B, \{p \cdot \iota(c)\})$$

- for $X = \{p \cdot \iota(c) + q \cdot \iota(\neg c)\}$ we have the *switching probabilistic choice*, which is equivalent to a probabilistic choice with parameter p if c holds, with parameter q if c does not hold:

$$A \triangleleft_{p \triangleleft c \triangleright q} \oplus B = \mathit{choice}(A, B, \{p \cdot \iota(c) + q \cdot \iota(\neg c)\})$$

- for $X = \mathcal{D}_w(c)$ we have the *conditional nondeterministic choice*, which behaves like $A \sqcap B$ if c holds, but it behaves like B if c does not hold:

$$A_c \sqcap B = \mathit{choice}(A, B, \mathcal{D}_w(c))$$

- for $X = \{\pi \mid \forall \sigma \bullet p \leq \pi(\sigma) \leq 1 - q\}$, where $p + q \leq 1$, we have the *nondeterministic probabilistic choice*, which guarantees a probability p to behave like A and a probability q to behave like B :

$$A_{p \oplus q} B = \mathit{choice}(A, B, \{\pi \mid \forall \sigma \bullet p \leq \pi(\sigma) \leq 1 - q\})$$

- for $X = \{p \cdot \iota \mid p \in [0..1]\}$ we have the *fair nondeterministic choice*, which reweighs by p or $(1 - p)$ the entire before-distributions — and therefore multiplies the weight of each state in the before-distribution by the same number p or $(1 - p)$ — as opposed to the nondeterministic choice which can change individually the weight of each state:

$$A \overset{\text{fair}}{\sqcap} B = \mathit{choice}(A, B, X = \{p \cdot \iota \mid p \in [0..1]\}) = \exists p \bullet A \oplus_p B$$

It is worth noticing that this kind of choice is different from nondeterministic choice (we can view it as a less general form of it), in fact from this definition we have that:

$$\forall \delta \bullet (A \overset{\text{fair}}{\sqcap} B)(\delta) \subset (A \sqcap B)(\delta)$$

A few laws on choice operators

Here is a non-comprehensive list of interesting laws on choice operators, that hold in our framework and that can also be found in *pGCL*:

Idempotency of choice operators : $\forall X \bullet \mathit{choice}(A, A, X) \equiv A$

Discarding right-hand option : $\mathit{choice}(A, B, \{\iota\}) \equiv A$

Distributivity of choice operators :

$$\mathit{choice}(A, (\mathit{choice}(B, C, X_2)), X_1) \equiv \mathit{choice}\left(\left(\mathit{choice}(A, B, X_1)\right), \left(\mathit{choice}(A, C, X_1)\right), X_2\right)$$

Sequential composition : $choice(A, B, X); C \equiv choice(A; C, B; C, X)$

Choice flipping : $\forall X \bullet choice(A, B, X) \equiv choice(B, A, \bar{X}) \wedge \bar{X} = \bigcup_{\pi \in X} \bar{\pi}$

Monotonicity of generic choice : $\forall \delta \bullet X_1 \subseteq X_2 \Rightarrow choice(A, B, X_1)(\delta) \subseteq choice(A, B, X_2)(\delta)$

These properties are proven in Appendix C.

3.4.2 Program structure

We can see that all of the program encountered so far can be written as a predicate of the following shape⁴:

$$\exists QuantOf(A) \bullet \delta' = BodyOf(A) \circ \delta \wedge OtherCndOf(A)$$

where:

- $BodyOf(A)$ is a sequence of modifications (*i.e.* interleaved restrictions and remapping operations) that are applied to δ in order to obtain the corresponding δ' ;
- $QuantOf(A)$ is a list of weighting distributions — all of the quantified probability distributions can be eliminated via the one-point rule, so that δ' can be expressed as $BodyOf(A) \circ \delta$;
- $OtherCndOf(A)$ is a list of any other conditions that are asserted by the program — for example in the generic choice operator.

3.5 Healthiness conditions

In *UTP* we have the key notion of “healthiness condition”, which is a property that characterises all *healthy* predicates, *i.e.* all those predicates that “make sense”; before moving further on, we are going to list quickly the healthiness conditions that characterise this framework.

The first one (*feasibility*, $Dist1$) assures that for any program $\mathcal{P}(\delta, \delta')$ the probability of termination cannot be greater than that of having started:

$$\|\delta'\| \leq \|\delta\|$$

It is worth noticing that *abort* is often defined as *true*: the definition we gave in this framework is the weakest one that is healthy (and feasible) as well.

Another healthiness condition (*monotonicity*, $Dist2$), states that, for any deterministic program \mathcal{P} , increasing δ implies that the resulting δ' increases as well:

$$\mathcal{P}(\delta_1, \delta'_1) \wedge \mathcal{P}(\delta_2, \delta'_2) \wedge \delta_2 > \delta_1 \Rightarrow \delta'_2 \geq \delta'_1$$

A third healthiness conditions is that multiplication by a (not too large and non-negative⁵) constant distributes through commands (*scaling*, $Dist3$):

$$\forall a \in \mathbb{R}^+ \wedge \|a \cdot \delta\| \leq 1 \bullet \mathcal{P}(\delta, \delta') \Leftrightarrow \mathcal{P}(a \cdot \delta, a \cdot \delta')$$

⁴We can prove this by structural induction of the language syntax.

⁵Mathematically the relation holds also if this is not met, but in that case $a \cdot \delta$ is not a probability distribution.

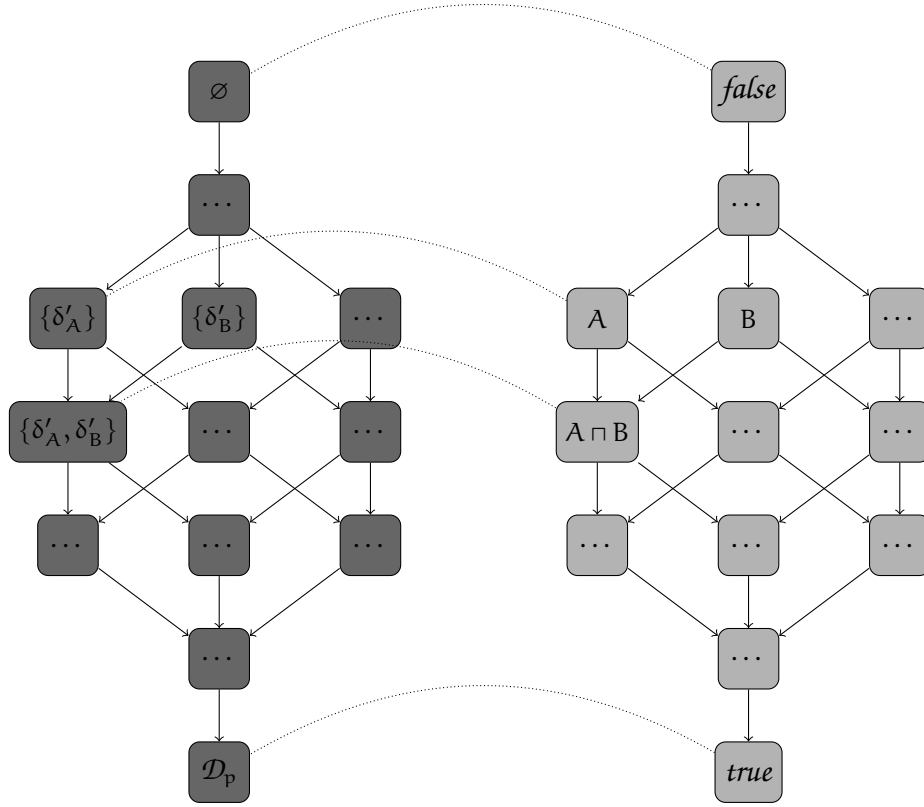


Figure 3.2: Program image lattice (\subseteq relation) and program lattice (\Rightarrow relation), represented in the case when $\delta \neq \epsilon$.

Proofs of these healthiness conditions are straightforward to be derived if we see the space of distributions as a vector space and can be found in §B.3.2.

Finally the purely random nondeterministic model adopted in the distributional framework yields a fourth healthiness condition Dist4 (*convexity*):

$$(\mathcal{P}_1 \sqcap \mathcal{P}_2)(\delta, \delta') \Rightarrow \delta' \geq \min(\mathcal{P}_1(\delta) \cup \mathcal{P}_2(\delta))$$

This poses restrictions on the space of possible program images, which is strictly a subset of $\wp\mathcal{D}$: this is analogous to the He set \mathcal{H} [MM04] as it is the set of all up-closed, Cauchy-closed, convex-closed sets of distributions.

3.6 The program lattice

Programs in standard *UTP* form a complete lattice with respect to the partial order induced by the implication relation [HH98].

Here we have a similar situation, as program images (parametric in δ) form a lattice with respect to the partial order induced by the set-inclusion relation: this relation among program images is isomorphic to the implication relation among programs:

$$A(\delta) \subseteq B(\delta) \Leftrightarrow A(\delta, \delta') \Rightarrow B(\delta, \delta')$$

In the case of $\delta = \epsilon$, the lattice collapses to a single element $\{\epsilon\}$. Otherwise the bottom element is clearly the set \mathcal{D}_p of all probability distributions, which is nothing but the program image of the aborting program *abort*, and a top element which is the empty set \emptyset .

In standard *UTP* the program lattice is completed by the top element *miracle*, which is *false*: we can see that we have a similar situation here, as a program which is satisfied by no after-distribution (*viz.* for which the corresponding program image is empty) is *false*.

Nevertheless we cannot define *miracle* in this way, because for any A we would have:

$$A \sqcap \textit{miracle} = \textit{false} = \textit{miracle}$$

whereas, according to the standard theory:

$$A \sqcap \textit{miracle} = A.$$

We obtain this same behaviour with the following definition:

$$\textit{miracle}(\delta, \delta') \triangleq (\delta = \epsilon) \wedge (\delta' = \epsilon).$$

Which means that miracles *can* happen, but only with null probability.

The program image $\textit{miracle}(\delta)$ is the empty set \emptyset if $\delta \neq \epsilon$ and $\{\epsilon\}$ otherwise: this shows that *miracle* is the top element of the program lattice.

3.7 Refinement

In standard *UTP* the refinement relation is the universal closure of the reverse implication, but when probability comes into play this is not enough any longer.

We want to capture as refinement the concept of a program being at least “as good” as another for all conditions, when it comes to the probability of satisfying them: this can be formalized by saying that a program A is refined by a program B when for all conditions and (before-)distributions, the minimal probability that an (after-)distribution from $A(\delta)$ satisfies a condition is less than that for $B(\delta)$.

We are going to give a definition for this in terms of a relation between the corresponding program images:

$$A \sqsubseteq B \triangleq \forall z, \delta \bullet \min(\|A(\delta)\langle z \rangle\|) \leq \min(\|B(\delta)\langle z \rangle\|)$$

The use of \min here mimics the way it is used in *pGCL* to define demonic choice.

This notion of refinement creates an order relation that is exactly the one created by the refinement relation used for *pGCL* [MM04]: the sets $\|A(\delta)\langle z \rangle\|$ and $\|B(\delta)\langle z \rangle\|$ contain the probabilities that the condition z is verified according to the possible after-distributions, and this definition requires that for any before-distributions the minimal probability for program A to satisfy z must be lower or equal than the minimal probability for program B to satisfy z — this is the definition given for *pGCL*.

The whole point of defining refinement this way was to show the similarity with *pGCL*; moving further and taking advantage of the structure of our framework, we can give an alternative

definition:

$$A \sqsubseteq B \triangleq \forall \delta \bullet B(\delta) \subseteq (A(\delta))^\Delta$$

where the *refinement set* $(A(\delta))^\Delta$ is the (up-, convex- and Cauchy-closed) set defined as:

$$(A(\delta))^\Delta \triangleq \{\delta_\Delta \mid \delta' \leq \delta_\Delta \leq \iota \wedge \delta' = \sum_{\delta'_i \in A(\delta(\pi_i))} \delta'_i \wedge \sum \pi_i = \iota\}$$

This set includes all after-distributions that are at least as great as those obtainable because of the nondeterminism in the behaviour of A : a program whose image lies in this set for all δ is a refinement of A , and hence the term “refinement set”.

From the above definition(s) we can easily demonstrate familiar refinement relations — the proofs boil down to expressing the refinement set of the left-hand side and the program images of the right-hand side:

$$A \sqsubseteq \textit{miracle}$$

$$A \sqcap B \sqsubseteq A$$

$$A \sqcap B \sqsubseteq B$$

And also:

$$A \sqcap B \sqsubseteq A \textit{ }_p \oplus B$$

$$A \sqcap B \sqsubseteq A \triangleleft c \triangleright B$$

This comes as no surprise, in fact:

$$A \textit{ }_p \oplus B = \exists \pi, \delta_A, \delta_B \bullet A(\delta(\pi), \delta_A) \wedge B(\delta(\bar{\pi}), \delta_B) \wedge \delta' = \delta_A + \delta_B \wedge \pi = p \cdot \iota$$

$$A \triangleleft c \triangleright B = \exists \pi, \delta_A, \delta_B \bullet A(\delta(\pi), \delta_A) \wedge B(\delta(\bar{\pi}), \delta_B) \wedge \delta' = \delta_A + \delta_B \wedge \pi = \iota \langle c \rangle$$

Concerning disjunction, we have that refinement fails to distinguish it from nondeterministic choice.

In fact we clearly have that:

$$A \sqcap B \sqsubseteq A \vee B$$

because

$$A \vee B = \exists \pi, \delta_A, \delta_B \bullet A(\delta(\pi), \delta_A) \wedge B(\delta(\bar{\pi}), \delta_B) \wedge \delta' = \delta_A + \delta_B \wedge \pi \in \{\epsilon, \iota\},$$

but we also have that

$$A \vee B \sqsubseteq A \sqcap B$$

as their refinement sets are the same:

$$\forall \delta \bullet ((A \vee B)(\delta))^\Delta = (A \sqcap B)(\delta) = ((A \sqcap B)(\delta))^\Delta$$

We can use mutual refinement as a notion of equivalence:

$$A \sqcap B \Leftrightarrow A \vee B.$$

This result is due to the definition we have used for refinement, as we have used the traditional view of nondeterminism as *demonic* nondeterminism, *i.e.* that returning the worst possible result for any desired outcome: this is in line with the traditional use of disjunction as a definition

for demonic choice.

Alternative definitions of refinement may take advantage of the possibility to distinguish between the operators \sqcap and \vee — this is left for future work.

In general, from the definition of refinement and the monotonicity of generic choice, we can show that:

$$\mathcal{X}_2 \sqsubseteq \mathcal{X}_1 \Rightarrow \text{choice}(A, B, \mathcal{X}_1) \sqsubseteq \text{choice}(A, B, \mathcal{X}_2)$$

It is worth stressing that the reverse implication is false — a counterexample is given by the case of the disjunction operator, where we have that:

$$\begin{aligned} A \vee B &\sqsubseteq A \text{ }_p\oplus B \\ A \vee B &\sqsubseteq A \triangleleft c \triangleright B \end{aligned}$$

This can be explained by comparing the lattice induced by refinement with that induced by the implication ordering: the latter is a sublattice of the former, in fact elements that were not comparable before are now related by refinement.

3.7.1 Probabilistic refinement

We want to generalise things even further, and introduce a notion of *probabilistic refinement*:

$$A \stackrel{p}{\sqsubseteq} B \triangleq \forall z, \delta \bullet p \cdot \min(\|A(\delta)\langle z \rangle\|) \leq \min(\|B(\delta)\langle z \rangle\|)$$

We call this a *p-accurate* refinement, meaning that the refinement relation \sqsubseteq is true in a fraction p of the possible cases.

We can give this alternative definition as well, similarly as we did above:

$$A \stackrel{p}{\sqsubseteq} B \triangleq \forall \delta \bullet B(\delta) \sqsubseteq (p \cdot A(\delta))^\Delta$$

where $p \cdot A(\delta)$ is the set made of all elements of $A(\delta)$ multiplied by p .

Let p^* be the highest positive real number such that $A \stackrel{p^*}{\sqsubseteq} B$: this is the *accuracy* with which B refines A and is a measure of how “better” B is when compared to A in any possible case — and of course $p < 1$ implies that B is not as “good” as A .

It is immediate to see that the refinement relation we have defined before is a special case of this more generic operator for $p = 1$, *i.e.* it is a 1-accurate refinement⁶:

$$A \sqsubseteq B = A \stackrel{1}{\sqsubseteq} B$$

This definition makes it much more meaningful to have a deterministic program on the left-hand side of the refinement relation⁷: the utility of such a thing is for example that a deterministic specification can be refined probabilistically by a (potentially) nondeterministic imple-

⁶Or a 100%-accurate refinement, in case we prefer expressing p as a percentage.

⁷It is immediate to prove that a deterministic program A can be refined only by another program B , which has to be equivalent to A , *i.e.* such that $A \sqsubseteq B \Leftrightarrow B \sqsubseteq A$.

mentation, and the implementation accuracy is a piece of information of great value.

This notion of refinement may seem like generalisation for its own sake, but it has useful real-world applications — an example on medical devices can be found in [SSL10].

3.8 Summary

In this chapter we have introduced a framework which is suitable to model the state space of a program by means of probability distributions.

By using predicates on homogeneous relations among probability distributions, we can give a *UTP* semantics to programs, which include both probabilism and nondeterminism.

All programs satisfy certain healthiness conditions and form a complete lattice.

The next chapters present some case studies where we have fruitfully applied this framework towards their treatment in the style of *UTP*.

In §4 and §5 we are going to present two quite general ones, namely that of *pGCL* followed by a probabilistic version of the *UTP* design theory: our aim is to show how different probabilistic frameworks can be given a *UTP* semantics by means of predicates on distributions.

Other examples (two well-known problems, namely the Monty-Hall one and Rabin's coordination algorithm) and a discussion on the applicability of this methodology to protocol verification (as a representative example of other domains where probability plays an important role) are collected in Appendix D.

CHAPTER 4

pGCL

The first case study we have addressed is that of giving a *UTP* semantics to *pGCL*. The result is very similar to the semantics for the programs in §3.3.1 and §3.4:

$$\begin{aligned}
 \textit{abort} &\triangleq \|\delta'\| \leq \|\delta\| \\
 \textit{miracle} &\triangleq (\delta = \epsilon) \wedge (\delta' = \epsilon) \\
 \textit{skip} &\triangleq \delta' = \delta \\
 \underline{v} := \underline{e} &\triangleq \delta' = \delta \{\underline{e}/\underline{v}\} \\
 A; B &\triangleq \exists \delta_m \bullet A(\delta, \delta_m) \wedge B(\delta_m, \delta') \\
 A \triangleleft c \triangleright B &\triangleq \exists \delta_A, \delta_B \bullet A(\delta(c), \delta_A) \wedge B(\delta(-c), \delta_B) \wedge \delta' = \delta_A + \delta_B \\
 A \textit{ }_p\oplus B &\triangleq \exists \delta_A, \delta_B \bullet A(p \cdot \delta, \delta_A) \wedge B((1-p) \cdot \delta, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
 A \sqcap B &\triangleq \exists \pi, \delta_A, \delta_B \bullet A(\delta(\pi), \delta_A) \wedge B(\delta(\bar{\pi}), \delta_B) \wedge \delta' = \delta_A + \delta_B \\
 c * A &\triangleq \textit{lfp} X \bullet (A; X) \triangleleft c \triangleright \textit{skip}
 \end{aligned}$$

Figure 4.1: *UTP* semantics of *pGCL*.

To be noted that conditional, probabilistic and nondeterministic choice are all instances of the generic choice introduced in §3.4.1.

It is quite straightforward to link the relational semantics model from [HSM97; MM04]: in fact it is like *pGCL* restricts its scope to working only with point distributions, thus relating a before-state to an after-distribution, whereas we are combining different point distributions into a single probability distribution and provide an after-distribution given by the same combination of the after-distributions corresponding to each point distributions, thus relating a before-distribution to an after-distribution.

If we note the program space $(\mathcal{S} \rightarrow \wp \mathcal{D}, \Xi)$ from the relational semantics model as \mathcal{P}_R , and use \mathcal{P}_D for the program space $(\mathcal{D} \rightarrow \wp \mathcal{D}, \Xi)$ from our distributional model, the elements representing a program A in each of the two program spaces will be noted A_R and A_D respectively:

$$\begin{aligned}
 \mathcal{P}_R &\triangleq (\mathcal{S} \rightarrow \wp \mathcal{D}, \Xi) \\
 \mathcal{P}_D &\triangleq (\mathcal{D} \rightarrow \wp \mathcal{D}, \Xi) \\
 A_R &\in \mathcal{P}_R \\
 A_D &\in \mathcal{P}_D.
 \end{aligned}$$

For example:

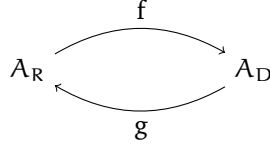
$$\textit{skip}_{\mathcal{P}_D}(\delta, \delta') = (\delta' = \delta)$$

and

$$\textit{skip}_{\mathcal{P}_R}(\sigma, \delta') = (\delta' = \eta_\sigma).$$

For the sake of clarity, in the following Figure 4.2 we will compare the relational semantics model of pGCL with the distributional semantics model of pGCL presented in Figure 4.1.

Let $f : \mathcal{P}_R \rightarrow \mathcal{P}_D$ be the function such that $A_D = f(A_R)$ and $g : \mathcal{P}_D \rightarrow \mathcal{P}_R$ the function such that $A_R = g(A_D)$:



If we take a *point distribution* $\eta_\sigma \triangleq \epsilon \uparrow \{\sigma \rightarrow 1\}$ we have that:

$$A_R(\sigma) = A_D(\eta_\sigma).$$

In the case of deterministic programs these are singleton sets:

$$A_R(\sigma) = A_D(\eta_\sigma) = \{\delta'_{(A, \sigma)}\},$$

where $\delta'_{(A, \sigma)}$ denotes the after-distribution reached by program A when it starts in the state σ ;

We can write these functions explicitly in the case of deterministic programs as:

$$\begin{aligned} A_R(\sigma) &= g(A_D)(\sigma) = A_D(\eta_\sigma) = \{\delta'_{(A, \sigma)}\} \\ A_D(\delta) &= f(A_R)(\delta) = \sum_{\zeta \in \mathcal{S}} \delta(\zeta) \cdot A_R(\zeta) = \sum_{\zeta \in \mathcal{S}} \delta(\zeta) \cdot \{\delta'_{(A, \zeta)}\}. \end{aligned}$$

We have that $f = g^{-1}$:

$$\begin{aligned} &g(f(A_R))(\sigma) \\ = &\quad \text{Definition of } f \\ &g\left(\sum_{\zeta \in \mathcal{S}} \delta(\zeta) \cdot A_R(\zeta)\right)(\sigma) \\ = &\quad \text{Definition of } g \\ &\sum_{\zeta \in \mathcal{S}} \eta_\sigma(\zeta) \cdot A_R(\zeta) \\ = &\quad \text{By definition, } \sigma \neq \zeta \Leftrightarrow \eta_\sigma(\zeta) = 0 \\ &\eta_\sigma(\sigma) \cdot A_R(\sigma) \\ = &\quad \text{By definition, } \eta_\sigma(\sigma) = 1 \\ &A_R(\sigma) \end{aligned}$$

Program A	Relational semantics $A_R(\sigma, \delta')$	Distributional semantics $A_D(\delta, \delta')$
<i>abort</i>	$\ \delta'\ < 1$	$\ \delta'\ \leq \ \delta\ $
<i>miracle</i>	<i>false</i>	$(\delta = \epsilon) \wedge (\delta' = \epsilon)$
<i>skip</i>	$\delta' = \eta_\sigma$	$\delta' = \delta$
$\underline{v} := \underline{e}$	$\exists \sigma' \bullet \sigma' = \sigma\{\underline{e}/\underline{v}\} \wedge \delta' = \eta_{\sigma'}$	$\delta' = \delta\{\underline{e}/\underline{v}\}$
$A; B$	$\exists \delta_m, \delta_1, \delta_2, \dots, \delta_n \bullet A(\sigma, \delta_m) \wedge (\bigwedge_{i=1}^{\#\text{dom}(\delta_m)} B(\sigma_i, \delta_i)) \wedge \delta' = \sum_{i=1}^{\#\text{dom}(\delta_m)} \delta_m(\sigma_i) \cdot \delta_i$	$\exists \delta_m \bullet A(\delta, \delta_m) \wedge B(\delta_m, \delta')$
$A < c \triangleright B$	$\sigma(c) \wedge A(\sigma, \delta') \vee \neg \sigma(c) \wedge B(\sigma, \delta')$	$\exists \delta_A, \delta_B \bullet A(\delta(c), \delta_A) \wedge B(\delta(-c), \delta_B) \wedge \delta' = \delta_A + \delta_B$
$A \oplus B$	$\exists \delta_A, \delta_B \bullet A(\sigma, \delta_A) \wedge B(\sigma, \delta_B) \wedge \delta' = p \cdot \delta_A + (1 - p) \cdot \delta_B$	$\exists \delta_A, \delta_B \bullet A(p \cdot \delta, \delta_A) \wedge B((1 - p) \cdot \delta, \delta_B) \wedge \delta' = \delta_A + \delta_B$
$A \sqcap B$	$A(\sigma, \delta') \vee B(\sigma, \delta')$	$\exists \pi, \delta_A, \delta_B \bullet A(\delta(\pi), \delta_A) \wedge B(\delta(\bar{\pi}), \delta_B) \wedge \delta' = \delta_A + \delta_B$
$c * A$	$\text{Ifp } X \bullet (A; X) < c \triangleright \text{skip}$	$\text{Ifp } X \bullet (A; X) < c \triangleright \text{skip}$

Figure 4.2: *pGCL* in the relational and distributional semantics model. It is worth to notice that we can recognise the Kleisli composition in the expression of sequential composition.

and

$$\begin{aligned}
& f(g(A_D))(\delta) \\
= & \text{Definition of } g \\
& f(A_R(\zeta))(\delta) \\
= & \text{Definition of } f \\
& \sum_{\zeta \in \mathcal{S}} \delta(\zeta) \cdot A_R(\zeta) \\
= & \text{Definition of } g \\
& A_D(\delta).
\end{aligned}$$

The generalization to the case of nondeterministic programs is trivial: in fact instead of dealing with singleton sets we have larger sets whose elements are related to the before-distribution by (at least) one deterministic execution path among the different alternatives made available by nondeterminism:

$$\begin{aligned}
A_R(\sigma) &= g(A_D)(\sigma) = A_D(\eta_\sigma) = \{\delta' \mid A_D(\eta_\sigma, \delta')\} \\
A_D(\delta) &= f(A_R)(\delta) = \sum_{\zeta \in \mathcal{S}} \delta(\zeta) \cdot A_R(\zeta) = \sum_{\zeta \in \mathcal{S}} \delta(\zeta) \cdot \{\delta' \mid A_D(\eta_\zeta, \delta')\}.
\end{aligned}$$

It is straightforward to relate the concept of equivalence from the relational semantics to equivalence according to the distributional framework: two programs are equivalent in the relational semantics when they map each state to the same set of distributions (*i.e.* they are the same function on \mathcal{P}_R), and similarly two programs are equivalent in the distributional framework when they map each before-distribution to the same set of after-distributions (*i.e.* they are the same function on \mathcal{P}_D).

If two programs are equivalent according to the relational semantics, the corresponding programs identified via the above link in the distributional framework (by application of the function f) are equivalent as well, as a result of the link being an isomorphism; clearly the converse is also true.

Things are slightly more complicated if we want to relate the *wp*-semantics from [MM04] to our semantic model. The way to do this is to observe that an expectation is a random variable (with non-negative real values), and as such it can be represented as a distribution χ in our framework. Then if χ' represents a post-expectation and A is a program, we can define the corresponding pre-expectation χ by computing the expected final weight of each state before A is run:

$$\chi(\sigma) = \min(\{\|\chi' \cdot \delta'\| \mid \delta' \in A(\eta_\sigma)\})$$

It shall be noted that this set is a singleton set for all deterministic constructs, and its cardinality can be larger only in the case that nondeterminism is present: in this semantics the model adopted is the demonic one and this results in the extraction of the (point-wise) minimum from that set.

4.1 Interaction of probabilistic and nondeterministic choice

In [MM04] the authors present a brief, well-known example on the interaction of probabilistic and nondeterministic choice: we are going to present it here as well, and solve it by means of our framework and compare the outcome and procedure with those presented in [MM04], where they use *pGCL*.

Let us take these two simple programs:

$$\begin{aligned} A &\triangleq x := 0 \sqcap x := 1 ; y := 0 \frac{1}{2} \oplus y := 1 \\ B &\triangleq x := 0 \frac{1}{2} \oplus x := 1 ; y := 0 \sqcap y := 1 \end{aligned}$$

We evaluate what is the probability that after each program has run we have $x = 1$, as well as the probability of having $x = y$.

We start by examining *A*:

$$\begin{aligned} &x := 0 \sqcap x := 1 ; y := 0 \frac{1}{2} \oplus y := 1 \\ \equiv &\quad \text{Translation: } A \\ &\exists \pi \bullet \delta' = \delta \langle \pi \rangle \{ \{0/x\} \} + \delta \langle \bar{\pi} \rangle \{ \{1/x\} \} ; \delta' = 1/2 \cdot \delta \{ \{0/y\} \} + 1/2 \cdot \delta \{ \{1/y\} \} \\ \equiv &\quad [d:P:Seq] \\ &\exists \pi, \delta_m \bullet \delta_m = \delta \langle \pi \rangle \{ \{0/x\} \} + \delta \langle \bar{\pi} \rangle \{ \{1/x\} \} \wedge \delta' = 1/2 \cdot \delta_m \{ \{0/y\} \} + 1/2 \cdot \delta_m \{ \{1/y\} \} \\ \equiv &\quad \text{One-point rule} \\ &\exists \pi \bullet \delta' = 1/2 \cdot (\delta \langle \pi \rangle \{ \{0/x\} \} + \delta \langle \bar{\pi} \rangle \{ \{1/x\} \}) \{ \{0/y\} \} + 1/2 \cdot (\delta \langle \pi \rangle \{ \{0/x\} \} + \delta \langle \bar{\pi} \rangle \{ \{1/x\} \}) \{ \{1/y\} \} \\ \equiv &\quad [p:D:Rmp:Lin] \\ &\exists \pi \bullet \delta' = 1/2 \cdot \delta \langle \pi \rangle \{ \{0/x\} \} \{ \{0/y\} \} + 1/2 \cdot \delta \langle \bar{\pi} \rangle \{ \{1/x\} \} \{ \{0/y\} \} + 1/2 \cdot \delta \langle \pi \rangle \{ \{0/x\} \} \{ \{1/y\} \} + 1/2 \cdot \delta \langle \bar{\pi} \rangle \{ \{1/x\} \} \{ \{1/y\} \} \end{aligned}$$

The final distribution $\delta'(\pi)$ is parametric in the weighting distribution π : let us try to use this to perform a worst-case analysis.

We can show that $\forall \pi \bullet \|\delta'(\pi)\langle x = y \rangle\| = 1/2$, which implies that We can show that $\forall \pi \bullet \|\delta'(\pi)\langle x = y \rangle\| = 1/2$, which implies that $\forall \delta \bullet \min(\|A(\delta)\langle x = y \rangle\|) = 1/2$:

$$\begin{aligned} \|\delta'(\pi)\langle x = y \rangle\| &= \left\| \left(1/2 \cdot \delta \langle \pi \rangle \{ \{0/x\} \} \{ \{0/y\} \} + 1/2 \cdot \delta \langle \bar{\pi} \rangle \{ \{1/x\} \} \{ \{1/y\} \} \right) \right\| \\ &= \left\| \left(1/2 \cdot \delta \langle \pi \rangle + 1/2 \cdot \delta \langle \bar{\pi} \rangle \right) \right\| = \|1/2 \cdot \delta\| = 1/2 \end{aligned}$$

But if we choose $\pi = \iota_{\delta'}$, we have $\delta'(\iota_{\delta'}) = 1/2 \cdot \delta \{ \{0/x\} \} \{ \{0/y\} \} + 1/2 \cdot \delta \{ \{0/x\} \} \{ \{1/y\} \}$ and therefore $\|\delta'(\iota_{\delta'})\langle x = 1 \rangle\| = 0$ — so the minimum of the weight of program *A* is 0.

Likewise, we examine B:

$$\begin{aligned}
& x := 0 \text{ } \frac{1}{2} \oplus x := 1 \text{ } ; y := 0 \sqcap y := 1 \\
\equiv & \quad \text{Translation: B} \\
& \delta' = 1/2 \cdot \delta\{0/x\} + 1/2 \cdot \delta\{1/x\} \text{ } ; \exists \pi \bullet \delta' = \delta\langle \pi \rangle\{0/y\} + \delta\langle \bar{\pi} \rangle\{1/y\} \\
\equiv & \quad [\text{d:P:Seq}] \\
& \exists \pi, \delta_m \bullet \delta_m = 1/2 \cdot \delta\{0/x\} + 1/2 \cdot \delta\{1/x\} \wedge \delta' = \delta_m\langle \pi \rangle\{0/y\} + \delta_m\langle \bar{\pi} \rangle\{1/y\} \\
\equiv & \quad \text{One-point rule} \\
& \exists \pi \bullet \delta' = (1/2 \cdot \delta\{0/x\} + 1/2 \cdot \delta\{1/x\})\langle \pi \rangle\{0/y\} + (1/2 \cdot \delta\{0/x\} + 1/2 \cdot \delta\{1/x\})\langle \bar{\pi} \rangle\{1/y\} \\
\equiv & \quad [\text{p:D:Rmp:Lin}] \\
& \exists \pi \bullet \delta' = 1/2 \cdot \delta\{0/x\}\langle \pi \rangle\{0/y\} + 1/2 \cdot \delta\{1/x\}\langle \pi \rangle\{0/y\} + 1/2 \cdot \delta\{0/x\}\langle \bar{\pi} \rangle\{1/y\} + 1/2 \cdot \delta\{1/x\}\langle \bar{\pi} \rangle\{1/y\}
\end{aligned}$$

The final distribution $\delta'(\pi)$ is parametric in a weighting distribution π and very similar to the resulting distribution after A, but with one crucial difference: $\langle \pi \rangle$ is put *after* the first occurrence of the remap operator.

We can show that $\|\delta'(\pi)\langle x = 1 \rangle\| = 1/2$.

But if we choose $\pi = \iota_{\delta'}\langle x = 1 \rangle$, we have $\delta'(\iota_{\delta'}\langle x = 1 \rangle) = 1/2 \cdot \delta\{0/x\}\{1/y\} + 1/2 \cdot \delta\{1/x\}\{0/y\}$ and therefore $\|\delta'(\iota_{\delta'}\langle x = 1 \rangle)\langle x = y \rangle\| = 0$

We have translated the programs and worked them out to express a predicate that links before-distributions with after-distributions: with this we can easily compute the minimum guaranteed probability that a condition will hold after the run of the program.

This is the same result we can achieve with pGCL, but:

- the notation is quite handy if we want to calculate the probability that some conditions holds, in the sense that we first derive the after-distribution and then we compute the probability that one of the conditions is true on that after-distribution, and then we go ahead by evaluating the other conditions on the same after-distribution. From the examples in [MM04] we can see how it is customary in pGCL to “proceed backwards” and derive preconditions step by step, so for each condition we would have to do the whole procedure from the start (or work with a parametric condition all along, which does not really make things simple);
- we are not forced to stick with the minimum probability (“hard-linked” in the pGCL definition of demonic choice), but we have a set containing the probabilities of every branch of the execution tree;
- it is straightforward to refine the demonic choice with any other kind of choice — we simply have to constrain the existentially quantified π .

CHAPTER 5

A probabilistic theory of designs

In §2.4 we have given a general overview of the *UTP* framework, and in particular we have presented the theory of designs in §2.4.1.

Here we are going to introduce a probabilistic version of this theory: through our distributional framework we obtain a richer theory where corresponding healthiness conditions hold (§5.1), even without the introduction of the auxiliary variables $o\mathcal{K}, o\mathcal{K}'$. Moreover the use of distributions enables us to evaluate the probability both of termination and of meeting a set of arbitrary postconditions as a function of the initial distribution (which determines the probability of meeting any required precondition).

A distinguishing characteristic of designs is the use of the auxiliary variables $o\mathcal{K}$ and $o\mathcal{K}'$. They are not sufficient in a probabilistic setting, as we need to be able to express quantitative information about the program also in terms of it having started or finished. We argue that this information is embedded in the distributions used to express programming constructs.

In fact the variable δ records implicitly if the program has started, as for each state σ it gives a precise probability that the program is in that initial state.

If δ is a full distribution (*i.e.* $\|\delta\| = 1$), then the program has started with probability 1: in some sense we can equate the predicate $o\mathcal{K} = \text{true}$ with the predicate $\|\delta\| = 1$. Conversely, a program for which $\delta = \epsilon$ has not started. Obviously there are all situations in between, where the fact of δ being a sub-distribution accounts for the program having started with probability $\|\delta\| < 1$.

Similarly if δ' is a full distribution, then the program terminates with probability 1: coherently we can equate the predicate $o\mathcal{K}' = \text{true}$ with the predicate $\|\delta'\| = 1$. In general the weight of δ' is the probability of termination: if the program reaches an after-distribution whose weight is strictly less than 1, then termination is not guaranteed (and in particular if $\delta' = \epsilon$ it is certain that it will not terminate).

Given a standard design $Pre \vdash Post$ we can easily derive the corresponding probabilistic design by using the observation above:

$$\begin{aligned} Pre \vdash Post &\equiv o\mathcal{K} \wedge Pre \Rightarrow o\mathcal{K}' \wedge Post \\ &\equiv \|\delta\| = 1 \wedge Pre \Rightarrow \|\delta'\| = 1 \wedge Post \\ &\equiv \|\delta\langle Pre \rangle\| = 1 \Rightarrow \|\delta'\langle Post \rangle\| = 1 \end{aligned}$$

This expression tells us that we have a valid design if whenever the before-distribution δ is a full distribution which is null everywhere Pre is not satisfied (and therefore $\delta = \delta\langle Pre \rangle$), then the resulting after-distribution δ' is a full distribution which is null everywhere $Post$ is not satisfied (and therefore $\delta' = \delta'\langle Post \rangle$).

This gives us a theory of *pGCL* programs that always terminate.

We can easily redefine assignment, in the same style as it has been redefined to make it a valid construct according to the theory of designs:

$$\begin{aligned} \underline{v} := \underline{e} &\hat{=} \text{true} \vdash \delta' = \delta \{\{e/v\}\} \\ &\equiv \text{ok} \wedge \text{true} \Rightarrow \text{ok}' \wedge \delta \{\{e/v\}\} \\ &\equiv \|\delta\| = 1 \Rightarrow \|\delta'\| = 1 \wedge \delta' = \delta \{\{e/v\}\} \end{aligned}$$

This states that an assignment is a valid design only if the expression e is defined everywhere in the state space: in fact undefinedness of e causes $\delta \{\{e/v\}\}$ to be a sub-distribution and therefore $\underline{v} := e$ reduces to *false*.

We can redefine *skip* in a similar way:

$$\begin{aligned} \text{skip} &\hat{=} \text{true} \vdash \delta' = \delta \\ &\equiv \text{ok} \wedge \text{true} \Rightarrow \text{ok}' \wedge \delta \\ &\equiv \|\delta\| = 1 \Rightarrow \|\delta'\| = 1 \wedge \delta' = \delta \\ &\equiv \|\delta\| = 1 \Rightarrow \delta' = \delta \end{aligned}$$

This new version of *skip* states that the after-distribution is the same as the before-distribution (and therefore it does not alter the weight, so this can be left implicit), but as any other design it reduces to *true* if δ is not a full distribution.

The bottom of the lattice is *abort*, which is again *true* as in the standard theory:

$$\begin{aligned} \text{abort} &\hat{=} \text{false} \vdash \text{false} \\ &\equiv \text{ok} \wedge \text{false} \Rightarrow \text{ok}' \wedge \text{false} \\ &\equiv \text{false} \Rightarrow \text{false} \\ &\equiv \text{true} \\ &\equiv \text{false} \Rightarrow \text{true} \\ &\equiv \text{ok} \wedge \text{false} \Rightarrow \text{ok}' \wedge \text{true} \\ &\equiv \text{false} \vdash \text{true} \end{aligned}$$

The standard definition of the construct *chaos* is

$$\begin{aligned} \text{chaos} &\hat{=} \text{true} \vdash \text{true} \\ &\equiv \text{ok} \wedge \text{true} \Rightarrow \text{ok}' \wedge \text{true} \\ &\equiv \text{ok} \Rightarrow \text{ok}' \\ &\equiv \|\delta\| = 1 \Rightarrow \|\delta'\| = 1 \end{aligned}$$

This is a program that guarantees termination, but in an unspecified state. It is equivalent to:

$$\text{chaos} \equiv \text{true} \vdash \text{abort}_R,$$

where the subscript R indicates that we are talking of the relational version of *abort*, from §3.4.

The top of the lattice is *miracle*:

$$\begin{aligned}
\text{miracle} &\triangleq \text{true} \vdash \text{false} \\
&\equiv \text{ok} \wedge \text{true} \Rightarrow \text{ok}' \wedge \text{false} \\
&\equiv \text{ok} \Rightarrow \text{false} \\
&\equiv \neg \text{ok} \\
&\equiv \neg(\|\delta\| = 1) \\
&\equiv \|\delta\| < 1
\end{aligned}$$

This is equivalent to

$$\text{miracle} \equiv \text{true} \vdash \text{miracle}_R.$$

5.1 Healthiness conditions

These new definitions relying on the distributional framework satisfy the healthiness conditions H1–H4 as well (§2.4.1).

We can in fact prove that the following laws hold:

- left unit law:

$$\begin{aligned}
\text{skip}; \text{Pre} \vdash \text{Post} &\equiv \|\delta\| = 1 \Rightarrow \delta' = \delta; \|\delta\langle \text{Pre} \rangle\| = 1 \Rightarrow \|\delta'\langle \text{Post} \rangle\| = 1 \\
&\equiv \exists \delta_m \bullet \|\delta\| = 1 \Rightarrow \delta_m = \delta \wedge \|\delta_m\langle \text{Pre} \rangle\| = 1 \Rightarrow \|\delta'\langle \text{Post} \rangle\| = 1 \\
&\equiv \|\delta\langle \text{Pre} \rangle\| = 1 \Rightarrow \|\delta'\langle \text{Post} \rangle\| = 1 \\
&\equiv \text{Pre} \vdash \text{Post}
\end{aligned}$$

- right unit law:

$$\begin{aligned}
\text{Pre} \vdash \text{Post}; \text{skip} &\equiv \|\delta\langle \text{Pre} \rangle\| = 1 \Rightarrow \|\delta'\langle \text{Post} \rangle\| = 1; \|\delta\| = 1 \Rightarrow \delta' = \delta \\
&\equiv \exists \delta_m \bullet \|\delta\langle \text{Pre} \rangle\| = 1 \Rightarrow \|\delta_m\langle \text{Post} \rangle\| = 1 \wedge \|\delta_m\| = 1 \Rightarrow \delta' = \delta_m \\
&\equiv \|\delta\langle \text{Pre} \rangle\| = 1 \Rightarrow \|\delta'\langle \text{Post} \rangle\| = 1 \\
&\equiv \text{Pre} \vdash \text{Post}
\end{aligned}$$

- left zero law:

$$\begin{aligned}
\text{true}; \text{Pre} \vdash \text{Post} &\equiv \text{true}; \|\delta\langle \text{Pre} \rangle\| = 1 \Rightarrow \|\delta'\langle \text{Post} \rangle\| = 1 \\
&\equiv \exists \delta_m \bullet \text{true} \wedge \|\delta_m\langle \text{Pre} \rangle\| = 1 \Rightarrow \|\delta'\langle \text{Post} \rangle\| = 1 \\
&\equiv \exists \delta_m \bullet \|\delta_m\langle \text{Pre} \rangle\| = 1 \Rightarrow \|\delta'\langle \text{Post} \rangle\| = 1 \\
&\equiv \text{true}
\end{aligned}$$

- right zero law:

$$\begin{aligned}
Pre \vdash Post; true &\equiv \|\delta\langle Pre \rangle\| = 1 \Rightarrow \|\delta'\langle Post \rangle\| = 1; true \\
&\equiv \exists \delta_m \bullet \|\delta\langle Pre \rangle\| = 1 \Rightarrow \|\delta_m\langle Post \rangle\| = 1 \wedge true \\
&\equiv \exists \delta_m \bullet \|\delta\langle Pre \rangle\| = 1 \Rightarrow \|\delta_m\langle Post \rangle\| = 1 \\
&\equiv true
\end{aligned}$$

5.2 Recasting total correctness

The reason that led to the standard theory of designs was that programs fail to satisfy the left zero law in the relational theory.

In the distributional framework programming constructs do satisfy this law, as for any programming construct \mathcal{P} other than *abort* or *miracle* it is never the case that $\delta \notin fv(\mathcal{P})$.

For this reason we have:

$$\begin{aligned}
true; \mathcal{P}(\delta, \delta') &\equiv \exists \delta_m \bullet true \wedge \mathcal{P}(\delta_m, \delta') \\
&\equiv \exists \delta_m \bullet \mathcal{P}(\delta_m, \delta') \\
&\equiv true
\end{aligned}$$

Similarly the right zero law is satisfied as well, along with the left and right unit laws: healthiness conditions equivalent to H1–H4 hold here as well.

Following this observation it appears that restricting the reasoning to programs with guaranteed termination is somehow limiting, as guaranteed termination is not an actual real-world feature of programs: programs must be reasonably reliable, but failure is always a possibility.

The reason for this may be inherent to the fact that programs are run on hardware which is susceptible of failure, as well as being a consequence of the way a program is designed (for example the implementation of a probabilistic algorithm where termination is probabilistic as well).

We can fully exploit the potential of the distributional framework towards modelling these situations by removing the constraints on the weights of the before- and after-distributions — so we use the programming constructs in Figure 4.1 exactly with the semantics presented there. The role of preconditions and postconditions is that of restricting the range of acceptable before- and after-distributions (and therefore act as restrictions to be applied to δ and δ' respectively) — this allows us to express desirable characteristics of a program in great detail, for example:

- $\mathcal{P} \wedge \|\delta'\| = 1$ requires \mathcal{P} to guarantee termination;
- $\mathcal{P} \wedge \|\delta'\| > 0.95$ requires \mathcal{P} to terminate with at least 95% probability;
- $\mathcal{P} \wedge \|\delta'\langle Post \rangle\| > 0.95$ requires \mathcal{P} to terminate with at least 95% probability in a state satisfying *Post*;
- $Pre \Rightarrow \mathcal{P} \wedge \|\delta'\langle Post \rangle\| > 0.95$ requires \mathcal{P} to terminate with at least 95% probability in a state satisfying *Post* whenever it starts in a state satisfying *Pre*;
- $\|\delta\langle Pre \rangle\| > 0.98 \Rightarrow \mathcal{P} \wedge \|\delta'\langle Post \rangle\| > 0.95$ requires \mathcal{P} to terminate with at least 95% probability in a state satisfying *Post* whenever the probability of *Pre* being satisfied at the beginning is at least 0.98;

- ...

All healthiness conditions deriving from the distributional framework (Dist1–Dist4) obviously hold here as well; with a small modification we can recast the notion of total correctness by restricting Dist1 to a variant Dist1-TC (which implies Dist1), stating that:

$$\|\delta\| = \|\delta'\|$$

This requires a program to terminate with the same probability p with which it has started:

$$\|\delta\| = p \wedge Pre \Rightarrow \|\delta'\| = p \wedge Post$$

5.3 Link with the standard model

Standard designs have observations $\sigma\mathcal{K}, \sigma'\mathcal{K}' \in \mathbb{B}$ and $\sigma, \sigma' \in \mathcal{S}$: a standard design is a predicate $\mathcal{P}_S(\sigma, \sigma', \sigma\mathcal{K}, \sigma'\mathcal{K}')$ that states that a program started (if $\sigma\mathcal{K}$ is true) in the state σ ends (if $\sigma'\mathcal{K}'$ is true) in the state σ' .

Probabilistic designs have observations $\delta, \delta' \in \mathcal{D}$: a probabilistic design is a predicate $\mathcal{P}_D(\delta, \delta')$ stating that a before-distribution δ will be transformed into the after-distribution δ' .

Informally we require the two approaches to yield the same results when we are dealing with point distributions, *i.e.* when the probability of being in a given state is 1.

In order to formalise the link between these two worlds, we define the linking predicate L as:

$$\begin{aligned} L((\delta, \delta'), (\sigma, \sigma', \sigma\mathcal{K}, \sigma'\mathcal{K}')) &\triangleq \sigma\mathcal{K} \Leftrightarrow (\|\delta\| = 1) \wedge \sigma'\mathcal{K}' \Leftrightarrow (\|\delta'\| = 1) \\ &\quad \wedge \delta = \eta_\sigma \wedge \delta' = \eta_{\sigma'} \end{aligned}$$

This linking predicate allows us to introduce the following *Galois connections*; first we define the weakest probabilistic design corresponding to a standard design \mathcal{P}_S :

$$\forall \sigma, \sigma', \sigma\mathcal{K}, \sigma'\mathcal{K}' \bullet L((\delta, \delta'), (\sigma, \sigma', \sigma\mathcal{K}, \sigma'\mathcal{K}')) \Rightarrow \mathcal{P}_S(\sigma, \sigma', \sigma\mathcal{K}, \sigma'\mathcal{K}')$$

Analogously, the strongest standard design corresponding to a probabilistic design \mathcal{P}_D is:

$$\exists \delta, \delta' \bullet L((\delta, \delta'), (\sigma, \sigma', \sigma\mathcal{K}, \sigma'\mathcal{K}')) \wedge \mathcal{P}_D(\delta, \delta')$$

It is easy to see that all programming constructs from the probabilistic theory that have homologue ones in the standard theory are linked to them, with the restriction of operating only on point distributions, otherwise they reduce to *abort*.

5.3.1 Weakening the link

This linking predicate is a bit too strong, as it maps many interesting program constructs to the aborting program: an example is that of generic choice, which has no homologue in the standard theory. Ideally a better option would be to relax some constraints and to map generic choice to nondeterministic choice rather than to *abort*.

In other words we are aiming at a link that loses all probabilistic information about the possible after-states and flattens it to a mere list of them.

This is not straightforward, as the linking predicate L in some sense verifies consistency of δ with respect to $\sigma, o\mathcal{K}$ and of δ' with respect to $\sigma', o\mathcal{K}'$: when the support¹ of the distribution has more than one element, the relation between δ and a state from its domain is too weak to be useful.

The situation is similar to that of a 3D-space, where dots are characterised by their x, y, z coordinates: a transformation creates a space with coordinates x', y', z' , whose relation with the undashed coordinates cannot in general be captured by a relation that mentions only one undashed and one dashed coordinate.

So far we have seen standard designs as relations:

$$\mathcal{P}_S : \mathcal{S} \times \mathbb{B} \rightarrow \mathcal{S} \times \mathbb{B}$$

but in order to build a more useful link we turn to this other interpretation:

$$\mathcal{P}_{\wp S} : \mathcal{S} \times \mathbb{B} \rightarrow \wp \mathcal{S} \times \mathbb{B}$$

which maps a state to what we may term its program image $\mathcal{P}(\sigma)$ (as it is a similar concept to that of program image introduced in §A.2.3), which contains all of the possible after-states reachable from a given before-state:

$$\mathcal{P}(\sigma) = \{\sigma' \mid \mathcal{P}_S(\sigma, \sigma')\}$$

All deterministic standard constructs map a state to a singleton set, whereas nondeterministic choice maps it to larger sets.

The interpretation of the predicate $\mathcal{P}_{\wp S}(\sigma, \alpha', o\mathcal{K}, o\mathcal{K}')$ is therefore that \mathcal{P} has started (if $o\mathcal{K}$ is true) in the state σ and has ended (if $o\mathcal{K}'$ is true) in a state $\sigma' \in \alpha'$:

$$\mathcal{P}_{\wp S}(\sigma, \alpha', o\mathcal{K}, o\mathcal{K}') \equiv \bigvee_{\sigma' \in \alpha'} \mathcal{P}_S(\sigma, \sigma', o\mathcal{K}, o\mathcal{K}')$$

With this in mind we can define the following linking predicate:

$$\begin{aligned} L_{\wp}((\delta, \delta'), (\sigma, \alpha', o\mathcal{K}, o\mathcal{K}')) &\triangleq o\mathcal{K} \Leftrightarrow (\|\delta\| = 1) \wedge o\mathcal{K}' \Leftrightarrow (\|\delta'\| = 1) \\ &\quad \wedge \delta = \eta_{\sigma} \wedge \text{supp}(\delta') = \alpha' \end{aligned}$$

We can state the variants of the Galois connections above as:

$$\begin{aligned} \forall \sigma, \alpha', o\mathcal{K}, o\mathcal{K}' \bullet L_{\wp}((\delta, \delta'), (\sigma, \alpha', o\mathcal{K}, o\mathcal{K}')) &\Rightarrow \mathcal{P}_{\wp S}(\sigma, \alpha', o\mathcal{K}, o\mathcal{K}') \\ \exists \delta, \delta' \bullet L_{\wp}((\delta, \delta'), (\sigma, \alpha', o\mathcal{K}, o\mathcal{K}')) &\wedge \mathcal{P}_D(\delta, \delta') \end{aligned}$$

5.4 Considerations on a pCSP theory

We have seen that the *UTP* theory of *CSP* is built on that of designs, with the introduction of three other pairs of auxiliary variables, notably *wait*, *tr*, *ref* and their dashed counterparts.

We recall their roles in the theory:

- *wait*, *wait'* are boolean variables recording if the program is waiting for interaction with

¹We remind the reader that the support of a function is the set of points where the function is not zero-valued: $\text{supp}(\delta) \triangleq \text{dom}(\delta) \setminus \ker(\delta)$.

the environment;

- tr, tr' record the list of events happened during the program run;
- ref, ref' are sets containing the event refused by the program.

They are in addition to $o\mathcal{K}, o\mathcal{K}'$, already added when going from the relational theory towards the concept of designs: the distributional framework presented in §3 spared us from having to add these variables when creating the concept of probabilistic designs, as we do not need to use them — we have in fact argued that this information is contained implicitly in the distributions δ, δ' , as their weight corresponds exactly to the probability that a particular program step has started or finished, respectively.

Information about divergent states remains implicit in the distributions: the probability of being in such a situation is precisely $(1 - \|\delta'\|)$.

In some sense the “ok” part of a distribution is mapped to the support of δ' , whereas the “not-ok” part gets disregarded.

We can therefore build on the theory of probabilistic designs presented in §5 to get to a probabilistic theory of CSP only by adding the remaining three pairs of auxiliary variables.

Their meaning will be the same as in the standard theory. The question is: what is the best way to embed them in the probabilistic theory of designs? We may be tempted to introduce them as auxiliary variables alongside with the program distribution, but the same reasons that were brought up to decide in favour of an approach that lumps all of the variables together into a single composite observation variable, require us to work on states with the following shape:

$$\sigma : (\underline{v}, wait, tr, ref) \rightarrow \mathcal{W} \times \mathbb{B} \times \text{Event-seq} \times \text{Event-set},$$

where \mathcal{W} is the set of possible values for the program variables.

This allows us to embed all of the remaining auxiliary variables in the state domain, and therefore this simplifies the definitions of the different programming constructs and healthiness conditions, compared to the traditional reactive definitions that use $o\mathcal{K}, wait, tr, ref$ as auxiliary variables — this is a novel approach.

5.4.1 R1

For example let us take the traditional R1, which states:

$$P = P \wedge (tr \leq tr')$$

In a probabilistic world this must hold point-wise for each couple of states (σ, σ') from the before- and after-distributions that are related by the program.

If we write this in the case of a single state σ (i.e. we take a point distribution η_σ as the before-distribution), the trace in the before-state σ must be a prefix of the trace in all of the possible after-states σ' from the support² of the resulting after-distribution δ' .

This must hold true for all states in the state space, so the formulation of the probabilistic R1 is:

$$P(\delta, \delta') = P(\delta, \delta') \wedge \left(\forall \sigma \bullet P(\eta_\sigma, \delta') \implies (\forall \sigma' \in \text{supp}(\delta') \bullet \sigma(tr) \leq \sigma'(tr)) \right)$$

where we have used the functional notation $\sigma(tr)$ to stand for the evaluation of tr on σ .

²The support of a function is the subset of its domain where the function is non-null.

From this formulation we can clearly see that divergent states do not take part in the verification of the condition R1; in addition, it is worth pointing out that, according to this definition, a totally divergent program (which yields $\delta' = \epsilon$ for any initial δ) is R1-healthy.

5.4.2 R2

Healthiness condition R2 states that the initial value of tr cannot have any influence on the evolution of the program, which determines only the tail ($tr' - tr$):

$$P(tr, tr') = \exists s \bullet P(s, s \sim (tr' - tr))$$

As we did above we first look at the case of point distributions, where a possible formulation is the following:

$$P(\eta_\sigma, \delta') = \exists s \bullet P(\eta_\sigma \{s/tr\}, \delta' \{s \sim (tr - \sigma(tr))/tr\})$$

Here we have used the remap operator to “change” the value of the trace in the spirit of R2 over all states.

This gives a sort of “substitution rule” that allows us to replace a state σ with another state ζ that differs only for the value of tr in the before-distribution, whereas in the after-distribution a part δ'_σ (accounting for the contribution of σ) is replaced by a new part δ'_ζ (accounting for the contribution of ζ):

$$P(\delta, \delta') = \forall \sigma \exists s \bullet (\zeta = \sigma \{s/tr\}) \wedge P((\delta - \delta_\sigma + \delta_\zeta), (\delta' - \delta'_\sigma + \delta'_\zeta))$$

where δ_σ and δ_ζ are point distributions scaled down by the probability of σ , i.e. $\delta_\sigma = \delta(\sigma) \cdot \eta_\sigma$ and $\delta_\zeta = \delta(\zeta) \cdot \eta_\zeta$.

5.4.3 R3

Before getting to R3 we have to define the probabilistic version of the reactive *skip*, denoted *skip*.

According to the standard theory of reactive designs [HH98], *skip* is defined as:

$$skip \triangleq (\neg ok \wedge tr \leq tr') \vee (ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref)$$

This definition has to distinguish the case of divergence (when it does not enforce anything other than trace elongation) from the case of non-divergence (when it states that all variables are left unchanged), and as a result it is much more complicated than the pure relational skip which is simply:

$$\underline{v}' = \underline{v}$$

The choice of embedding the auxiliary variables in the state function σ (and having left all information about divergence implicit in δ, δ') starts to pay out here, as it enables us to keep such an easy definition as well:

$$skip \triangleq \delta' = \delta$$

In other words all non-divergent states are preserved as they are, whereas now there is no statement on divergent states — other than the implicit one that the overall probability of divergence must be left unchanged.

R3 does not mention tr, tr' :

$$P = \text{skip} \triangleleft \text{wait} \triangleright P$$

As a result this is pretty straightforward to express in a probabilistic setting, as we can use directly the semantics of the conditional construct presented in §3.3:

$$\begin{aligned}
& \text{skip} \triangleleft \text{wait} \triangleright P \\
\equiv & \text{definition of conditional} \\
& \exists \delta_A, \delta_B \bullet \text{skip}(\delta\langle \text{wait} \rangle, \delta_A) \wedge P(\delta\langle \neg \text{wait} \rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
\equiv & \text{definition of skip} \\
& \exists \delta_A, \delta_B \bullet \text{skip}(\delta\langle \text{wait} \rangle, \delta_A) \wedge \delta_A = \delta\langle \text{wait} \rangle \wedge P(\delta\langle \neg \text{wait} \rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
\equiv & \text{one-point rule on } \delta_A \\
& \exists \delta_B \bullet \text{skip}(\delta\langle \text{wait} \rangle, \delta\langle \text{wait} \rangle) \wedge P(\delta\langle \neg \text{wait} \rangle, \delta_B) \wedge \delta_B = \delta' - \delta\langle \text{wait} \rangle \\
\equiv & \text{one-point rule on } \delta_B \\
& \text{skip}(\delta\langle \text{wait} \rangle, \delta\langle \text{wait} \rangle) \wedge P(\delta\langle \neg \text{wait} \rangle, \delta' - \delta\langle \text{wait} \rangle)
\end{aligned}$$

And therefore.

$$P(\delta, \delta') = \text{skip}(\delta\langle \text{wait} \rangle, \delta\langle \text{wait} \rangle) \wedge P(\delta\langle \neg \text{wait} \rangle, \delta' - \delta\langle \text{wait} \rangle)$$

We split the before-distribution into two parts, one where wait is true and that equals the corresponding after-distribution, and one where it is not and that has evolved into the difference of the total after-distribution δ' and the part $\delta\langle \text{wait} \rangle$ that did not evolve.

This can be simplified down to:

$$P(\delta, \delta') = P(\delta\langle \neg \text{wait} \rangle, \delta' - \delta\langle \text{wait} \rangle).$$

5.4.4 CSP1 and CSP2

Another advantage of the distributional framework is that compliance with the remaining two healthiness conditions, namely CSP1 and CSP2, is subsumed by other conditions, as we are now going to show.

In standard CSP, CSP1 states that:

$$P = P \vee (\neg \text{ok} \wedge tr \leq tr')$$

As all information about divergent states is kept implicit in distributions, we can argue that this healthiness condition is stripped down to the identity $P = P$.

In some sense, all states which are “ok” evolve from the support of the before-distribution towards a state in the support of the after-distribution, which is “ok”, or diverge to a state, which is “not-ok” and is not part of the support of the after-distribution, effectively getting out of the game; on the other hand all states which are “not-ok” are not part of the support of the before-distribution and have no means to get back in the game.

Probabilistic reactive programs are therefore CSP1-healthy by design, as $P(\delta, \delta')$ already states

that either a state evolves according to what is described by δ, δ' or diverges. Our formalism does not allow us to express the trace-elongation property for divergent states, but after all it is not crucial information — they diverge, that's already bad enough! The other healthiness condition, CSP2, states that:

$$P; J = P$$

where

$$J \triangleq \underline{v}' = \underline{v} \wedge (o\mathcal{K} \Rightarrow o\mathcal{K}') \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref$$

In the probabilistic world based on distribution this reduces to:

$$P; skip = P$$

which is nothing but H3. In fact:

$$\begin{aligned} J &\triangleq (\underline{v}' = \underline{v} \wedge (o\mathcal{K} \Rightarrow o\mathcal{K}') \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref) \\ &\equiv (\underline{v}' = \underline{v} \wedge o\mathcal{K}' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref) \vee \\ &\quad \vee (\underline{v}' = \underline{v} \wedge \neg o\mathcal{K} \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref) \\ &\equiv skip \vee (\underline{v}' = \underline{v} \wedge \neg o\mathcal{K} \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref) \end{aligned}$$

And again the part with $\neg o\mathcal{K}$ gets disregarded, thus the reactive program J in the probabilistic world coincides with $skip$ — and there we have that CSP2 collapses to H3.

These brief considerations on a $pCSP$ theory based on distributions are preliminary work, that indicates possible directions of future research.

CHAPTER 6

Conclusion

In the previous chapters we have presented in detail a framework specifically conceived to deal with programs featuring both probabilism and nondeterminism in the style of *UTP*.

The novelty in our framework is its approach to probability, which arises from a distributional model of the state space of a program: thanks to this view of the world we are able to express concisely the relations between the situation before and after the execution of a program.

The algebraic properties of the distributional model allow us to reason on distributions and programs using theories and tools borrowed from the domain of vector spaces.

According to the programs-are-predicates view of the world shared by the *UTP* research community, we use relations among distributions to give a predicate semantics to several program constructs: as a result this has enabled us to treat efficiently different cases, all collected in §D.

From these case studies we can see the key strengths of this framework, as well as understanding what the inevitable weaknesses are.

On the plus side the notation is very compact: all probabilistic information is hidden within the distributions used in the predicates representing the different programs, and side conditions are kept to a minimum; ordinary logic and algebraic rules make it straightforward to reason on the semantics of a program.

The algebraic properties of the framework make it suitable for mechanization of several steps in a verification procedure (left for future work): an example is the use of vectors and matrices as an elegant formalism to deal with boring and error-prone computations, which can be handled by a computer.

Moreover several program properties can be inferred by inspection of the corresponding matrices in the vector formalism.

Our framework is very flexible, as some features are not “hard-wired” within. An example is nondeterminism: we have adopted a neutral view of it, because it shows no demonic behaviour *per se*, as instead it is customary in several other frameworks. The demonic behaviour is a consequence of a particular definition of refinement, but nothing forbids us from using a different definition to look at programs from a different perspective.

Last, but not least, our framework integrates well with *UTP*: this was a central requirement for us, as one of the goal of our research was the integration of probability into the *UTP* framework — as we have discussed in §2.4 the different approaches used so far were not deemed totally satisfactory by many people.

What perhaps we feel as the main shortcoming of our approach is highlighted by §5.4: the treatment of traces is quite complicated and far from being intuitive. As a result we feel that

our framework may not be the most efficient approach to $pCSP$ and, consequently, the approach to a probabilistic version of *Circus* should take this into account.

This is one of the reasons why finding a slightly different approach to model $pCSP$ would be an interesting direction for future work.

Another interesting line of research would involve probability and security, as they are two sides of the same medal. We have shown a possible application of our framework in the domain of protocol verification, but we can probably go beyond that and this would probably yield interesting results in the field of security.

All of this would benefit greatly from the presence of some tool support, as on one hand it would make the framework more effective and easy to use, on the other hand a mechanized approach could take advantage of the different mathematical properties of the framework towards an efficient implementation.

States and distributions

A.1 Variables, types and expressions

Variables are the elements of a program that we can use to observe and model the behaviour of a program.

In *UTP* it is customary to distinguish between *observational* and *auxiliary variables*: the former directly correspond to the variables that a program can access and modify, whereas the latter are an abstraction which records some particular behaviour of the program, for example termination or being in a waiting state.

We use the notation \mathcal{V} to stand for the set of all variables of a program.

UTP offers constructs to introduce new variables within a given scope, nevertheless we will not take this possibility into account: the framework we are going to present can be modified to support this construct in a conceptually easy and straightforward way, but in spite of the conceptual simplicity this requires the introduction of complicated machinery to handle this — we will get back to this point to clarify what we mean in §A.2 and §A.3.

We assume therefore that \mathcal{V} is fixed and cannot change dynamically as the program runs.

Variables of a given *type* can assume a value from a set characteristic of that type; for the variable $v_i \in \mathcal{V}$ we note the set of its possible values as \mathcal{W}_i :

$$\mathcal{W}_i \triangleq \text{type}(v_i).$$

Having a possibly different type for each variable adds unnecessary complexity to the framework we are going to introduce; this is easily manageable and does not require a big deal of effort, but we believe that it shifts attention away from more delicate matters: for this reason we assume that there is no type distinction among the different variables, whose possible values will therefore lie in the set of all types \mathcal{W} :

$$\forall i \bullet \mathcal{W}_i \subset \mathcal{W}.$$

For the sake of simplicity, let us assume that \mathcal{W} contains integers and booleans only.

An *expression* on variables is a combination of constants and variables, combined by operators; the set of all expressions is \mathcal{E} .

A notable subset of \mathcal{E} is that of boolean expressions, which we will refer to as *conditions*.

A.2 States

Program states define the mapping that associates each variable to its corresponding value, in other words they are functions from the set of all variables to the set of their possible values:

$$\sigma : \mathcal{V} \rightarrow \mathcal{W}.$$

We do not allow for the case of a variable not being associated to any value (and hence σ is a total function): as in the real world a variable points to a location in the memory, in the worst case that location contains garbage, but still the operation of retrieving the value of a variable returns a result, which will be interpreted as a value of the appropriate type.

The domain of σ , which is \mathcal{V} , is defined as its *alphabet*:

$$\text{alph}(\sigma) \triangleq \text{dom}(\sigma) = \mathcal{V}.$$

So the choice we made in §A.1, which disallows for dynamic changes in \mathcal{V} , results in dealing with states with the same alphabet.

The set of all states is the state space \mathcal{S} .

It is handy to lump all the variables of \mathcal{V} together into a single vector of variables \underline{v} :

$$\underline{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}.$$

so that we can give an alternative definition of state as

$$\sigma : \{\underline{v}\} \rightarrow \underline{\mathcal{W}},$$

where $\underline{\mathcal{W}}$ is the cartesian product of n copies of \mathcal{W} .

As a result each state σ maps the variable vector \underline{v} to the corresponding vector of values \underline{w} :

$$\sigma = \underline{v} \mapsto \underline{w}$$

where

$$\underline{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \in \underline{\mathcal{W}}$$

and the map operator has been lifted to operate element-wise on vectors:

$$\underline{v} \mapsto \underline{w} \triangleq \{v_i \mapsto w_i \mid 1 \leq i \leq n\}.$$

A.2.1 Evaluation of an expression

An expression e can be evaluated in a state σ by replacing each variable v_i it mentions with the value $\sigma(v_i)$ that is contained by that variable in that state: doing the calculations with these values returns the *evaluation of the expression e on the state σ* , which is the value $\text{eval}_\sigma(e)$.

Here is a recursive definition, where k is a constant, \mathcal{F} a n -ary function and e_i an expression:

$$\begin{aligned}\text{eval}_\sigma(k) &\triangleq k \\ \text{eval}_\sigma(v_i) &\triangleq \sigma(v_i) \\ \text{eval}_\sigma(\mathcal{F}(e_1, e_2, \dots, e_n)) &\triangleq \mathcal{F}(\text{eval}_\sigma(e_1), \text{eval}_\sigma(e_2), \dots, \text{eval}_\sigma(e_n))\end{aligned}$$

As a shorthand notation for the evaluation function, we overload the function state:

$$\sigma(e) \triangleq \text{eval}_\sigma(e)$$

When an expression e contains only values and operators, we have that its evaluation is the same on any state, thus when the notation is clear from the context we will simply write e instead of $\text{eval}_\sigma(e)$ (or $\sigma(e)$, using the shorthand notation).

Using this, we can write that:

$$\sigma(e) = \text{eval}_\sigma(e^{\{\sigma(v_i)/v_i\}}) = \sigma(e^{\{\sigma(v_i)/v_i\}}) = e^{\{\sigma(v_i)/v_i\}}$$

In the case of a boolean expression (*condition*), we say that a state *satisfies* a condition c when it evaluates to *true* in that state.

As with variables and values, it is useful (in view of the §A.2.3 on assignments) to introduce some vector notation for expressions as well:

$$\sigma(\underline{e}) \triangleq \begin{pmatrix} \sigma(e_1) \\ \sigma(e_2) \\ \vdots \\ \sigma(e_n) \end{pmatrix},$$

where obviously

$$\underline{e} = \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix} \in \underline{\mathcal{E}}.$$

We use the following notation for simultaneous substitutions¹ $\{f_1/g_1\}\{f_2/g_2\}\dots\{f_n/g_n\}$:

$$\{f/g\} \triangleq \{f_1/g_1\}\{f_2/g_2\}\dots\{f_n/g_n\},$$

¹For this to make sense, it must be the case that $\forall i \neq j \bullet g_i \neq g_j$.

where

$$\underline{f} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix} \quad \text{and} \quad \underline{g} = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{pmatrix}$$

When the substitution $\{f/g\}$ is applied to a vector of expressions \underline{e} , the meaning is the following:

$$\underline{e}\{f/g\} \triangleq \begin{pmatrix} e_1\{f/g\} \\ e_2\{f/g\} \\ \vdots \\ e_n\{f/g\} \end{pmatrix}$$

The composition of two expression vectors \underline{f} and \underline{e} is defined as a particular substitution that involves the variable vector \underline{v} :

$$\underline{f} \circ \underline{e} \triangleq \underline{f}\{e/v\} = \begin{pmatrix} f_1\{e/v\} \\ f_2\{e/v\} \\ \vdots \\ f_n\{e/v\} \end{pmatrix}$$

We can read the notation $\underline{f} \circ \underline{e}$ as \underline{f} after \underline{e} .

Concerning the evaluation of this vector we have

$$\sigma(\underline{f} \circ \underline{e}) = \sigma(\underline{f}\{e/v\}) = \sigma(\underline{f}\{\sigma(e)/v\}) = \underline{f}\{\sigma(e)/v\}$$

This is equivalent to evaluating \underline{f} in a state ζ such that $\zeta(v) = \sigma(e)$.

Now it should be clear why we intentionally use a symbol like \circ and the word ‘‘after’’, which both remind of functional composition: if for every expression and variable vectors \underline{e} and \underline{v} we define an associated function $\underline{e}_v : \underline{\mathcal{W}} \rightarrow \underline{\mathcal{W}}$ as:

$$\underline{e}_v(\underline{w}) = \text{eval}_{v \rightarrow w}(\underline{e})$$

then for any state $\sigma = v \mapsto w$, we have that $\sigma(\underline{f} \circ \underline{e}) = \underline{f}_v(\underline{e}_v(w))$:

$$\begin{cases} \sigma(\underline{f} \circ \underline{e}) = \underline{f}_v(\underline{w}') \\ \underline{w}' = \underline{e}_v(\underline{w}) \end{cases}$$

When composing the same expression for $k \geq 1$ times, we use the following notation:

$$\underline{e}^k \triangleq \underbrace{\underline{e} \circ \underline{e} \circ \dots \circ \underline{e}}_{k \text{ times}}$$

We define that for $k = 0$ this notation has the following meaning:

$$\underline{e}^0 \triangleq v$$

A.2.2 Abstract states

An *abstract state* $\alpha \subseteq \mathcal{S}$ is a set of states:

$$\alpha \triangleq \{\sigma_1, \sigma_2, \dots, \sigma_n, \dots\}$$

The alphabet of an abstract state is defined as the set of all the different alphabets that appear in the abstract state:

$$\text{alph}(\alpha) \triangleq \{\mathcal{A} \mid \mathcal{A} = \text{alph}(\sigma) \wedge \sigma \in \alpha\}$$

So in the case of an abstract state containing all states having the same alphabet \mathcal{A} , its alphabet is the singleton set $\{\mathcal{A}\}$; when alphabets may in general vary from state to state, the largest such abstract state is noted $\mathcal{S}_{\mathcal{A}}$:

$$\mathcal{S}_{\mathcal{A}} \triangleq \{\sigma \mid \text{alph}(\sigma) = \mathcal{A}\}$$

We write it this way as it is the largest subset of \mathcal{S} , whose elements are all those states with alphabet \mathcal{A} : our assumption of all states having the same alphabet simplifies the presentation of our framework, as the state space \mathcal{S} we are dealing with is actually $\mathcal{S}_{\mathcal{A}}$.

We say that an abstract state satisfies a condition c when all its elements do.

We define the *restriction* of an abstract state through a condition c as a total function $_ \langle _ \rangle : (\wp \mathcal{S} \times \mathcal{E}) \rightarrow \wp \mathcal{S}$, defined as follows:

$$\alpha \langle c \rangle \triangleq \{\sigma \mid \sigma \in \alpha \wedge \sigma(c) = \text{true}\}$$

We have that:

$$\alpha \langle c \rangle = \mathcal{S} \langle c \rangle \cap \alpha$$

Clearly if the condition is *true* we have:

$$\alpha \langle \text{true} \rangle = \alpha$$

And obviously if the condition is *false* we have:

$$\alpha \langle \text{false} \rangle = \emptyset$$

A.2.3 Assignments

An assignment performed in a state σ is an operation $v_i := e_i$, that updates the value contained in v_i with $\sigma(e_i)$.

We use the following notation for n simultaneous assignments of the expressions e_1, e_2, \dots, e_n to the variables $v_1, v_2, \dots, v_n \in \mathcal{V}$:

$$\underline{v} := \underline{e} \triangleq \bigwedge_{i=1}^n (v_i := e_i),$$

where we remind that:

$$\underline{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \quad \text{and} \quad \underline{e} = \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix}.$$

Let us now define the *inverse-image set* for a generic assignment $\underline{v} := \underline{e}$, after which the new state σ' describes the new mapping for the updated vector of variables \underline{v} :

$$\text{Inv}(\underline{v} := \underline{e}, \sigma') \triangleq \{ \sigma \mid \sigma'(\underline{v}) = \sigma(\underline{e}) \wedge \sigma \in \mathcal{S}_{\text{alph}(\sigma')} \}.$$

Clearly the above definition is simplified by the assumption that all states σ have the same alphabet, as per assumption in §A.1.

We can generalize this to an abstract state α' :

$$\text{Inv}(\underline{v} := \underline{e}, \alpha') \triangleq \bigcup_{\sigma' \in \alpha'} \text{Inv}(\underline{v} := \underline{e}, \sigma').$$

The abstract state $\text{Inv}(\underline{v} := \underline{e}, \alpha')$ is the set of all the possible states before the assignment that are compatible with the result of the new mapping being in the abstract state α' .

Due to the fact that the evaluation of an expression is an injective function we have that:

$$\text{Inv}(\underline{v} := \underline{e}, \sigma_1) \cap \text{Inv}(\underline{v} := \underline{e}, \sigma_2) = \emptyset \Leftrightarrow \sigma_1 \neq \sigma_2.$$

Thanks to this property, if the evaluation of an expression \underline{e} is defined on all of the states belonging to an abstract state α , we have that it is possible to partition α through \underline{e} .

In fact if we have a relation \mathcal{R}_e defined as:

$$\sigma_1 \mathcal{R}_e \sigma_2 \Leftrightarrow \sigma_1(\underline{e}) = \sigma_2(\underline{e}).$$

This is an equivalence relation among states belonging to an abstract state α , that is partitioned into equivalence classes corresponding to inverse-image sets α' :

$$\alpha = \bigcup_{\sigma' \in \alpha'} \text{Inv}(\underline{v} := \underline{e}, \sigma'),$$

where each class is represented by a state σ such that $\sigma(\underline{e}) = \sigma'(\underline{v})$.

Nested inverse-image set : $\text{Inv}(\underline{v} := \underline{e}, \text{Inv}(\underline{v} := \underline{f}, \{\sigma\})) = \text{Inv}(\underline{v} := \underline{f}\{\underline{e}/\underline{v}\}, \{\sigma\})$

The inverse-image set will play an important role, as the *remap* operator (which is to be introduced in §A.3.4) will be defined in terms of it and is a crucial component of our framework, as it is needed to give semantics to assignment statements.

A.3 Distributions

In §3.1 we have informally introduced the concept of *probability distribution* over the state space, as a means of assigning a probability to each state in \mathcal{S} : this is a particular instance of a

more general concept.

A *distribution* χ is in general a partial function χ , that maps some states from \mathcal{S} to real numbers:

$$\chi : \mathcal{S} \rightarrow \mathbb{R}.$$

We refer to each real number χ_i as the *weight* of the corresponding state σ_i ; we use \mathcal{D} to note the set of all possible distributions.

The partiality of χ is a technical device that allows us to treat efficiently cases when it is assumed implicitly that some states are mapped to the value 0, but the corresponding pair does not belong to χ : the whole framework is built in such a way that the operators do not distinguish between a given distribution and another one, which differs only for the addition of some otherwise undefined states that are mapped to a null weight.

For many application we need to have a measure the collective weight of all states of a distribution χ : we refer to this as to the *distribution weight*, and it is trivially the sum over its domain of all the state weights:

$$\|\chi\| \triangleq \sum_{\sigma \in \text{dom}(\chi)} \chi(\sigma)$$

This operation can be lifted to a set $\mathcal{X} \subseteq \mathcal{D}$ of distributions in an obvious way:

$$\|\mathcal{X}\| \triangleq \{\|\chi\| \mid \chi \in \mathcal{X}\}$$

In general the *alphabet* of a distribution is defined as the set of all the different alphabets that appear in the distribution domain:

$$\text{alph}(\chi) \triangleq \text{alph}(\text{dom}(\chi)).$$

Clearly in the case of a fixed alphabet \mathcal{A} shared by all states, this reduces to the singleton set $\{\mathcal{A}\}$.

A particular distribution is the *empty distribution* $\epsilon_\alpha : \mathcal{S} \rightarrow \mathbb{R}$, which is a distribution such that $\text{dom}(\epsilon_\alpha) = \alpha$ and $\text{img}(\epsilon_\alpha) = \{0\}$, viz. it maps each state in the abstract state α to 0:

$$\epsilon_\alpha \triangleq \{\sigma \mapsto 0 \mid \sigma \in \alpha\}$$

Another particular distribution is the *unity distribution* $\iota_\alpha : \mathcal{S} \rightarrow \mathbb{R}$, which is a distribution such that $\text{dom}(\iota_\alpha) = \alpha$ and $\text{img}(\iota_\alpha) = \{1\}$, viz. it maps each state in the abstract state α to 1:

$$\iota_\alpha \triangleq \{\sigma \mapsto 1 \mid \sigma \in \alpha\}$$

We define the following shortcuts:

$$\begin{array}{ll} \epsilon_{\mathcal{A}} \triangleq \epsilon_{\mathcal{S}_{\mathcal{A}}} & \iota_{\mathcal{A}} \triangleq \iota_{\mathcal{S}_{\mathcal{A}}} \\ \epsilon_{\chi} \triangleq \epsilon_{\text{dom}(\chi)} & \iota_{\chi} \triangleq \iota_{\text{dom}(\chi)} \\ \epsilon \triangleq \emptyset & \iota \triangleq \iota_{\mathcal{S}} \end{array}$$

We define the *restriction of a distribution through a condition* c as follows:

$$\chi\langle c \rangle \triangleq \left\{ \sigma \mapsto \chi(\sigma) \mid \sigma \in \text{dom}(\chi)\langle c \rangle \right\}$$

This is a distribution where all states satisfying the condition c are mapped to the same weight as in the original distribution χ , whereas those on which c evaluates to false are remapped to the null weight.

The following properties hold for a restricted distribution, some of which are immediately inferable from the definition and some others which are proven in Appendix C:

Restriction through conjunction of conditions : $\chi\langle c_1 \wedge c_2 \rangle = \chi\langle c_1 \rangle\langle c_2 \rangle = \chi\langle c_2 \rangle\langle c_1 \rangle$

Restriction through equivalent condition : $(c_1 \Leftrightarrow c_2) \Rightarrow \chi\langle c_1 \rangle = \chi\langle c_2 \rangle$

Restriction through implied condition (I) : $(c_2 \Rightarrow c_1) \Leftrightarrow \chi\langle c_1 \rangle\langle c_2 \rangle = \chi\langle c_2 \rangle$

Restriction through implied condition (II) : $(c_1 \Rightarrow \neg c_2) \Rightarrow \chi\langle c_1 \rangle\langle c_2 \rangle = \epsilon$

In case we have conditions c_σ and c_α selecting (*i.e.* evaluating true only on) a single state σ and an abstract state α respectively, we simplify the notation as follows:

$$\delta\langle \sigma \rangle \triangleq \delta\langle c_\sigma \rangle \qquad \delta\langle \alpha \rangle \triangleq \delta\langle c_\alpha \rangle$$

The expression of the distribution weight, in the case of restricted distributions, can be simplified by excluding from the sum all states which are mapped to 0 by the restriction and therefore we obtain the following:

- $\|\delta\langle c \rangle\| = \sum_{\sigma \in \text{dom}(\chi)\langle c \rangle} \delta(\sigma)$
- $\|\delta\langle \sigma \rangle\| = \delta(\sigma)$
- $\|\delta\langle \alpha \rangle\| = \sum_{\sigma \in \alpha} \delta(\sigma)$.

We define the *point distribution* (with domain α) as the restriction of a unity distribution to a single state, *viz.* all the distribution weight is concentrated in a single state which maps to 1:

$$\eta_{\sigma, \alpha} \triangleq \iota_\alpha\langle \sigma \rangle$$

And clearly we have that:

$$\|\eta_{\sigma, \alpha}\| = 1$$

We also define the *restriction of a distribution through another distribution* as follows:

$$\chi_1\langle \chi_2 \rangle \triangleq \left\{ \sigma \mapsto \chi_1(\sigma) \cdot \chi_2(\sigma) \mid \sigma \in \text{dom}(\chi_1) \cap \text{dom}(\chi_2) \right\}$$

Commutativity of this operation derives directly from the definition:

$$\chi_1\langle \chi_2 \rangle = \chi_2\langle \chi_1 \rangle.$$

The reason why we call these operations in a similar way is that if we can see that the restriction of a distribution through a condition as a generalization to distributions of the restriction of abstract states through a condition, the restriction of a distribution through a distribution can be seen as a further generalization:

$$\chi\langle c \rangle = \chi\langle \iota_\chi\langle c \rangle \rangle$$

All of this can be lifted to a set $\mathcal{X} \subseteq \mathcal{D}$ of distributions in an obvious way:

- $\mathcal{X}\langle c \rangle \triangleq \{\chi\langle c \rangle \mid \chi \in \mathcal{X}\}$
- $\mathcal{X}\langle \chi \rangle \triangleq \{\xi\langle \chi \rangle \mid \xi \in \mathcal{X}\}$

A.3.1 Operations on distributions

Arithmetical operations can intuitively be lifted point-wise to operate on distributions².

The *sum* of distributions χ_1 and χ_2 is a mapping where each state is mapped to the sum of the weights from the two distributions:

$$\chi_1 + \chi_2 \triangleq \left\{ \sigma \mapsto (\chi_1(\sigma) + \chi_2(\sigma)) \right\}$$

From this definition we can derive that:

- $\|\chi_1 + \chi_2\| = \|\chi_1\| + \|\chi_2\|$
- $(\chi_1 + \chi_2)\langle \pi \rangle = \chi_1\langle \pi \rangle + \chi_2\langle \pi \rangle$

This can be lifted elementwise to the case of two sets of distributions $\mathcal{X}, \mathcal{Y} \subseteq \mathcal{D}$:

$$\mathcal{X} + \mathcal{Y} \triangleq \{\chi + \xi \mid \chi \in \mathcal{X}, \xi \in \mathcal{Y}\}.$$

Thanks to the latter property we can split a distribution into two other distributions, where all the elements of one satisfy a given condition c , while the elements of the other do not:

$$\chi = \chi\langle c \rangle + \chi\langle \neg c \rangle$$

This is a key property as whenever a program working on some distribution behaves differently according to the state it is acting on, it is necessary to be able to split the distribution in this way.

Distributions can be scaled through point-wise multiplication by a real number. This is the *multiplication by a scalar number*, which is then defined as:

$$n \cdot \chi \triangleq \left\{ \sigma \mapsto (n \cdot \chi(\sigma)) \right\}$$

We have previously the restriction of a distribution through another distribution in terms of a point-wise product: depending on the situation it is useful to think of this alternatively as a restriction or as a product of distributions, so we define the *product* of two distributions as:

$$\chi_1 \cdot \chi_2 \triangleq \chi_1\langle \chi_2 \rangle$$

As this is just a make-up for the restriction of a distribution through another distribution, commutativity of the product of distributions derives directly from its definition:

$$\chi_1 \cdot \chi_2 = \chi_2 \cdot \chi_1$$

²We are assuming that we are dealing with distributions on the same state space — a trivial generalization can be used if this is not the case, by adding all states that are missing from either distribution and have them mapped to 0.

All of this can be lifted to a set $\mathcal{X} \subseteq \mathcal{D}$ of distributions in an obvious way:

- $\mathcal{X} \cdot \mathcal{X} \triangleq \{\xi \cdot \chi \mid \xi, \chi \in \mathcal{X}\}$
- $n \cdot \mathcal{X} \triangleq \{n \cdot \xi \mid \xi \in \mathcal{X}\}$

It is possible to introduce a partial order among distributions:

$$\chi_1 \leq \chi_2 \triangleq \forall \sigma \in \text{dom}(\chi_1) \cup \text{dom}(\chi_2) \bullet \chi_1(\sigma) \leq \chi_2(\sigma)$$

A.3.2 Specific types of distributions

Some specific types of distributions play special roles in our framework, so we are going to term them accordingly.

A *weighting distribution* π is a distribution mapping states from its domain to real values in the range $[0..1]$:

$$\pi: \mathcal{S} \rightarrow [0..1]$$

We use \mathcal{D}_w to note the subset of \mathcal{D} of all weighting distributions; the partial order defined above results in a complete partial order on the \mathcal{D}_w , where the top element is ι and the bottom element is ϵ .

Given a weighting distribution π , we define its complementary weighting distribution $\bar{\pi}$ as:

$$\bar{\pi} \triangleq \iota_\pi - \pi$$

Restriction : $\pi_1 \langle \pi_2 \rangle \in \mathcal{D}_w$

A *probability distribution* δ is a weighting distribution such that $\|\delta\| \leq 1$.

We can further specify by using the term *full probability distribution* when $\|\delta\| = 1$ and the term *probability subdistribution*³ when $\|\delta\| < 1$

We use \mathcal{D}_p to note the subset of \mathcal{D}_w of all probability distributions.

Restriction : $\delta \langle \pi \rangle \in \mathcal{D}_p$

In the case of probability distributions we can recognise that $\delta(\sigma)$ is the function of σ which is usually referred to as the *probability mass function*: it represents the way the probability is distributed depending on σ .

So for a pair $(\sigma_i \mapsto p_i) \in \delta$ we will refer to the weight p_i as to the *probability* of the state σ_i . Likewise we will talk of the probability of an abstract state rather than of its weight: in fact if we see a state as an *outcome*, we can see an abstract state as an *event* (*i.e.* a set of outcomes).

A.3.3 A simpler notation

It is apparent that having to deal with distributions with different domains requires the use of a lot of different subscripts and side-conditions, which are conceptually void and are rather an exercise of patience and due diligence.

³A small *caveat* here: when Morgan *et al.* talk about “probability subdistributions”, they refer to the case of the cumulative probability being *less or equal to 1*; we chose to use a stricter connotation of the term “subdistribution”, as we found it less confusing.

For this reason in some cases, for example when the state space is finite, it is helpful to think of a distribution as a total function $\chi : \mathcal{S} \rightarrow \mathbb{R}$, where the undefined mappings are replaced by the null mapping; in this case it is handy to use the vector notation, so that:

$$\chi = \begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \vdots \\ \sigma_n \end{pmatrix} \mapsto \begin{pmatrix} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_n \end{pmatrix}.$$

This is justified by the fact that we can see a distribution as an element of a vector space: we will explore later on (§B) in more detail the concept of distributions as vectors along with its implications.

In order to be able to better focus on more important matters, we are going to work under this assumption from now on — and coherently we will omit any subscript carrying information on the distribution domain.

Moreover, in this work we will be talking mostly of probability distributions, so we will usually be referring to them simply as “distributions”, eventually distinguishing between the case of a “full distribution” and that of a “subdistribution”.

Whenever we want to use this term in the more general meaning we have used so far, we will rather use “general distributions”.

A.3.4 The *remap* operator

We have previously hinted at the importance of the *remap* operator within our framework. The reason for its importance is that it is a technical device to deal with all the complicated machinery that is responsible for the correct modelling of assignments.

When states distributed according to a probability distribution are modified by an assignment $\underline{v} := \underline{e}$, the original before-distribution δ is transformed into the after-distribution $\delta' = \delta \{\!\{ \underline{e} / \underline{v} \}\!\}$, where the postfix operator $\{\!\{ \underline{e} / \underline{v} \}\!\}$ is the remap operator: it is therefore an effective way of keeping track of the changes affecting a distribution δ as it “evolves” assignment after assignment towards a final distribution δ' .

The *remap* operator is defined in terms of the weight of the inverse-image set for the corresponding assignment:

$$\delta \{\!\{ \underline{e} / \underline{v} \}\!\} \triangleq \left\{ \sigma' \mapsto \left\| \delta \left(\text{Inv}(\underline{v} := \underline{e}, \{\sigma'\}) \right) \right\| \mid \text{alph}(\sigma') \in \text{alph}(\delta) \right\}$$

In other words, for each after-state σ' from the domain of the resulting distribution $\delta \{\!\{ \underline{e} / \underline{v} \}\!\}$, we have that the corresponding weight is made up of the original weight of all before-states σ that have been transformed into σ' because of the assignment $\underline{v} := \underline{e}$:

$$(\delta \{\!\{ \underline{e} / \underline{v} \}\!\})(\sigma') = \left\{ \sum \delta(\sigma) \mid \sigma' = \sigma \dagger \{ \underline{v} \mapsto \text{eval}_\sigma(\underline{e}) \} \right\}$$

From the definition we can see that after applying the remap operator the alphabet of the resulting distribution is the same as the alphabet of the original distribution:

$$\text{alph}(\delta \{\!\{ \underline{e} / \underline{v} \}\!\}) = \text{alph}(\delta)$$

Quite often it is the case that we are dealing with an assignment e_i to a single variable v_i from \mathcal{V} : in this case we overload the notation adopted so far and use the postfix operator $\{\{e_i/v_i\}\}$.

Sometimes the same assignment is repeated several times, one after the other, so we define a compact notation for this case:

$$\delta \{\{e/v\}\}^k \triangleq \underbrace{\delta \{\{e/v\}\} \{\{e/v\}\} \dots \{\{e/v\}\}}_{k \text{ times}}$$

Properties

From the definitions of sum and multiplication, we have that the remap operator is a linear one:

$$(x \cdot \delta \{\{e/v\}\} + y \cdot \delta \{\{f/v\}\}) \{\{g/v\}\} = x \cdot \delta \{\{e/v\}\} \{\{g/v\}\} + y \cdot \delta \{\{f/v\}\} \{\{g/v\}\}$$

Here are some other properties:

Composition (I) : $\delta \{\{e/v\}\} \{\{f/v\}\} = \delta \{\{f\{e/v\}/v\}\}$

Composition (II) : $\delta \{\{e/v\}\} \{\{f/v\}\} = \delta \{\{f \circ e/v\}\}$

Composition (III) : $\delta \{\{e/v_i\}\} \{\{f/v_j\}\} = \delta \{\{e, f\{e/v_i\}\}/(v_i, v_j)\}$

Composition (IV) : $\delta \{\{e/v_i\}\} \{\{f/v_i\}\} = \delta \{\{f\{e/v_i\}/v_i\}\}$

Iteration : $\delta \{\{e/v\}\}^k = \delta \{\{e^k/v\}\}$

Commutativity (I) : $\delta \{\{e/v_i\}\} \{\{f/v_j\}\} = \delta \{\{f\{e/v_i\}/v_j\}\} \{\{e/v_i\}\}$ iff $v_j \notin fv(e)$

Commutativity (II) : $\delta \{\{e/v_i\}\} \{\{f/v_j\}\} = \delta \{\{f/v_j\}\} \{\{e/v_i\}\}$ iff $v_i \notin fv(f) \wedge v_j \notin fv(e)$

Expression substitution : $\delta \{\{f = g\}\} \{\{e/v\}\} = \delta \{\{f = g\}\} \{\{e\{f/g\}/v\}\}$

Contradiction : $\forall \sigma \in \text{dom}(\delta) \bullet \sigma(c\{e/v\}) = \text{false} \wedge \delta \neq \epsilon \Leftrightarrow \delta \{\{e/v\}\}(c) = \epsilon$

Assertion : $\forall \sigma \in \text{dom}(\delta) \bullet \sigma(c\{e/v\}) = \text{true} \Leftrightarrow \delta \{\{e/v\}\}(c) = \delta \{\{e/v\}\}$

Remapping a condition : $\delta \{\{e/v\}\}(c) = \delta \{\{c\{e/v\}\}\} \{\{e/v\}\}$

Weight of a distribution after remapping : $\|\delta \{\{e/v\}\}\| = \|\delta\|$ iff $\sigma(\underline{e})$ is defined in $\text{dom}(\delta)$

APPENDIX B

Distributions as vectors

When we are working with distributions, we are in effect dealing with a vector space with size equal to the cardinality of the state space \mathcal{S} : a distribution χ over the (finite¹) state space \mathcal{S} can be seen as a linear combination of *point distributions* η_σ with coefficients x_σ ranging in \mathbb{R} :

$$\chi = \sum_{\sigma \in \mathcal{S}} x_\sigma \cdot \eta_\sigma$$

Similarly a weighting distribution π can be seen as the same linear combination, *but* with coefficients w_σ ranging over $[0..1]$:

$$\pi = \sum_{\sigma \in \mathcal{S}} w_\sigma \cdot \eta_\sigma$$

Finally a probability distribution δ can be seen as the same linear combination, with coefficients p_σ ranging in $[0..1]$ (as for weighting distributions), which have the additional property of being (at most) one-summing:

$$\delta = \sum_{\sigma \in \mathcal{S}} p_\sigma \cdot \eta_\sigma \wedge \sum_{\sigma \in \mathcal{S}} p_\sigma \leq 1$$

When writing the distributions as a linear combination of point distributions, we have implicitly chosen the set of all point distribution as a basis of the vector space. In particular it is an orthonormal basis, which we refer to as the *canonical basis* (made of the *canonical generators*). We can therefore represent all distributions as vectors of the coefficients from the corresponding linear combinations:

$$\underline{\chi} \triangleq \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad \underline{\pi} \triangleq \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \quad \underline{\delta} \triangleq \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}$$

For the sake of clarity we assume that states are numbered from 1 to n and use the subscript i instead of σ_i ; nevertheless we will be very pedantic regarding the vector notation and underline a distribution whenever we refer to its vector representation.

B.1 Operations on vectors

First of all we can introduce a partial order among these vectors by overloading the \leq operator:

$$\underline{\chi}_1 \leq \underline{\chi}_2 \triangleq \forall i \bullet x_{i1} \leq x_{i2}$$

¹We assume the state space \mathcal{S} to be finite and with cardinality n , but we can deal also with an infinite state space; moreover we assume that the domain of all distributions coincides with the state space — in other words we are taking the trivial completion, that maps to 0 all states in $\mathcal{S} \setminus \text{dom}(\chi)$.

This definition matches that given in A.3.1; analogously this results in a complete partial order on the subset of weighting distributions, where we have a top element $\underline{1}$, *i.e.* the unit vector, and a bottom element $\underline{\epsilon}$, *i.e.* the zero vector:

$$\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \leq \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

We use the L_1 -norm (also known as *Manhattan norm*) as the norm of choice for this space²:

$$\|\underline{x}\| \triangleq \|\underline{x}\|_1 = \sum |x_i|$$

In the case of distributions from the positive cone of \mathcal{D} , noted \mathcal{D}^+ , this coincides with the notion of distribution weight given in A.3.

Occasionally we will use also the L_∞ -norm (also known as *Chebyshev norm*), which we will use later on:

$$\|\underline{x}\|_\infty = \max_i \{|x_i|\}$$

\mathcal{D} with the Manhattan norm can therefore be seen as a *metric space*, where the distance function is defined as:

$$d(\underline{x}_1, \underline{x}_2) \triangleq \|\underline{x}_1 - \underline{x}_2\|$$

We use the conventional definition of the scalar product of two vectors:

$$\underline{x}_1 \cdot \underline{x}_2 \triangleq \sum x_{i1} x_{i2} = \underline{x}_1^T \underline{x}_2$$

With this definition we have that $\|\underline{x}\| = \underline{1} \cdot \underline{x}$ for all distributions from the positive cone \mathcal{D}^+ (which includes weighting and probability distributions).

We use the conventional definition of the entry-wise product³ of two vectors:

$$\underline{x}_1 \circ \underline{x}_2 \triangleq \underline{x}_H \quad \text{where } x_{iH} = x_{i1} \cdot x_{i2}$$

If we use the notation $\text{diag}(\underline{v})$ to denote the diagonal matrix whose element (i, i) is the i -th component of \underline{v} , we can write the following equality:

$$\underline{x}_1 \circ \underline{x}_2 = \text{diag}(\underline{x}_1) \underline{x}_2$$

Addition and multiplication by a scalar have the usual definitions as well.

²From now on we will systematically omit the indication of the sum index whenever it is obvious from the context and it ranges from 1 to n .

³Also known as *Hadamard product*.

B.1.1 The set \mathcal{D}_p

Using the above definition of norm we can express the property that probability distributions are at most 1-summing:

$$\|\underline{\delta}\| \leq 1$$

The set \mathcal{D}_p of all probability distributions is therefore the intersection of the positive cone of \mathcal{D} and the (closed) n -ball of radius 1, centred in ϵ : $\mathcal{D}_p = \mathcal{D}^+ \cap \mathcal{B}_1[\epsilon]$.

B.2 Programs as matrices

We have already stated that a deterministic program A can be seen as a *distribution-transformer*, as it “turns” a probability distribution δ into a post-distribution δ' .

From a different angle we can see this as a homeomorphism in the vector space of distributions⁴, and as such it can be described as a square matrix \underline{A} of size $n \times n$ (so that it is conformable to the product of a vector with n elements, *i.e.* a $n \times 1$ matrix), which we will refer to as the *program matrix*:

$$\underline{\delta}' = \underline{A} \underline{\delta}$$

which is, explicitly:

$$\begin{pmatrix} p'_1 \\ p'_2 \\ \vdots \\ p'_n \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}$$

We use the notations \underline{a}_{i*} and \underline{a}_{*j} to refer respectively to the i -th row (transposed) and to the j -th column of \underline{A} .

Coherently with the adoption of the L_1 -norm for vectors, we use it also as the norm of choice for matrices:

$$\|\underline{A}\| \triangleq \|\underline{A}\|_1 = \max_j \{\|\underline{a}_{*j}\|\}$$

The L_∞ -norm for matrices is the following:

$$\|\underline{A}\|_\infty = \max_i \{\|\underline{a}_{i*}\|_1\}$$

Therefore we have that:

$$\|\underline{A}\| \triangleq \|\underline{A}^T\|_\infty$$

We define a partial order among matrices by extending column-wise (or, equivalently, row-wise) the \leq -order on vectors:

$$\underline{A} \leq \underline{B} \triangleq \forall j \bullet \underline{a}_{*j} \leq \underline{b}_{*j} \equiv \forall i \bullet \underline{a}_{i*} \leq \underline{b}_{i*}$$

We use this to define a partial order among programs:

$$A \preceq B \triangleq \underline{A} \leq \underline{B}$$

⁴In particular, we will show that this homeomorphism maps probability distributions to probability distributions.

We can notice in passing that $A \leq B$ implies that $A \subseteq B$, as for any δ , $A \leq B$ implies that $B(\delta) \subseteq (A(\delta))^\Delta$ — graphically, the refinement set comprises all of the distributions contained between the hyperplane (with the smallest dimension) containing all distributions of $A(\delta)$ and the hyperplane of all distributions with unitary weight —; we cannot give a sensible definition for $A \leq B$ in the case of nondeterministic programs, so the reverse implication is false.

B.2.1 Interpretation of the columns of the program matrix

We can see that $\underline{\delta}'$ is a linear combination of the columns of \underline{A} , with coefficients in the range $[0..1]$ that sum up to 1 at most.

If $\underline{\delta} = \underline{\eta}_i$ we have that $\underline{\delta}' = \underline{\delta}'_{A_i}$, defined as:

$$\underline{\delta}'_{A_i} \hat{=} \underline{A} \underline{\eta}_i = \begin{pmatrix} a_{11} & \dots & a_{1i} & \dots & a_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & \dots & a_{ii} & \dots & a_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{ni} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} a_{1i} \\ \vdots \\ a_{ii} \\ \vdots \\ a_{ni} \end{pmatrix} = \underline{a}_{*i}$$

is the probability distribution we obtain if program A is run from state σ_i — in other words $\underline{\delta}'_{A_i}$ is the vector representation of the post-distribution $\delta'_{(A, \sigma_i)} = A(\eta_{\sigma_i})$.

We can therefore derive a healthiness condition for the elements in each column of A , which is that they are all positive and cannot sum up to more than 1 — and when the sum is exactly 1 this means that A is guaranteed to terminate whenever starting in state σ_i .

From this healthiness condition we can derive that probability distributions are indeed mapped to probability distributions.

Thus we can see $\underline{\delta}'$ as a linear combination of the probability distributions accounting for all of the possible outcomes of A , where the coefficient of the i -th possible outcome relative to state σ_i is the probability that A starts running in that state.

In other words the columns of \underline{A} are the generators of a vector space, which has the property that the representation of the before-distribution δ in that space coincides with the representation of δ' in \mathcal{D} : for this reason we will refer to $\underline{\delta}'_{A_i}$ as to the i -th *generator of A* ; therefore the canonical generators are those of the identity program \underline{I} .

We can relate this to how deterministic programs are viewed in [MM04, §5.1], where the space \mathcal{P}_d of deterministic programs is defined as:

$$\mathcal{P}_d \hat{=} (\mathcal{S} \rightarrow \mathcal{D}_p, \Xi)$$

A program A is therefore seen as the following relation:

$$A = \{(\sigma_i, \delta'_{A_i}) \mid \sigma_i \in \mathcal{S}\}$$

The matrix \underline{A} is a complete description of program A , as it contains all of the information

provided in the above relation:

$$\underline{\Delta} = \begin{pmatrix} \delta'_{A1} & \delta'_{A2} & \dots & \delta'_{An} \end{pmatrix}$$

B.2.2 Random Variables and *pGCL* Expectations

If we have a random variable $X' : \mathcal{S} \rightarrow \mathbb{R}$ which assigns a real value to every state in \mathcal{S} , we can compute its *expected value*⁵ $\mathcal{E}_{\delta'}(X')$ by summing over all states the value assigned to each state weighted by the probability of that state:

$$\sum_{\sigma \in \mathcal{S}} p_{\sigma} \cdot X'(\sigma)$$

We are interested in the case when the probability distribution over the state space is represented by δ' , as if we represent X' as a generic distribution $\underline{\chi}'$, then we can express its expected value as:

$$\mathcal{E}_{\delta'}(X') = \underline{\chi}' \cdot \underline{\delta}' = (\underline{\chi}')^T \underline{\delta}'$$

In [MM04] *expectations* are defined as functions that map each state to the expected value that a non-negative random variable will have at the end of the program.

This expectation, written for final states, specialises to a *post-expectation*:

$$\mathcal{E}_{\delta'}(X' | \delta' = \eta_i) = x_i$$

So this is actually the random variable X' itself seen as an expectation — in this way we can see programs as *expectation transformers*.

It is more interesting to write the corresponding *pre-expectation*, which gives the expected value of X when the program A has started in state σ_i , *i.e.* when the final distribution of program states is δ'_{Ai} :

$$\mathcal{E}_{\delta'}(X' | \delta = \eta_i) = \mathcal{E}_{\delta'}(X' | \delta' = \delta'_{Ai}) = \underline{\chi}' \cdot \underline{\delta}'_{Ai} = (\underline{\chi}')^T \underline{\delta}'_{Ai} = (\underline{\chi}')^T \underline{\Delta} \underline{\eta}_i$$

We can therefore express this pre-expectation as a generic distribution $\underline{\chi}$:

$$\underline{\chi}^T = (\underline{\chi}')^T \underline{\Delta} \begin{pmatrix} \underline{\eta}_1 & \underline{\eta}_2 & \dots & \underline{\eta}_n \end{pmatrix} = (\underline{\chi}')^T \underline{\Delta} \underline{I} = (\underline{\chi}')^T \underline{\Delta}$$

The pre-expectation is a random variable X that has the property of having the same expected value as X' if the initial and final probability distribution over the state space are represented by δ and δ' respectively:

$$(\underline{\chi}')^T \underline{\delta}' = (\underline{\chi}')^T \underline{\Delta} \underline{\delta} = \underline{\chi}^T \underline{\delta}$$

Because of the way they operate on expectations, these random variables are constrained to having non-negative values in [MM04]: here we can relax this constraint, as we are using random variables in a slightly different way, that allows more flexibility.

⁵Usually it is customary to refer to the expected value of a random variable as to its *expectation*: we will refrain from doing so, as this same term is used with a different (although not totally unrelated) meaning in the context of *pGCL*.

B.2.3 Interpretation of the rows of the program matrix

We can see that we are able to relate the above random variables using the transposition of program matrix \underline{A} :

$$\underline{\chi} = ((\underline{\chi}')^T \underline{A})^T = \underline{A}^T \underline{\chi}'$$

Similarly as above, we can see $\underline{\chi}$ as a linear combination of the columns of \underline{A}^T , *i.e.* as a linear combination of the (transposed) rows of \underline{A} .

Let us remember that it is possible to express any generic distribution as a linear combination of point distributions; for $\underline{\chi}'$ we have that the coefficients in the linear combination are the values x_i that X' has in each state σ_i :

$$\underline{\chi}' = \sum x_i \eta_i$$

If we define $\underline{\omega}_{\Lambda_i}$ in the following way:

$$\underline{\omega}_{\Lambda_i} \triangleq \underline{A}^T \underline{\eta}_i = \begin{pmatrix} a_{11} & \dots & a_{i1} & \dots & a_{n1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{1i} & \dots & a_{ii} & \dots & a_{ni} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{1n} & \dots & a_{in} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} a_{i1} \\ \vdots \\ a_{ii} \\ \vdots \\ a_{in} \end{pmatrix} = \underline{a}_{i*}$$

we can write that:

$$\underline{\chi} = \underline{A}^T \underline{\chi}' = \underline{A}^T (\sum x_i \eta_i) = \sum x_i \underline{A}^T \eta_i = \sum x_i \underline{\omega}_{\Lambda_i}$$

As the columns of \underline{A} are at most 1-summing, these vectors have the property of summing to no more than the unit vector $\underline{1}$:

$$\sum \underline{\omega}_{\Lambda_i} \leq \underline{1}$$

We can compute the expected value of X on δ (which is also the expected value of X' on δ') in the following way:

$$\mathcal{E}_\delta(X) = \underline{\chi}^T \underline{\delta} = \sum x_i \underline{\omega}_{\Lambda_i}^T \underline{\delta} = \sum x_i p'_i = \mathcal{E}_{\delta'}(X')$$

We see that we have expressed the probability p'_i in terms of $\underline{\delta}$, *i.e.* in terms of all the probabilities p_1, p_2, \dots, p_n , and $\underline{\omega}_{\Lambda_i}$ defines the weight of each probability in the sum: its j -th element can be seen as the probability that the program will end in state σ_i when starting in state σ_j .

B.2.4 Probability of an event

It is interesting to consider the case when we have a random variable Z' that assigns real values in the range $[0..1]$ to each state in \mathcal{S} : if this variable describes the probability that a certain event happens depending on each state (*i.e.* the conditional probability), then we can compute its expected value to evaluate the overall probability of that event:

$$\mathcal{P}(\text{event}) = \mathcal{E}_{\delta'}(Z') = \mathcal{E}_\delta(Z)$$

If we represent Z' with a weighting distribution $\underline{\omega}'$, we can write that:

$$\underline{\omega} = \underline{A}^T \underline{\omega}' = \sum z_i \underline{\omega}_{\Lambda_i}$$

$\underline{\omega}$ represents a random variable that describes the probability that the event happens depending on the initial states, and is a linear combination of the (transposed) rows of \underline{A} with coefficients in the range $[0..1]$.

We can then see a program as a *random-variable transformer*.

B.3 Deterministic Programs

We can take advantage of the matrix approach to rewrite the predicates corresponding to deterministic programs in a different way:

- the program *skip* can be represented by the identity matrix \underline{I} , as the after-distribution equals the before-distribution:

$$\text{skip} \triangleq \underline{\delta}' = \underline{I} \underline{\delta};$$

- an assignment takes the weight assigned to a state σ_i and reassigns it to a state σ'_j : this is represented by a matrix \underline{E} whose (i, j) -th element e_{ij} is 1 for all states σ_j being remapped to σ'_i , and 0 otherwise. In other words, the element in j -th position in the i -th row is the value of characteristic function of the inverse-image set $\text{Inv}(\underline{v} := \underline{e}, \sigma'_i)$ for the argument σ_j :

$$\underline{v} := \underline{e} \triangleq \underline{\delta}' = \underline{E} \underline{\delta};$$

- it is quite trivial to render sequential composition, as this can be easily done by means of the usual matrix composition:

$$A; B \triangleq \underline{\delta}' = \underline{B} \underline{A} \underline{\delta};$$

- for conditional constructs we use the diagonal matrices \underline{C} and $\bar{\underline{C}}$ to “select” respectively the states which satisfy the condition c and those that don’t: the (i, j) -th element c_{ij} in matrix \underline{C} is $c_{ii} = \sigma_i(c)$ (i.e. the boolean value 1 if c is satisfied by σ_i , and 0 otherwise) in case $i = j$, whereas $c_{ij} = 0$ if $i \neq j$; the matrix $\bar{\underline{C}}$ is defined as $\underline{I} - \underline{C}$. If we compose these matrices with those representing the programs A and B and sum the results, what we obtain is the representation of the conditional construct, i.e. $\underline{A} \underline{C} + \underline{B} \bar{\underline{C}}$: it is interesting to notice that the i -th column in this matrix is a_{*i} if $c_{ii} = 1$ and b_{*i} otherwise:

$$A \triangleleft c \triangleright B \triangleq \underline{\delta}' = \underline{A} \underline{C} \underline{\delta} + \underline{B} \bar{\underline{C}} \underline{\delta};$$

- the conditional choice construct scales the matrices representing the programs A and B by p and $(1 - p)$ respectively and sums the results:

$$A \text{ }_p\oplus\text{ } B \triangleq \underline{\delta}' = p \cdot \underline{A} \underline{\delta} + (1 - p) \cdot \underline{B} \underline{\delta};$$

- in a loop the body, represented by \underline{A} , is executed over and over again as long as a condition \underline{c} is satisfied (\underline{C} “selects” all states satisfying the condition, whereas $\bar{\underline{C}}$ accounts for those satisfying the complementary exit condition):

$$c * A \hat{=} \nu \underline{X} \bullet (\underline{XAC} + \underline{IC}).$$

B.3.1 Some considerations on loops

We can write that:

$$\begin{aligned} \nu \underline{X} \bullet (\underline{XAC} + \underline{IC}) &\equiv \underline{\delta}' = (\bar{\underline{C}} + \bar{\underline{C}}\underline{AC} + \bar{\underline{C}}(\underline{AC})^2 + \bar{\underline{C}}(\underline{AC})^3 + \dots) \underline{\delta} \\ &\equiv \underline{\delta}' = (\bar{\underline{C}} \sum_{i=0}^{\infty} (\underline{AC})^i) \underline{\delta} \end{aligned}$$

If the series converges than we have that $\sum_{i=0}^{\infty} (\underline{AC})^i = (\underline{I} - \underline{AC})^{-1}$ and thus:

$$\nu \underline{X} \bullet (\underline{XAC} + \underline{IC}) \equiv \underline{\delta}' = \bar{\underline{C}}(\underline{I} - \underline{AC})^{-1} \underline{\delta}$$

We have convergence if:

- \underline{AC} is *nilpotent* of some order N — and in that case the loop is guaranteed to terminate at most after N loops;
- $(\underline{AC})^i \rightarrow \underline{0}$ — and in that case termination is probabilistic, as the probability of non-termination tends to 0.

It is interesting to notice that if the element \bar{c}_{ii} of $\bar{\underline{C}}$ is 1, then the probability of exiting the loop by reaching the state σ_i depends on the i -th row of the matrix \underline{AC} . Conversely if the element \bar{c}_{jj} of $\bar{\underline{C}}$ is 0, then the probability of continuing the loop because the intermediate state σ_j depends on the j -th row of the matrix \underline{AC} .

This observation allows us to derive decision procedures to establish if a loop is (probabilistically) guaranteed to terminate⁶: the basic requirement is that all columns of \underline{A} are one-summing (if this is not fulfilled then there is some intrinsic non-termination probability in A).

We say that a column is *terminal of order 0* if it is null; a column is *terminal of order $i + 1$* if the only non-null elements have row index equal to the column index of a terminal column of order i ; provided that A is terminating, we have that:

- a loop is guaranteed to terminate if all columns of \underline{AC} are terminal of some order, *i.e.* the null elements are disposed according to an appropriate pattern;
- a loop is probabilistically guaranteed to terminate if at least one of the non-terminal columns has at least one non-null element with row index equal to the column index of a terminating column.

⁶This procedure is always applicable in the case of matrices of finite rank, *i.e.* when the state space is finite. In the case of matrices of non-finite rank it may be possible to apply the algorithm depending on the properties of the matrices — they must have a finite set of terminal columns of order greater than 0.

In order to present an agile decision criterion we introduce the concept of *reduction* M^R of a *non-null* matrix M , as the matrix obtained by individuating the null columns $m_{*,j}$ and removing them along with the rows $m_{j,*}$.

If we subsequently reduce M , we arrive to what we term the *everlooping matrix* M^E , which is not further reducible (*i.e.* the null matrix or a matrix with no null columns).

The properties of the everlooping matrix $(\underline{AC})^E$ allow us to conclude that:

- if $(\underline{AC})^E = \underline{0}$ then we have guaranteed termination;
- if $(\underline{AC})^E \rightarrow \underline{0}$ then we have probabilistically guaranteed termination;
- otherwise there is the possibility of being caught in an infinite loop.

More details and a proof for this can be found in [BPB11].

B.3.2 Healthiness conditions

If we look at the first three healthiness conditions from §3.5 from this different angle, we can restate them in a slightly different fashion:

- as δ' is a linear combination of a matrix whose columns are at most one-summing with the elements of δ as coefficients, we have that the norm of δ' cannot exceed that of δ (*feasibility*,Dist1/D);
- if we increase δ , the corresponding $\delta' = \underline{A}\delta$ is increasing as well: similarly as above, this is implied by the non-negativity of all matrix elements (*monotonicity*,Dist2/D);
- multiplication by a non-negative constant distributes through matrices (*scaling*,Dist3/D).

In the case of random variables we obtain something closer to the presentation of the healthiness conditions for *pGCL* expectations from [MM04]:

- as χ is a linear combination of a matrix whose rows are at most one-summing with the elements of χ' as coefficients, we have that the norm of χ cannot exceed that of χ' . As a consequence the weight of a distribution ξ from the positive cone \mathcal{D}^+ cannot exceed that of ξ' (*feasibility*,Dist1/RV);
- if we increase χ' , the corresponding $\chi = \underline{A}^\top \chi'$ increases as well: this is implied by the non-negativity of all matrix elements (*monotonicity*,Dist2/RV);
- multiplication by a non-negative constant distributes through matrices (*scaling*,Dist3/RV).

Some examples in a two-element space

Before proceeding any further, we think it is useful to present the reader with a few examples, to help visualize the concepts presented so far.

Let us consider a state space with only two elements:

$$\mathcal{S} = \{\sigma_1, \sigma_2\}$$

The possible probability distributions on this state space can be graphically presented as in

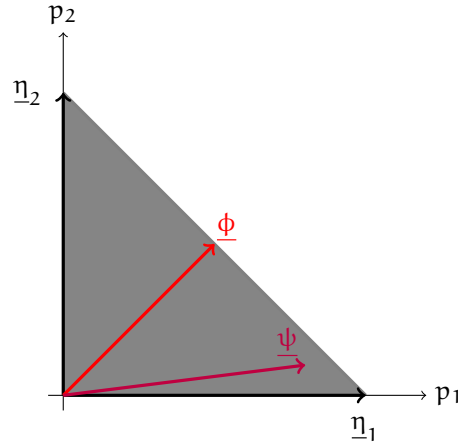


Figure B.1: Representation of probability distributions on a 2-element space.

figure B.1:

We have drawn the vectors representing the point distributions η_1 and η_2 , the 1-summing probability distribution ϕ and the probability distribution ψ , which sums up to 0.9 instead:

$$\underline{\eta}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \underline{\eta}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \underline{\phi} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} = 0.5\underline{\eta}_1 + 0.5\underline{\eta}_2 \quad \underline{\psi} = \begin{pmatrix} 0.8 \\ 0.1 \end{pmatrix} = 0.8\underline{\eta}_1 + 0.1\underline{\eta}_2$$

We use the following naming conventions to refer to points in the plane:

- the point $(0, 0)$ is O ;
- the point ψ_i is that connected to O by $\underline{\psi}_i$;
- the point X_i is that connected to O by $\underline{\psi}_{X_i}$.

1-summing probability distributions as ϕ are represented by points on the thicker line limiting the shaded area, all other probability distributions as ψ are represented by points in the shaded area.

A program is then represented by a 2×2 matrix; let A be the always-terminating program characterised by the following matrix:

$$\underline{A} = \begin{pmatrix} 0.25 & 0.625 \\ 0.75 & 0.375 \end{pmatrix} = \begin{pmatrix} \underline{\delta}'_{A1} & \underline{\delta}'_{A2} \end{pmatrix}$$

The probability distributions that can possibly result from running this program can be represented as the darker area in figure B.2a.

We can see that the vector space generated by $\underline{\eta}_1$ and $\underline{\eta}_2$ has been transformed through a homeomorphism to the vector space generated by $\underline{\delta}'_{A1}$ and $\underline{\delta}'_{A2}$, and all vectors have undergone the same transformation:

$$\underline{\phi}' = \underline{A} \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} = 0.5\underline{\delta}'_{A1} + 0.5\underline{\delta}'_{A2} = \begin{pmatrix} 0.4375 \\ 0.5625 \end{pmatrix} \quad \underline{\psi}' = \underline{A} \begin{pmatrix} 0.8 \\ 0.1 \end{pmatrix} = 0.8\underline{\delta}'_{A1} + 0.1\underline{\delta}'_{A2} = \begin{pmatrix} 0.2625 \\ 0.6375 \end{pmatrix}$$

We can verify that $\|\underline{\phi}\| = \|\underline{\phi}'\| = 1$ and $\|\underline{\psi}\| = \|\underline{\psi}'\| = 0.9$: as A is always terminating the after-distribution has always the same weight as the before-distribution.

Let us now consider the program B , which is almost like A with the difference that it has probability 0.1 of non-terminating when starting from state σ_1 :

$$\underline{B} = \underline{A} - \begin{pmatrix} 0.1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0.15 & 0.625 \\ 0.75 & 0.375 \end{pmatrix} = (\underline{\delta}'_{B1} \quad \underline{\delta}'_{A2})$$

We see that the second column of \underline{B} is the same as in \underline{A} .

The situation for program B is represented in figure B.2b.

The thicker line denotes maximal elements, which are mostly probability subdistributions (the only probability distribution is represented by the right end point, which we obtain in case we run the program from the state σ_2 , that guarantees termination).

ϕ' and ψ' have changed in the following way:

$$\underline{\phi}' = \underline{B} \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} = 0.5 \underline{\delta}'_{B1} + 0.5 \underline{\delta}'_{A2} = \begin{pmatrix} 0.3875 \\ 0.5625 \end{pmatrix} \quad \underline{\psi}' = \underline{B} \begin{pmatrix} 0.8 \\ 0.1 \end{pmatrix} = 0.8 \underline{\delta}'_{B1} + 0.1 \underline{\delta}'_{A2} = \begin{pmatrix} 0.1825 \\ 0.6375 \end{pmatrix}$$

Their weights have decreased, as $\|\underline{\phi}'\| = 0.95$ and $\|\underline{\psi}'\| = 0.82$. If we compare the two programs we can analyse where this difference comes from:

$$\underline{B}\underline{\phi} - \underline{A}\underline{\phi} = \begin{pmatrix} 0.1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} = \begin{pmatrix} 0.05 \\ 0 \end{pmatrix} \quad \underline{B}\underline{\psi} - \underline{A}\underline{\psi} = \begin{pmatrix} 0.1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0.8 \\ 0.1 \end{pmatrix} = \begin{pmatrix} 0.08 \\ 0 \end{pmatrix}$$

Now let us introduce a random variable Z' , describing for example the probability that the result given by a program is correct, as follows:

$$Z' = \{(\sigma_1, 0.8), (\sigma_2, 0.9)\}$$

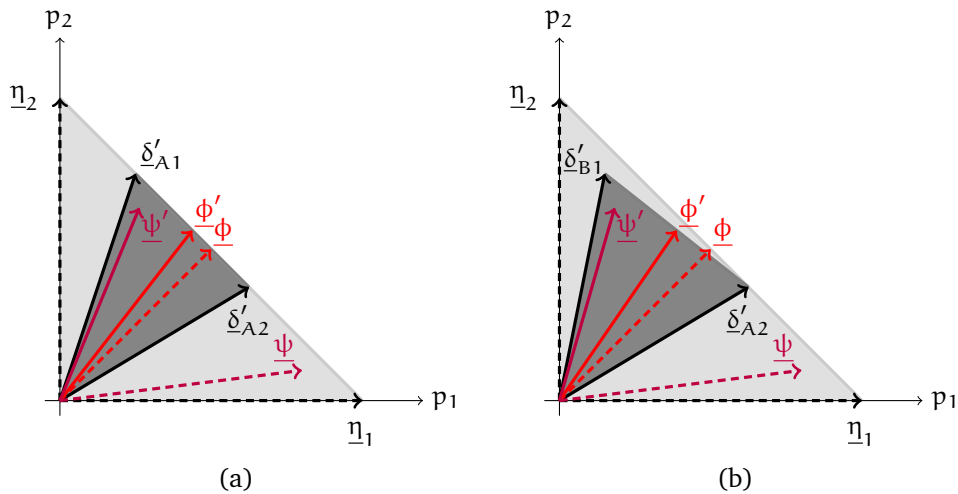


Figure B.2: Representation of probability after-distributions: (a) after the application of program A ; (b) after the application of program B .

This can be represented by the weighting distribution $\underline{\omega}'$:

$$\underline{\omega}' = \begin{pmatrix} 0.8 \\ 0.9 \end{pmatrix}$$

The space of all random variables like Z' , i.e. with values in $[0..1]$, can be represented as in figure B.3:

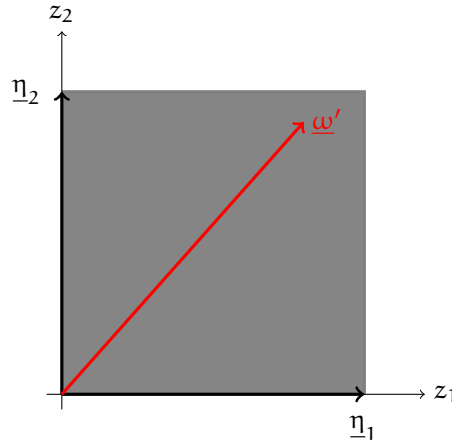


Figure B.3: Representation of random variables with values in $[0..1]$ on a 2-element space.

If we take program A, we can relate $\underline{\omega}'$ to $\underline{\omega}$ through the homeomorphism described by the matrix \underline{A}^\top :

$$\underline{\omega} = \underline{A}^\top \begin{pmatrix} 0.8 \\ 0.9 \end{pmatrix} = 0.8 \underline{\omega}_{A1} + 0.9 \underline{\omega}_{A2} = \begin{pmatrix} 0.875 \\ 0.8375 \end{pmatrix}$$

This is represented in figure B.4a.

We can compute the expected value of Z' after program A has run, in the cases when the before-distribution is respectively ϕ and ψ .

$$\begin{aligned} \mathcal{E}_\phi(Z') &= \underline{\omega}^\top \underline{\phi} = \begin{pmatrix} 0.875 & 0.8375 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} = 0.85625 \\ \mathcal{E}_\psi(Z') &= \underline{\omega}^\top \underline{\psi} = \begin{pmatrix} 0.875 & 0.8375 \end{pmatrix} \begin{pmatrix} 0.8 \\ 0.1 \end{pmatrix} = 0.78375 \end{aligned}$$

We can easily verify that these are the same results we would have obtained if we had calculated the after-distributions and then computed the expected value:

$$\begin{aligned} \mathcal{E}_\phi(Z') &= (\underline{\omega}')^\top \underline{\phi}' = \begin{pmatrix} 0.8 & 0.9 \end{pmatrix} \begin{pmatrix} 0.4375 \\ 0.5625 \end{pmatrix} = 0.85625 \\ \mathcal{E}_\psi(Z') &= (\underline{\omega}')^\top \underline{\psi}' = \begin{pmatrix} 0.8 & 0.9 \end{pmatrix} \begin{pmatrix} 0.2625 \\ 0.6375 \end{pmatrix} = 0.78375 \end{aligned}$$

In the case of program B we have that:

$$\underline{\omega} = \underline{B}^\top \begin{pmatrix} 0.8 \\ 0.9 \end{pmatrix} = 0.8 \underline{\omega}_{B1} + 0.9 \underline{\omega}_{A2} = \begin{pmatrix} 0.6825 \\ 0.8375 \end{pmatrix}$$

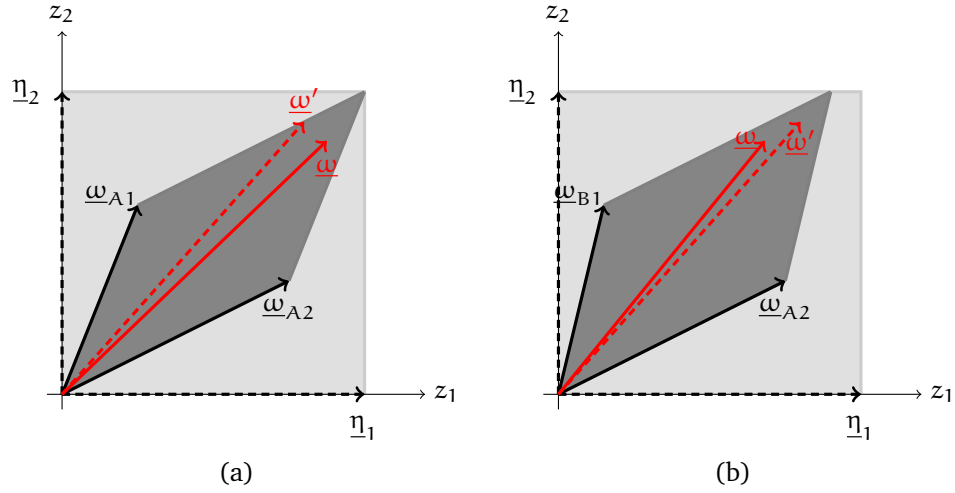


Figure B.4: Representation of random variables with values in $[0..1]$: (a) after the application of program A; (b) after the application of program B.

This is represented in figure B.4b.

We can compute the expected value of Z' after program B has run:

$$\begin{aligned} \mathcal{E}_\phi(Z') &= \underline{\omega}^T \underline{\phi} = \begin{pmatrix} 0.6825 & 0.8375 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} = 0.76 \\ \mathcal{E}_\psi(Z') &= \underline{\omega}^T \underline{\psi} = \begin{pmatrix} 0.6825 & 0.8375 \end{pmatrix} \begin{pmatrix} 0.8 \\ 0.1 \end{pmatrix} = 0.62975 \end{aligned}$$

B.4 Nondeterministic choice

Here we discuss briefly nondeterminism in the case of nondeterministic choice (the case of generic choice is a sub-case of this).

When we use the matrix notation, the re-weighting operation can conveniently be represented by the matrices $\text{diag}(\underline{\pi})$ and $\text{diag}(\bar{\underline{\pi}})$.

$$A \sqcap B \triangleq \exists \pi \bullet \underline{\delta}' = \underline{A} \text{diag}(\underline{\pi}) \underline{\delta} + \underline{B} \text{diag}(\bar{\underline{\pi}}) \underline{\delta}.$$

Let us go on with the examples to see how this works; we pick a nondeterministic program $C \sqcap D$, where:

$$\underline{C} = \begin{pmatrix} 0.7 & 0.15 \\ 0.3 & 0.85 \end{pmatrix} = \begin{pmatrix} \underline{\delta}'_{C1} & \underline{\delta}'_{C2} \end{pmatrix} \quad \underline{D} = \begin{pmatrix} 0.6 & 0.1 \\ 0.4 & 0.9 \end{pmatrix} = \begin{pmatrix} \underline{\delta}'_{D1} & \underline{\delta}'_{D2} \end{pmatrix}$$

Let us focus on one of the possible after-distributions, parametric in π :

$$\underline{\delta}'_\pi = \underline{C} (\underline{\pi} \circ \underline{\delta}) + \underline{D} (\bar{\underline{\pi}} \circ \underline{\delta}) = (\underline{C} \text{diag}(\underline{\pi})) \underline{\delta} + (\underline{D} \text{diag}(\bar{\underline{\pi}})) \underline{\delta}$$

If $\pi = (w_1 \ w_2)^T$, we can write that:

$$\underline{C} \text{diag}(\underline{\pi}) + \underline{D} \text{diag}(\bar{\underline{\pi}}) = \begin{pmatrix} w_1 \cdot 0.7 + \bar{w}_1 \cdot 0.6 & w_2 \cdot 0.15 + \bar{w}_2 \cdot 0.1 \\ w_1 \cdot 0.3 + \bar{w}_1 \cdot 0.4 & w_2 \cdot 0.85 + \bar{w}_2 \cdot 0.9 \end{pmatrix}$$

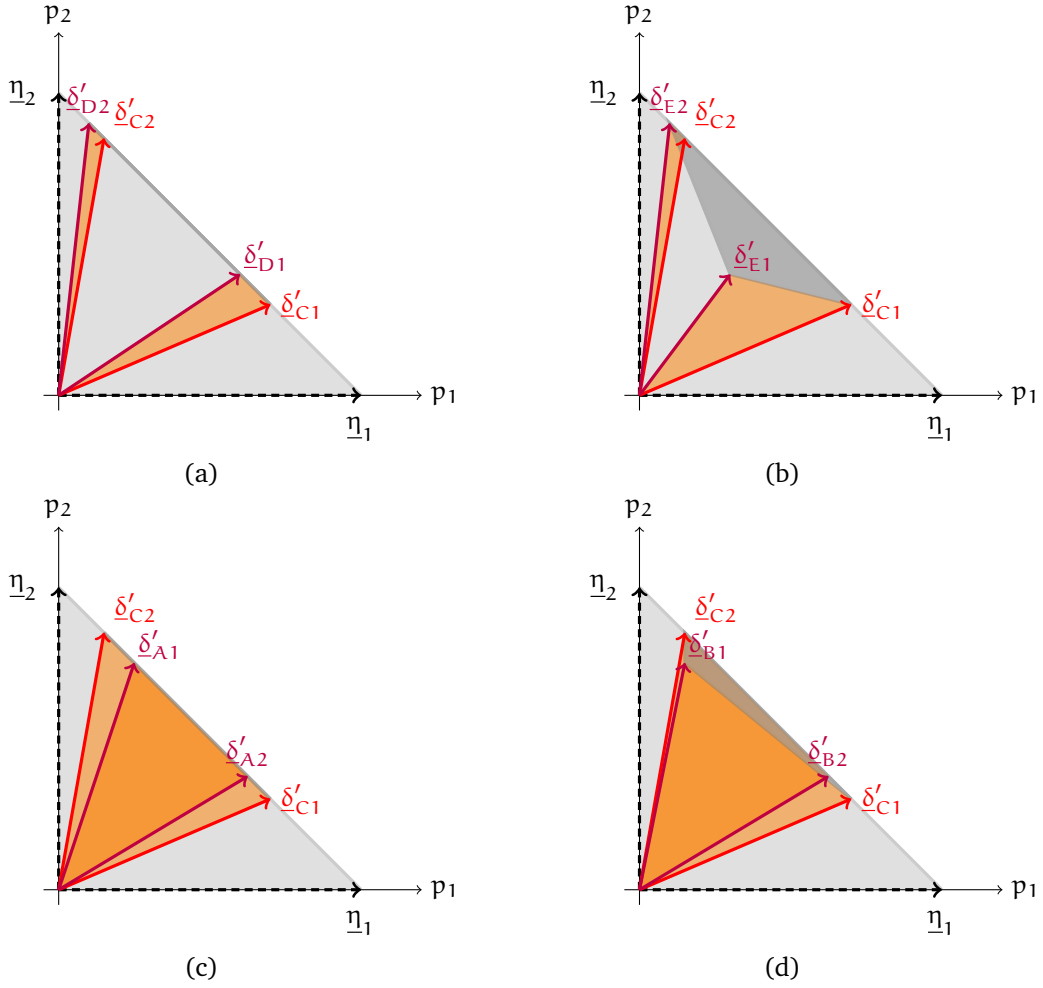


Figure B.5: Representation of probability after-distributions: (a) after the application of program $C \cap D$; (b) after the application of program $C \cap E$; (c) after the application of program $C \cap A$; (d) after the application of program $C \cap B$.

In this way we can clearly see that it is a linear combination of the columns of the two matrices:

$$\underline{C} \text{diag}(\underline{\pi}) \delta + \underline{D} \text{diag}(\bar{\underline{\pi}}) = \left(w_1 \cdot \underline{\delta}'_{C1} + \bar{w}_1 \cdot \underline{\delta}'_{D1} \quad w_2 \cdot \underline{\delta}'_{C2} + \bar{w}_2 \cdot \underline{\delta}'_{D2} \right)$$

The situation is represented in figure B.5a: the i -th generator of the program is an element of $D_i C_i$ and is determined by the i -th component of $\underline{\pi}$.

Full probability before-distributions are mapped to after-distributions lying on the segment connecting the two generators, which is a part of the segment containing the maximal elements of \mathcal{D}_p (that connecting the canonical generators): this is because both programs are guaranteed to terminate, and therefore if we start with a full probability before-distribution we get to a full probability after-distribution, as distribution weight is preserved in case of terminating programs.

The space of all possible outcomes varies depending on $\underline{\pi}$, but for sure we have that:

- it can be no wider than the area $D_2 O C_1$;
- regardless of $\underline{\pi}$ it has to contain the area $C_2 O D_1$, *i.e.* contiguous parts from the shaded

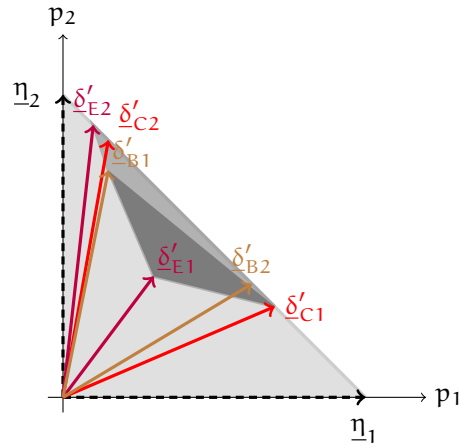


Figure B.6: Representation of probability after-distributions after the application of program $(C \sqcap E) \sqcap A$.

areas D_2OC_2 and D_1OC_1 may or may not belong to this space;

- it is limited by a segment containing C_2D_1 and contained by D_2C_1 .

Let us pick a case where the nondeterministic choice includes also a program which is not always guaranteed to terminate, *i.e.* $C \sqcap E$ where:

$$\underline{E} = \underline{D} - \begin{pmatrix} 0.3 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0.3 & 0.1 \\ 0.4 & 0.9 \end{pmatrix} = \begin{pmatrix} \delta'_{E1} & \delta'_{E2} \end{pmatrix}$$

From figure B.5b we can see that in this case one of the generators lies on a segment which is not part of the segment containing the maximal elements of \mathcal{D}_p : this is because of non-termination and its impact varies depending on $\underline{\pi}$.

Full probability before-distributions are therefore mapped to after-distributions belonging to the area $E_2E_1C_1$ and we can clearly see that even starting with a full probability before-distribution cannot guarantee that we obtain a full probability after-distribution (in this particular case, this happens only if the before-distribution is $\underline{\eta}_2$ as $\underline{\delta}_{A2}$ and $\underline{\delta}_{C2}$ are 1-summing and thus account for certain termination when starting in state σ_2).

The space of all possible outcomes varies depending on $\underline{\pi}$, but for sure we have that:

- it can be no wider than the area E_2OC_1 ;
- regardless of $\underline{\pi}$ it has to contain the area XOE_1 , where X is the intersection of the E_1E_2 and OC_2 ;
- it is limited by a segment whose vertices are respectively on E_2C_2 and E_1C_1 .

If the nondeterministic choice were $C \sqcap A$ we would have had:

$$\underline{C} \text{diag}(\underline{\pi}) + \underline{A} \text{diag}(\underline{\bar{\pi}}) = \begin{pmatrix} w_1 \cdot 0.7 + \bar{w}_1 \cdot 0.25 & w_2 \cdot 0.15 + \bar{w}_2 \cdot 0.625 \\ w_1 \cdot 0.3 + \bar{w}_1 \cdot 0.75 & w_2 \cdot 0.85 + \bar{w}_2 \cdot 0.375 \end{pmatrix}$$

This is the situation of figure B.5c, where we can immediately see that the segments A_iC_i containing the i -th generator overlap: as a result the area which belongs to the space of

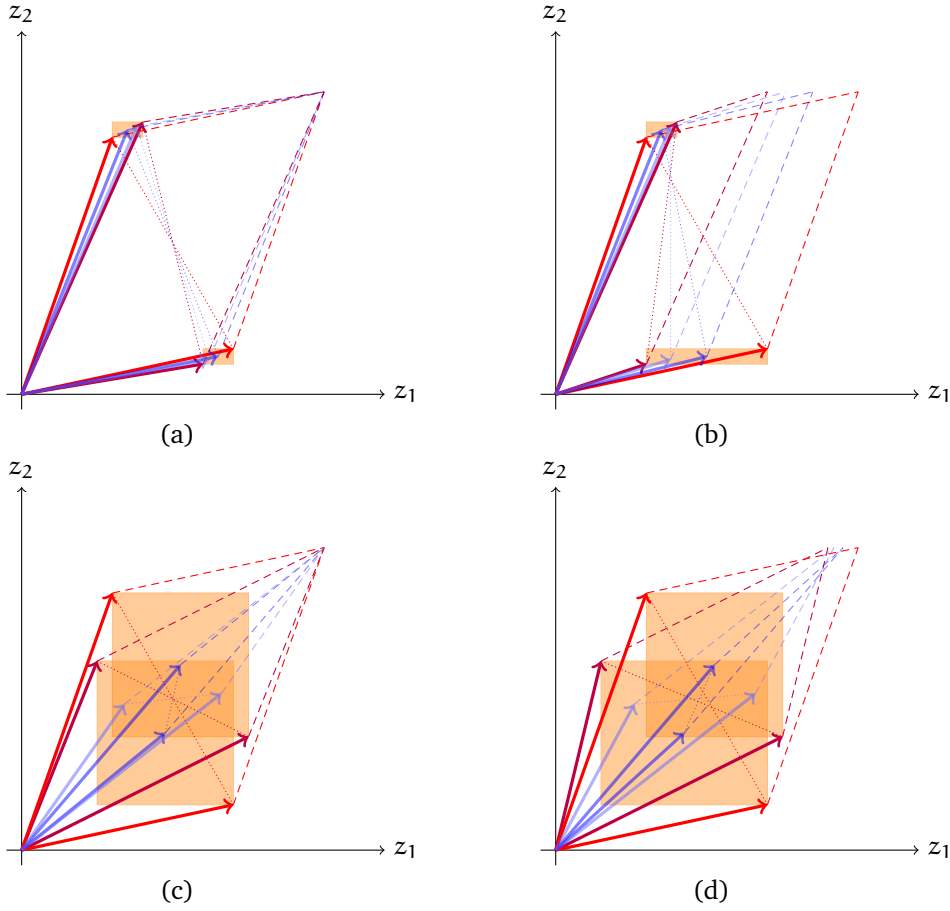


Figure B.7: Representation of random variables with values in $[0..1]$: (a) after the application of program $C \sqcap D$; (b) after the application of program $C \sqcap E$; (c) after the application of program $C \sqcap A$; (d) after the application of program $C \sqcap B$.

possible after-distributions collapses to a segment individuated by the condition $w_1 \cdot \delta'_{A1} + w_2 \cdot \delta'_{C1} = w_1 \cdot \delta'_{A2} + w_2 \cdot \delta'_{C2}$.

Likewise the segment of possible full distributions (both A and C are terminating programs) is a part of $C_1 C_2$, which has to contain the end point of the above segment.

Similarly if the nondeterministic choice were $C \sqcap B$ we would have had:

$$\underline{C} \text{diag}(\underline{\pi}) + \underline{B} \text{diag}(\underline{\bar{\pi}}) = \begin{pmatrix} w_1 \cdot 0.7 + \bar{w}_1 \cdot 0.15 & w_2 \cdot 0.15 + \bar{w}_2 \cdot 0.625 \\ w_1 \cdot 0.3 + \bar{w}_1 \cdot 0.75 & w_2 \cdot 0.85 + \bar{w}_2 \cdot 0.375 \end{pmatrix}$$

In this case we can see that figure B.5d has elements of similarity both with $C \sqcap A$ and $C \sqcap E$, so similar considerations apply.

Generally speaking, in the case of nondeterministic choice between two deterministic programs, we can say that the maximal elements in the space of possible after-distributions lie on a segment whose end points belong to the segment connecting homologue program generators.

If we have nondeterminism on either side of the nondeterministic choice, homologue program generators define an area that contains all of them: the maximal elements in the space of possible after-distributions lie on a segment whose end points belong to these areas — we can see this in figure B.6, representing the situation for $(C \sqcap E) \sqcap A$.

If we look at the transposed matrix in the case of $C \sqcap D$, we have that:

$$\underline{\omega}_\pi = (\underline{C} \text{diag}(\underline{\pi}) + \underline{D} \text{diag}(\underline{\bar{\pi}}))^T \underline{\omega}' = \left(\text{diag}(\underline{\pi}) \underline{\omega}_{C1} + \text{diag}(\underline{\bar{\pi}}) \underline{\omega}_{D1} \quad \text{diag}(\underline{\pi}) \underline{\omega}_{C2} + \text{diag}(\underline{\bar{\pi}}) \underline{\omega}_{D2} \right) \underline{\omega}'$$

$$(\underline{C} \text{diag}(\underline{\pi}) + \underline{D} \text{diag}(\underline{\bar{\pi}}))^T = \begin{pmatrix} w_1 \cdot 0.7 + \bar{w}_1 \cdot 0.6 & w_1 \cdot 0.3 + \bar{w}_1 \cdot 0.4 \\ w_2 \cdot 0.15 + \bar{w}_2 \cdot 0.1 & w_2 \cdot 0.85 + \bar{w}_2 \cdot 0.9 \end{pmatrix}$$

We can notice that the j -th component of the i -th generator of the program $C \sqcap D$ is the weighted average of the j -th components of the i -th generators of C and D , where the weights w_i and \bar{w}_i are the same for all i .

In figure B.7a we can see that the representation of the two programs are deformed in complementary ways (*i.e.*, if the z_i component is scaled by w_i for C , the z_i components is being scaled \bar{w}_i for D) and then composed together to form the representation of $C \sqcap D$.

The shaded rectangles represent the areas where the generators lie and the dotted lines connect corresponding generators (picking one generator determines the other, as the scaling factors are the same for all generators of each program).

Figures B.7b, B.7c and B.7d show the representations in the cases of all other programs we have taken as examples in this section — to be noted the effect of non-termination in figures B.7b and B.7d.

Additional figures on nondeterminism

In the next pages there are additional figures which (may) give a clearer view of some parts of the presentation relating to nondeterminism.

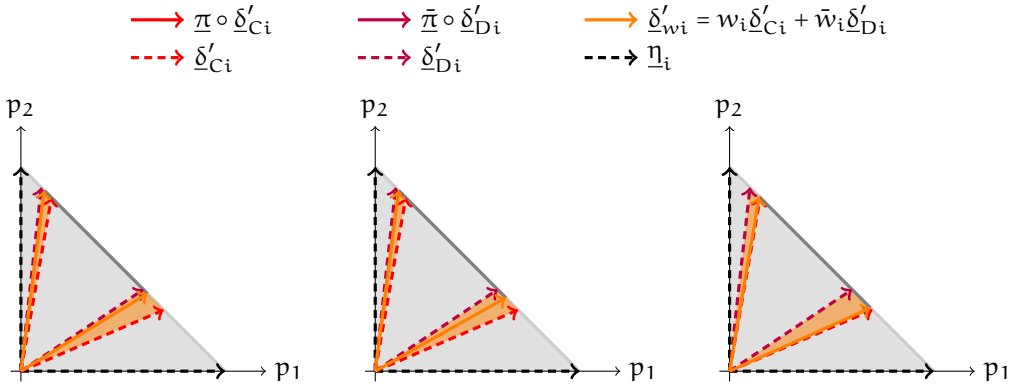


Figure B.8: Representation of probability after-distributions after the application of program $C \cap D$, in case that: (a) $\underline{\pi} = (0.2, 0.3)$; (b) $\underline{\pi} = (0.4, 0.3)$; (c) $\underline{\pi} = (0.9, 0.9)$.

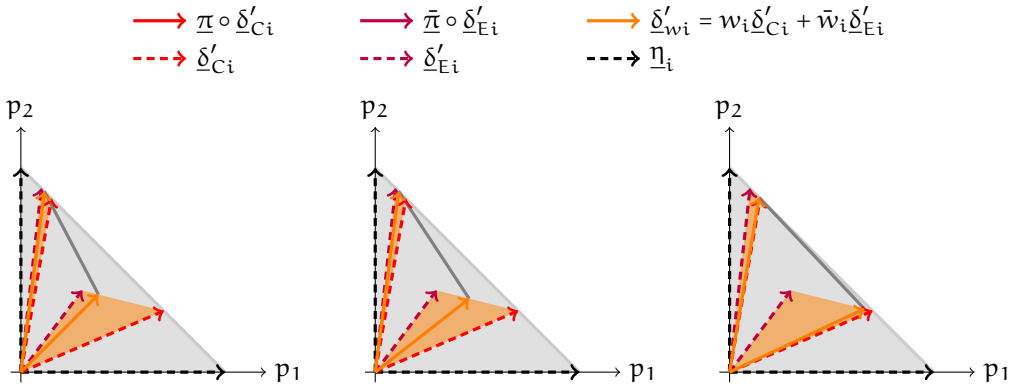


Figure B.9: Representation of probability after-distributions after the application of program $C \cap E$, in case that: (a) $\underline{\pi} = (0.2, 0.3)$; (b) $\underline{\pi} = (0.4, 0.3)$; (c) $\underline{\pi} = (0.9, 0.9)$.

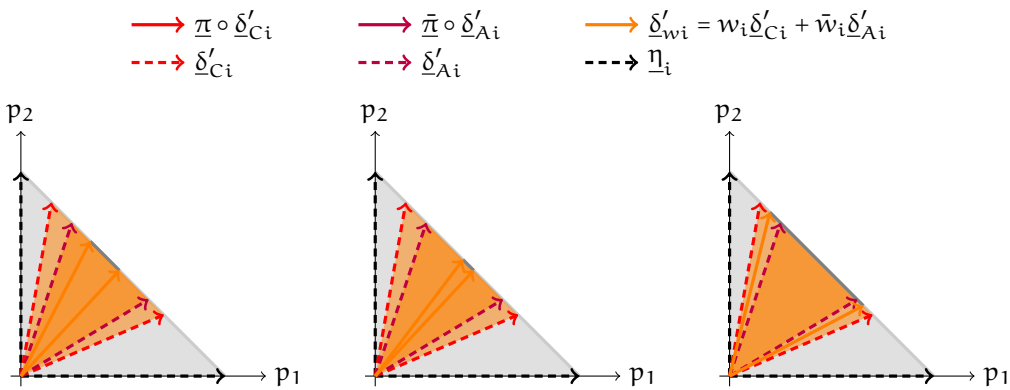


Figure B.10: Representation of probability after-distributions after the application of program $C \cap A$, in case that: (a) $\underline{\pi} = (0.2, 0.3)$; (b) $\underline{\pi} = (0.4, 0.3)$; (c) $\underline{\pi} = (0.9, 0.9)$.

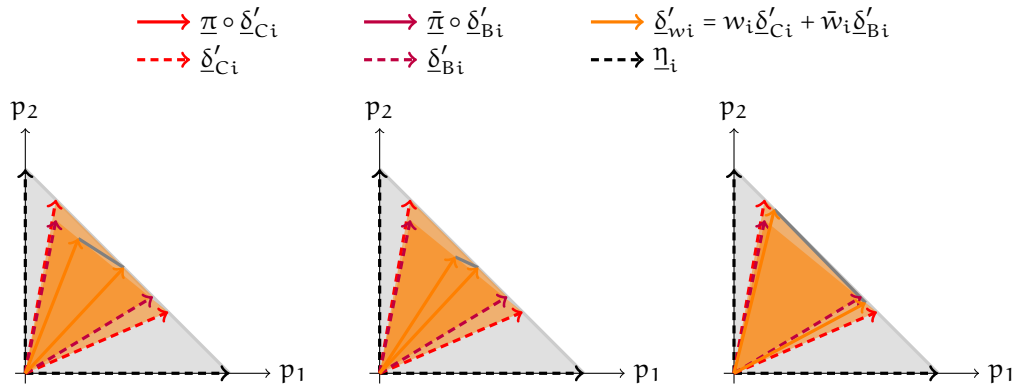


Figure B.11: Representation of probability after-distributions after the application of program $C \sqcap B$, in case that: (a) $\underline{\pi} = (0.2, 0.3)$; (b) $\underline{\pi} = (0.4, 0.3)$; (c) $\underline{\pi} = (0.9, 0.9)$.

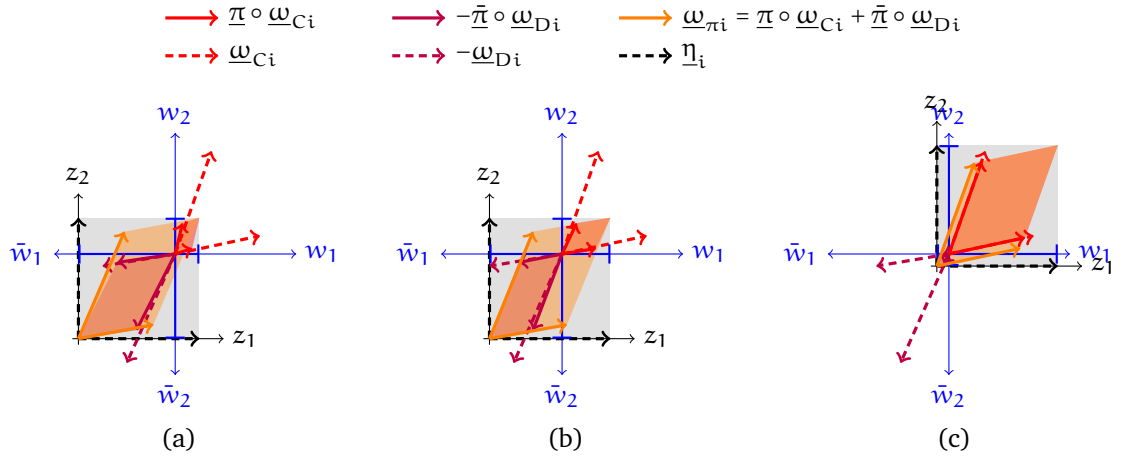


Figure B.12: Representation of random variables with values in $[0..1]$ after the application of program $C \sqcap D$, in case that: (a) $\underline{\pi} = (0.2, 0.3)$; (b) $\underline{\pi} = (0.4, 0.3)$; (c) $\underline{\pi} = (0.9, 0.9)$.

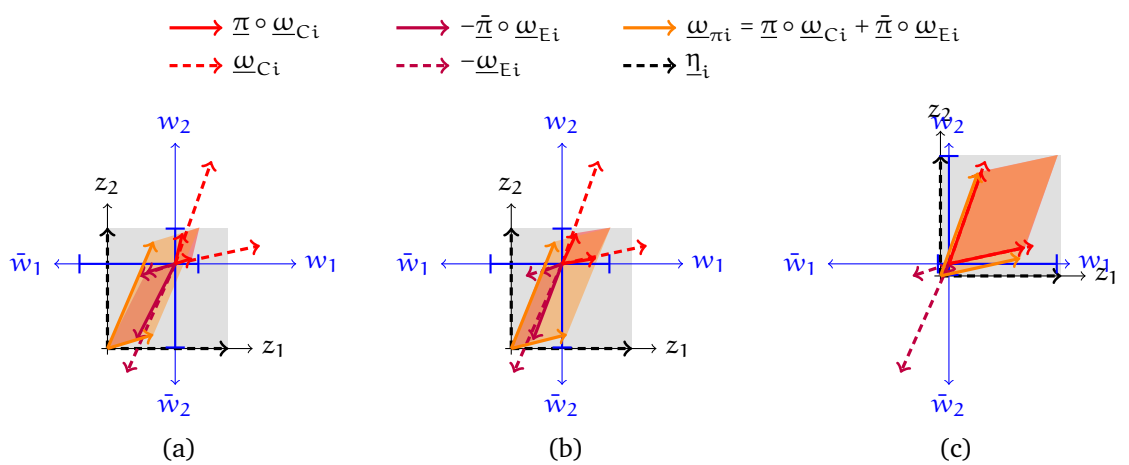


Figure B.13: Representation of random variables with values in $[0..1]$ after the application of program $C \sqcap E$, in case that: (a) $\underline{\pi} = (0.2, 0.3)$; (b) $\underline{\pi} = (0.4, 0.3)$; (c) $\underline{\pi} = (0.9, 0.9)$.

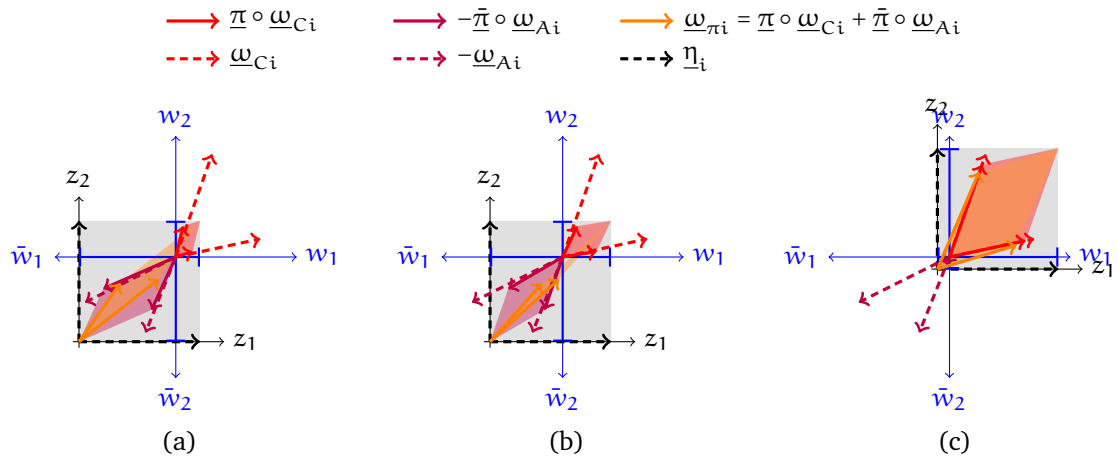


Figure B.14: Representation of random variables with values in $[0..1]$ after the application of program $C \cap A$, in case that: (a) $\pi = (0.2, 0.3)$; (b) $\pi = (0.4, 0.3)$; (c) $\pi = (0.9, 0.9)$.

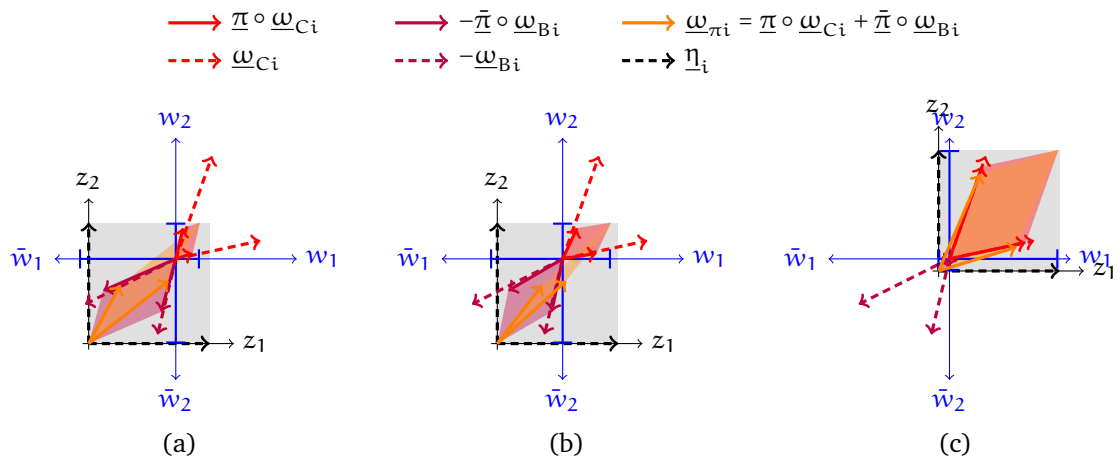


Figure B.15: Representation of random variables with values in $[0..1]$ after the application of program $C \cap B$, in case that: (a) $\pi = (0.2, 0.3)$; (b) $\pi = (0.4, 0.3)$; (c) $\pi = (0.9, 0.9)$.

APPENDIX C

Proofs

C.1 Restriction of the state space

$$\boxed{\alpha\langle c \rangle = \mathcal{S}\langle c \rangle \cap \alpha}$$

Proof:

$$\begin{aligned} & \alpha\langle c \rangle \\ = & [d:A:Rst] \text{ --- } \S A.2.2 \\ & \{\sigma \mid \sigma \in \alpha \wedge \sigma\langle c \rangle = \text{true}\} \\ = & [d:S] \text{ --- } \S A.2 \\ & \{\sigma \mid \sigma \in \mathcal{S} \wedge \sigma \in \alpha \wedge \sigma\langle c \rangle = \text{true}\} \\ = & \text{Set theory} \\ & \{\sigma \mid \sigma \in \mathcal{S} \wedge \sigma\langle c \rangle = \text{true}\} \cap \{\sigma \mid \sigma \in \alpha\} \\ = & [d:A:Rst] \\ & \mathcal{S}\langle c \rangle \cap \alpha \end{aligned}$$

□

C.2 Restriction through equivalent condition

$$\boxed{(c_1 \Leftrightarrow c_2) \Rightarrow \chi\langle c_1 \rangle = \chi\langle c_2 \rangle}$$

Proof:

$$\text{dom}(\chi)\langle c_1 \rangle = \text{dom}(\chi)\langle c_2 \rangle$$

□

C.3 Restriction through implied condition (I)

$$\boxed{(c_2 \Rightarrow c_1) \Leftrightarrow \chi\langle c_1 \rangle\langle c_2 \rangle = \chi\langle c_2 \rangle}$$

Proof:

$$\begin{aligned} & \chi\langle c_1 \rangle\langle c_2 \rangle \\ = & [p:D:Rst:Cnj] \text{ --- } \S A.3 \\ & \chi\langle c_1 \wedge c_2 \rangle \\ = & [p:D:Rst:EqC] \text{ --- } \S C.2: (c_2 \Rightarrow c_1) \wedge (c_1 \wedge c_2) \Leftrightarrow c_2 \\ & \chi\langle c_2 \rangle \end{aligned}$$

□

C.4 Restriction through implied condition (II)

$$\boxed{(c_1 \Rightarrow \neg c_2) \Rightarrow \chi\langle c_1 \rangle\langle c_2 \rangle = \epsilon}$$

Proof:

$$\begin{aligned} & \chi\langle c_1 \rangle\langle c_2 \rangle \\ = & \quad [p:D:Rst:Cnj] \text{ — §A.3} \\ & \chi\langle c_1 \wedge c_2 \rangle \\ = & \quad [p:D:Rst:EqC] \text{ — §C.2: } (c_1 \Rightarrow \neg c_2) \wedge (c_1 \wedge c_2) \Leftrightarrow \textit{false} \\ & \chi\langle \textit{false} \rangle = \epsilon \end{aligned}$$

□

C.5 Restriction through a restricted unitary distribution

$$\boxed{\chi\langle c \rangle = \chi\langle \iota_\chi\langle c \rangle \rangle}$$

Proof:

$$\begin{aligned} & \chi\langle \iota_\chi\langle c \rangle \rangle \\ = & \quad [d:D:RstD] \text{ — §A.3} \\ & \left\{ \sigma \mapsto \chi(\sigma) \cdot \iota_\chi(\sigma) \mid \sigma \in \text{dom}(\chi) \cap \text{dom}(\iota_\chi\langle c \rangle) \right\} \\ = & \quad \text{Set theory: } \text{dom}(\iota_\chi\langle c \rangle) = \text{dom}(\chi\langle c \rangle) \subseteq \text{dom}(\chi) \\ & \left\{ \sigma \mapsto \chi(\sigma) \cdot \iota_\chi(\sigma) \mid \sigma \in \text{dom}(\chi\langle c \rangle) \right\} \\ = & \quad [d:D:UD] \text{ — §A.3} \\ & \left\{ \sigma \mapsto \chi(\sigma) \mid \sigma \in \text{dom}(\chi\langle c \rangle) \right\} \\ = & \quad [d:D:Rst] \text{ — §A.3} \\ & \chi\langle c \rangle \end{aligned}$$

□

C.6 Case Split

$$\chi = \chi\langle c \rangle + \chi\langle -c \rangle$$

Proof:

$$\begin{aligned}
& \chi\langle c \rangle + \chi\langle -c \rangle \\
= & \quad [d:D:Sum] \text{ — §A.3.1} \\
& \{\sigma \mapsto (\chi\langle c \rangle(\sigma) + \chi\langle -c \rangle(\sigma)) \mid \sigma \in \text{dom}(\chi\langle c \rangle) \cup \text{dom}(\chi\langle -c \rangle)\} \\
= & \quad \text{Set theory} \\
& \{\sigma \mapsto (\chi\langle c \rangle(\sigma) + 0) \mid \sigma \in \text{dom}(\chi\langle c \rangle)\} \cup \{\sigma \mapsto (0 + \chi\langle -c \rangle(\sigma)) \mid \sigma \in \text{dom}(\chi\langle -c \rangle)\} \\
= & \quad [d:D:Rst] \text{ — §A.3} \\
& \{\sigma \mapsto \chi(\sigma) \mid \sigma \in \text{dom}(\chi\langle c \rangle)\} \cup \{\sigma \mapsto \chi(\sigma) \mid \sigma \in \text{dom}(\chi\langle -c \rangle)\} \\
= & \quad \text{Set theory} \\
& \{\sigma \mapsto \chi(\sigma) \mid \sigma \in \text{dom}(\chi\langle c \rangle) \cup \text{dom}(\chi\langle -c \rangle)\} \\
= & \quad \text{Set theory} \\
& \{\sigma \mapsto \chi(\sigma) \mid \sigma \in \text{dom}(\chi)\} \\
= & \quad [d:D] \text{ — §A.3} \\
& \chi
\end{aligned}$$

□

C.7 Restriction

$$\pi_1 \langle \pi_2 \rangle \in \mathcal{D}_w$$

Proof:

$$\pi_1 \langle \pi_2 \rangle(\sigma) = \pi_1(\sigma) \cdot \pi_2(\sigma) \leq \pi_1(\sigma)$$

□

C.8 Restriction

$$\delta \langle \pi \rangle \in \mathcal{D}_p$$

Proof:

$$\delta \langle \pi \rangle(\sigma) = \delta(\sigma) \cdot \pi(\sigma) \leq \delta(\sigma)$$

□

C.9 Nested inverse-image set

$$\boxed{Innv(\underline{v} := \underline{e}, Innv(\underline{v} := \underline{f}, \{\sigma\})) = Innv(\underline{v} := \underline{f}\{\underline{e}/\underline{v}\}, \{\sigma\})}$$

Proof:

$$\begin{aligned}
& Innv(\underline{v} := \underline{e}, Innv(\underline{v} := \underline{f}, \{\sigma'\})) \\
= & [d:S:Inv] \text{ — §A.2.3} \\
& Innv(\underline{v} := \underline{e}, \{\sigma \mid \sigma'(\underline{v}) = \sigma(\underline{f}) \wedge \sigma \in \mathcal{S}_{\text{alph}(\sigma')}\}) \\
= & [d:S:Inv] \\
& \bigcup_{\zeta' \in \{\sigma \mid \sigma'(\underline{v}) = \sigma(\underline{f}) \wedge \sigma \in \mathcal{S}_{\text{alph}(\sigma')}\}} \{\zeta \mid \zeta'(\underline{v}) = \zeta(\underline{e}) \wedge \zeta \in \mathcal{S}_{\text{alph}(\zeta')}\} \\
= & \text{Property of distributed union} \\
& \{\zeta \mid \sigma(\underline{v}) = \zeta(\underline{e}) \wedge \sigma'(\underline{v}) = \sigma(\underline{f}) \wedge \sigma, \zeta \in \mathcal{S}_{\text{alph}(\sigma')}\} \\
= & [p:E:Ev:Comp] \text{ — §A.2.1} \\
& \{\zeta \mid \sigma'(\underline{v}) = \zeta(\underline{f} \circ \underline{e}) \wedge \zeta \in \mathcal{S}_{\text{alph}(\sigma')}\} \\
= & [d:E:Comp] \text{ — §A.2.1} \\
& \{\zeta \mid \sigma'(\underline{v}) = \zeta(\underline{f}\{\underline{e}/\underline{v}\}) \wedge \zeta \in \mathcal{S}_{\text{alph}(\sigma')}\} \\
= & [d:S:Inv] \\
& Innv(\underline{v} := \underline{f}\{\underline{e}/\underline{v}\}, \{\sigma'\})
\end{aligned}$$

□

C.10 Linearity of the remap operator

$$\boxed{(x \cdot \delta\{\underline{e}/\underline{v}\} + y \cdot \delta\{\underline{f}/\underline{v}\})\{\underline{g}/\underline{v}\} = x \cdot \delta\{\underline{e}/\underline{v}\}\{\underline{g}/\underline{v}\} + y \cdot \delta\{\underline{f}/\underline{v}\}\{\underline{g}/\underline{v}\}}$$

Proof:

$$\begin{aligned}
& (x \cdot \delta\{\underline{e}/\underline{v}\} + y \cdot \delta\{\underline{f}/\underline{v}\})\{\underline{g}/\underline{v}\}(\sigma) \\
= & [d:D:Rmp] \text{ — §A.3.4} \\
& \|(x \cdot \delta\{\underline{e}/\underline{v}\} + y \cdot \delta\{\underline{f}/\underline{v}\})\{Innv(\underline{v} := \underline{g}, \{\sigma\})\}\| \\
= & [p:D:Sum:Wt] \text{ — §A.3.1} \\
& \|x \cdot \delta\{\underline{e}/\underline{v}\}\{Innv(\underline{v} := \underline{g}, \{\sigma\})\} + y \cdot \delta\{\underline{f}/\underline{v}\}\{Innv(\underline{v} := \underline{g}, \{\sigma\})\}\| \\
= & [d:D:Rmp] \\
& x \cdot \delta\{\underline{e}/\underline{v}\}\{\underline{g}/\underline{v}\}(\sigma) + y \cdot \delta\{\underline{f}/\underline{v}\}\{\underline{g}/\underline{v}\}(\sigma) \\
= & [d:D:Sum] \\
& (x \cdot \delta\{\underline{e}/\underline{v}\}\{\underline{g}/\underline{v}\} + y \cdot \delta\{\underline{f}/\underline{v}\}\{\underline{g}/\underline{v}\})(\sigma)
\end{aligned}$$

□

C.11 Composition (I)

$$\delta\{\underline{e}/\underline{v}\}\{\underline{f}/\underline{v}\} = \delta\{\underline{f}\{\underline{e}/\underline{v}\}/\underline{v}\}$$

Proof:

$$\begin{aligned}
& \delta\{\underline{e}/\underline{v}\}\{\underline{f}/\underline{v}\}(\sigma) \\
= & [d:D:Rmp] \text{ — §A.3.4} \\
& \|\delta\{\underline{e}/\underline{v}\}\{\text{Inv}(\underline{v} := \underline{f}, \{\sigma\})\}\| \\
= & [p:D:RstA:Wt] \text{ — §A.3} \\
& \sum_{\zeta \in \text{Inv}(\underline{v} := \underline{f}, \{\sigma\})} \delta\{\underline{e}/\underline{v}\}(\zeta) \\
= & [d:D:Rmp] \\
& \sum_{\zeta \in \text{Inv}(\underline{v} := \underline{f}, \{\sigma\})} \|\delta\{\text{Inv}(\underline{v} := \underline{e}, \{\zeta\})\}\| \\
= & [d:A:Inv] : \bigcup_{\zeta \in \text{Inv}(\underline{v} := \underline{f}, \{\sigma\})} \text{Inv}(\underline{v} := \underline{e}, \{\zeta\}) = \text{Inv}(\underline{v} := \underline{e}, \text{Inv}(\underline{v} := \underline{f}, \{\sigma\})) \\
& \|\delta\{\text{Inv}(\underline{v} := \underline{e}, \text{Inv}(\underline{v} := \underline{f}, \{\sigma\}))\}\| \\
= & [p:S:Inv:Nest] \text{ — §C.9} \\
& \|\delta\{\text{Inv}(\underline{v} := \underline{f}\{\underline{e}/\underline{v}\}, \{\sigma\})\}\| \\
= & [d:D:Rmp] \\
& \delta\{\underline{f}\{\underline{e}/\underline{v}\}/\underline{v}\}(\sigma)
\end{aligned}$$

□

C.12 Composition (II)

$$\delta\{\underline{e}/\underline{v}\}\{\underline{f}/\underline{v}\} = \delta\{\underline{f} \circ \underline{e}/\underline{v}\}$$

Proof:

$$\begin{aligned}
& \delta\{\underline{e}/\underline{v}\}\{\underline{f}/\underline{v}\} \\
= & [p:D:Rmp:Comp1] \text{ — §C.11} \\
& \delta\{\underline{f}\{\underline{e}/\underline{v}\}/\underline{v}\} \\
= & [d:E:Comp] \text{ — §A.2.1} \\
& \delta\{\underline{f} \circ \underline{e}/\underline{v}\}
\end{aligned}$$

□

C.13 Composition (III)

$$\delta\{\underline{e}/v_i\}\{\underline{f}/v_j\} = \delta\{(e, \underline{f}\{e/v_i\})/(v_i, v_j)\}$$

Proof: Special case of C.11, where $\underline{v} = \begin{pmatrix} v_i \\ v_j \end{pmatrix}$, $\underline{e} = \begin{pmatrix} e \\ v_j \end{pmatrix}$ and $\underline{f} = \begin{pmatrix} v_i \\ f \end{pmatrix}$.

□

C.14 Composition (IV)

$$\delta\{e/v_i\}\{f/v_i\} = \delta\{f\{e/v_i\}/v_i\}$$

Proof: Special case of C.11, where $\underline{v} = (v_i)$, $\underline{e} = (e)$ and $\underline{f} = (f)$.

□

C.15 Iteration

$$\delta\{e/\underline{v}\}^k = \delta\{e^k/\underline{v}\}$$

Proof: By induction, the base case is trivial for $k = \{0, 1\}$.

Inductive hypothesis: $\delta\{e/\underline{v}\}^n = \delta\{e^n/\underline{v}\}$

$$\begin{aligned} & \delta\{e/\underline{v}\}^{n+1} \\ = & [d:D:Rmp:Iter] \text{ — §A.3.4} \\ & \delta\{e/\underline{v}\}^n \{e/\underline{v}\} \\ = & \text{Inductive hypothesis} \\ & \delta\{e^n/\underline{v}\} \{e/\underline{v}\} \\ = & [p:D:Rmp:Comp1] \text{ — §C.11} \\ & \delta\{e\{e^n/\underline{v}\}/\underline{v}\} \\ = & [d:E:Comp] \text{ — §A.2.1} \\ & \delta\{e \circ e^n/\underline{v}\} \\ = & [d:E:Comp:Iter] \text{ — §A.2.1} \\ & \delta\{e^{n+1}/\underline{v}\} \end{aligned}$$

□

C.16 Commutativity (I)

$$\delta\{e/v_i\}\{f/v_j\} = \delta\{f\{e/v_i\}/v_j\}\{e/v_i\} \text{ iff } v_j \notin fv(e)$$

Proof:

$$\begin{aligned} & \delta\{e/v_i\}\{f/v_j\} \\ = & [p:D:Rmp:Comp3] \text{ — §C.13} \\ & \delta\{(e, f\{e/v_i\})/(v_i, v_j)\} \\ = & \text{Substitution: } v_j \notin fv(e) \Rightarrow e\{x/v_j\} = e \\ & \delta\{(e\{f\{e/v_i\}/v_j\}, f\{e/v_i\})/(v_i, v_j)\} \\ = & \text{Substitution: } x = y\{x/y\} \\ & \delta\{(e\{f\{e/v_i\}/v_j\}, v_j\{f\{e/v_i\}/v_j\})/(v_i, v_j)\} \\ = & [p:D:Rmp:Comp3] \\ & \delta\{f\{e/v_i\}/v_j\}\{e/v_i\} \end{aligned}$$

□

C.17 Commutativity (II)

$$\delta\{e/v_i\}\{f/v_j\} = \delta\{f/v_j\}\{e/v_i\} \text{ iff } v_i \notin fv(f) \wedge v_j \notin fv(e)$$

Proof:

$$\begin{aligned} & \delta\{e/v_i\}\{f/v_j\} \\ = & [p:D:Rmp:Cmm1] \text{ — } \S C.16 \\ & \delta\{f\{e/v_i\}/v_j\}\{e/v_i\} \\ = & \text{Substitution: } v_i \notin fv(f) \Rightarrow f\{e/v_i\} = f \\ & \delta\{f/v_j\}\{e/v_i\} \end{aligned}$$

□

C.18 Expression substitution

$$\delta\{f = g\}\{e/v\} = \delta\{f = g\}\{e\{e/g\}/v\}$$

Proof:

$$\begin{aligned} & \delta\{f = g\}\{e\{e/g\}/v\} \\ = & [d:E:Ev] \text{ — } \S A.2.1 \\ & \delta\{f = g\}\{e/v\} \end{aligned}$$

□

C.19 Contradiction

$$\forall \sigma \in \text{dom}(\delta) \bullet \sigma(c\{e/v\}) = \text{false} \wedge \delta \neq \epsilon \Leftrightarrow \delta\{e/v\}\langle c \rangle = \epsilon$$

Proof:

$$\begin{aligned} & \forall \sigma \in \text{dom}(\delta) \bullet \sigma(c\{e/v\}) = \text{false} \wedge \delta \neq \epsilon \\ \equiv & [d:D:Rmp] \text{ — } \S A.3.4 \\ & \forall \sigma' \in \text{dom}(\delta\{e/v\}) \bullet \sigma'(c) = \text{false} \wedge \delta \neq \epsilon \\ \equiv & [d:D:Rst] \text{ — } \S A.3 \\ & \delta\{e/v\}\langle c \rangle = \epsilon \end{aligned}$$

□

C.20 Assertion

$$\forall \sigma \in \text{dom}(\delta) \bullet \sigma(c\{e/v\}) = \text{true} \Leftrightarrow \delta\{e/v\}\langle c \rangle = \delta\{e/v\}$$

Proof:

$$\begin{aligned} & \forall \sigma \in \text{dom}(\delta) \bullet \sigma(c\{e/v\}) = \text{true} \\ \equiv & [d:D:Rmp] \text{ — } \S A.3.4 \\ & \forall \sigma' \in \text{dom}(\delta\{e/v\}) \bullet \sigma'(c) = \text{true} \\ \equiv & [d:D:Rst] \text{ — } \S A.3 \\ & \delta\{e/v\}\langle c \rangle = \delta\{e/v\} \end{aligned}$$

□

C.21 Remapping a condition

$$\delta\{\underline{e}/\underline{v}\}\langle c \rangle = \delta\langle c\{\underline{e}/\underline{v}\}\rangle\{\underline{e}/\underline{v}\}$$

Proof:

$$\begin{aligned} & \delta\{\underline{e}/\underline{v}\}\langle c \rangle \\ = & [p:D:Sum:CS] \text{ --- } \S C.6 \\ & \delta\langle c\{\underline{e}/\underline{v}\}\rangle\{\underline{e}/\underline{v}\}\langle c \rangle + \delta\langle -c\{\underline{e}/\underline{v}\}\rangle\{\underline{e}/\underline{v}\}\langle c \rangle \\ = & [p:D:Rmp:Rst1] \text{ --- } \S C.19 \\ & \delta\langle c\{\underline{e}/\underline{v}\}\rangle\{\underline{e}/\underline{v}\}\langle c \rangle + \epsilon \\ = & [p:D:Rmp:Rst2] \text{ --- } \S C.20 \\ & \delta\langle c\{\underline{e}/\underline{v}\}\rangle\{\underline{e}/\underline{v}\} \end{aligned}$$

□

C.22 Weight of a distribution after remapping

$$\|\delta\{\underline{e}/\underline{v}\}\| = \|\delta\| \text{ iff } \sigma(\underline{e}) \text{ is defined in } \text{dom}(\delta)$$

Proof:

$$\begin{aligned} & \|\delta\{\underline{e}/\underline{v}\}\| \\ = & [d:D:Wt] \text{ --- } \S A.3 \\ & \sum_{\sigma' \in \text{dom } \delta\{\underline{e}/\underline{v}\}} \delta\{\underline{e}/\underline{v}\}(\sigma') \\ = & [d:D:Rmp] \text{ --- } \S A.3.4 \\ & \sum_{\sigma' \in \text{dom } \delta\{\underline{e}/\underline{v}\}} \|\delta\langle \text{Inv}(\underline{v} := \underline{e}, \{\sigma'\}) \rangle\| \\ = & [p:D:RstA:Wt] \text{ --- } \S A.3 \\ & \sum_{\sigma' \in \text{dom } \delta\{\underline{e}/\underline{v}\}} \left(\sum_{\sigma \in \text{Inv}(\underline{v} := \underline{e}, \{\sigma'\})} \delta(\sigma) \right) \\ = & [p:A:Inv:EqR] \text{ --- } \S A.2.3 : \bigcup_{\sigma' \in \text{dom } \delta\{\underline{e}/\underline{v}\}} \text{Inv}(\underline{v} := \underline{e}, \{\sigma'\}) = \text{dom } \delta \text{ iff } \sigma(\underline{e}) \text{ is defined in } \text{dom}(\delta) \\ & \sum_{\sigma \in \text{dom } \delta} \delta(\sigma) \\ = & [d:D:Wt] \\ & \|\delta\| \end{aligned}$$

□

C.23 Pseudo-associativity of probabilistic choice

$$\boxed{A_{p \oplus} (B_{q \oplus} C) \equiv (A_{r \oplus} B)_{s \oplus} C \wedge p = rs \wedge (1-s) = (1-p)(1-q)}$$

Proof:

$$\begin{aligned} & A_{p \oplus} (B_{q \oplus} C) \\ \equiv & [d:P:Ch:Prb] \text{ — §B.3} \\ & \exists \delta_A, \delta_{BC} \bullet A(p \cdot \delta, \delta_A) \wedge (B_{q \oplus} C)((1-p) \cdot \delta, \delta_{BC}) \wedge \delta' = \delta_A + \delta_{BC} \\ \equiv & [d:P:Ch:Prb] \wedge \delta_{BC} = \delta_B + \delta_C \text{ (One-point rule)} \\ & \exists \delta_A, \delta_B, \delta_C \bullet A(p \cdot \delta, \delta_A) \wedge B(q(1-p) \cdot \delta, \delta_B) \wedge C((1-q)(1-p) \cdot \delta, \delta_C) \wedge \delta' = \delta_A + \delta_B + \delta_C \\ \equiv & (1-p)(1-q) = (1-s) \wedge p = rs \Rightarrow q(1-p) = (1-r)s \\ & \exists \delta_A, \delta_B, \delta_C \bullet A(rs \cdot \delta, \delta_A) \wedge B((1-r)s \cdot \delta, \delta_B) \wedge C((1-s) \cdot \delta, \delta_C) \wedge \delta' = \delta_A + \delta_B + \delta_C \\ \equiv & [d:P:Ch:Prb] \wedge \delta_{AB} = \delta_A + \delta_B \text{ (One-point rule)} \\ & \exists \delta_{AB}, \delta_C \bullet (A_{r \oplus} B)(s \cdot \delta, \delta_{AB}) \wedge C((1-s) \cdot \delta, \delta_C) \wedge \delta' = \delta_{AB} + \delta_C \\ \equiv & [d:P:Ch:Prb] \\ & (A_{r \oplus} B)_{s \oplus} C \end{aligned}$$

□

C.24 Idempotency of choice operators

$$\boxed{\forall X \bullet \text{choice}(A, A, X) \equiv A}$$

Proof:

$$\begin{aligned} & \text{choice}(A, A, X) \\ \equiv & [d:P:Ch] \text{ — §3.4.1} \\ & \exists \pi, \delta_A, \delta_{\bar{A}} \bullet \pi \in X \wedge A(\delta\langle\pi\rangle, \delta_A) \wedge A(\delta\langle\bar{\pi}\rangle, \delta_{\bar{A}}) \wedge \delta' = \delta_A + \delta_{\bar{A}} \\ \equiv & [d:P:Structure] \text{ — §3.4.2} \\ & \exists \pi, \delta_A, \delta_{\bar{A}}, \text{QuantOf}(A) \bullet \pi \in X \wedge \delta_A = \text{BodyOf}(A) \circ \delta\langle\pi\rangle \wedge \delta_{\bar{A}} = \text{BodyOf}(A) \circ \delta\langle\bar{\pi}\rangle \wedge \delta' = \delta_A + \delta_{\bar{A}} \\ \equiv & \text{One-point rule} \\ & \exists \pi, \text{QuantOf}(A) \bullet \pi \in X \wedge \delta' = \text{BodyOf}(A) \circ \delta\langle\pi\rangle + \text{BodyOf}(A) \circ \delta\langle\bar{\pi}\rangle \\ \equiv & [p:D:Sum:CS] \text{ — §C.6} \\ & \exists \pi, \text{QuantOf}(A) \bullet \pi \in X \wedge \delta' = \text{BodyOf}(A) \circ \delta \\ \equiv & [d:P] \text{ — §3.3} \\ & A \end{aligned}$$

□

C.25 Discarding right-hand option

$$\boxed{\text{choice}(A, B, \{\iota\}) \equiv A}$$

Proof:

$$\begin{aligned}
& \text{choice}(A, B, \{\iota\}) \\
\equiv & [d:P:Ch] \text{ — §3.4.1} \\
& \exists \pi, \delta_A, \delta_B \bullet \pi \in \{\iota\} \wedge A(\delta\langle\pi\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
\equiv & \text{One-point rule} \\
& \exists \delta_A, \delta_B \wedge A(\delta\langle\iota\rangle, \delta_A) \wedge B(\delta\langle\epsilon\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
\equiv & [d:D:Rst] \text{ — §A.3} \\
& \exists \delta_A, \delta_B \wedge A(\delta, \delta_A) \wedge B(\epsilon, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
\equiv & \text{One-point rule} \\
& A
\end{aligned}$$

□

C.26 Distributivity of choice operators

$$\boxed{\text{choice}(A, (\text{choice}(B, C, X_2)), X_1) \equiv \text{choice}((\text{choice}(A, B, X_1)), (\text{choice}(A, C, X_1)), X_2)}$$

Proof:

$$\begin{aligned}
& \text{choice}(A, (\text{choice}(B, C, X_2)), X_1) \\
\equiv & [d:P:Ch] \text{ — §3.4.1} \\
& \exists \pi_1, \delta_A, \delta_{BC} \bullet \pi_i \in X_i \wedge A(\delta\langle\pi_1\rangle, \delta_A) \wedge (\text{choice}(B, C, X_2))(\delta\langle\bar{\pi}_1\rangle, \delta_{BC}) \wedge \delta' = \delta_A + \delta_{BC} \\
\equiv & [d:P:Ch] \wedge \delta_{BC} = \delta_B + \delta_C \text{ (One-point rule)} \\
& \exists \pi_1, \pi_2, \delta_A, \delta_B, \delta_C \bullet \pi_i \in X_i \wedge A(\delta\langle\pi_1\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}_1\rangle\langle\pi_2\rangle, \delta_B) \wedge C(\delta\langle\bar{\pi}_1\rangle\langle\bar{\pi}_2\rangle, \delta_C) \\
& \wedge \delta' = \delta_A + \delta_B + \delta_C \\
\equiv & [p:D:Sum:CS] \text{ — §C.6} \\
& \exists \pi_1, \pi_2, \delta_A, \delta_B, \delta_C \bullet \pi_i \in X_i \wedge A(\delta\langle\pi_1\rangle\langle\pi_2\rangle + \delta\langle\pi_1\rangle\langle\bar{\pi}_2\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}_1\rangle\langle\pi_2\rangle, \delta_B) \wedge \\
& \wedge C(\delta\langle\bar{\pi}_1\rangle\langle\bar{\pi}_2\rangle, \delta_C) \wedge \delta' = \delta_A + \delta_B + \delta_C \\
\equiv & \text{Linearity} \\
& \exists \pi_1, \pi_2, \delta_A, \delta_{\bar{A}}, \delta_B, \delta_C \bullet \pi_i \in X_i \wedge A(\delta\langle\pi_1\rangle\langle\pi_2\rangle, \delta_A) \wedge A(\delta\langle\pi_1\rangle\langle\bar{\pi}_2\rangle, \delta_{\bar{A}}) \wedge \\
& \wedge B(\delta\langle\bar{\pi}_1\rangle\langle\pi_2\rangle, \delta_B) \wedge C(\delta\langle\bar{\pi}_1\rangle\langle\bar{\pi}_2\rangle, \delta_C) \wedge \delta' = \delta_A + \delta_{\bar{A}} + \delta_B + \delta_C \\
\equiv & [d:P:Ch] \wedge \delta_{AB} = \delta_A + \delta_B \wedge \delta_{\bar{A}C} = \delta_{\bar{A}} + \delta_C \text{ (One-point rule)} \\
& \exists \pi_2, \delta_{AB}, \delta_{\bar{A}C} \bullet \pi_2 \in X_i \wedge (\text{choice}(A, B, X_1))(\delta\langle\pi_2\rangle, \delta_{AB}) \wedge (\text{choice}(\bar{A}, C, X_1))(\delta\langle\bar{\pi}_2\rangle, \delta_{\bar{A}C}) \\
& \wedge \delta' = \delta_{AB} + \delta_{\bar{A}C} \\
\equiv & [d:P:Ch] \\
& \text{choice}((\text{choice}(A, B, X_1)), (\text{choice}(A, C, X_1)), X_2)
\end{aligned}$$

□

C.27 Sequential composition

$$\boxed{choice(A, B, X); C \equiv choice((A; C), (B; C), X)}$$

Proof:

$$\begin{aligned}
& choice(A, B, X); C \\
\equiv & [d:P:Seq] \text{ — §B.3} \\
& \exists \delta_m \bullet choice(A, B, X)(\delta, \delta_m) \wedge C(\delta_m, \delta') \\
\equiv & [d:P:Ch] \text{ — §3.4.1} \\
& \exists \pi, \delta_A, \delta_B, \delta_m \bullet \pi \in X \wedge A(\delta\langle\pi\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}\rangle, \delta_B) \wedge \delta_m = \delta_A + \delta_B \wedge C(\delta_m, \delta') \\
\equiv & \text{One-point rule} \\
& \exists \pi, \delta_A, \delta_B \bullet \pi \in X \wedge A(\delta\langle\pi\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}\rangle, \delta_B) \wedge C(\delta_A + \delta_B, \delta') \\
\equiv & \text{Linearity} \\
& \exists \pi, \delta_A, \delta_B, \delta_C, \delta_{\bar{C}} \bullet \pi \in X \wedge A(\delta\langle\pi\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}\rangle, \delta_B) \wedge C(\delta_A, \delta_C) \wedge C(\delta_B, \delta_{\bar{C}}) \wedge \delta' = \delta_C + \delta_{\bar{C}} \\
\equiv & [d:P:Seq] \\
& \exists \pi, \delta_C, \delta_{\bar{C}} \bullet \pi \in X \wedge (A; C)(\delta\langle\pi\rangle, \delta_C) \wedge (B; \bar{C})(\delta\langle\bar{\pi}\rangle, \delta_{\bar{C}}) \wedge \delta' = \delta_C + \delta_{\bar{C}} \\
\equiv & [d:P:Ch] \\
& choice((A; C), (B; \bar{C}), X)
\end{aligned}$$

□

C.28 Choice flipping

$$\boxed{\forall X \bullet choice(A, B, X) \equiv choice(B, A, \bar{X}) \wedge \bar{X} = \bigcup_{\pi \in X} \bar{\pi}}$$

Proof:

$$\begin{aligned}
& choice(A, B, X) \\
\equiv & [d:P:Ch] \text{ — §3.4.1} \\
& \exists \pi, \delta_A, \delta_B \bullet \pi \in X \wedge A(\delta\langle\pi\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
\equiv & \bar{X} = \bigcup_{\pi \in X} \bar{\pi} \\
& \exists \bar{\pi}, \delta_A, \delta_B \bullet \bar{\pi} \in \bar{X} \wedge A(\delta\langle\bar{\pi}\rangle, \delta_A) \wedge B(\delta\langle\pi\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
\equiv & [d:P:Ch] \\
& choice(B, A, \bar{X})
\end{aligned}$$

□

C.29 Monotonicity of generic choice

$$\boxed{\forall \delta \bullet \mathcal{X}_1 \subseteq \mathcal{X}_2 \Rightarrow \text{choice}(A, B, \mathcal{X}_1)(\delta) \subseteq \text{choice}(A, B, \mathcal{X}_2)(\delta)}$$

Proof:

$$\begin{aligned}
& \text{choice}(A, B, \mathcal{X}_2)(\delta) \\
= & \quad [d:P:Ch] \text{ — §3.4.1} \\
& (\exists \pi, \delta_A, \delta_B \bullet \pi \in \mathcal{X}_2 \wedge A(\delta(\pi), \delta_A) \wedge B(\delta(\bar{\pi}), \delta_B) \wedge \delta' = \delta_A + \delta_B)(\delta) \\
= & \quad \text{Set theory} \wedge \mathcal{X}_1 \subseteq \mathcal{X}_2 \\
& (\exists \pi, \delta_A, \delta_B \bullet \pi \in \mathcal{X}_1 \cup (\mathcal{X}_2 \setminus \mathcal{X}_1) \wedge A(\delta(\pi), \delta_A) \wedge B(\delta(\bar{\pi}), \delta_B) \wedge \delta' = \delta_A + \delta_B)(\delta) \\
= & \quad [d:P:Ch] \\
& \text{choice}(A, B, \mathcal{X}_1)(\delta) \cup \text{choice}(A, B, \mathcal{X}_2 \setminus \mathcal{X}_1)(\delta)
\end{aligned}$$

□

C.30 Refinement relation for choices involving $\mathcal{X}_2 \subseteq \mathcal{X}_1$

$$\boxed{\mathcal{X}_2 \subseteq \mathcal{X}_1 \Rightarrow \text{choice}(A, B, \mathcal{X}_1) \sqsubseteq \text{choice}(A, B, \mathcal{X}_2)}$$

Proof:

$$\begin{aligned}
& \text{choice}(A, B, \mathcal{X}_1) \sqsubseteq \text{choice}(A, B, \mathcal{X}_2) \\
\equiv & \quad [d:P:Rfn:Alt] \text{ — §3.7} \\
& \forall \delta \bullet \text{choice}(A, B, \mathcal{X}_2)(\delta) \subseteq (\text{choice}(A, B, \mathcal{X}_1)(\delta))^\Delta \\
\equiv & \quad [p:P:Ch:Mntn] \text{ — §C.29} \wedge \forall X \bullet X \subseteq (X)^\Delta \\
& \forall \delta \bullet \text{true}
\end{aligned}$$

□

C.31 Refinement of the disjunction of two programs

$$\boxed{A \vee B \sqsubseteq A \text{ }_p\text{ } \oplus B}$$

Proof:

$$\begin{aligned}
& A \vee B \sqsubseteq A \text{ }_p\text{ } \oplus B \\
\equiv & \quad [d:P:Rfn:Alt] \text{ — §3.7} \\
& \forall \delta \bullet (A \text{ }_p\text{ } \oplus B)(\delta) \subseteq ((A \vee B)(\delta))^\Delta \\
\equiv & \quad \text{Set theory} \\
& \forall \delta, \delta' \bullet \delta' \in (A \text{ }_p\text{ } \oplus B)(\delta) \wedge \delta' \in ((A \vee B)(\delta))^\Delta \\
\equiv & \quad [d:P:Ch:Prb] \text{ — §B.3} \\
& \forall \delta, \delta', \delta'_A, \delta'_B \bullet \delta'_A \in A(\delta), \delta'_B \in B(\delta) \wedge \delta' = (p \cdot \delta'_A + (1-p) \cdot \delta'_B) \wedge \delta' \in ((A \vee B)(\delta))^\Delta \\
\equiv & \quad [d:P:RfnSet] \text{ — §3.7} \\
& \forall \delta \bullet \text{true}
\end{aligned}$$

□

C.32 Refinement of the disjunction of two programs

$$\boxed{A \vee B \sqsubseteq A \triangleleft c \triangleright B}$$

Proof:

$$\begin{aligned}
& A \vee B \sqsubseteq A \triangleleft c \triangleright B \\
\equiv & \quad [d:P:Rfn:Alt] \text{ — } \S 3.7 \\
& \forall \delta \bullet (A \triangleleft c \triangleright B)(\delta) \sqsubseteq ((A \vee B)(\delta))^\Delta \\
\equiv & \quad \text{Set theory} \\
& \forall \delta, \delta' \bullet \delta' \in (A \triangleleft c \triangleright B)(\delta) \wedge \delta' \in ((A \vee B)(\delta))^\Delta \\
\equiv & \quad [d:P:Ch:Cnd] \text{ — } \S B.3 \\
& \forall \delta, \delta', \delta'_A, \delta'_B \bullet \delta'_A \in A(\delta(c)), \delta'_B \in B(\delta(-c)) \wedge \delta' = \delta'_A + \delta'_B \wedge \delta' \in ((A \vee B)(\delta))^\Delta \\
\equiv & \quad [d:P:RfnSet] \text{ — } \S 3.7 \\
& \forall \delta \bullet \text{true}
\end{aligned}$$

□

C.33 Linking functions

$$\boxed{f = g^{-1}}$$

Proof:

$$\begin{aligned}
A_D(\delta) &= f(A_R)(\delta) = \sum_{\zeta \in \mathcal{S}} \delta(\zeta) \cdot A_R(\zeta) = \sum_{\zeta \in \mathcal{S}} \delta(\zeta) \cdot \delta'_{(A, \zeta)} \\
A_R(\sigma) &= g(A_D)(\sigma) = A_D(\eta_\sigma) = \delta'_{(A, \sigma)}.
\end{aligned}$$

$$\begin{aligned}
& g(f(A_R))(\sigma) \\
= & \quad \text{Definition of } f \\
& g\left(\sum_{\zeta \in \mathcal{S}} \delta(\zeta) \cdot A_R(\zeta)\right)(\sigma) \\
= & \quad \text{Definition of } g \\
& \sum_{\zeta \in \mathcal{S}} \eta_\sigma(\zeta) \cdot A_R(\zeta) \\
= & \quad \text{By definition, } \sigma \neq \zeta \Leftrightarrow \eta_\sigma(\zeta) = 0 \\
& \eta_\sigma(\sigma) \cdot A_R(\sigma) \\
= & \quad \text{By definition, } \eta_\sigma(\sigma) = 1 \\
& A_R(\sigma)
\end{aligned}$$

and

$$\begin{aligned}
& f(g(A_D))(\delta) \\
= & \text{Definition of } g \\
& f(A_R(\zeta))(\delta) \\
= & \text{Definition of } f \\
& \sum_{\zeta \in \mathcal{S}} \delta(\zeta) \cdot A_R(\zeta) \\
= & \text{Definition of } g \\
& A_D(\delta).
\end{aligned}$$

□

C.34 Feasibility

See B.3.2

Proof: As χ is a linear combination of a matrix whose rows are at most one-summing with the elements of χ' as coefficients, we have that the norm of χ cannot exceed that of χ' :

$$\begin{aligned}
& \|\chi\| \\
= & \\
& \sum_{i=0}^n |x_i| \\
= & \\
& \sum_{i=0}^n \left| \sum_{j=0}^n a_{ji} x'_j \right| \\
\leq & \quad \forall i, j \bullet a_{ji} \geq 0 \\
& \sum_{i=0}^n \sum_{j=0}^n a_{ji} |x'_j| \\
\leq & \quad \forall i \bullet \sum_{j=0}^n a_{ji} \leq 1 \\
& \sum_{i=0}^n |x'_i| \\
= & \\
& \|\chi'\|
\end{aligned}$$

□

C.35 Feasibility

See B.3.2

Proof: As δ' is a linear combination of a matrix whose columns are at most one-summing with the elements of δ as coefficients, we have that the norm of δ' cannot exceed that of δ :

$$\begin{aligned}
 & \|\delta'\| \\
 = & \\
 & \sum_{i=0}^n |p'_i| \\
 = & \\
 & \sum_{i=0}^n \left| \sum_{j=0}^n a_{ij} p_j \right| \\
 \leq & \quad \forall i, j \bullet a_{ij} \geq 0 \\
 & \sum_{i=0}^n \sum_{j=0}^n a_{ij} |p_j| \\
 \leq & \quad \forall j \bullet \sum_{i=0}^n a_{ij} \leq 1 \\
 & \sum_{j=0}^n |p_j| \\
 = & \\
 & \|\delta\|
 \end{aligned}$$

□

C.36 Monotonicity of A

See B.3.2

Proof: If we increase χ' , the corresponding $\chi = \underline{A}^T \chi'$ is increasing as well:

$$\begin{aligned}
 & \chi'_x \geq \chi'_y \\
 \equiv & \\
 & \begin{pmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_n \end{pmatrix} \geq \begin{pmatrix} y'_1 \\ y'_2 \\ \vdots \\ y'_n \end{pmatrix} \\
 \Rightarrow & \\
 & \exists i \bullet x'_i \geq y'_i \\
 \Rightarrow & \quad \forall j, k \bullet a_{jk} \geq 0 \\
 & \sum_{j=0}^n x'_j a_{j*}^T \leq \sum_{j=0}^n y'_j a_{j*}^T \\
 \equiv & \\
 & \underline{A}^T \chi'_x \geq \underline{A}^T \chi'_y \\
 \equiv & \\
 & \chi_x \geq \chi_y
 \end{aligned}$$

□

C.37 Monotonicity of A

See B.3.2

Proof: If we increase δ , the corresponding $\delta' = \underline{A}\delta$ is increasing as well:

$$\begin{aligned}
 & \delta'_p \geq \delta_q \\
 \equiv & \\
 & \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix} \geq \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \end{pmatrix} \\
 \Rightarrow & \\
 & \exists i \bullet p_i \geq q_i \\
 \Rightarrow & \quad \forall j, k \bullet a_{jk} \geq 0 \\
 & \sum_{k=0}^n p_k \underline{a}_{*k} \leq \sum_{k=0}^n q_k \underline{a}_{*k} \\
 \equiv & \\
 & \underline{A}\delta_p \geq \underline{A}\delta_q \\
 \equiv & \\
 & \delta'_p \geq \delta'_q
 \end{aligned}$$

□

C.38 Scaling

See B.3.2

Proof:

All definitions of the different program constructs contain the application of a matrix to a vector, when we see distributions as elements of a vector space: as this operation is linear, we can see that the constant can be placed anywhere in the a matrix composition.

□

C.39 Convexity

$$\boxed{(A \sqcap B)(\delta, \delta') \Rightarrow \delta' \geq \min(A(\delta) \cup B(\delta))}$$

Proof:

$$\begin{aligned}
 & \delta' \in (A \sqcap B)(\delta) \\
 \equiv & \quad \text{Definition of program image} \\
 & \delta' \in \bigcup_{\pi \in \mathcal{D}_w} (A(\delta\langle\pi\rangle) + B(\delta\langle\bar{\pi}\rangle)) \\
 \Rightarrow & \\
 & \delta' \geq \min(A(\delta) \cup B(\delta))
 \end{aligned}$$

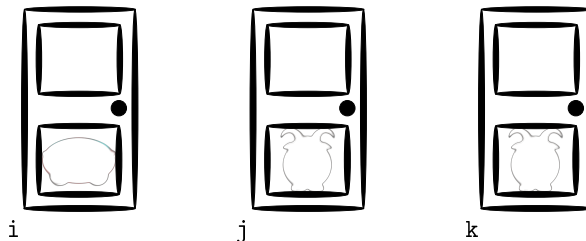
□

APPENDIX D

Other case studies

D.1 Monty Hall



In the Monty Hall game a player is challenged to guess behind which of the three doors in front of him hides a car.

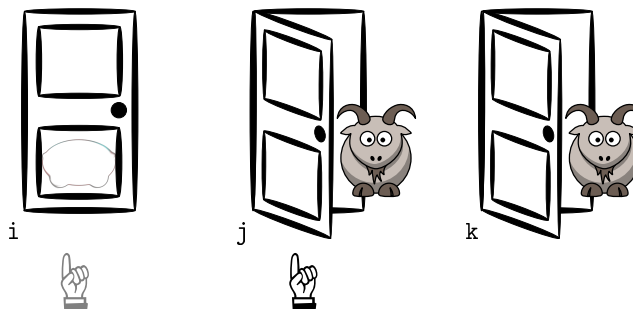


After having chosen a door among the three possible options, Monty Hall will open one of the remaining two doors. Monty Hall knows where the car is, so he is going to open one of the other two.

The player is given the chance to change his guess at this point.

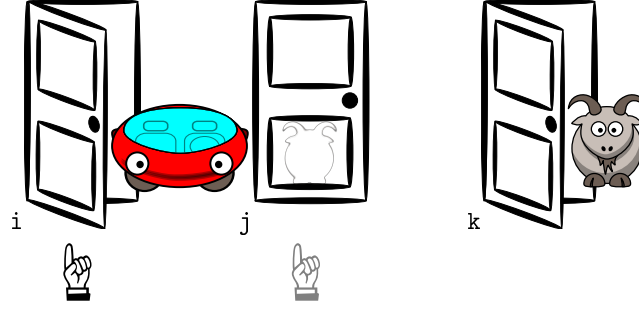
It is known from the literature¹ that the player will maximize the probability of finding the car if now he changes the door he has chosen (the probability will be $2/3$).

In fact the player can lose only if his first choice — indicated with  — was the i -th door, which is hiding the car (and this happens with probability $1/3$) so after Monty Hall has opened the k -th door, that is one of the two hiding a goat, the switching strategy leads the player's final choice — indicated with  — to be the j -th door, which is hiding a goat:



Nevertheless this is a winning strategy with probability $2/3$, as the chances of winning equal the chances of choosing a door hiding a goat, when all doors are closed. In fact choosing the j -th door forces Monty Hall to open the k -th door, and switching makes the player choose the i -th door:

¹Also back in 1935, it was known as Bertrand's box paradox (1889). This problem is often used as an example: among the papers cited as references, we can find it in McIver and Morgan [MM04] as well as in the more recent Chen and Sanders [CS09].



A short program, which uses the program constructs defined in §3, to implement the game is the following:

$$P \triangleq \text{setup}; \text{player}; \text{host}; \text{guess}$$

Let us use three variables a , b and c with the following meaning:

$a \triangleq$ the position of the car

$b \triangleq$ the player's guess

$c \triangleq$ Monty Hall's hint

we can then define each instruction as follows:

$$\text{setup} \triangleq a := 1 \sqcap (a := 2 \sqcap a := 3) \quad [1]$$

$$\text{player} \triangleq b := 1 \frac{1}{3} \oplus (b := 2 \frac{1}{2} \oplus b := 3) \quad [2]$$

$$\text{host} \triangleq c := \mathcal{S}(a, b) \triangleleft (a \neq b) \triangleright (c := \mathcal{H}_m(a) \sqcap c := \mathcal{H}_M(a)) \quad [3]$$

$$\text{guess} \triangleq b := \mathcal{S}(b, c) \quad [4]$$

Here is the definition of the functions mentioned in the program:

$$\mathcal{S}(x, y) \triangleq \min(\{1, 2, 3\} \setminus \{x, y\})$$

$$\mathcal{H}_m(x) \triangleq \min(\{1, 2, 3\} \setminus \{x\})$$

$$\mathcal{H}_M(x) \triangleq \max(\{1, 2, 3\} \setminus \{x\})$$

Let $\underline{v} = (a, b, c)$ and $\text{type}(a) = \text{type}(b) = \text{type}(c) = \{1, 2, 3\}$: the state space is

$$\mathcal{S} = \{\sigma \mid \sigma = \underline{v} \mapsto \underline{w} \wedge \underline{w} \in \text{type}(a) \times \text{type}(b) \times \text{type}(c)\}$$

For convenience we use σ_{ijk} to identify the state where $a = i$, $b = j$ and $c = k$; we represent the state distribution δ with a 27-element vector $\underline{\delta}$, whose components refer to the 27 possible states (in lexicographic order, *i.e.* the first element refers to σ_{111} , the second to σ_{112} and so on, till the 27th referring to σ_{333}) — and we use this notation to index the elements of all distributions and matrices.

The initial distribution is a parameter of the problem: we assume its weight is 1, but make no further assumptions on the individual weight of each state.

Let us now go through the first instruction:

$$\begin{aligned} a := i &= \delta' = \delta\{i/a\} \\ \text{setup} &= \exists \pi_1, \pi_2 \bullet \delta' = \delta\langle \pi_1 \rangle\{1/a\} + \delta\langle \pi_2 \rangle\{2/a\} + \delta\langle \iota - \pi_1 - \pi_2 \rangle\{3/a\} \end{aligned}$$

We leave implicit the condition that π_1 , π_2 and π_3 are weighting distributions, i.e. $\forall i \bullet \epsilon \leq \pi_i \leq \iota$.

After the second instruction we have:

$$\begin{aligned} b := i &= \delta'_j = \delta\{i/b\} \\ \text{player} &= \delta' = 1/3 \cdot \delta\{1/b\} + 1/3 \cdot \delta\{2/b\} + 1/3 \cdot \delta\{3/b\} \end{aligned}$$

We have an if-statement in the third instruction, so we have:

$$\begin{aligned} c := \mathcal{S}(a, b) &= \delta' = \delta\{\mathcal{S}(a, b)/c\} \\ c := \mathcal{H}_m(a) &= \delta' = \delta\{\mathcal{H}_m(a)/c\} \\ c := \mathcal{H}_M(a) &= \delta' = \delta\{\mathcal{H}_M(a)/c\} \\ c := \mathcal{H}_m(a) \sqcap c := \mathcal{H}_M(a) &= \exists \pi_{\mathcal{H}} \bullet \delta' = \delta\langle \pi_{\mathcal{H}} \rangle\{\mathcal{H}_m(a)/c\} + \delta\langle \iota - \pi_{\mathcal{H}} \rangle\{\mathcal{H}_M(a)/c\} \\ \text{host} &= \exists \pi_{\mathcal{H}} \bullet \delta' = \delta\langle a \neq b \rangle\{\mathcal{S}(a, b)/c\} + \\ &\quad + \delta\langle a = b \rangle\langle \pi_{\mathcal{H}} \rangle\{\mathcal{H}_m(a)/c\} + \delta\langle a = b \rangle\langle \iota - \pi_{\mathcal{H}} \rangle\{\mathcal{H}_M(a)/c\} \end{aligned}$$

Finally the fourth instruction gives

$$b := \mathcal{S}(b, c) = \delta' = \delta\{\mathcal{S}(b, c)/b\}$$

Let us now compose sequentially these constructs:

$$\begin{aligned}
& \text{setup}; \text{player}; \text{host}; \text{guess} \\
\equiv & \quad \text{Translation: } \text{setup}; \text{player} \text{ — with the position } \bar{\pi}_{12} = \iota - \pi_1 - \pi_2 \\
& \exists \pi_1, \pi_2 \bullet \delta' = \delta \langle \pi_1 \rangle \{1/a\} + \delta \langle \pi_2 \rangle \{2/a\} + \delta \langle \bar{\pi}_{12} \rangle \{3/a\} ; \\
& \quad ; \delta' = 1/3 \cdot \delta \{1/b\} + 1/3 \cdot \delta \{2/b\} + 1/3 \cdot \delta \{3/b\} ; \text{host}; \text{guess} \\
\equiv & \quad [\text{d:P:Seq}] \\
& \exists \pi_1, \pi_2, \delta_m \bullet \delta_m = \delta \langle \pi_1 \rangle \{1/a\} + \delta \langle \pi_2 \rangle \{2/a\} + \delta \langle \bar{\pi}_{12} \rangle \{3/a\} \wedge \\
& \quad \wedge \delta' = 1/3 \cdot \delta_m \{1/b\} + 1/3 \cdot \delta_m \{2/b\} + 1/3 \cdot \delta_m \{3/b\} ; \text{host}; \text{guess} \\
\equiv & \quad \text{One-point rule} \\
& \exists \pi_1, \pi_2 \bullet \delta' = 1/3 \cdot (\delta \langle \pi_1 \rangle \{1/a\} + \delta \langle \pi_2 \rangle \{2/a\} + \delta \langle \bar{\pi}_{12} \rangle \{3/a\}) \{1/b\} + \\
& \quad + 1/3 \cdot (\delta \langle \pi_1 \rangle \{1/a\} + \delta \langle \pi_2 \rangle \{2/a\} + \delta \langle \bar{\pi}_{12} \rangle \{3/a\}) \{2/b\} + \\
& \quad + 1/3 \cdot (\delta \langle \pi_1 \rangle \{1/a\} + \delta \langle \pi_2 \rangle \{2/a\} + \delta \langle \bar{\pi}_{12} \rangle \{3/a\}) \{3/b\} ; \text{host}; \text{guess} \\
\equiv & \quad \text{Translation: } \text{host} \\
& \exists \pi_1, \pi_2 \bullet \delta' = 1/3 \cdot (\delta \langle \pi_1 \rangle \{1/a\} + \delta \langle \pi_2 \rangle \{2/a\} + \delta \langle \bar{\pi}_{12} \rangle \{3/a\}) \{1/b\} + \\
& \quad + 1/3 \cdot (\delta \langle \pi_1 \rangle \{1/a\} + \delta \langle \pi_2 \rangle \{2/a\} + \delta \langle \bar{\pi}_{12} \rangle \{3/a\}) \{2/b\} + \\
& \quad + 1/3 \cdot (\delta \langle \pi_1 \rangle \{1/a\} + \delta \langle \pi_2 \rangle \{2/a\} + \delta \langle \bar{\pi}_{12} \rangle \{3/a\}) \{3/b\} ; \\
& \quad ; \exists \pi_{\mathcal{H}} \bullet \delta' = \delta \langle \mathbf{a} \neq \mathbf{b} \rangle \{ \mathcal{S}(\mathbf{a}, \mathbf{b})/c \} + \\
& \quad + \delta \langle \mathbf{a} = \mathbf{b} \rangle \langle \pi_{\mathcal{H}} \rangle \{ \mathcal{H}_m(\mathbf{a})/c \} + \delta \langle \mathbf{a} = \mathbf{b} \rangle \langle \bar{\pi}_{\mathcal{H}} \rangle \{ \mathcal{H}_M(\mathbf{a})/c \} ; \text{guess}
\end{aligned}$$

$$\begin{aligned}
&\equiv [p:D:Rmp:Lin] \\
&\exists \pi_1, \pi_2, \pi_{\mathcal{H}} \bullet \delta' = 1/3 \cdot \delta \langle \pi_2 \rangle \{ \{2/a\} \{1/b\} \langle a \neq b \rangle \{ \mathcal{S}(a,b)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \bar{\pi}_{12} \rangle \{ \{3/a\} \{1/b\} \langle a \neq b \rangle \{ \mathcal{S}(a,b)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \pi_1 \rangle \{ \{1/a\} \{2/b\} \langle a \neq b \rangle \{ \mathcal{S}(a,b)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \bar{\pi}_{12} \rangle \{ \{3/a\} \{2/b\} \langle a \neq b \rangle \{ \mathcal{S}(a,b)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \pi_1 \rangle \{ \{1/a\} \{3/b\} \langle a \neq b \rangle \{ \mathcal{S}(a,b)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \pi_2 \rangle \{ \{2/a\} \{3/b\} \langle a \neq b \rangle \{ \mathcal{S}(a,b)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \pi_1 \rangle \{ \{1/a\} \{1/b\} \langle a = b \rangle \langle \pi_{\mathcal{H}} \rangle \{ \mathcal{H}_m(a)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \pi_2 \rangle \{ \{2/a\} \{2/b\} \langle a = b \rangle \langle \pi_{\mathcal{H}} \rangle \{ \mathcal{H}_m(a)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \bar{\pi}_{12} \rangle \{ \{3/a\} \{3/b\} \langle a = b \rangle \langle \pi_{\mathcal{H}} \rangle \{ \mathcal{H}_m(a)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \pi_1 \rangle \{ \{1/a\} \{1/b\} \langle a = b \rangle \langle \bar{\pi}_{\mathcal{H}} \rangle \{ \mathcal{H}_M(a)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \pi_2 \rangle \{ \{2/a\} \{2/b\} \langle a = b \rangle \langle \bar{\pi}_{\mathcal{H}} \rangle \{ \mathcal{H}_M(a)/c \} \{ \mathcal{S}(b,c)/b \} \} + \\
&\quad + 1/3 \cdot \delta \langle \bar{\pi}_{12} \rangle \{ \{3/a\} \{3/b\} \langle a = b \rangle \langle \bar{\pi}_{\mathcal{H}} \rangle \{ \mathcal{H}_M(a)/c \} \{ \mathcal{S}(b,c)/b \} \}
\end{aligned}$$

Now that we have a statement describing the final distribution that results after the execution of the program, we can recognize two kind of terms:

- $\delta \{ \{i/a\} \{j/b\} \langle a \neq b \rangle \{ \mathcal{S}(a,b)/c \} \{ \mathcal{S}(b,c)/b \} \}$
- $\delta \{ \{i/a\} \{j/b\} \langle a = b \rangle \langle \pi \rangle \{ \mathcal{H}(a)/c \} \{ \mathcal{S}(b,c)/b \} \}$

where $i \neq j$.

We can see that the ones of the first kind account for cases when the player wins, while those of the second kind account for the cases when the player loses — let us see this by working out these terms, under the winning condition, *i.e.* $a = b$.

For terms of the first kind we have:

$$\begin{aligned}
&\delta \{ \{i/a\} \{j/b\} \langle a \neq b \rangle \{ \mathcal{S}(a,b)/c \} \{ \mathcal{S}(b,c)/b \} \langle a = b \rangle \} \\
&= [p:D:Rmp:Comp1] \\
&\delta \{ \{i/a\} \{j/b\} \langle a \neq b \rangle \{ \mathcal{S}(a,b), \mathcal{S}(b,c) \{ \mathcal{S}(a,b)/c \} /c, b \} \langle a = b \rangle \} \\
&= \text{Substitution: } c = \mathcal{S}(a, b) \\
&\delta \{ \{i/a\} \{j/b\} \langle a \neq b \rangle \{ \mathcal{S}(a,b), \mathcal{S}(b, \mathcal{S}(a,b)) /c, b \} \langle a = b \rangle \} \\
&= [p:D:Sum:CS] \\
&\delta \{ \{i/a\} \{j/b\} \langle a \neq b \rangle \{ \mathcal{S}(b, \mathcal{S}(a,b)) = a \} \{ \mathcal{S}(a,b), \mathcal{S}(b, \mathcal{S}(a,b)) /c, b \} \langle a = b \rangle \} + \\
&\quad + \delta \{ \{i/a\} \{j/b\} \langle a \neq b \rangle \{ \mathcal{S}(b, \mathcal{S}(a,b)) \neq a \} \{ \mathcal{S}(a,b), \mathcal{S}(b, \mathcal{S}(a,b)) /c, b \} \langle a = b \rangle \} \\
&= [p:D:Rmp:Rst1] \\
&\delta \{ \{i/a\} \{j/b\} \langle a \neq b \rangle \{ \mathcal{S}(b, \mathcal{S}(a,b)) = a \} \{ \mathcal{S}(a,b), \mathcal{S}(b, \mathcal{S}(a,b)) /c, b \} \langle a = b \rangle \} + \epsilon \\
&= [d:D:Sum] \\
&\delta \{ \{i/a\} \{j/b\} \langle a \neq b \rangle \{ \mathcal{S}(b, \mathcal{S}(a,b)) = a \} \{ \mathcal{S}(a,b), \mathcal{S}(b, \mathcal{S}(a,b)) /c, b \} \langle a = b \rangle \} \\
&= [p:D:Rst:Rmp] \\
&\delta \{ \{i/a\} \{j/b\} \langle a \neq b \rangle \{ \mathcal{S}(b, \mathcal{S}(a,b)) = a \} \{ \mathcal{S}(a,b), a/c, b \} \langle a = b \rangle \} \\
&= [p:D:Rmp:Rst2]
\end{aligned}$$

$$\begin{aligned}
& \delta \{i/a\} \{j/b\} \langle a \neq b \rangle \langle \mathcal{S}(b, \mathcal{S}(a, b)) = a \rangle \{ \mathcal{S}(a, b), a/c, b \} \\
= & \quad [\text{p:D:Rst:ImC1}] \\
& \delta \{i/a\} \{j/b\} \langle a \neq b \rangle \{ \mathcal{S}(a, b), a/c, b \} \\
= & \quad [\text{p:D:Rmp:Comp1}] \\
& \delta \{i, j/a, b\} \langle a \neq b \rangle \{ \mathcal{S}(a, b), a/c, b \} \\
= & \quad [\text{p:D:Rmp:Rst2}] \\
& \delta \{i, j/a, b\} \{ \mathcal{S}(a, b), a/c, b \}
\end{aligned}$$

As both remapping operations use expressions defined everywhere, thanks to [p:D:Rmp:Wt] we have that:

$$\| \delta \{i, j/a, b\} \{ \mathcal{S}(a, b), a/c, b \} \| = \| \delta \|$$

For terms of the second kind we have:

$$\begin{aligned}
& \delta \{i/a\} \{j/b\} \langle a = b \rangle \langle \pi \rangle \{ \mathcal{H}(a)/c \} \{ \mathcal{S}(b, c)/b \} \langle a = b \rangle \\
= & \quad [\text{p:D:Rmp:Comp1}] \\
& \delta \{i/a\} \{j/b\} \langle a = b \rangle \langle \pi \rangle \{ \mathcal{H}(a), \mathcal{S}(b, c) \{ \mathcal{H}(a)/c \} / c, b \} \langle a = b \rangle \\
= & \quad \text{Substitution: } c = \mathcal{H}(a) \\
& \delta \{i/a\} \{j/b\} \langle a = b \rangle \langle \pi \rangle \{ \mathcal{H}(a), \mathcal{S}(b, \mathcal{H}(a))/c, b \} \langle a = b \rangle \\
= & \quad [\text{p:D:Rst:ES}] \\
& \delta \{i/a\} \{j/b\} \langle a = b \rangle \langle \pi \rangle \{ \mathcal{H}(a), \mathcal{S}(a, \mathcal{H}(a))/c, b \} \langle a = b \rangle \\
= & \quad [\text{p:D:Rst:Wt}] \\
& \delta \{i/a\} \{j/b\} \langle a = b \rangle \langle \pi \rangle \{ \mathcal{H}(a), \mathcal{S}(a, \mathcal{H}(a))/c, b \} \langle a = b \rangle \\
= & \quad [\text{p:D:Rmp:Rst1}] \\
& \epsilon
\end{aligned}$$

Therefore we have:

$$\| \delta' \langle a = b \rangle \| = \| 2 \cdot (1/3 \cdot \delta \langle \pi_1 \rangle) + 1/3 \cdot \delta \langle \pi_2 \rangle + 1/3 \cdot \delta \langle \pi_3 \rangle \| = 2/3 \cdot \| \delta \|$$

We have assumed that the weight of the initial distribution is 1, so the weight of all winning states is $2/3$ — it is now clear why we did not need to make any other assumption, as this is all that matters, as all the variables undergo at least an assignment during the run of the program. $2/3$ is also the expected value for each of the initial states, so the pre-expectation assigning this weight to every state corresponds to the post-expectation of the predicate $\iota \langle a = b \rangle$.

We are now going to use the vector notation to solve this problem in a slightly different way. The predicate for the first instruction

$$\text{setup} = \exists \pi_1, \pi_2, \pi_3 \bullet \delta' = \delta \langle \pi_1 \rangle \{1/a\} + \delta \langle \pi_2 \rangle \{2/a\} + \delta \langle \pi_3 \rangle \{3/a\} \wedge \pi_1 + \pi_2 + \pi_3 = \iota$$

can be rewritten as:

$$\exists \underline{\pi}_1, \underline{\pi}_2, \underline{\pi}_3 \bullet \underline{\delta}' = \underline{A}_1 \underline{\pi}_1 \circ \underline{\delta} + \underline{A}_2 \underline{\pi}_2 \circ \underline{\delta} + \underline{A}_3 \underline{\pi}_3 \circ \underline{\delta} \wedge \underline{\pi}_1 + \underline{\pi}_2 + \underline{\pi}_3 = \underline{\iota},$$

where:

$$\underline{\underline{A}}_1 = \begin{pmatrix} \underline{\underline{I}} & \underline{\underline{I}} & \underline{\underline{I}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{0}} \end{pmatrix} \quad \underline{\underline{A}}_2 = \begin{pmatrix} \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{I}} & \underline{\underline{I}} & \underline{\underline{I}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{0}} \end{pmatrix} \quad \underline{\underline{A}}_3 = \begin{pmatrix} \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{I}} & \underline{\underline{I}} & \underline{\underline{I}} \end{pmatrix}.$$

The matrices above are 27×27 , and are made of blocks which are 9×9 .

For convenience we can rewrite this as:

$$\exists \underline{\underline{\pi}}_1, \underline{\underline{\pi}}_2 \bullet \underline{\underline{\delta}} = \underline{\underline{S}}_1 \underline{\underline{\delta}} \wedge \underline{\underline{S}}_1 = \underline{\underline{A}}_1 \text{diag}(\underline{\underline{\pi}}_1) + \underline{\underline{A}}_2 \text{diag}(\underline{\underline{\pi}}_2) + \underline{\underline{A}}_3 \text{diag}(\underline{\underline{\pi}}_3) \wedge \underline{\underline{\pi}}_1 + \underline{\underline{\pi}}_2 + \underline{\underline{\pi}}_3 = \underline{\underline{1}}$$

Therefore, if we use $\pi_{(l,ijk)}$ to note the element of π_l with index ijk , we have that $\underline{\underline{S}}_1$ has the following shape:

$$\underline{\underline{S}}_1 = \begin{pmatrix} \pi_{(1,111)} & 0 & \cdots & \pi_{(1,211)} & 0 & \cdots & 0 & \pi_{(1,311)} & 0 & \cdots \\ 0 & \pi_{(1,112)} & \cdots & 0 & \pi_{(1,212)} & \cdots & 0 & 0 & \pi_{(1,312)} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \pi_{(2,111)} & 0 & \cdots & \pi_{(2,211)} & 0 & \cdots & 0 & \pi_{(2,311)} & 0 & \cdots \\ 0 & \pi_{(2,112)} & \cdots & 0 & \pi_{(2,212)} & \cdots & 0 & 0 & \pi_{(2,312)} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \pi_{(3,111)} & 0 & \cdots & \pi_{(3,211)} & 0 & \cdots & 0 & \pi_{(3,311)} & 0 & \cdots \\ 0 & \pi_{(3,112)} & \cdots & 0 & \pi_{(3,212)} & \cdots & 0 & 0 & \pi_{(3,312)} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

With the position:

$$\underline{\underline{P}}_{li} = \begin{pmatrix} \pi_{(l,i11)} & 0 & \cdots & 0 \\ 0 & \pi_{(l,i12)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \pi_{(l,i33)} \end{pmatrix}$$

we can write that:

$$\text{diag}(\underline{\underline{\pi}}_1) = \begin{pmatrix} \underline{\underline{P}}_{11} & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{P}}_{12} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{P}}_{13} \end{pmatrix}$$

and therefore:

$$\underline{\underline{S}}_1 = \begin{pmatrix} \underline{\underline{P}}_{11} & \underline{\underline{P}}_{12} & \underline{\underline{P}}_{13} \\ \underline{\underline{P}}_{21} & \underline{\underline{P}}_{22} & \underline{\underline{P}}_{23} \\ \underline{\underline{P}}_{31} & \underline{\underline{P}}_{32} & \underline{\underline{P}}_{33} \end{pmatrix}$$

We should note that:

$$\forall i \bullet \underline{P_{1i}} + \underline{P_{2i}} + \underline{P_{3i}} = \underline{I}$$

After the second instruction we have:

$$p_{layer} = \delta' = 1/3 \cdot \delta\{1/b\} + 1/3 \cdot \delta\{2/b\} + 1/3 \cdot \delta\{3/b\}.$$

This corresponds to the predicate:

$$\underline{\delta} = 1/3 \underline{B_1} \underline{\delta} + 1/3 \underline{B_2} \underline{\delta} + 1/3 \underline{B_3} \underline{\delta}$$

where:

$$\underline{B_1} = \begin{pmatrix} \begin{pmatrix} \underline{I} & \underline{I} & \underline{I} \\ \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{Q} & \underline{Q} & \underline{Q} \end{pmatrix} & \underline{Q} & \underline{Q} \\ \underline{Q} & \begin{pmatrix} \underline{I} & \underline{I} & \underline{I} \\ \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{Q} & \underline{Q} & \underline{Q} \end{pmatrix} & \underline{Q} \\ \underline{Q} & \underline{Q} & \begin{pmatrix} \underline{I} & \underline{I} & \underline{I} \\ \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{Q} & \underline{Q} & \underline{Q} \end{pmatrix} \end{pmatrix}$$

$$\underline{B_2} = \begin{pmatrix} \begin{pmatrix} \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{I} & \underline{I} & \underline{I} \\ \underline{Q} & \underline{Q} & \underline{Q} \end{pmatrix} & \underline{Q} & \underline{Q} \\ \underline{Q} & \begin{pmatrix} \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{I} & \underline{I} & \underline{I} \\ \underline{Q} & \underline{Q} & \underline{Q} \end{pmatrix} & \underline{Q} \\ \underline{Q} & \underline{Q} & \begin{pmatrix} \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{I} & \underline{I} & \underline{I} \\ \underline{Q} & \underline{Q} & \underline{Q} \end{pmatrix} \end{pmatrix}$$

$$\underline{B_3} = \begin{pmatrix} \begin{pmatrix} \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{I} & \underline{I} & \underline{I} \end{pmatrix} & \underline{Q} & \underline{Q} \\ \underline{Q} & \begin{pmatrix} \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{I} & \underline{I} & \underline{I} \end{pmatrix} & \underline{Q} \\ \underline{Q} & \underline{Q} & \begin{pmatrix} \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{I} & \underline{I} & \underline{I} \end{pmatrix} \end{pmatrix}$$

The matrices above are 27×27 , and are made of blocks which are 9×9 .

For convenience we can rewrite this as:

$$\underline{\delta} = \underline{S}_2 \underline{\delta}$$

where:

$$\underline{S}_2 = 1/3(\underline{B}_1 + \underline{B}_2 + \underline{B}_3) = 1/3 \begin{pmatrix} \begin{pmatrix} \underline{I} & \underline{I} & \underline{I} \\ \underline{I} & \underline{I} & \underline{I} \\ \underline{I} & \underline{I} & \underline{I} \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} \underline{I} & \underline{I} & \underline{I} \\ \underline{I} & \underline{I} & \underline{I} \\ \underline{I} & \underline{I} & \underline{I} \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} \underline{I} & \underline{I} & \underline{I} \\ \underline{I} & \underline{I} & \underline{I} \\ \underline{I} & \underline{I} & \underline{I} \end{pmatrix} \end{pmatrix}$$

The predicate for the third instruction

$$\begin{aligned} \text{host} &= \exists \pi_{\mathcal{H}} \bullet \delta' = \delta \langle a \neq b \rangle \{\mathcal{L}(a,b)/c\} + \\ &\quad + \delta \langle a = b \rangle \langle \pi_{\mathcal{H}} \rangle \{\mathcal{H}_m(a)/c\} + \delta \langle a = b \rangle \langle \iota - \pi_{\mathcal{H}} \rangle \{\mathcal{H}_M(a)/c\} \end{aligned}$$

corresponds to the predicate:

$$\exists \pi_{\mathcal{H}} \bullet \underline{\delta} = \underline{\mathcal{L}} \underline{C} \underline{\delta} + \underline{\mathcal{H}_m} \pi_{\mathcal{H}} \circ \underline{\bar{C}} \underline{\delta} + \underline{\mathcal{H}_M} \bar{\pi}_{\mathcal{H}} \circ \underline{\bar{C}} \underline{\delta}$$

The conditional is rendered through the diagonal matrix \underline{C} :

$$\underline{C} = \begin{pmatrix} c_{111} & 0 & 0 & \dots & 0 \\ 0 & c_{112} & 0 & \dots & 0 \\ 0 & 0 & c_{113} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & c_{333} \end{pmatrix}$$

where c_{ijk} equals 0 if $i = j$ and 1 otherwise, and therefore:

$$\underline{C} = \begin{pmatrix} \begin{pmatrix} \underline{0} & \underline{0} & \underline{0} \\ \underline{0} & \underline{I} & \underline{0} \\ \underline{0} & \underline{0} & \underline{I} \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} \underline{I} & \underline{0} & \underline{0} \\ \underline{0} & \underline{0} & \underline{0} \\ \underline{0} & \underline{0} & \underline{I} \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} \underline{I} & \underline{0} & \underline{0} \\ \underline{0} & \underline{I} & \underline{0} \\ \underline{0} & \underline{0} & \underline{0} \end{pmatrix} \end{pmatrix}$$

The assignments are represented by the following matrices:

$$\underline{\mathcal{L}} = \begin{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} & \underline{0} & \underline{0} & \dots & \underline{0} \\ \underline{0} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} & \underline{0} & \dots & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} & \dots & \underline{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \underline{0} & \underline{0} & \underline{0} & \dots & \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{pmatrix}$$

This accounts for the operation $c := \mathcal{S}(a, b)$, therefore the blocks \underline{Q}_c on the diagonal have 1s in the c -th row:

$$\underline{Q}_1 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \underline{Q}_2 = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \quad \underline{Q}_3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

and therefore:

$$\underline{\mathcal{L}} = \begin{pmatrix} \begin{pmatrix} \underline{Q}_2 & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_3 & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_2 \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} \underline{Q}_3 & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_1 & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_1 \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} \underline{Q}_2 & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_1 & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_1 \end{pmatrix} \end{pmatrix}$$

$$\underline{\mathcal{H}}_m = \begin{pmatrix} \begin{pmatrix} \underline{Q}_2 & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_2 & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_2 \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} \underline{Q}_1 & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_1 & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_1 \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} \underline{Q}_1 & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_1 & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_1 \end{pmatrix} \end{pmatrix}$$

$$\underline{\mathcal{H}}_M = \begin{pmatrix} \begin{pmatrix} \underline{Q}_3 & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_3 & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_3 \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} \underline{Q}_3 & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_3 & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_3 \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} \underline{Q}_2 & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_2 & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_2 \end{pmatrix} \end{pmatrix}$$

We can rewrite the predicate as:

$$\exists \pi_{\mathcal{H}} \bullet \delta = \underline{S}_3 \delta \wedge \underline{S}_3 = \underline{L}\underline{C} + \underline{\mathcal{H}}_m \text{diag}(\pi_{\mathcal{H}})\underline{\bar{C}} + \underline{\mathcal{H}}_M \text{diag}(\bar{\pi}_{\mathcal{H}})\underline{\bar{C}}$$

Therefore we have that \underline{S}_3 has the following shape:

$$\underline{S}_3 = \begin{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ \pi_{(\mathcal{H},111)} & \pi_{(\mathcal{H},112)} & \pi_{(\mathcal{H},113)} \\ \bar{\pi}_{(\mathcal{H},111)} & \bar{\pi}_{(\mathcal{H},112)} & \bar{\pi}_{(\mathcal{H},113)} \end{pmatrix} & \underline{0} & \underline{0} & \dots & \underline{0} \\ \underline{0} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} & \underline{0} & \dots & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} & \dots & \underline{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \underline{0} & \underline{0} & \underline{0} & \dots & \begin{pmatrix} \pi_{(\mathcal{H},331)} & \pi_{(\mathcal{H},332)} & \pi_{(\mathcal{H},333)} \\ \bar{\pi}_{(\mathcal{H},331)} & \bar{\pi}_{(\mathcal{H},332)} & \bar{\pi}_{(\mathcal{H},333)} \\ 0 & 0 & 0 \end{pmatrix} \end{pmatrix}$$

We can clearly recognize the blocks operating on states where the condition $a \neq b$ is verified from those operating on states where $a = b$.

With the position:

$$\underline{R}_{ij} = \begin{pmatrix} \pi_{(\mathcal{H}, ij1)} & 0 & 0 \\ 0 & \pi_{(\mathcal{H}, ij2)} & 0 \\ 0 & 0 & \pi_{(\mathcal{H}, ij3)} \end{pmatrix}$$

we can write that:

$$\text{diag}(\underline{\pi}_{\mathcal{H}}) = \begin{pmatrix} \begin{pmatrix} \underline{R}_{11} & \underline{0} & \underline{0} \\ \underline{0} & \underline{R}_{12} & \underline{0} \\ \underline{0} & \underline{0} & \underline{R}_{13} \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} \underline{R}_{21} & \underline{0} & \underline{0} \\ \underline{0} & \underline{R}_{22} & \underline{0} \\ \underline{0} & \underline{0} & \underline{R}_{23} \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} \underline{R}_{31} & \underline{0} & \underline{0} \\ \underline{0} & \underline{R}_{32} & \underline{0} \\ \underline{0} & \underline{0} & \underline{R}_{33} \end{pmatrix} \end{pmatrix}$$

and therefore:

$$\underline{S}_3 = \begin{pmatrix} \begin{pmatrix} (\underline{Q}_2 \underline{R}_{11} + \underline{Q}_3 \bar{\underline{R}}_{11}) & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_3 & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_2 \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} \underline{Q}_3 & \underline{0} & \underline{0} \\ \underline{0} & (\underline{Q}_1 \underline{R}_{22} + \underline{Q}_3 \bar{\underline{R}}_{22}) & \underline{0} \\ \underline{0} & \underline{0} & \underline{Q}_1 \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} \underline{Q}_2 & \underline{0} & \underline{0} \\ \underline{0} & \underline{Q}_1 & \underline{0} \\ \underline{0} & \underline{0} & (\underline{Q}_1 \underline{R}_{33} + \underline{Q}_2 \bar{\underline{R}}_{33}) \end{pmatrix} \end{pmatrix}$$

Finally the fourth instruction gives

$$b := \mathcal{S}(b, c) = \delta' = \delta \{ \mathcal{S}(b, c) / b \},$$

and this corresponds to the predicate:

$$\underline{\delta}' = \underline{S}_4 \underline{\delta}$$

where:

$$\underline{S}_4 = \begin{pmatrix} \begin{pmatrix} 000000000 \\ 000010010 \\ 000001001 \\ 100000100 \\ 000000000 \\ 001000000 \\ 000100000 \\ 010000000 \\ 000000000 \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} 000000000 \\ 000010010 \\ 000001001 \\ 100000100 \\ 000000000 \\ 001000000 \\ 000100000 \\ 010000000 \\ 000000000 \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} 000000000 \\ 000010010 \\ 000001001 \\ 100000100 \\ 000000000 \\ 001000000 \\ 000100000 \\ 010000000 \\ 000000000 \end{pmatrix} \end{pmatrix}$$

With the positions:

$$\underline{Z}_1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \underline{Z}_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \underline{Z}_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \underline{Z}_4 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

we have:

$$\underline{S}_4 = \begin{pmatrix} \begin{pmatrix} \underline{0} & \underline{Z}_1 & \underline{Z}_1 \\ \underline{Z}_2 & \underline{0} & \underline{Z}_3 \\ \underline{Z}_4 & \underline{Z}_3 & \underline{0} \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} \underline{0} & \underline{Z}_1 & \underline{Z}_1 \\ \underline{Z}_2 & \underline{0} & \underline{Z}_3 \\ \underline{Z}_4 & \underline{Z}_3 & \underline{0} \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} \underline{0} & \underline{Z}_1 & \underline{Z}_1 \\ \underline{Z}_2 & \underline{0} & \underline{Z}_3 \\ \underline{Z}_4 & \underline{Z}_3 & \underline{0} \end{pmatrix} \end{pmatrix}$$

Putting all of it together, we have that the program can be represented by the following predicate:

$$\exists \pi_1, \pi_2, \pi_{\mathcal{H}} \bullet \delta = \underline{S_4} \underline{S_3} \underline{S_2} \underline{S_1} \delta \wedge \text{conditions relating } \underline{S_1} \text{ and } \underline{S_3} \text{ to } \pi_1, \pi_2, \pi_{\mathcal{H}}$$

Let us do the maths now: we will take advantage of the matrices being sparse and with easily recognisable blocks².

$$\underline{S_4} \underline{S_3} = \begin{pmatrix} \begin{pmatrix} \underline{Q} & \underline{Z_1} & \underline{Z_1} \\ \underline{Z_2} & \underline{Q} & \underline{Z_3} \\ \underline{Z_4} & \underline{Z_3} & \underline{Q} \end{pmatrix} & \underline{Q} & \underline{Q} \\ \underline{Q} & \begin{pmatrix} \underline{Q} & \underline{Z_1} & \underline{Z_1} \\ \underline{Z_2} & \underline{Q} & \underline{Z_3} \\ \underline{Z_4} & \underline{Z_3} & \underline{Q} \end{pmatrix} & \underline{Q} \\ \underline{Q} & \underline{Q} & \begin{pmatrix} \underline{Q} & \underline{Z_1} & \underline{Z_1} \\ \underline{Z_2} & \underline{Q} & \underline{Z_3} \\ \underline{Z_4} & \underline{Z_3} & \underline{Q} \end{pmatrix} \end{pmatrix} \begin{pmatrix} \begin{pmatrix} (\underline{Q_2} \underline{R_{11}} + \underline{Q_3} \underline{\bar{R}_{11}}) & \underline{Q} & \underline{Q} \\ \underline{Q} & \underline{Q_3} & \underline{Q} \\ \underline{Q} & \underline{Q} & \underline{Q_2} \end{pmatrix} & \underline{Q} & \underline{Q} \\ \underline{Q} & \begin{pmatrix} \underline{Q_3} & \underline{Q} & \underline{Q} \\ \underline{Q} & (\underline{Q_1} \underline{R_{22}} + \underline{Q_3} \underline{\bar{R}_{22}}) & \underline{Q} \\ \underline{Q} & \underline{Q} & \underline{Q_1} \end{pmatrix} & \underline{Q} \\ \underline{Q} & \underline{Q} & \begin{pmatrix} \underline{Q_2} & \underline{Q} & \underline{Q} \\ \underline{Q} & \underline{Q_1} & \underline{Q} \\ \underline{Q} & \underline{Q} & (\underline{Q_1} \underline{R_{33}} + \underline{Q_2} \underline{\bar{R}_{33}}) \end{pmatrix} \end{pmatrix}$$

which is:

$$\underline{S_4} \underline{S_3} = \begin{pmatrix} \begin{pmatrix} \underline{Q} & \underline{Z_1} \underline{Q_3} & \underline{Z_1} \underline{Q_2} \\ \underline{Z_2} (\underline{Q_2} \underline{R_{11}} + \underline{Q_3} \underline{\bar{R}_{11}}) & \underline{Q} & \underline{Z_3} \underline{Q_2} \\ \underline{Z_4} (\underline{Q_2} \underline{R_{11}} + \underline{Q_3} \underline{\bar{R}_{11}}) & \underline{Z_3} \underline{Q_3} & \underline{Q} \end{pmatrix} & \underline{Q} & \underline{Q} \\ \underline{Q} & \begin{pmatrix} \underline{Q} & \underline{Z_1} (\underline{Q_1} \underline{R_{22}} + \underline{Q_3} \underline{\bar{R}_{22}}) & \underline{Z_1} \underline{Q_1} \\ \underline{Z_2} \underline{Q_3} & \underline{Q} & \underline{Z_3} \underline{Q_1} \\ \underline{Z_4} \underline{Q_3} & \underline{Z_3} (\underline{Q_1} \underline{R_{22}} + \underline{Q_3} \underline{\bar{R}_{22}}) & \underline{Q} \end{pmatrix} & \underline{Q} \\ \underline{Q} & \underline{Q} & \begin{pmatrix} \underline{Q} & \underline{Z_1} \underline{Q_1} & \underline{Z_1} (\underline{Q_1} \underline{R_{33}} + \underline{Q_2} \underline{\bar{R}_{33}}) \\ \underline{Z_2} \underline{Q_2} & \underline{Q} & \underline{Z_3} (\underline{Q_1} \underline{R_{33}} + \underline{Q_2} \underline{\bar{R}_{33}}) \\ \underline{Z_4} \underline{Q_2} & \underline{Z_3} \underline{Q_1} & \underline{Q} \end{pmatrix} \end{pmatrix}$$

As we have that:

$$\begin{aligned} \underline{Z_1} \underline{Q_1} &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \underline{Q} & \underline{Z_2} \underline{Q_2} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \underline{Q} \\ \underline{Z_3} \underline{Q_2} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \underline{Q} & \underline{Z_3} \underline{Q_3} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \underline{Q} \\ \underline{Z_4} \underline{Q_3} &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \underline{Q} \end{aligned}$$

and therefore:

$$\begin{aligned} \underline{Z_2} \underline{Q_2} \underline{R_{11}} &= \underline{Q} & \underline{Z_4} \underline{Q_3} \underline{\bar{R}_{11}} &= \underline{Q} & \underline{Z_1} \underline{Q_1} \underline{R_{22}} &= \underline{Q} \\ \underline{Z_3} \underline{Q_3} \underline{\bar{R}_{22}} &= \underline{Q} & \underline{Z_1} \underline{Q_1} \underline{R_{33}} &= \underline{Q} & \underline{Z_3} \underline{\bar{R}_{33}} \underline{Q_3} &= \underline{Q} \end{aligned}$$

²I did try to do a good part of the calculation on my HP49g+ with actual values and variables, but given the size of the matrices it is quite painful and error-prone.

we can further simplify this expression as:

$$\underline{S}_4 \underline{S}_3 = \begin{pmatrix} \begin{pmatrix} \underline{0} & \underline{Z}_1 \underline{Q}_3 & \underline{Z}_1 \underline{Q}_2 \\ \underline{Z}_2 \underline{Q}_3 \bar{R}_{11} & \underline{0} & \underline{0} \\ \underline{Z}_4 \underline{Q}_2 \bar{R}_{11} & \underline{0} & \underline{0} \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} \underline{0} & \underline{Z}_1 \underline{Q}_3 \bar{R}_{22} & \underline{0} \\ \underline{Z}_2 \underline{Q}_3 & \underline{0} & \underline{Z}_3 \underline{Q}_1 \\ \underline{0} & \underline{Z}_3 \underline{Q}_1 \bar{R}_{22} & \underline{0} \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} \underline{0} & \underline{0} & \underline{Z}_1 \underline{Q}_2 \bar{R}_{33} \\ \underline{0} & \underline{0} & \underline{Z}_3 \underline{Q}_1 \bar{R}_{33} \\ \underline{Z}_4 \underline{Q}_2 & \underline{Z}_3 \underline{Q}_1 & \underline{0} \end{pmatrix} \end{pmatrix}$$

We can also notice that:

$$\underline{Z}_1 \underline{Q}_2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \underline{Q}_2 \quad \underline{Z}_1 \underline{Q}_3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \underline{Q}_3$$

$$\underline{Z}_2 \underline{Q}_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \underline{Q}_1 \quad \underline{Z}_2 \underline{Q}_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \underline{Q}_3$$

$$\underline{Z}_3 \underline{Q}_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \underline{Q}_1 \quad \underline{Z}_4 \underline{Q}_2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \underline{Q}_2$$

and therefore:

$$\underline{S}_4 \underline{S}_3 = \begin{pmatrix} \begin{pmatrix} \underline{0} & \underline{Q}_3 & \underline{Q}_2 \\ \underline{Q}_3 \bar{R}_{11} & \underline{0} & \underline{0} \\ \underline{Q}_2 \bar{R}_{11} & \underline{0} & \underline{0} \end{pmatrix} & \underline{0} & \underline{0} \\ \underline{0} & \begin{pmatrix} \underline{0} & \underline{Q}_3 \bar{R}_{22} & \underline{0} \\ \underline{Q}_3 & \underline{0} & \underline{Q}_1 \\ \underline{0} & \underline{Q}_1 \bar{R}_{22} & \underline{0} \end{pmatrix} & \underline{0} \\ \underline{0} & \underline{0} & \begin{pmatrix} \underline{0} & \underline{0} & \underline{Q}_2 \bar{R}_{33} \\ \underline{0} & \underline{0} & \underline{Q}_1 \bar{R}_{33} \\ \underline{Q}_2 & \underline{Q}_1 & \underline{0} \end{pmatrix} \end{pmatrix}$$

Then we add \underline{S}_2 :

$$\exists \pi_1, \pi_2, \pi_{\mathcal{H}} \bullet \delta' = 1/3 \left(\begin{array}{c} \left(\begin{array}{ccc} \underline{(Q_2 + Q_3)} & \underline{(Q_2 + Q_3)} & \underline{(Q_2 + Q_3)} \\ \underline{Q_3 \bar{R}_{11}} & \underline{Q_3 \bar{R}_{11}} & \underline{Q_3 \bar{R}_{11}} \\ \underline{Q_2 \bar{R}_{11}} & \underline{Q_2 \bar{R}_{11}} & \underline{Q_2 \bar{R}_{11}} \end{array} \right) \underline{P_{11}} \quad \left(\begin{array}{ccc} \underline{(Q_2 + Q_3)} & \underline{(Q_2 + Q_3)} & \underline{(Q_2 + Q_3)} \\ \underline{Q_3 \bar{R}_{11}} & \underline{Q_3 \bar{R}_{11}} & \underline{Q_3 \bar{R}_{11}} \\ \underline{Q_2 \bar{R}_{11}} & \underline{Q_2 \bar{R}_{11}} & \underline{Q_2 \bar{R}_{11}} \end{array} \right) \underline{P_{12}} \quad \left(\begin{array}{ccc} \underline{(Q_2 + Q_3)} & \underline{(Q_2 + Q_3)} & \underline{(Q_2 + Q_3)} \\ \underline{Q_3 \bar{R}_{11}} & \underline{Q_3 \bar{R}_{11}} & \underline{Q_3 \bar{R}_{11}} \\ \underline{Q_2 \bar{R}_{11}} & \underline{Q_2 \bar{R}_{11}} & \underline{Q_2 \bar{R}_{11}} \end{array} \right) \underline{P_{13}} \\ \left(\begin{array}{ccc} \underline{Q_3 \bar{R}_{22}} & \underline{Q_3 \bar{R}_{22}} & \underline{Q_3 \bar{R}_{22}} \\ \underline{(Q_1 + Q_3)} & \underline{(Q_1 + Q_3)} & \underline{(Q_1 + Q_3)} \\ \underline{Q_1 \bar{R}_{22}} & \underline{Q_1 \bar{R}_{22}} & \underline{Q_1 \bar{R}_{22}} \end{array} \right) \underline{P_{21}} \quad \left(\begin{array}{ccc} \underline{Q_3 \bar{R}_{22}} & \underline{Q_3 \bar{R}_{22}} & \underline{Q_3 \bar{R}_{22}} \\ \underline{(Q_1 + Q_3)} & \underline{(Q_1 + Q_3)} & \underline{(Q_1 + Q_3)} \\ \underline{Q_1 \bar{R}_{22}} & \underline{Q_1 \bar{R}_{22}} & \underline{Q_1 \bar{R}_{22}} \end{array} \right) \underline{P_{22}} \quad \left(\begin{array}{ccc} \underline{Q_3 \bar{R}_{22}} & \underline{Q_3 \bar{R}_{22}} & \underline{Q_3 \bar{R}_{22}} \\ \underline{(Q_1 + Q_3)} & \underline{(Q_1 + Q_3)} & \underline{(Q_1 + Q_3)} \\ \underline{Q_1 \bar{R}_{22}} & \underline{Q_1 \bar{R}_{22}} & \underline{Q_1 \bar{R}_{22}} \end{array} \right) \underline{P_{23}} \\ \left(\begin{array}{ccc} \underline{Q_2 \bar{R}_{33}} & \underline{Q_2 \bar{R}_{33}} & \underline{Q_2 \bar{R}_{33}} \\ \underline{Q_1 \bar{R}_{33}} & \underline{Q_1 \bar{R}_{33}} & \underline{Q_1 \bar{R}_{33}} \\ \underline{(Q_1 + Q_2)} & \underline{(Q_1 + Q_2)} & \underline{(Q_1 + Q_2)} \end{array} \right) \underline{P_{31}} \quad \left(\begin{array}{ccc} \underline{Q_2 \bar{R}_{33}} & \underline{Q_2 \bar{R}_{33}} & \underline{Q_2 \bar{R}_{33}} \\ \underline{Q_1 \bar{R}_{33}} & \underline{Q_1 \bar{R}_{33}} & \underline{Q_1 \bar{R}_{33}} \\ \underline{(Q_1 + Q_2)} & \underline{(Q_1 + Q_2)} & \underline{(Q_1 + Q_2)} \end{array} \right) \underline{P_{32}} \quad \left(\begin{array}{ccc} \underline{Q_2 \bar{R}_{33}} & \underline{Q_2 \bar{R}_{33}} & \underline{Q_2 \bar{R}_{33}} \\ \underline{Q_1 \bar{R}_{33}} & \underline{Q_1 \bar{R}_{33}} & \underline{Q_1 \bar{R}_{33}} \\ \underline{(Q_1 + Q_2)} & \underline{(Q_1 + Q_2)} & \underline{(Q_1 + Q_2)} \end{array} \right) \underline{P_{33}} \end{array} \right) \delta$$

with side conditions relating \underline{R}_{ii} and \underline{P}_{ij} to $\pi_1, \pi_2, \pi_{\mathcal{H}}$.

We can infer the program properties by analysing this matrix, here are a few examples:

- the rows with index iii are null everywhere, so the program will never terminate in a state σ_{iii} where $a = b = c$;
- more in general, the rows with index iji and ijj are null everywhere, so the program will never terminate in a state σ_{iji} where $a = c$ or $b = c$;
- all of the columns are one-summing, so the program is always terminating.

We now want to focus on the probability $\|\delta'\langle a = b \rangle\|$ of the program ending in a winning state σ_{ijj} , where $a = b$, so let us extract the submatrix \underline{G} from the one above, by selecting the rows relative to winning states:

If we do all of the multiplications by \underline{P}_{ij} we obtain:

$$\underline{G} = \begin{pmatrix} \underline{g}_{*111} \\ \underline{g}_{*112} \\ \underline{g}_{*113} \\ \underline{g}_{*121} \\ \vdots \\ \underline{g}_{*133} \\ \underline{g}_{*221} \\ \underline{g}_{*222} \\ \underline{g}_{*223} \\ \underline{g}_{*231} \\ \vdots \\ \underline{g}_{*323} \\ \underline{g}_{*331} \\ \underline{g}_{*332} \\ \underline{g}_{*333} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 \\ \frac{\pi(1,111)}{3} & \frac{\pi(1,112)}{3} & \frac{\pi(1,113)}{3} & \dots & \frac{\pi(1,333)}{3} \\ \frac{\pi(1,111)}{3} & \frac{\pi(1,112)}{3} & \frac{\pi(1,113)}{3} & \dots & \frac{\pi(1,333)}{3} \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \\ \frac{\pi(2,111)}{3} & \frac{\pi(2,112)}{3} & \frac{\pi(2,113)}{3} & \dots & \frac{\pi(2,333)}{3} \\ 0 & 0 & 0 & \dots & 0 \\ \frac{\pi(2,111)}{3} & \frac{\pi(2,112)}{3} & \frac{\pi(2,113)}{3} & \dots & \frac{\pi(2,333)}{3} \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \\ \frac{\pi(3,111)}{3} & \frac{\pi(3,112)}{3} & \frac{\pi(3,113)}{3} & \dots & \frac{\pi(3,333)}{3} \\ \frac{\pi(3,111)}{3} & \frac{\pi(3,112)}{3} & \frac{\pi(3,113)}{3} & \dots & \frac{\pi(3,333)}{3} \\ 0 & 0 & 0 & \dots & 0 \end{pmatrix}$$

We can see that all of the generators of matrix \underline{G} have norm $2/3$ as $\pi(1,ijk) + \pi(2,ijk) + \pi(3,ijk) = 1$ for all i, j, k :

$$\forall i, j, k \bullet 2 \cdot \frac{\pi(1,ijk)}{3} + 2 \cdot \frac{\pi(2,ijk)}{3} + 2 \cdot \frac{\pi(3,ijk)}{3} = \frac{2}{3}$$

We can conclude that this is the probability of the program ending in a winning state, and it does not depend on the starting state (and therefore it does not depend on the initial distribution, as long as it is one-summing) as all of the generators have the same norm.

We would definitely like to work on smaller matrices: when is it possible and what is the price we pay for that?

To see this let us approach the problem from a different angle; first of all we partition the state space into 5 abstract states as in figure D.1:

$$\begin{aligned} \alpha_0 &= \{\sigma_{iii} \mid \sigma_{iii} \in \mathcal{S}\} = \{\sigma_{111}, \sigma_{222}, \sigma_{333}\} \\ \alpha_1 &= \{\sigma_{ijj} \mid \sigma_{ijj} \in \mathcal{S} \wedge i \neq j\} = \{\sigma_{112}, \sigma_{113}, \sigma_{221}, \sigma_{223}, \sigma_{331}, \sigma_{332}\} \\ \alpha_2 &= \{\sigma_{iji} \mid \sigma_{iji} \in \mathcal{S} \wedge i \neq j\} = \{\sigma_{121}, \sigma_{131}, \sigma_{212}, \sigma_{232}, \sigma_{313}, \sigma_{323}\} \\ \alpha_3 &= \{\sigma_{jii} \mid \sigma_{jii} \in \mathcal{S} \wedge i \neq j\} = \{\sigma_{122}, \sigma_{133}, \sigma_{211}, \sigma_{233}, \sigma_{311}, \sigma_{322}\} \\ \alpha_4 &= \{\sigma_{ijk} \mid \sigma_{ijk} \in \mathcal{S} \wedge i \neq j \wedge i \neq k \wedge j \neq k\} = \{\sigma_{123}, \sigma_{132}, \sigma_{213}, \sigma_{231}, \sigma_{312}, \sigma_{321}\} \end{aligned}$$

Let ξ be a distribution over the set $\{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4\}$: the vector $\underline{\xi}$ representing the distribution is a 5-element one.

The instruction *setup* = $a := 1 \sqcap (a := 2 \sqcap a := 3)$ can do the following:

- remap a state σ_{iii} to itself or to σ_{jii} , *i.e.* remap the abstract state α_0 to α_0 or α_3 ;
- remap a state σ_{ijj} to itself, to σ_{jij} or to σ_{kij} , *i.e.* remap the abstract state α_1 to α_1 , α_2 or α_4 ;

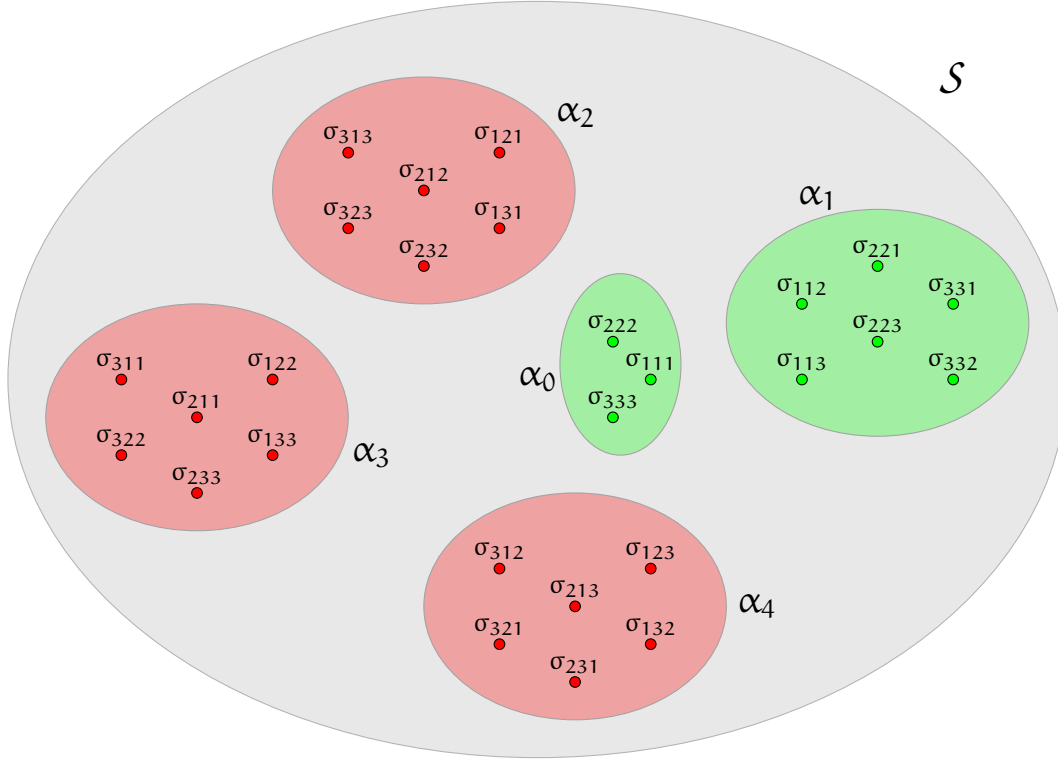


Figure D.1: The partition of the state space of program \mathcal{MH}

- remap a state σ_{iji} to itself, to σ_{ijj} or to σ_{kji} , *i.e.* remap the abstract state α_2 to α_2 , α_1 or α_4 ;
- remap a state σ_{jii} to itself or to σ_{iii} , *i.e.* remap the abstract state α_3 to α_0 or α_3 ;
- remap a state σ_{ijk} to itself, to σ_{kjk} or to σ_{jjk} , *i.e.* remap the abstract state α_4 to α_4 , α_2 or α_1 .

The choice among the different possibilities is done nondeterministically; the probability π_{ij} of remapping α_i to α_j can therefore vary arbitrarily with the following constraints:

$$\begin{aligned}\pi_{00} + \pi_{03} &= 1 \\ \pi_{11} + \pi_{12} + \pi_{14} &= 1 \\ \pi_{21} + \pi_{22} + \pi_{24} &= 1 \\ \pi_{30} + \pi_{33} &= 1 \\ \pi_{41} + \pi_{42} + \pi_{44} &= 1\end{aligned}$$

This operation can be expressed by the predicate³:

$$\exists \pi_{ij}, \pi_{kl} \bullet \underline{\xi}' = \underline{\mathbb{T}}_1 \underline{\xi} \wedge i, j \in \{1, 2, 4\} \wedge k, l \in \{0, 3\} \wedge \text{conditions on } \pi_{ij}, \pi_{kl} \text{ above}$$

³The quantification on the different π_{ij} is equivalent to the usual quantification on weighting distributions.

where

$$\underline{\underline{T}}_1 = \begin{pmatrix} \pi_{00} & 0 & 0 & \pi_{30} & 0 \\ 0 & \pi_{11} & \pi_{21} & 0 & \pi_{41} \\ 0 & \pi_{12} & \pi_{22} & 0 & \pi_{42} \\ \pi_{03} & 0 & 0 & \pi_{33} & 0 \\ 0 & \pi_{13} & \pi_{12} & 0 & \pi_{44} \end{pmatrix}$$

The instruction $player = b := 1 \frac{1}{3} \oplus (b := 2 \frac{1}{2} \oplus b := 3)$ can do the following:

- remap with probability $1/3$ a state σ_{iii} to itself, to σ_{iji} or to state σ_{iki} , *i.e.* remap the abstract state α_0 to α_0 with probability $1/3$ or to α_2 with probability $2/3$;
- remap with probability $1/3$ a state σ_{ijj} to itself, to σ_{ijj} or to σ_{ikj} , *i.e.* remap the abstract state α_1 to α_1 , α_4 or α_3 , with probability $1/3$ each;
- remap with probability $1/3$ a state σ_{iji} to itself, to σ_{iki} or to σ_{iij} , *i.e.* remap the abstract state α_2 to α_2 with probability $2/3$ or to α_0 with probability $1/3$;
- remap with probability $1/3$ a state σ_{jii} to itself to σ_{jii} or to σ_{jki} , *i.e.* remap the abstract state α_3 to α_3 , α_1 or α_3 , with probability $1/3$ each;
- remap with probability $1/3$ a state σ_{ijk} to itself, to σ_{iik} or to σ_{ikk} , *i.e.* remap the abstract state α_4 to α_4 , α_1 or α_3 , with probability $1/3$ each.

This operation can be expressed by the predicate:

$$\xi' = \underline{\underline{T}}_2 \xi$$

where

$$\underline{\underline{T}}_2 = \begin{pmatrix} 1/3 & 0 & 1/3 & 0 & 0 \\ 0 & 1/3 & 0 & 1/3 & 1/3 \\ 2/3 & 0 & 2/3 & 0 & 0 \\ 0 & 1/3 & 0 & 1/3 & 1/3 \\ 0 & 1/3 & 0 & 1/3 & 1/3 \end{pmatrix}$$

The instruction $host = c := \mathcal{S}(a, b) \triangleleft (a \neq b) \triangleright (c := \mathcal{H}_m(a) \sqcap c := \mathcal{H}_M(a))$ can do the following:

- remap a state σ_{iii} or a state σ_{ijj} to the state σ_{ijj} or σ_{iik} , *i.e.* remap the abstract states α_0 and α_1 to α_1 ;
- remap a state σ_{iji} , a state σ_{ijj} or a state σ_{ijk} to the state σ_{ijk} , *i.e.* remap the abstract states α_2 , α_3 and α_4 to α_4 ;

This operation can be expressed by the predicate:

$$\xi' = \underline{\underline{T}}_3 \xi$$

where

$$\underline{\underline{T}}_3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Finally the instruction $guess = b := \mathcal{S}(b, c)$ can do the following:

- remap a state σ_{iii} or a state σ_{iji} or a state to the state σ_{ili} , where $j \leq l$, i.e. remap the abstract states α_0 and α_2 to α_2 ;
- remap a state σ_{ijj} to the state σ_{ijk} , i.e. remap the abstract state α_1 to α_4 ;
- remap a state σ_{jii} to the state σ_{jli} , where $j \leq l$, i.e. remap the abstract state α_3 to α_1 or α_4 depending on j ;
- remap a state σ_{ijk} to the state σ_{iik} , i.e. remap the abstract state α_4 to α_2 .

This operation can be expressed by the predicate:

$$\underline{\xi}' = \underline{T}_4 \underline{\xi}$$

where

$$\underline{T}_4 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & p & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & \bar{p} & 0 \end{pmatrix}$$

Let us compute $\underline{T}_4 \underline{T}_3 \underline{T}_2 \underline{T}_1$ now:

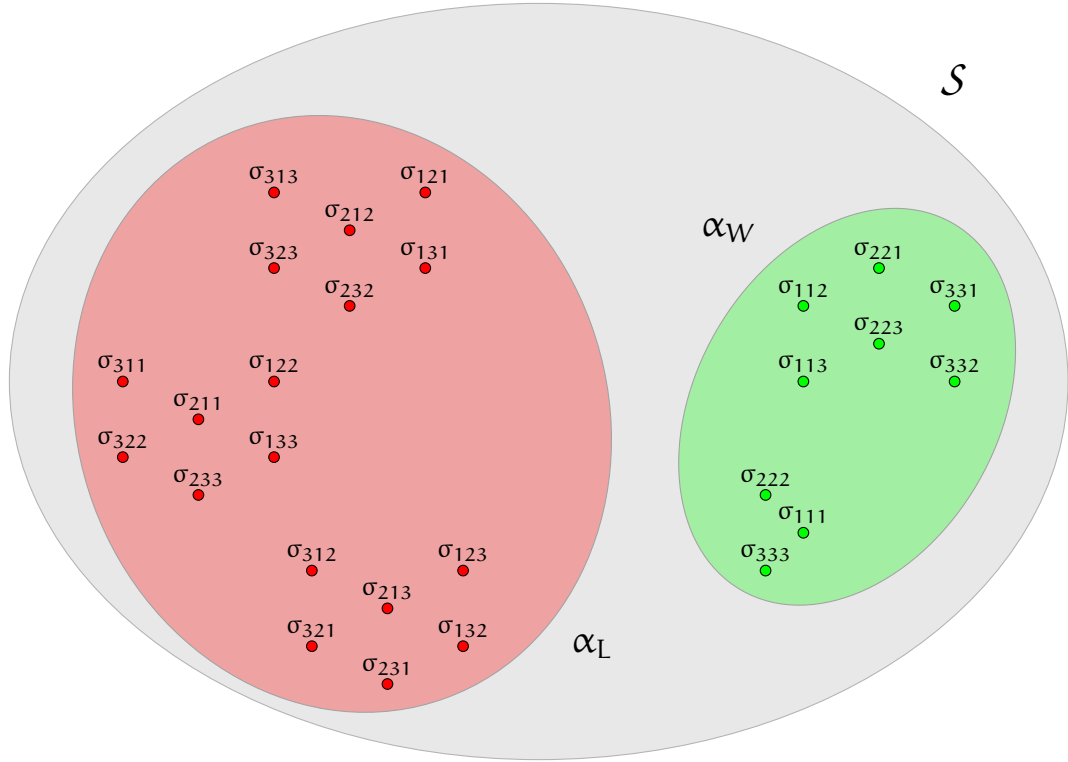
$$\underline{T}_4 \underline{T}_3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & p & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & \bar{p} & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$$\underline{T}_4 \underline{T}_3 \underline{T}_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1/3 & 0 & 1/3 & 0 & 0 \\ 0 & 1/3 & 0 & 1/3 & 1/3 \\ 2/3 & 0 & 2/3 & 0 & 0 \\ 0 & 1/3 & 0 & 1/3 & 1/3 \\ 0 & 1/3 & 0 & 1/3 & 1/3 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 2/3 & 2/3 & 2/3 & 2/3 & 2/3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 1/3 & 1/3 \end{pmatrix}$$

$$\underline{T}_4 \underline{T}_3 \underline{T}_2 \underline{T}_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 2/3 & 2/3 & 2/3 & 2/3 & 2/3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 1/3 & 1/3 \end{pmatrix} \begin{pmatrix} \pi_{00} & 0 & 0 & \pi_{30} & 0 \\ 0 & \pi_{11} & \pi_{21} & 0 & \pi_{41} \\ 0 & \pi_{12} & \pi_{22} & 0 & \pi_{42} \\ \pi_{03} & 0 & 0 & \pi_{33} & 0 \\ 0 & \pi_{13} & \pi_{12} & 0 & \pi_{44} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 2/3 & 2/3 & 2/3 & 2/3 & 2/3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 1/3 & 1/3 \end{pmatrix}$$

So we have that we can describe the program \mathcal{MH} with the following predicate:

$$\underline{\xi}' = \underline{T}_4 \underline{T}_3 \underline{T}_2 \underline{T}_1 \underline{\xi} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 2/3 & 2/3 & 2/3 & 2/3 & 2/3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 1/3 & 1/3 \end{pmatrix} \underline{\xi}$$

Figure D.2: A different partition of the state space of program \mathcal{MH}

If ξ is one-summing, then we can simply conclude that:

$$\underline{\xi}' = \begin{pmatrix} 0 \\ 2/3 \\ 0 \\ 0 \\ 1/3 \end{pmatrix}$$

Needless to say that we did not have to sweat as much as we did with the 27×27 matrices: we pay this by losing some details in the description, nevertheless if we are only interested in evaluating conditions that have the same truth value within every abstract state, then it is a perfectly reasonable approach.

Depending on the applications, we may be happy with this approach also if it introduces “false negatives”, *i.e.* we include some states where a condition evaluates to true in an abstract state where the condition is false, as the probability we estimate for a given condition is going to be lower than the actual probability.

Let us give it a try with a different partition of the state space (figure D.2), in order to show how to go from a description with a certain level of detail to a less precise one:

$$\begin{aligned} \alpha_W &= \alpha_0 \cup \alpha_1 = \{\sigma_{ij} \mid \sigma_{ij} \in \mathcal{S}\} \\ \alpha_L &= \alpha_2 \cup \alpha_3 \cup \alpha_4 = \{\sigma_{ijk} \mid \sigma_{ijk} \in \mathcal{S} \wedge i \neq j\} \end{aligned}$$

Let ζ be a distribution on the set $\{\alpha_W, \alpha_L\}$, we want to express the program behaviour with the predicate:

$$\exists \text{ variables due to demonic choice } \bullet \underline{\zeta}' \underline{U}_4 \underline{U}_3 \underline{U}_2 \underline{U}_1 \underline{\zeta}$$

We can write immediately the matrices \underline{U}_i :

- *setup* picks one random abstract state, and therefore $\underline{U}_1 = \begin{pmatrix} \pi_{WW} & \pi_{WL} \\ \pi_{LW} & \pi_{LL} \end{pmatrix}$,
where $\pi_{WW} + \pi_{LW} = \pi_{WL} + \pi_{LL} = 1$;
- *player* maps any abstract state to α_W with probability $1/3$ and to α_L with probability $2/3$,
and therefore $\underline{U}_2 = \begin{pmatrix} 1/3 & 1/3 \\ 2/3 & 2/3 \end{pmatrix}$;
- *host* does not change the current situation, and therefore $\underline{U}_3 = \underline{I}$
- *inverts* the current situation, mapping α_L to α_W and vice versa, and therefore $\underline{U}_4 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

If we do the maths we obtain:

$$\underline{\zeta}' = \begin{pmatrix} 2/3 & 2/3 \\ 1/3 & 1/3 \end{pmatrix} \underline{\zeta}$$

Which means that the probability of ending in the abstract state α_W is $2/3$.

Now, what is the relation between the matrices \underline{U}_i and \underline{T}_i ?

We always want that:

$$\zeta(\alpha_W) = \xi(\alpha_0) + \xi(\alpha_1) \wedge \zeta(\alpha_L) = \xi(\alpha_2) + \xi(\alpha_3) + \xi(\alpha_4)$$

and therefore, if we partition the matrices \underline{T}_i as follows:

$$\begin{aligned} \underline{T}_i &= \begin{pmatrix} \begin{pmatrix} t_{i11} & t_{i12} \\ t_{i21} & t_{i22} \end{pmatrix} & \begin{pmatrix} t_{i13} & t_{i14} & t_{i15} \\ t_{i23} & t_{i24} & t_{i25} \end{pmatrix} \\ \begin{pmatrix} t_{i31} & t_{i32} \\ t_{i41} & t_{i42} \\ t_{i51} & t_{i52} \end{pmatrix} & \begin{pmatrix} t_{i33} & t_{i34} & t_{i35} \\ t_{i43} & t_{i44} & t_{i45} \\ t_{i53} & t_{i54} & t_{i55} \end{pmatrix} \end{pmatrix} \\ &= \begin{pmatrix} \left(\underline{t}_{(1,iWW)} & \underline{t}_{(2,iWW)} \right) & \left(\underline{t}_{(1,iLW)} & \underline{t}_{(2,iLW)} & \underline{t}_{(3,iLW)} \right) \\ \left(\underline{t}_{(1,iWL)} & \underline{t}_{(2,iWL)} \right) & \left(\underline{t}_{(1,iLL)} & \underline{t}_{(2,iLL)} & \underline{t}_{(3,iLL)} \right) \end{pmatrix} = \begin{pmatrix} \underline{T}_{iWW} & \underline{T}_{iLW} \\ \underline{T}_{iWL} & \underline{T}_{iLL} \end{pmatrix} \end{aligned}$$

we can reduce the problem to that of relating the matrices \underline{T}_{iJK} , where $J, K \in \{W, L\}$, to the elements u_{iJK} of the matrices \underline{U}_i .

We have no problems in picking a value for u_{iJK} whenever the columns of \underline{T}_{iJK} have the same norm:

$$u_{iJK} = \|\underline{t}_{(1,iJK)}\|$$

This is for example the case of \underline{u}_2 w.r.t. \underline{T}_2 :

$$\underline{T}_2 = \left(\begin{array}{cc} \begin{pmatrix} 1/3 & 0 \\ 0 & 1/3 \end{pmatrix} & \begin{pmatrix} 1/3 & 0 & 0 \\ 0 & 1/3 & 1/3 \end{pmatrix} \\ \begin{pmatrix} 2/3 & 0 \\ 0 & 1/3 \\ 0 & 1/3 \end{pmatrix} & \begin{pmatrix} 2/3 & 0 & 0 \\ 0 & 1/3 & 1/3 \\ 0 & 1/3 & 1/3 \end{pmatrix} \end{array} \right) \sim \underline{u}_2 = \begin{pmatrix} 1/3 & 1/3 \\ 2/3 & 2/3 \end{pmatrix}$$

and that of \underline{u}_3 w.r.t. \underline{T}_3 :

$$\underline{T}_3 = \left(\begin{array}{cc} \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \end{array} \right) \sim \underline{u}_3 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

In the case of \underline{u}_4 w.r.t. \underline{T}_4 things are not so trivial, as the columns in the submatrices \underline{T}_{4JK} do not have the same norm, so we have to find an appropriate criterion to pick a value for u_{4JK} , with the constraint that:

$$u_{iJK} = \|\text{some linear combination of } \underline{t}_{(l,iJK)}\|$$

The coefficients of the linear combination have to be one-summing and must not vary for the same l and J ; it is not certain that we can find appropriate coefficients in all cases, but in this case we are happy with the following solution:

$$\begin{aligned} u_{4WW} &= \|\underline{t}_{(1,4WW)}\| = \|\underline{t}_{(2,4WW)}\| \\ u_{4LW} &= \|\underline{t}_{(1,4LW)}\| = \|\underline{t}_{(2,4LW)}\| \\ u_{4WL} &= \|\underline{t}_{(3,4WL)}\| \\ u_{4LL} &= \|\underline{t}_{(3,4LL)}\| \end{aligned}$$

$$\underline{T}_4 = \left(\begin{array}{cc} \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & p & 1 \end{pmatrix} \\ \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & \bar{p} & 0 \end{pmatrix} \end{array} \right) \sim \underline{u}_4 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

This is a good choice, as we can justify it by observing that we could easily extract $\underline{u}_4 \underline{u}_3 = \underline{u}_4$ from $\underline{T}_4 \underline{T}_3$:

$$\underline{T}_4 \underline{T}_3 = \begin{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{pmatrix} \rightsquigarrow \underline{u}_4 \underline{u}_3 = \underline{u}_4 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

In case of \underline{u}_1 w.r.t. \underline{T}_1 , we can see that the elements u_{1JK} are simply combinations of existentially quantified variables, so we can simply quantify over them, and therefore:

$$\exists \pi_{WW}, \pi_{LW}, \pi_{WL}, \pi_{LL} \bullet \zeta' \underline{u}_4 \underline{u}_3 \underline{u}_2 \underline{u}_1 \zeta \wedge \pi_{WW} + \pi_{LW} = \pi_{WL} + \pi_{LL} = 1$$

D.2 Rabin's choice coordination algorithm

The scenario which is customarily used to present Rabin's choice coordination algorithm is the following: N tourists have to gather in the same (indoor) place and have no means to communicate and agree where to meet among two alternatives.

The tourists have one notebook each and there is a blackboard in front of each of the two meeting points: at the beginning all tourists write a 0 on their notebooks and there is a 0 written on both blackboards.

Each tourist picks arbitrarily one of the possible meeting points to be visited first, and adopt the following strategy that has to be repeated until a final meeting point is elected (*i.e.* when he finds "HERE" on the blackboard in front of the meeting point):

- if the number on the blackboard is larger than that on his notebook, he erases the number on his notebook, notes down that on the blackboard and goes to the other meeting point;
- if the number on the blackboard is equal to that on his notebook, he replaces the number on the blackboard with the next even number and then he flips a (fair) coin: if it is head he increments the number on the blackboard by 1 and does nothing otherwise. Finally he replaces the number on his notebook with the one on the blackboard and heads to the other meeting point;
- if the number on his notebook is larger than that on the blackboard, he writes "HERE" on the blackboard and goes inside;
- obviously if he finds "HERE" on the blackboard, he simply goes inside.

The events happen symmetrically in both meeting places until the tourists flipping the coins obtain a different result: breaking the symmetry allows the election of a meeting place. Informally we can say that the probability that symmetry is not broken after K repetitions is $1/2^K$, so we probabilistically the whole process is going to terminate.

Let us write a program that simulates the tourists' behaviour. We use the following variables:

- $t_i \triangleq$ the position of the i -th tourist
- $a_i \triangleq$ the number on the i -th tourist's notebook
- $b_j \triangleq$ the number on the blackboard in the j -th meeting point

where

$$\begin{aligned} t_i &\in \mathbb{B} = \{0, 1\} \\ a_i &\in \mathbb{N} \\ b_j &\in \mathbb{N} \cup \{\text{HERE}\} \wedge \text{HERE} \triangleq -1 \end{aligned}$$

Let $\underline{v} = (b_0, b_1, t_1, a_1, t_2, a_2, \dots, t_N, a_N)$: the state space is

$$\mathcal{S} = \{\sigma \mid \sigma = \underline{v} \mapsto \underline{w} \wedge \underline{w} \in (\mathbb{B} \times \mathbb{N})^N \times (\mathbb{N} \cup \{\text{HERE}\})^2\}$$

As in the Monty Hall example, we choose the lexicographic order to sort the states and define the order in which they will appear in the vector representing the corresponding distribution. The first problem we immediately see is that the state base is infinite: for this reason we will not be able to write explicitly the matrices accounting for the different operations, but we will simply write down their characteristics.

The basic step can then be formalized as follows:

$$\begin{aligned} \text{tourist} t_i \triangleq \delta' = & \delta(t_i = z) (\langle a_i < b_z \rangle \{\{b_z/a_i\}\} \{\{\bar{t}_i/t_i\}\} + \\ & + \langle a_i = b_z \rangle (1/2 \{\{\tilde{b}_z/b_z\}\} + 1/2 \{\{\tilde{b}_z+1/b_z\}\}) \{\{b_z/a_i\}\} \{\{\bar{t}_i/t_i\}\} + \\ & + \langle a_i > b_z \rangle \{\{\text{HERE}/b_z\}\}) \end{aligned}$$

where

$$\begin{aligned} \bar{t} &\triangleq -t \\ \tilde{b}_z &\triangleq \begin{cases} b_z + 1 & \text{if } b_z \text{ is odd} \\ b_z + 2 & \text{if } b_z \text{ is even} \end{cases} \end{aligned}$$

Using the matrix notation, we get:

$$\underline{\delta}' = (\underline{C}_i \underline{N} \underline{S}_i + \underline{C}_i \underline{N} \underline{F} \underline{E}_i + \underline{H} \underline{G}_i) \underline{W}_{i,z} \underline{\delta}$$

where

- the matrix $\underline{W}_{i,z}$ (W as in “Where”) selects the states where $t_i = z$: this is a diagonal matrix where the diagonal element $w_{jj} = 1$ if the j -th state satisfies the condition $t_i = z$. Clearly we have that $\underline{W}_{i,0} + \underline{W}_{i,1} = \underline{I}$;
- the matrix \underline{S}_i (S as in “Smaller”) selects the states where $a_i < b_z$, the matrix \underline{E}_i (E as in “Equal”) selects the states where $a_i = b_z$ and the matrix \underline{G}_i (G as in “Greater”) selects the states where $a_i > b_z$: clearly we have that $\underline{S}_i + \underline{E}_i + \underline{G}_i = \underline{I}$;
- the matrix \underline{N} (N as in “Note”) is responsible for the assignment $a_i := b_z$: the columns of \underline{N} are null everywhere with the exception of a single value equal to 1 — it is so because this assignment is defined everywhere;
- the matrix \underline{E} (F as in “Flip”) is $\underline{E} = \underline{F}_H + \underline{F}_T$, *i.e.* the weighted sum of the two matrices \underline{F}_H and \underline{F}_T responsible for the assignments $b_z := \tilde{b}_z$ and $b_z := \tilde{b}_z + 1$, respectively: these matrices have columns that are null everywhere, with the exception of one value which

equals 1 (in a different position for each matrix), and as a result the columns of matrix \underline{E} are null everywhere, with the exception of two values which are equal to $1/2$;

- the matrix \underline{H} (H as in “Here”) is responsible for the assignment $b_z := \text{HERE}$, whereas the matrix \underline{C}_i (C as in “Change”) is responsible for the assignment $t_i := \bar{t}_i$: this matrices account for assignments defined everywhere, so they have a 1 in some position in each column, exactly as in the previous cases.

We initially work with the assumption that all of the tourists visit, in any sequence, the n -th place before a tourist can visit the $(n+1)$ -th one. The i -th tourist is the first one to arrive at the meeting place z and he finds on the blackboard the same number he has on his notebook, so he behaves as follows:

$$\delta' = \delta\{t_i = z\}\{a_i = b_z\}(1/2\{\bar{b}_z/b_z\} + 1/2\{\bar{b}_z+1/b_z\})\{b_z/a_i\}\{\bar{t}_i/t_i\}$$

The first three operations can be expressed as (we use ξ_1 instead of δ and add a tilde \sim on top of the matrix symbol to stress that the vector elements are sorted in a different order):

$$\underline{\xi}'_1 = \begin{pmatrix} \vdots \\ p'_{SS} \\ \vdots \\ p'_{S-} \\ p'_S \\ p'_E \\ \vdots \\ p'_{GG} \\ \vdots \\ p'_z \\ \vdots \end{pmatrix} = \widetilde{\underline{E}}_i \widetilde{W}_{iz} \xi_1$$

where the upper part of the vector $\underline{\xi}_1$ contains all states where $t_i = \bar{z}$ (and the lower part all others), and in particular:

- p_{SS} is the weight relative to some state where $a_i < b_z - 3$ and $t_i = z$;
- p_{S-} is the weight relative to some state σ where $a_i = b_z - 2$;
- p_S is the weight relative to some state where $a_i = b_z - 1$ and all other variables map to the same value as in σ ;
- p_E is the weight relative to some state where $a_i = b_z$ and all other variables map to the same value as in σ ;
- p_G is the weight relative to some state where $a_i > b_z + 2$;
- p_z is the weight relative to the first state where $t_i = z$;

and:

$$\begin{aligned}
 \underline{\tilde{W}}_{iz} &= \begin{pmatrix} \begin{pmatrix} \underline{I} & \vdots & \underline{Q} \\ \dots 0 & 1 & 0 \dots \\ \underline{Q} & 0 & \underline{I} \end{pmatrix} & \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{Q} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \underline{Q} & \underline{Q} \\ \underline{Q} & \underline{Q} & \begin{pmatrix} \underline{I} & \vdots & \underline{Q} \\ \dots 0 & 1 & 0 \dots \\ \underline{Q} & 0 & \underline{I} \end{pmatrix} & \underline{Q} \\ \underline{Q} & \underline{Q} & \underline{Q} & \begin{pmatrix} 0 & 0 \dots \\ 0 & \underline{Q} \\ \vdots & \underline{Q} \end{pmatrix} \end{pmatrix} \\
 \underline{\tilde{E}}_i &= \begin{pmatrix} \begin{pmatrix} \underline{Q} & \vdots & \underline{Q} \\ \dots 0 & 0 & 0 \dots \\ \underline{Q} & 0 & \underline{Q} \end{pmatrix} & \underline{Q} & \underline{Q} & \underline{Q} \\ \underline{Q} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \underline{Q} & \underline{Q} \\ \underline{Q} & \underline{Q} & \begin{pmatrix} \underline{Q} & \vdots & \underline{Q} \\ \dots 0 & 0 & 0 \dots \\ \underline{Q} & 0 & \underline{Q} \end{pmatrix} & \underline{Q} \\ \underline{Q} & \underline{Q} & \underline{Q} & \begin{pmatrix} - & 0 \dots \\ 0 & \text{diag}(-) \\ \vdots & \text{diag}(-) \end{pmatrix} \end{pmatrix} \\
 \underline{\tilde{E}} &= \begin{pmatrix} \begin{pmatrix} - & - & - \\ - & - & - \\ - & - & - \end{pmatrix} & - & - & - \\ \underline{Q} & \begin{pmatrix} 0 & 1/2 & 1/2 \\ 0 & 0 & 1/2 \\ 0 & 0 & 0 \end{pmatrix} & \underline{Q} & \underline{Q} \\ - & - & \begin{pmatrix} - & - & - \\ - & - & - \\ - & - & - \end{pmatrix} & - \\ - & - & - & \begin{pmatrix} - & - \\ - & - \end{pmatrix} \end{pmatrix}
 \end{aligned}$$

The two final operations can be expressed as (we use ξ_2 instead of δ or ξ_1 and add a hat $\hat{\cdot}$ on top of the matrix symbol to stress that the vector elements are sorted in a different order):

$$\underline{\xi}'_2 = \begin{pmatrix} \vdots \\ p'_{zSS} \\ \vdots \\ p'_{zE} \\ \vdots \\ p'_{zGG} \\ \vdots \\ p'_{\bar{z}SS} \\ \vdots \\ p'_{\bar{z}E} \\ \vdots \\ p'_{\bar{z}GG} \\ \vdots \end{pmatrix} = \underline{\hat{C}}_i \underline{\hat{N}} \xi_2$$

where the upper part of the vector $\underline{\xi}_1$ contains all states where $t_i = \bar{z}$ (and the lower part all others), and in particular:

- p_{zSS} is the weight relative to some state σ where $a_i < b_z - 2$ and $t_i = z$;
- p_{zE} is the weight relative to some state where $a_i = b_z$ and all other variables map to the same value as in σ ;
- p_{zGG} is the weight relative to some state where $a_i = b_z + 2$ and all other variables map to the same value as in σ ;
- $p_{\bar{z}SS}$ is the weight relative to some state ζ where $t_i = \bar{z}$ and all other variables map to the same value as in σ ;
- $p_{\bar{z}E}$ is the weight relative to some state where $a_i = b_z$ and all other variables map to the same value as in ζ ;
- $p_{\bar{z}GG}$ is the weight relative to some state where $a_i = b_z + 2$ and all other variables map to the same value as in ζ ;

and:

$$\underline{\hat{N}} = \begin{pmatrix} - & \underline{0} & - & \underline{0} & - \\ - & \begin{pmatrix} 0 \dots 0 \\ \dots \\ 1 \dots 1 \\ \dots \\ 0 \dots 0 \end{pmatrix} & - & \underline{0} & - \\ - & \underline{0} & - & \underline{0} & - \\ - & \underline{0} & - & \begin{pmatrix} 0 \dots 0 \\ \dots \\ 1 \dots 1 \\ \dots \\ 0 \dots 0 \end{pmatrix} & - \\ - & \underline{0} & - & \underline{0} & - \end{pmatrix}$$

$$\underline{\underline{\widehat{C}_i}} = \begin{pmatrix} - & \underline{\underline{0}} & - & \underline{\underline{0}} & - \\ - & \underline{\underline{0}} & - & \underline{\underline{I}} & - \\ - & \underline{\underline{0}} & - & \underline{\underline{0}} & - \\ - & \underline{\underline{I}} & - & \underline{\underline{0}} & - \\ - & \underline{\underline{0}} & - & \underline{\underline{0}} & - \end{pmatrix}$$

We can express all of this in terms of δ by using suitable permutation matrices:

$$\underline{\underline{\delta'}} = (\underline{\underline{P_2^{-1}}} \underline{\underline{\widehat{C}_i}} \underline{\underline{\widehat{N}}} \underline{\underline{P_2}}) (\underline{\underline{P_1^{-1}}} \underline{\underline{\widetilde{E}_i}} \underline{\underline{\widetilde{W}_{iz}}} \underline{\underline{P_1}}) \underline{\underline{\delta}}$$

where:

$$\underline{\underline{P_1^{-1}}} \underline{\underline{\xi_1}} = \underline{\underline{\delta}} \qquad \underline{\underline{P_2^{-1}}} \underline{\underline{\xi_2}} = \underline{\underline{\delta}}$$

Moreover we have that:

$$\begin{aligned} \underline{\underline{P_1^{-1}}} \underline{\underline{\widetilde{W}_{iz}}} \underline{\underline{P_1}} &= \underline{\underline{W_{iz}}} \\ \underline{\underline{P_1^{-1}}} \underline{\underline{\widetilde{E}_i}} \underline{\underline{P_1}} &= \underline{\underline{E_i}} \\ \underline{\underline{P_1^{-1}}} \underline{\underline{\widetilde{E}}} \underline{\underline{P_1}} &= \underline{\underline{E}} \\ \underline{\underline{P_2^{-1}}} \underline{\underline{\widehat{N}}} \underline{\underline{P_2}} &= \underline{\underline{N}} \\ \underline{\underline{P_2^{-1}}} \underline{\underline{\widehat{C}_i}} \underline{\underline{P_2}} &= \underline{\underline{C_i}} \end{aligned}$$

Although this shows how things are in line of principle, this is an example where it is not practical to work with matrices, so we will rely on the other formalism to verify (probabilistic) termination properties.

Going on with the description of the program, we look at the j -th tourist, who is the first to arrive at the meeting place \bar{z} , behaves as follows:

$$\delta' = \delta \langle t_j = \bar{z} \rangle \langle a_j = b_{\bar{z}} \rangle (1/2 \{ \bar{b}_{\bar{z}}/b_{\bar{z}} \} + 1/2 \{ \bar{b}_{\bar{z}+1}/b_{\bar{z}} \}) \{ b_{\bar{z}}/a_i \} \{ \bar{t}_j/t_j \}$$

All other tourists find a number which is larger than the one they have on their notebook, as the numbers on the blackboards have been modified by the i -th and the j -th tourists, so they simply update what's written on their notebook:

$$\delta' = \delta \langle t_k = y \rangle \langle a_k < b_y \rangle \{ b_y/a_k \} \{ \bar{t}_k/t_k \} \wedge (y = z \vee y = \bar{z})$$

If we sequentially compose the behaviours of all tourists, we have an expression where two terms stand out:

$$\delta' = \delta \dots (1/2 \{ \bar{b}_z/b_z \} + 1/2 \{ \bar{b}_{z+1}/b_z \}) \dots (1/2 \{ \bar{b}_{\bar{z}}/b_{\bar{z}} \} + 1/2 \{ \bar{b}_{\bar{z}+1}/b_{\bar{z}} \}) \dots$$

and this can be rewritten as:

$$\begin{aligned} \delta' = & 1/4 \cdot \delta \dots \{\{\bar{b}_z/b_z\} \dots \{\bar{b}_{\bar{z}}/b_{\bar{z}}\} \dots + 1/4 \cdot \delta \dots \{\{\bar{b}_z/b_z\} \dots \{\bar{b}_{\bar{z}+1}/b_{\bar{z}}\} \dots + \\ & + 1/4 \cdot \delta \dots \{\{\bar{b}_z+1/b_z\} \dots \{\bar{b}_{\bar{z}}/b_{\bar{z}}\} \dots + 1/4 \cdot \delta \dots \{\{\bar{b}_z+1/b_z\} \dots \{\bar{b}_{\bar{z}+1}/b_{\bar{z}}\} \dots \end{aligned}$$

Therefore all tourists that have visited the meeting point z have b_z on their notebook, whereas all others have $b_{\bar{z}}$. If the i -th and the j -th tourists have obtained the same result when flipping the coin, we have that $b_z = b_{\bar{z}}$, so everything is repeated again — and this happens each time with probability $1/2$, as we are dealing with full distributions.

If they obtained a different result — and this happens with probability for each coin flipping $1/2$ —, then we have $b_z \neq b_{\bar{z}}$, which leads to termination: let us consider the case when $b_z < b_{\bar{z}}$ — the complementary case is symmetrical.

The first tourist arriving to location z has $b_{\bar{z}}$ on his notebook and so behaves as follows:

$$\delta' = \delta \{a_i > b_z\} \{\text{HERE}/b_z\}$$

The following tourists arriving there will “find” $b_z = \text{HERE}$, so they have the same behaviour. All other tourists will arrive at location \bar{z} , which displays on the blackboard the value $b_{\bar{z}}$, which larger than that on their notebook (which is b_z), so their behaviour is:

$$\delta' = \delta \{t_k = \bar{z}\} (\{a_k < b_{\bar{z}}\} \{\bar{b}_z/a_k\} \{\bar{t}_k/t_k\})$$

They will finally head to location z , where they will find the other tourists.

The initialization of the program is a demonic choice, that sets the initial values of a_i , *i.e.* the first location to visit; then the whole program is a while-loop, where the guard is the condition $c \triangleq \exists z \forall i \bullet a_i = z$:

$$\text{init}; c * \text{sync}$$

where

$$\text{sync} \triangleq \text{any permutation of the } N \text{ different } \textit{tourist}_i$$

We have that:

$$\begin{aligned} c * \text{sync} = \delta' = & 1/4 \cdot \delta \dots \{\{\bar{b}_z+1/b_z\} \dots \{\bar{b}_{\bar{z}}/b_{\bar{z}}\} \dots + 1/4 \cdot \delta \dots \{\{\bar{b}_z/b_z\} \dots \{\bar{b}_{\bar{z}+1}/b_{\bar{z}}\} \dots + \\ & + 1/16 \cdot \delta \dots \{\{\bar{b}_z/b_z\} \dots \{\bar{b}_{\bar{z}}/b_{\bar{z}}\} \dots \{\{\bar{b}_z+1/b_z\} \dots \{\bar{b}_{\bar{z}}/b_{\bar{z}}\} \dots + \\ & + 1/16 \cdot \delta \dots \{\{\bar{b}_z/b_z\} \dots \{\bar{b}_{\bar{z}}/b_{\bar{z}}\} \dots \{\{\bar{b}_z/b_z\} \dots \{\bar{b}_{\bar{z}+1}/b_{\bar{z}}\} \dots + \\ & + 1/16 \cdot \delta \dots \{\{\bar{b}_z+1/b_z\} \dots \{\bar{b}_{\bar{z}+1}/b_{\bar{z}}\} \dots \{\{\bar{b}_z+1/b_z\} \dots \{\bar{b}_{\bar{z}}/b_{\bar{z}}\} \dots + \\ & + 1/16 \cdot \delta \dots \{\{\bar{b}_z+1/b_z\} \dots \{\bar{b}_{\bar{z}+1}/b_{\bar{z}}\} \dots \{\{\bar{b}_z/b_z\} \dots \{\bar{b}_{\bar{z}+1}/b_{\bar{z}}\} \dots + \\ & + \dots \end{aligned}$$

As we have full distribution, the probability of termination of each path is the initial coefficient, and therefore the overall probability $p(K)$ of termination after K steps is:

$$p(K) = \sum_{i=1}^K \frac{1}{2^i}$$

and therefore:

$$\lim_{K \rightarrow +\infty} p(K) = 1$$

Removing the assumption that they all visit the n -th location before a tourist can visit the $(n + 1)$ -th one does not compromise the algorithm, as any tourist that stays behind will update the content of his notebook with the current value he finds on the first blackboard he visits when he starts moving again, and this will lead him either to the elected meeting place (if one has already been picked) or in the same situation as all other tourists — to have probabilistic termination we have to make sure that no tourist can stay idle forever.

D.3 Protocol verification

In the last years research has strongly focused on proofs of security: the verification step to ensure that a computer program or a protocol have certain requested properties is a crucial one, and this task has to be done preferentially by formal reasoning, rather than by tests and simulations, as the latter approach is not as exhaustive as the formal one.

For a quite recent survey on the state of the art, one can refer to [ABCL09].

There are two possible approaches to protocol verification: the formal model and the computational model.

In the first model, we are in a highly idealized setting, whose properties can be expressed through logic and manipulated with formal techniques (for example rewriting rules or theorem proving), and therefore this can be effectively implemented in fully-automated protocol verifiers. A popular model is that described by Dolev and Yao in 1983 [DY83], which is presented in §D.3.1.

In this model we can reason about an idealized version of the protocol, so we can abstract from the implementation issues: for example a flaw in an implementation of a protocol due to overflow will not be detected in the formal model, but an error due to misconception of the protocol will be found by a protocol verifier.

The second approach adopts a computational perspective, focusing on the actual computations underlying a protocol, and borrows ideas from complexity theory. The goal is to provide a more accurate analysis, in terms of providing quantitative considerations on the security of a protocol, by estimating the number and the kind of operations required to break a protocol when using a certain attack. It requires much more human intervention in proofs, and is only recently being automated. [Bla08]

Bridging the gap between proofs in the formal model and in the computational model is one of the issues in protocol verification: in [AR02] the authors present a computational-soundness theorem, that relates the two views. Through this theorem it is possible to relate formally equivalent terms with computationally indistinguishable terms.

These verification techniques allow us to uncover design faults that may remain hidden for years. There are several successful episodes that can be recalled on this topic, and probably one of the best-known example is the formal verification of the popular Needham-Schroeder protocol by Gavin Lowe [Low95; Low96]: this protocol dates back to 1978, but it was just in 1995 that it was shown that a *Man-in-the-Middle* attack can effectively be mounted against this protocol — now the corrected version of the protocol is known as Needham-Schroeder-Lowe

protocol. To achieve this goal Gavin Lowe used the FDR tool [Fdr], which is a model checker for CSP.

Besides generic model checkers, there are tools which have been conceived specifically with communication protocols in mind. An example is Bruno Blanchet's ProVerif [Bla01; AB05; Bla08], which is the tool we will be using.

Having an adequate tool support has been a key aspect that has made protocol verification a mainstream activity in computer science, as it made protocol verification techniques available to the protocol development community.

To model cryptographic protocols we use *ad hoc* languages. In [AG97] the spi-calculus is formalised as an extension of Miller's π -calculus [Mil99], where the authors add cryptographic primitives.

This is a first step towards the applied π -calculus [AF01]: here functions and equations are added to the standard π -calculus, as well as the possibility of sending more complicated terms through channels. For this reason the authors felt the need of adding also a way to declare a short name for a more complicate expression (a kind of substitution).

Destructors and error handling are embedded in the applied π -calculus in [AB05].

D.3.1 The Dolev-Yao Model

The Dolev-Yao model (sketched in figure D.3) dates back to 1983 and it is still widely used in protocol verification [DY83]. In this model the net is seen as a star, where the attacker is in the central node and can act on every communication:

- the net is under the intruder's control: messages can be intercepted and altered. New messages can be injected to the net;
- the cryptographic primitives are perfect;
- the protocol admits any number of participants and any number of parallel sessions;
- the protocol messages can be of any size.

What the attacker cannot do is to break cryptographic functions, as they are assumed to be perfect. This means that a cyphered message can be decrypted only with the appropriate key or that a hashing function is collision-free.

The size of messages, keys and of any other term is irrelevant: this allows us to reason about the protocol abstracting from the actual implementations, as issues like overflows, different strength of keys, channel capacity and so on are not taken into account. This is useful, because the flaws that may eventually be found depend on the protocol, and not on the particular implementation of it.

Finally in the run of the protocol any number of participants is admitted: this means that there can be any number of parallel sessions that may interact.

Probabilistic variations on the Model

The scenario depicted in the Dolev-Yao model is a highly idealized one: it can be effectively implemented in protocol verifiers, but it is somehow far from the reality of things, as it relies on strong assumptions.

The idea of using a probabilistic calculus for cryptographic protocols has been investigated in a variety of paper, such as [Mit⁺01], where the authors present a calculus, which is a variation of CCS that focuses on probabilistic polynomial time.

The calculus is very similar to the π -calculus by Milner [Mil99] and to the spi-calculus [AG97], and bases the verification of security properties on observational equivalences on processes.

In this calculus polynomials are associated with terms: as the aim of this process calculus is to account for probabilistic polynomial-time adversaries, this calculus must be able to express some information about the width of a channel or about the number of feasible replications of a process. This results in a complicated semantics.

Similar ideas are expressed in [ZD04; ZD05], where the authors discuss the setting that results from having a probabilistic Dolev-Yao attacker who is able to guess a key with a given probability, and the related transition system. Generally speaking, the inference rules that an attacker can use are weighed by the probability that their application will be successful: the rules that characterise a classic Dolev-Yao attacker are weighed by probability $p = 1$, while the rules characterising a probabilistic Dolev-Yao attacker are weighed by a probability $p \leq 1$.

This preliminary work does not account for private channels or other operations normally forbidden to a Dolev-Yao attacker, such as decryption without key, signature forge, and so on — and the authors have now abandoned this line of research.

Nonetheless we can find a generalization of this in [Bau06]: here the Dolev-Yao model is seen

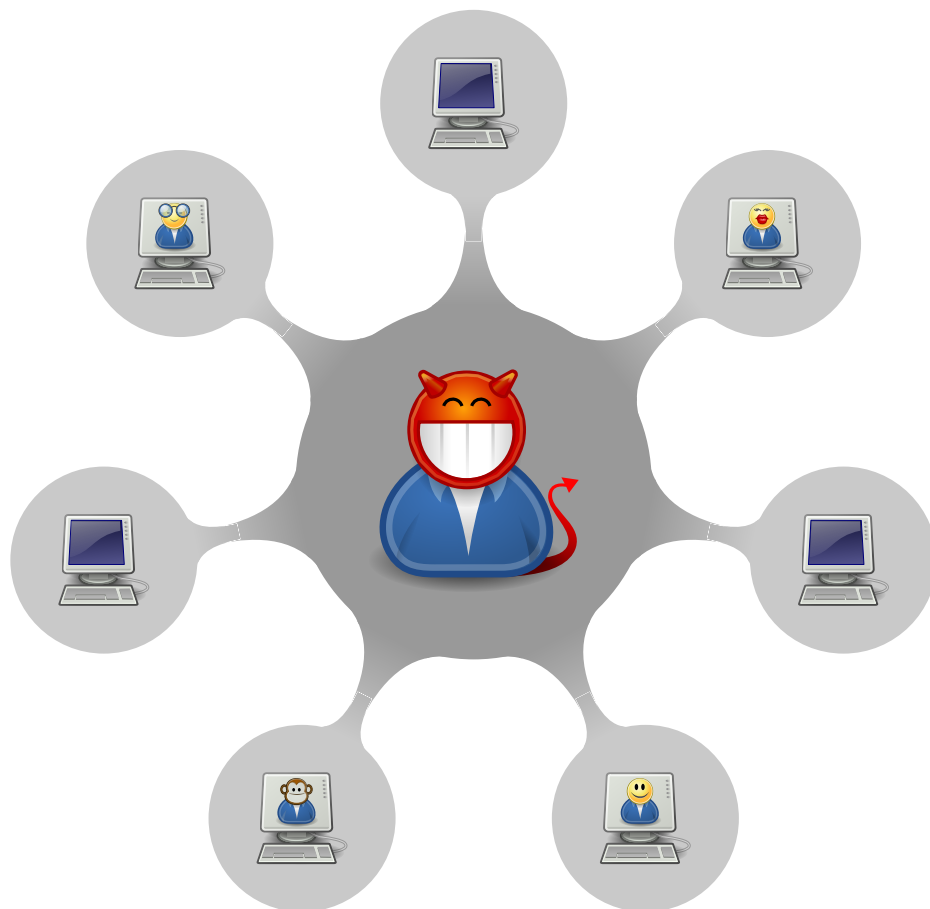


Figure D.3: The Dolev-Yao model.

as a transition system, probabilities and computational times are added by labelling the transitions.

In this transition system each transition stands for a computation and is labeled with a function that relates the computation with its cost: the weight function returns a non-negative real number or infinity (in case of impossible computations). The use of this framework allows us to evaluate the feasibility of an attack.

Later on in the paper the assumption that computation times are deterministic is removed and there are probabilistic computation times that label the transitions. Thanks to this a low probability can be used to label transitions accounting for a forbidden operation, such as breaking encryption without having the key: in this way what should not happen can be given the appropriate weight, and one can try to reason in the unlikely case that it does happen.

A technique for evaluating the probability of an attack in the case of a more powerful DY-attacker (that can guess keys and break some encryptions with a given probability) is presented in [Adã⁺06]: when building an attack trace it is possible to build a tree of it and by adding labels to the transitions among the states in the tree it is possible to evaluate the probability of such an attack by multiplication of these probabilities.

The authors of [DMV04] focus in particular on off-line guessing, which is a kind of attack that does not need to interact with the protocol. An attacker has got to be able to verify if a guess he has made makes sense or not — for example when decrypting messages with a guessed key, there must be at least a portion of them which is known and that can confirm that the key is the right one. The model proposed accounts for the intermediate computations done by an attacker, by means of maps.

Lately part of the research community has started to look to secure refinement calculus as to a new technique to be applied to security problems and this could extend the range of security applications that can be verified. [MM09; MMM09]

We are also moving in this direction, as we aim at using a probabilistic *UTP* framework, which bears the concept of refinement, to reason about protocols: in a probabilistic setting this will enable us to compare different protocols, in terms of the probability they may be broken.

D.3.2 A strategy to evaluate the probability of successful attacks by means of standard protocol verifiers

We relax the hypotheses underlying the Dolev-Yao model, in particular we aim at reasoning in a setting where the perfect cryptography assumption is weakened.

To remove the hypothesis that cryptographic primitives are perfect means that it is possible for a one-way function to be inverted, thus revealing its argument.

If we are dealing with a sound cryptosystem, the probability of this to happen is negligible, though non-zero. It is useful to evaluate this probability: a trivial application is to estimate the security which is gained by using larger keys. An evaluation of the strength of these functions implicitly carries information about the channel width, which the calculus proposed in [Mit⁺01] explicitly accounts for.

We can think of two extreme cases when having to violate a cryptosystem: one option is a completely random guess, the other is collecting enough data that can be used to break cryp-

tography. The general case is somewhere in between, ranging between these extremes.

Protocol verifiers such as ProVerif apply inference rules to derive terms from the data exchanged among the agents. The hypothesis of perfect cryptography may be instantiated in a protocol model by providing a constructor function, but no destructor function: once a constructor function is applied to some arguments it will not be possible to reverse it and find their values. This is the case for example of hashing functions: from the cryptographic hash of a string it is not possible to recover the original string.

In the case of public key cryptography the perfect cryptography assumption is rendered as a couple of constructor-destructor functions: the destructor function will return a result only if the appropriate key is provided.

When testing the model of a safe protocol, the protocol verifier will state that no attack on the protocol is possible. Conversely if the protocol was not safe, the protocol verifier will return the trace of a possible attack.

Introducing new destructors

Starting from a safe model of the protocol, what we are aiming at is a suitable modification of the model, that accounts for a possible break in the cryptography.

We do so by inserting new destructors: if the security of the protocol was relying on the perfect cryptography assumption, this will enable the protocol verifier to find an attack on the protocol.

The naive destructors that can be added are the ones accounting for random guesses: they simply behave as oracles that invert the one-way function. For example we can provide the attacker with a destructor returning the key that is used to encrypt a message, as well as providing him with a destructor that recovers the plain text from an encrypted message.

Finer destructors can be added, which take more than one argument. For example this may be used to take into account cryptoanalysis attacks: we can imagine a destructor enabling the attacker to recover a key after having collected a certain number of messages cyphered under that key. More in general we can model different information leaks and add destructor exploiting those leaks.

Examining the trace of the attack

We can think of these new destructors as functions that can be used by paying a price, and the price is that the probability of success of an attack is diminished accordingly to the probability that the functions used to perform that attack will return a correct value.

For example if an attack has been found and it does not use those destructors, it will succeed with probability $p = 1$: each inference rule which is applied gives a result with that probability, so the product of all those probabilities is $p = 1$.

Conversely an attack which uses once only one of such destructors will succeed with the probability that the destructor works properly: only one inference rule will succeed with probability $p \leq 1$, so the final probability of a successful attack coincides with that value.

Obviously in the case that an attack needs to use more destructors, its probability of success will be the probability that all the destructors return a correct result.

Considerations on the most successful attack

Unluckily once we have evaluated the probability for the discovered attack to succeed, still we cannot trivially be sure that there is not another attack that can succeed with a higher probability. But at least it is possible to give upper and lower bounds for the success probability, as well as an upper bound to the number of times when the attacker will have to rely on the new destructors.

The success probability p of breaking a protocol with the most successful attack is greater or equal to the success probability \hat{p} of the discovered attack.

$$p \geq \hat{p}$$

Similarly we can also see that the success probability p cannot be greater than the probability p_{med} of the most effective destructor to succeed, *i.e.* the one which most likely will return a correct result: as the protocol was safe before adding the new destructors, the most favourable situation for the attacker is when he needs to apply once only the most effective destructor, as the success probability of the attack coincides with the success probability of the destructor.

$$p \leq p_{\text{med}}$$

If this is not the case, we can at least estimate what the maximum number of application of these destructors is: this is the number of times that the most effective destructor can be applied before the probability of success drops below the success probability of the discovered attack.

These considerations can be used to guide the research for a more successful attack.

D.3.3 An example: using ProVerif to verify the Yahalom protocol

To illustrate the propose methodology, we will reason about the Yahalom protocol: the protocol verifier that will be used is ProVerif. It comes along with some examples files, among which there is a model of the Yahalom protocol (coded as a sequence of Horn clauses), which will be modified to suit our needs.

ProVerif

First of all a brief description of the tool used in this example, ProVerif: it is a protocol verifier written by Bruno Blanchet [AB05; Bla01].

The tool processes input files formatted as a sequence of Horn clauses or as a process in the applied π -calculus (a cryptographically-oriented variation of the π -calculus), which will be translated into Horn clauses before being run.

In particular, by means of ProVerif it is possible to verify secrecy properties, *i.e.* whether a Dolev-Yao attacker is able to derive a term from the messages exchanged among the agents: for example this can be used to prove the correctness of a key agreement protocol, by proving that a term, encrypted under the negotiated key and sent on a public channel, is not derivable by an attacker. This is the way we will use ProVerif to test the Yahalom protocol.

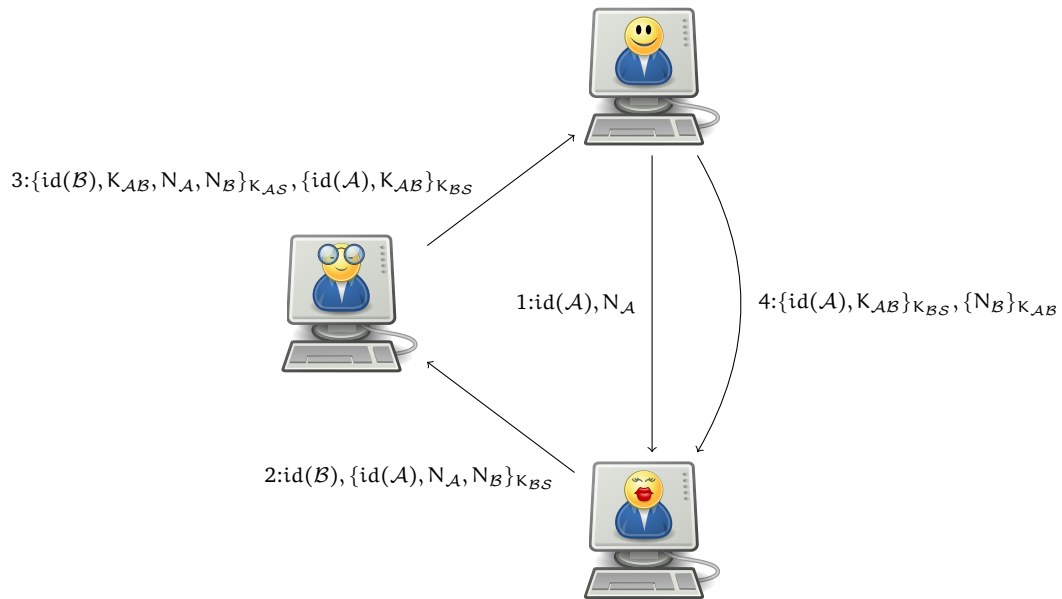


Figure D.4: The Yahalom protocol.

Protocol Description

During the protocol flow (see figure D.4) a total of 4 messages are exchanged among two agents, \mathcal{A} and \mathcal{B} , and a server, \mathcal{S} . Both agents share a key with the server and can generate fresh nonces and keys. The server knows the identity of the agents and its own identity is publicly known. One agent knows the identity of the other agent (the one he wants to send a message to), but not viceversa.

Here are the contents of the exchanged messages:

$\mathcal{A} \rightarrow \mathcal{B}$ the identity of \mathcal{A} , together with a fresh nonce $N_{\mathcal{A}}$, is sent to \mathcal{B} ;

$\mathcal{B} \rightarrow \mathcal{S}$ the identity of \mathcal{B} and a triplet (made of a fresh nonce $N_{\mathcal{B}}$, the nonce $N_{\mathcal{A}}$ and the identity of \mathcal{A}) encrypted under the key K_{BS} is sent to \mathcal{S} ;

$\mathcal{S} \rightarrow \mathcal{A}$ the fresh key K_{AB} , the identity of \mathcal{B} and the nonces $N_{\mathcal{A}}$ and $N_{\mathcal{B}}$, all encrypted under the key K_{AS} , along with the couple made by the identity of \mathcal{A} and the key K_{AB} , encrypted under the key K_{BS} , are sent to \mathcal{A} ;

$\mathcal{A} \rightarrow \mathcal{B}$ the couple made by the identity of \mathcal{A} and the key K_{AB} , encrypted under the key K_{BS} , and the nonce $N_{\mathcal{B}}$, encrypted under the key K_{AB} , are sent to \mathcal{B} .

Adding Destructors to Break the Protocol

When analysing the protocol, ProVerif proves that it is safe in the classical Dolev-Yao model.

Let us now weaken the model by providing the attacker with some useful destructors, that will enable him to break the protocol: ProVerif will find that the protocol is not safe anymore.

The challenge for the adversary is to be able to decrypt the first message $M_{K_{AB}}$, which is sent encrypted under the fresh key K_{AB} .

A Destructor to Guess the Key As a first option we can imagine that the attacker can guess the key K used to encrypt a message with probability p_G . If no other knowledge about the key is available, we have that $p_G = 2^{-\ell_K}$, where ℓ_K is the length of the key K .

The attack is trivial:

- the attacker can decrypt the message $M_{K_{AB}}$ if he knows K_{AB} ;
- the key K_{AB} can be guessed with probability p_G .

ProVerif gives the following attack trace instead:

- the attacker can decrypt the message $M_{K_{AB}}$ if he knows K_{AB} ;
- the key K_{AB} is sent from \mathcal{S} to \mathcal{A} in a message which is encrypted under the key K_{AS} ;
- the key K_{AS} can be guessed with probability p_G .

In both cases the probability of this attack to succeed is therefore p_G , as the new destructor is used only once.

A Destructor to Decrypt a Message without Knowing the Key As another option we can imagine that the attacker can decrypt an encrypted message M_K without knowing the key K with probability p_D . If no other knowledge about the contents M of the message is available, we have that $p_D = 2^{-\ell_M}$, where ℓ_M is the length of the plaintext of the message.

The attack is trivial and is the one found by ProVerif:

- the attacker can decrypt the message $M_{K_{AB}}$ with probability p_D .

The probability of this attack to succeed is therefore p_D , as also in this case the new destructor is used only once.

A Destructor to Spoof the Server Identity Yet another possible scenario is the one when the attacker can successfully pretend to be the server: this means that he is able to recreate the server authentication credentials. The probability of being able to do this is p_S . If no other knowledge about the private key P used to generate the server credentials is available, we have that $p_S = 2^{-\ell_P}$, where ℓ_P is the length of the key P .

ProVerif gives the following attack trace:

- the attacker breaks the server credentials with probability p_S ;
- the attacker can successfully mount a *Man-in-the-Middle* attack.

The probability of this attack to succeed is therefore p_S , as again the new destructor is used only once.

Considerations on the Attack Traces

The attack traces found by ProVerif show that the Yahalom protocol is not fully safe anymore if we weaken the perfect encryption hypothesis, but it can be broken with probability

$$p = \max\{p_G, p_D, p_S\}$$

If the attacker can use all of the three destructors described above, instead of only one of them at once, ProVerif will output a trace which is similar to one of the attacks described above (actually the one using the message decryption without knowing the key), as in each one of them the new destructor is used only once, so there can be no more effective combination of using the destructors.

It must be noted that the trace given by ProVerif in this case may not correspond to the trace of the most successful attack — ProVerif knows nothing about probabilities, as this is something that we add *a posteriori*.

It must also be noted that the traces given by ProVerif are not always the most trivial ones.

D.3.4 Protocol runs as predicates

We intend to use the framework presented in §3 to reason about protocols, and we aim at doing so by writing predicates that account for the evolution of an attacker's knowledge, when interacting with the protocol.

The attacker's knowledge can be described by a state σ , where the observation variables are boolean variables: the variable v_t assumes the boolean value 1 when the attacker knows the term t , and vice-versa it assumes the value 0 when the attacker does not know t ; the state space \mathcal{K} is formed by all such states accounting for all possible attacker's knowledge patterns.

The inference rules that describe the possible steps to be taken in a run of the protocol determine the possible assignments that transform a state σ into a σ' , to account for the knowledge gathered by the attacker after that step was taken: the only assignments we will have to deal with are those assigning the boolean value 1 to v_t upon discovery of term t by the attacker.

An assignment can account for the application of a single rule, as well as for the application of a concatenation of inference rules.

In a setting such as that one of ProVerif, we deal with two kinds of predicates:

- $\text{message}(c,m)$ — the message m is sent on channel c ;
- $\text{attacker}(t)$ — the attacker may have the term t .

We aim at keeping track of what the attacker knows, without really caring of the messages he has had to exchange to gather that knowledge: when we have subsequent applications of rules involving the predicate message (in the antecedent or in the subsequent), we can work them out to infer a rule involving only the predicate attacker , that is equivalent to the original rules for what concerns the attacker's knowledge.

This means that we will be working with predicates of the following shape:

$$\bigwedge_{i=0}^N \text{attacker}(t_i) \Rightarrow \text{attacker}(t_{(N+1)})$$

Here is an example of how easily we can “purge” the protocol rules from all message predicates (this means that we are not focusing on the message flow but only on the evolution of the attacker’s knowledge).

Let us take the following set of rules:

- 1 : $\text{attacker}(c)$
- 2 : $\text{attacker}(c) \Rightarrow \text{message}(c, r)$
- 3 : $\text{message}(c, r) \Rightarrow \text{message}(c, m)$
- 4 : $\text{attacker}(c) \wedge \text{message}(c, m) \Rightarrow \text{attacker}(m)$

Using rules (1), (2), (3) and (4) we can derive $\text{attacker}(m)$ from the axiom $\text{attacker}(c)$. We can do this also if the set is reduced to:

- 5 : $\text{attacker}(c)$
- 6 : $\text{attacker}(c) \Rightarrow \text{attacker}(m)$

We can see that rule (6) can be derived from rules (2), (3) and (4).

In this way we have a set of rules giving an equivalent protocol model limited to the aspects concerning the attacker’s knowledge pool.

Finally we are ready to add probability to the picture: we can do so by taking advantage of our framework and monitor the possible lines of evolution of the attacker’s knowledge by using a probability distribution on the state space \mathcal{K} of all possible knowledge patterns.

The initial distribution δ_0 evolves as the protocol runs; the evolution of δ_0 has the interesting property that any state can evolve only towards a state where the attacker’s knowledge has increased, *i.e.* the attacker does not forget acquired knowledge; consequently state probability is “being transferred” as the protocol runs, from states where the attacker knows some terms to states accounting for a wider knowledge pool.

If we are given an attack trace, this formalism allows us to keep track of the evolution of the intruder’s knowledge as the attack is being carried out.

In the case of a non-probabilistic i -th step, we have that it is responsible for the assignment of the expression \underline{e}_i to \underline{v} , causing the distribution δ to become the distribution $\delta' = \delta \{\underline{e}_i / \underline{v}\}$.

For example if such a step is the inference of the term \hat{t} by application of the following rule:

$$\bigwedge_{t \in \text{knowledge}} \text{attacker}(t) \Rightarrow \text{attacker}(\hat{t})$$

the assignment will be:

$$\delta' = \delta\{1/v_t\}$$

We can generalise this to the case of a probabilistic j -th step, say with N possible outcomes. In that case we have that the expression describing the distribution δ' is the linear combination of the different outcomes of the distribution δ undergoing the N different assignments $\underline{v} := \underline{e}_{jk}$:

$$\delta' = \sum_{k=1}^N p_k \cdot \delta\{e_{jk}/\underline{v}\}$$

where we have that $\sum_{k=1}^N p_k \leq 1$.

What we obtain by sequentially composing all of the different steps is a predicate that describes the relation between the distribution δ of the attacker's initial knowledge and the distribution δ' of the knowledge he can gain, which has the following shape:

$$\delta' = \sum p \cdot \delta\{e/\underline{v}\}\{f/\underline{v}\}\dots\{g/\underline{v}\}$$

For an attack trace \mathcal{T} — which we assume to be deterministic — and an initial distribution δ we note the final distribution δ' as⁴:

$$\mathcal{T}(\delta) \triangleq \delta'$$

We are particularly interested in the case where an attacker starts with zero knowledge, and so when $\delta = \eta_0$ — we are using η_0 to note the distribution where there is a single state mapping to probability 1, *i.e.* that having all variables v_t mapping to 0, whereas all other states have probability 0.

Finally if we evaluate the probability of the attacker reaching his goal G , which is a boolean expression that assumes the value *true* on the states where the attacker has enough knowledge to violate the protocol, on the final distribution $\mathcal{T}(\delta)$, we have also the probability of the corresponding attack to be performed successfully:

$$p \triangleq \mathcal{T}(\delta)(G)$$

The same considerations on the probability of a successful attack from §D.3.2 apply to this case, with the opportune generalisations.

D.3.5 An example: key-guessing on the Yahalom protocol

As in §D.3.3 we take the Yahalom protocol as an example to illustrate the proposed methodology, and we will focus in particular on the case when the attacker tries to guess a key — the other two cases from the example in [BB09] (the attacker spoofing the server's identity or being able to decrypt a message without knowing the key) are other examples of similar complexity.

For the protocol description one should go back to §D.3.3.

⁴If we wanted to be more formal, this should be the program image for \mathcal{T} , which is a singleton set because of the deterministic nature of \mathcal{T} .

Adding a destructor to account for key-guessing

The challenge for the adversary is to be able to decrypt the first message $M_{K_{AB}}$, which is sent encrypted under the fresh key K_{AB} : let v_m be the boolean variable accounting for the attacker knowing the plaintext M , the security goal to be broken is expressed by the condition $v_m = 1$ (all states satisfying this condition are states where the attacker has successfully compromised the protocol).

One of the possible destructors we can add to the model is that accounting for the attacker guessing the key K used to encrypt a message with probability p_G . If no other knowledge about the key is available, we have that $p_G = 2^{-\ell_K}$, where ℓ_K is the length of the key K .

The attack is trivial:

- the attacker can decrypt the message $M_{K_{AB}}$ if he knows K_{AB} ;
- the key K_{AB} can be guessed with probability p_G .

Let v_{kab} and v_{mkab} be the boolean variables accounting for the attacker knowing the terms K_{AB} and $M_{K_{AB}}$ respectively, the attack trace seen as a predicate can be written as (when the attacker starts with zero knowledge):

$$\begin{aligned} \mathcal{T}_G(\eta_0) = & p_G \cdot \eta_0 \{1/v_{kab}\} \{1/v_{mkab}\} \{1/v_m\} + \\ & + (1 - p_G) \cdot \eta_0 \{1/v_{mkab}\} \end{aligned}$$

The first term accounts for the attacker acquiring the term $M_{K_{AB}}$ (which flows on a public channel), then correctly guessing K_{AB} and thus being able to obtain M (breaching the security goal), whereas the second term describes the case when the attacker acquires $M_{K_{AB}}$ but cannot guess the right key.

The probability of this attack succeeding is therefore:

$$p = \mathcal{T}_G(\eta_0)(v_m = 1) = p_G$$

The attack trace found by ProVerif is slightly different:

- the attacker can decrypt the message $M_{K_{AB}}$ if he knows K_{AB} ;
- the key K_{AS} is sent from S to A in a message which is encrypted under the key K_{AS} — let us note it as Z ;
- the key K_{AS} can be guessed with probability p_G .

Let v_z and v_{kas} be the boolean variable accounting for the attacker knowing the terms Z and K_{AS} , similarly as above we have that the predicate for the attack trace can be written as:

$$\begin{aligned} \mathcal{T}_G(\eta_0) = & p_G \cdot \eta_0 \{1/v_{mkab}\} \{1/v_z\} \{1/v_{kas}\} \{1/v_{kab}\} \{1/v_m\} + \\ & + (1 - p_G) \cdot \eta_0 \{1/v_{mkab}\} \{1/v_z\} \end{aligned}$$

Therefore in both cases the probability of an attack based on guessing is p_G .

D.3.6 Towards a UTP-style protocol verification technique

Now that we have defined how to turn attack traces into predicates, it is interesting to address the concept of refinement: this is a key concept in the UTP framework, and in §3.7 we have

introduced this notion in our framework as well.

In the case of attack traces we are interested in a more specific definition, which takes into account a specific security goal G to be broken, so a reasonable informal definition could be that an attack trace \mathcal{T}_A refines another attack trace \mathcal{T}_B (with respect to G) whenever the success probability is greater when mounting the attack A than with attack B .

If we want to capture this definition formally, we can write:

$$\mathcal{T}_A \sqsupseteq_G \mathcal{T}_B \quad \triangleq \quad \forall \delta \bullet \mathcal{T}_A(\delta)\langle G \rangle \geq \mathcal{T}_B(\delta)\langle G \rangle$$

This relation induces a partial order over the set of possible traces. We can observe that a safe protocol run is an attack trace with success probability 0, *i.e.* interacting with the protocol in this way does not compromise the security goal G .

All those traces yielding a non-zero success probability are attack traces, and they can be ranked by effectiveness by the refinement relation: here we are dealing with a lattice, where the bottom elements are the safest traces and the top elements are those where the attacker can violate the protocol with the highest probability.

At the very bottom of the lattice we find the empty trace (*i.e.* the safest trace), which is the trace of a protocol which has not started; the exploration of the lattice is going to provide useful information to determine the most successful attack.

APPENDIX E

Notation

E.1 Logic

\neg : logical negation
 \wedge : logical conjunction
 \vee : logical disjunction
 \Rightarrow : implication
 \Leftrightarrow : double implication
true : logical true
false : logical false

E.2 Relations and functions

\mapsto : maps to
 \dagger : override
 \rightarrow : total function
 \rightarrowtail : partial function
 \mathcal{R} : relation
dom : domain operator
codom : codomain operator
img : image operator

E.3 Probability

p, q, r, s : probability
 P, Q : stochastic variable
 f_P : probability density function
 F_P : cumulative density function

E.4 Variables, values and expressions

$:=$: assignment

v : variable

w : value

e, f, g : expression

c, d, z : boolean expression

\underline{v} : vector of variables

\underline{w} : vector of values

$\underline{e}, \underline{f}, \underline{g}$: vector of expressions

\mathcal{V} : set of variables

\mathcal{W} : set of values

\mathcal{E} : set of expressions

eval : expression evaluation operator

type : variable type operator

fv : free variable operator

bv : bound variable operator

E.5 States and distributions

σ, ζ : state
 α : abstract state
 \mathcal{S} : set of all states (state space)
 \mathcal{A} : alphabet
 alph : alphabet operator
 χ, ξ : distribution
 ϵ : empty distribution
 ι : unitary distribution
 π : weighting distribution
 $\bar{\pi}$: complementary weighting distribution
 δ : probability distribution
 \mathcal{X}, \mathcal{Y} : set of distributions
 \mathcal{D} : set of all distributions
 \mathcal{D}_w : set of all weighting distributions
 \mathcal{D}_p : set of all probability distributions
 $\|_-$: weight
 $_{-}\langle _ \rangle$: restriction
 $Inv(_, _)$: inverse-image set
 $_{-}\{ / - \}$: remap

E.6 Programs

$skip$: skip
 $abort$: abort
 $miracle$: miracle
 $*$: iteration
 $_{-}\triangleleft _ \triangleright _$: conditional choice
 \sqcup : angelic choice
 \sqcap : demonic choice
 $p^{\oplus}, p^{\oplus}_{(1-p)}$: probabilistic choice
 $choice(_, _, _)$: choice
 \sqsubseteq : refinement
 $(_)^\Delta$: refinement set

APPENDIX F

Mathematical Background

This appendix revisits and expands the presentation given in [Koz81], in order to provide a very concise reference for the mathematical foundations of this work, assuming a basic knowledge of topology; we try to keep the notation as similar as possible to the one used throughout the present work.

F.1 General Notions

The *characteristic function* of a set $B \subseteq A$ is the function $\chi_B : A \rightarrow \{0, 1\}$ defined as follows:

$$\chi_B(\mathbf{a}) \triangleq \begin{cases} 1 & \text{if } \mathbf{a} \in B \\ 0 & \text{if } \mathbf{a} \notin B. \end{cases}$$

A *binary operation* op on a set A is a total function $op : A \times A \rightarrow A$: it should be noted that the totality of the function implies the *closure* property of the set A with respect to op .

F.2 Vector spaces

A *commutative group* (or *abelian group*) is a pair $(A, +)$, where A is a set with a unique distinguished element 0 and $+$ is a binary operator, such that for any elements $\mathbf{a}, \mathbf{b}, \mathbf{c} \in A$:

$$\begin{aligned} \mathbf{a} + \mathbf{b} &= \mathbf{b} + \mathbf{a} && \text{[commutativity]} \\ (\mathbf{a} + \mathbf{b}) + \mathbf{c} &= \mathbf{a} + (\mathbf{b} + \mathbf{c}) && \text{[associativity]} \\ \mathbf{a} + 0 &= \mathbf{a} && \text{[identity]} \\ \mathbf{a} + (-\mathbf{a}) &= 0 && \text{[inverse]}, \end{aligned}$$

where $-\mathbf{a}$ denotes the element of A called the *additive inverse* of \mathbf{a} (unique for each \mathbf{a}).

A *vector space* $(\underline{V}, +, \cdot)$ over a field F is a set \underline{V} , equipped with two binary operators $+$ and \cdot , such that $(\underline{V}, +)$ is a commutative group, and for which the following axioms hold for a unique distinguished element $1 \in F$, any elements $\mathbf{a}, \mathbf{b} \in F$ and $\underline{t}, \underline{u}, \underline{v} \in \underline{V}$:

$$\begin{aligned} \mathbf{a}(\underline{u} + \underline{v}) &= \mathbf{a}\underline{u} + \mathbf{a}\underline{v} && \text{[distributivity]} \\ (\mathbf{a} + \mathbf{b})\underline{v} &= \mathbf{a}\underline{v} + \mathbf{b}\underline{v} && \text{[distributivity]} \\ \mathbf{a}(\mathbf{b}\underline{v}) &= (\mathbf{a}\mathbf{b})\underline{v} && \text{[associativity]} \\ 1\underline{v} &= \underline{v} && \text{[identity]}. \end{aligned}$$

The elements of \underline{V} are called *vectors*, whereas the elements of F are scalars.

An example of this is the vector space where $\underline{V} = \mathbb{R}^N$, where the vectors are columns of N real numbers, the scalars are real numbers and the binary operators $+$ and \cdot correspond respectively to the element-wise sum and multiplication of each element by a scalar.

A *positive cone* \underline{V}^+ is a subset of \underline{V} such that:

$$\begin{aligned} \underline{u}, \underline{v} \in \underline{V}^+ \wedge a, b \in F^+ &\Rightarrow a\underline{u} + b\underline{v} \in \underline{V}^+ \\ \underline{v} \in \underline{V}^+ \wedge -\underline{v} \in \underline{V}^+ &\Leftrightarrow \underline{v} = 0, \end{aligned}$$

where F^+ denotes the set of all non-negative elements of the field F .

A *distance* on \underline{V} is a function $d : \underline{V} \times \underline{V} \rightarrow \mathbb{R}^+$ such that:

$$\begin{aligned} d(\underline{v}, \underline{v}) &= 0 \\ d(\underline{u}, \underline{v}) &= d(\underline{v}, \underline{u}) \\ d(\underline{u}, \underline{v}) &\leq d(\underline{u}, \underline{t}) + d(\underline{t}, \underline{v}). \end{aligned}$$

A *metric space* is a pair (\underline{V}, d) , where d is a distance on \underline{V} .

A sequence $\underline{v}_1, \underline{v}_2, \dots$ is a *Cauchy sequence* if for every positive real number ϵ there is a positive integer n such that:

$$\forall p, q > n \bullet d(\underline{v}_p, \underline{v}_q) \leq \epsilon.$$

A metric space is *complete* if every Cauchy sequence has a limit in \underline{V} .

A *norm* on a vector space \underline{V} is a function $\|\cdot\| : \underline{V} \rightarrow \mathbb{R}^+$ such that:

$$\begin{aligned} \|\underline{v}\| = 0 &\Leftrightarrow \underline{v} = 0 \\ \forall a \in F \bullet \|a\underline{v}\| &= |a| \cdot \|\underline{v}\| \\ \|\underline{u} + \underline{v}\| &\leq \|\underline{u}\| + \|\underline{v}\|. \end{aligned}$$

A norm induces a metric on \underline{V} , as it is possible to define a distance function as:

$$d(\underline{u}, \underline{v}) \triangleq \|\underline{u} - \underline{v}\|.$$

If the metric space $(\underline{V}, \|\cdot\|)$ is complete, then it is a *Banach space*.

It is possible to define a *partial order* \leq on \underline{V} by using its positive cone \underline{V}^+ :

$$\underline{u} \leq \underline{v} \triangleq (\underline{v} - \underline{u}) \in \underline{V}^+$$

Addition and scalar multiplication enjoy the following properties:

$$\begin{aligned} \underline{v} + \sup_{\underline{u} \in \underline{U}}(\underline{u}) &= \sup_{\underline{u} \in \underline{U}}(\underline{v} + \underline{u}) \\ \sup_{\underline{u} \in \underline{U}}(a\underline{u}) &= a \sup_{\underline{u} \in \underline{U}}(\underline{u}). \end{aligned}$$

where $\underline{U} \subseteq \underline{V}$ and $a \in F^+$, and therefore we say that they are *order-continuous* with respect to the order \leq .

A set $\underline{U} \subseteq \underline{V}$ is a *directed set* if there is a \leq -upper bound in \underline{U} for each pair $\underline{u}, \underline{v} \in \underline{U}$.

An *interval* is a set $[\underline{u}, \underline{v}] \triangleq \{\underline{t} \mid \underline{u} \leq \underline{t} \leq \underline{v}\}$; if a set is contained in an interval, then it is said to be *order-bounded*.

A *vector lattice* is a pair (\underline{V}, \leq) , where every pair $\underline{u}, \underline{v} \in \underline{U}$ has a \leq -least upper bound (or *join*) $\underline{u} \sqcup \underline{v}$, or equivalently every pair $\underline{u}, \underline{v} \in \underline{U}$ has a \leq -greatest lower bound (or *meet*) $\underline{u} \sqcap \underline{v}$.

A vector lattice is conditionally complete if every set of elements of \underline{V} with a \leq -upper bound has a least upper bound.

Both addition and scalar multiplication distribute over \sqcap and \sqcup ; moreover $\underline{u} + \underline{v} = \underline{u} \sqcap \underline{v} + \underline{u} \sqcup \underline{v}$.

The *Jordan decomposition* of a vector \underline{v} returns two unique positive vectors \underline{v}^+ and \underline{v}^- , whose meet is 0 and such that $\underline{v} = \underline{v}^+ - \underline{v}^-$. We have that:

$$\begin{aligned}\underline{v}^+ &= \underline{v} \sqcap 0 \\ \underline{v}^- &= -\underline{v} \sqcap 0 \\ \underline{u} \sqcap \underline{v} &= (\underline{u} - \underline{v})^+ + \underline{v} \\ \underline{u} \sqcup \underline{v} &= -(-\underline{u} \sqcap -\underline{v}).\end{aligned}$$

The *absolute value* of a vector \underline{v} is $|\underline{v}| \triangleq \underline{v}^+ + \underline{v}^- = \underline{v}^+ \sqcap \underline{v}^-$. We have that:

$$\begin{aligned}|\underline{v}| &\geq 0 \\ |\underline{v}| = 0 &\Leftrightarrow \underline{v} = 0 \\ |\underline{u} - \underline{v}| &= (\underline{u} \sqcap \underline{v}) - (\underline{u} \sqcup \underline{v}) \\ \underline{u} \sqcap \underline{v} &= 1/2(\underline{u} + \underline{v} + |\underline{u} - \underline{v}|).\end{aligned}$$

If $(\underline{V}, \|\cdot\|, \leq)$ is a *Banach lattice* if it is both a Banach space and a vector lattice and the following holds:

$$\begin{aligned}\|\underline{v}\| &= \|\underline{v}^+\| \\ 0 \leq \underline{u} \leq \underline{v} &\Rightarrow \|\underline{u}\| \leq \|\underline{v}\|\end{aligned}$$

Given a linear transformation $T : \underline{U} \rightarrow \underline{V}$, where \underline{U} and \underline{V} are two normed vector spaces, we say that T is *$\|\cdot\|$ -bounded* if:

$$\sup_{\underline{u} \in \mathcal{B}_1[0]} \|T(\underline{u})\| < +\infty,$$

where $\mathcal{B}_1[0] = \{\underline{u} \mid \|\underline{u}\| \leq 1\}$ is the (closed) ball of radius 1; the property of being $\|\cdot\|$ -bounded is equivalent to that of being continuous with respect to the metric induced by $\|\cdot\|$.

The space of all such $\|\cdot\|$ -bounded linear transformations is a normed vector space under point-wise addition and scalar multiplication, with the *uniform norm* defined as:

$$\|T\| = \sup_{\underline{u} \in \mathcal{B}_1(0)} \|T(\underline{u})\|.$$

We say that T is *monotone* if

$$\underline{u} \leq \underline{v} \Leftrightarrow T(\underline{u}) \leq T(\underline{v}).$$

We say that T is *order-bounded* if it maps order-bounded sets to order-bounded sets.

The set of order-bounded linear transformations from a vector lattice to a conditionally complete vector lattice is itself a vector lattice, where:

$$(S \sqcap T)(\underline{v}) = \sup_{0 \leq \underline{u} \leq \underline{v}} S(\underline{u}) + T(\underline{v} - \underline{u}).$$

A *linear operator* on \underline{V} is an endomorphism $op : \underline{V} \rightarrow \underline{V}$ which is also $\|\cdot\|$ -bounded linear transformation; the space of all linear operators on \underline{V} is a Banach space if \underline{V} is a Banach space.

F.2.1 The vector space \mathbb{R}^N

The elements of the vector space \mathbb{R}^N are vectors made of N real numbers, *viz.* linear combinations of the canonical generators e_i (which are null vectors with a 1 in the i -th position) with coefficients in \mathbb{R} .

The *total variation norm* (also known as *taxicab norm* or *Manhattan norm*) in \mathbb{R}^N is the norm defined as:

$$\|\underline{v}\|_1 \triangleq \sum_{i=1}^N |v_i|.$$

The *uniform norm* (also known as *supremum norm* or *Chebyshev norm*) in \mathbb{R}^N is the norm defined as:

$$\|\underline{v}\|_\infty \triangleq \max_{i \in [1..N]} |v_i|.$$

F.2.2 The vector space $\mathbb{R}^{N \times N}$

The elements of the vector space $\mathbb{R}^{N \times N}$ are matrices made with N elements from \mathbb{R}^N .

The *total variation norm* in $\mathbb{R}^{N \times N}$ is the norm defined as:

$$\|\underline{A}\|_1 \triangleq \max_{j \in [1..N]} \|\underline{a}_{*j}\|_1.$$

The *uniform norm* in $\mathbb{R}^{N \times N}$ is the norm defined as:

$$\|\underline{A}\|_\infty \triangleq \max_{i \in [1..N]} \|\underline{a}_{i*}\|_1.$$

F.3 Boolean algebra

A *boolean algebra* is a structure made of a set A , with two unique distinguished elements 0 and 1, equipped with two binary operators \wedge and \vee and the unary operator \neg , and for which the

following axioms hold for any elements $a, b, c \in A$:

$$\begin{array}{lll}
 a \vee (b \vee c) = (a \vee b) \vee c & a \wedge (b \wedge c) = (a \wedge b) \wedge c & \text{[associativity]} \\
 a \vee b = b \vee a & a \wedge b = b \wedge a & \text{[commutativity]} \\
 a \vee (a \wedge b) = a & a \wedge (a \vee b) = a & \text{[absorption]} \\
 a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c) & a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) & \text{[distributivity]} \\
 a \vee \neg a = 1 & a \wedge \neg a = 0 & \text{[complement].}
 \end{array}$$

An example of this is the *two-element boolean algebra*, where $A = \{0, 1\}$ and \wedge, \vee, \neg are respectively the logical conjunction, disjunction, negation.

Another example is the case where $A = \wp S$ for any set S , 0 and 1 are respectively the empty set and S itself, and \wedge, \vee, \neg are respectively the set intersection, union, complement.

A σ -algebra is a boolean algebra which is closed with respect to countable union.

F.4 Measure Theory

A *measurable space* is a pair $(\mathcal{S}, \Sigma_{\mathcal{S}})$, where \mathcal{S} is a set and $\Sigma_{\mathcal{S}} \subseteq \wp \mathcal{S}$ is a σ -algebra of subsets of \mathcal{S} .

A function $f : (\mathcal{S}, \Sigma_{\mathcal{S}}) \rightarrow (\mathcal{W}, \Sigma_{\mathcal{W}})$ is *measurable* if

$$\forall \beta \bullet \alpha = f^{-1}(\beta) \wedge \alpha \in \Sigma_{\mathcal{S}} \Leftrightarrow \beta \in \Sigma_{\mathcal{W}}.$$

A function is *countably additive* if, given finitely many pairwise disjoint sets $\alpha_1, \alpha_2, \dots, \alpha_N \in \Sigma_{\mathcal{S}}$, the following relation holds:

$$\mu\left(\bigcup_{i=1}^N \alpha_i\right) = \sum_{i=1}^N \mu(\alpha_i).$$

A *measure* (or *distribution*) on $(\mathcal{S}, \Sigma_{\mathcal{S}})$ is a countably additive function $\mu : \Sigma_{\mathcal{S}} \rightarrow \mathbb{R}$; a measure is said to be *positive* if $\mu(\alpha) \geq 0$ for every $\alpha \in \Sigma_{\mathcal{S}}$.

A measure is *discrete* if all its weight is distributed on countably many elements; a special case is when all the weight is distributed on a single element, which is called *point mass*; a measure is *continuous* if $\mu(\alpha) = 0$ for all countable α .

Every measure can be uniquely represented as the sum of a continuous measure and a discrete one.

The *Jordan decomposition* of a measure μ returns two unique positive measures μ^+ (*positive variation*) and μ^- (*negative variation*) such that:

$$\mu^+ = \mu(\alpha) \quad \wedge \quad \mu^- = -\mu(\bar{\alpha}),$$

for some $\alpha \in \Sigma_{\mathcal{S}}$, where $\bar{\alpha} = \mathcal{S} \setminus \alpha$.

The *total variation* of μ is defined as the measure $|\mu| = \mu^+ + \mu^-$.

The *total variation norm* here can be defined as $\|\mu\| \triangleq |\mu|$: the set of measures on $(\mathcal{S}, \Sigma_{\mathcal{S}})$, together with this norm and with addition, scalar multiplication and \leq relation all lifted point-wise, forms a Banach lattice.

A *measure space* $(\mathcal{S}, \Sigma_{\mathcal{S}}, \mu)$ is a measurable space equipped with a measure.

F.5 Probability Theory

A subprobability measure is a positive measure such that $\mu(\mathcal{S}) \leq 1$; in case $\mu(\mathcal{S}) = 1$ we talk of a *probability measure*.

A *probability space* is a measure space $(\mathcal{S}, \Sigma_{\mathcal{S}}, \mu)$ where μ is a probability measure: \mathcal{S} can be thought as the set of all possible *outcomes* σ , whereas $\Sigma_{\mathcal{S}}$ is the set of all possible *events* α .

A *random variable* is a partial measurable function $\nu : (\mathcal{S}, \Sigma_{\mathcal{S}}, \mu) \rightarrow (\mathcal{W}, \Sigma_{\mathcal{W}})$; its domain is a probability space and is usually referred to as the *sample space*, whereas its range is referred to as the *value space*.

A random variable induces a subprobability measure $\mu \circ \nu^{-1}$ on $(\mathcal{W}, \Sigma_{\mathcal{W}})$ (this is a probability measure in case ν is a total function):

$$\mu \circ \nu^{-1}(\alpha) = \mu(\nu^{-1}(\alpha)).$$

The probability of an event $\alpha \in \Sigma_{\mathcal{S}}$ is therefore:

$$\mathcal{P}(\sigma \in \alpha) \triangleq \mu \circ \nu^{-1}(\alpha).$$

A *random variate* is a particular outcome of a random variable.

A *random vector* is a random variable $\underline{\nu}$ where the value space is the cartesian product $(\mathcal{W}, \Sigma_{\mathcal{W}}) = \prod(\mathcal{W}_i, \Sigma_{\mathcal{W}_i})$: $\underline{\nu}$ is a list of the random variables ν_1, ν_2, \dots and induces a subprobability measure $\mu \circ \underline{\nu}^{-1}$ on $(\mathcal{W}, \Sigma_{\mathcal{W}})$, referred to as the *joint distribution* of ν_1, ν_2, \dots .

Two random variables ν_i and ν_j are *independent* if their joint distribution is $(\mu \circ \nu_i^{-1}) \cdot (\mu \circ \nu_j^{-1})$.

References

- [AB05] Martín Abadi and Bruno Blanchet. “Analyzing Security Protocols with Secrecy Types and Logic Programs”. In: *Journal of the ACM* 52.1 (Jan. 2005), pp. 102–146.
- [ABCL09] Martín Abadi, Bruno Blanchet, and Hubert Comon-Lundh. “Models and Proofs of Protocol Security: A Progress Report”. In: *21st International Conference on Computer Aided Verification (CAV’09)*. Lecture Notes on Computer Science. Grenoble, France: Springer Verlag, July 2009.
- [Adã⁺06] Pedro Adão et al. “Towards a Quantitative Analysis of Security Protocols”. In: *Electr. Notes Theor. Comput. Sci.* 164.3 (2006), pp. 3–25.
- [AF01] Martín Abadi and Cédric Fournet. “Mobile values, new names, and secure communication”. In: *POPL ’01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. London, United Kingdom: ACM, 2001, pp. 104–115.
- [AG97] Martín Abadi and Andrew D. Gordon. “A Calculus for Cryptographic Protocols: The Spi Calculus”. In: *Fourth ACM Conference on Computer and Communications Security*. ACM Press, 1997, pp. 36–47.
- [AR02] Martín Abadi and Phillip Rogaway. “Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)”. In: *J. Cryptology* 15.2 (2002), pp. 103–127.
- [Bau06] Mathieu Baudet. “Random Polynomial-Time Attacks and Dolev-Yao Models”. In: *Journal of Automata, Languages and Combinatorics* 11.1 (2006), pp. 7–21.
- [BB09] Riccardo Bresciani and Andrew Butterfield. “Weakening the Dolev-Yao model through probability”. In: *SIN ’09: Proceedings of the 2nd international conference on Security of information and networks*. North Cyprus: ACM, 2009, pp. 293–297.
- [BB11] Riccardo Bresciani and Andrew Butterfield. *Towards a UTP-style framework to deal with probabilities*. Tech. rep. TCD-CS-2011-09. FMG, Trinity College Dublin, Ireland, Aug. 2011.
- [BB12a] Riccardo Bresciani and Andrew Butterfield. “A probabilistic theory of designs based on distributions”. In: *UTP 2012*. 2012.
- [BB12b] Riccardo Bresciani and Andrew Butterfield. “A UTP approach towards probabilistic protocol verification”. In: *Security and Communication Networks* (2012).
- [BB12c] Riccardo Bresciani and Andrew Butterfield. “A UTP semantics of pGCL as a homogeneous relation”. In: *iFM 2012*. 2012.
- [Bla01] Bruno Blanchet. “An efficient cryptographic protocol verifier based on Prolog rules”. In: *14th IEEE Computer Security Foundations Workshop*. 2001, pp. 86–100.

- [Bla08] Bruno Blanchet. “Vérification automatique de protocoles cryptographiques : modèle formel et modèle calculatoire”. En français avec publications en anglais en annexe. In French with publications in English in appendix. Mémoire d’habilitation à diriger des recherches. Université Paris-Dauphine, Nov. 2008.
- [BPB11] Riccardo Bresciani, Mario Poletti, and Andrew Butterfield. *Nilpotency of square matrices with non-negative elements*. Tech. rep. TCD-CS-2011-17. FMG, Trinity College Dublin, Ireland, Dec. 2011.
- [But10] Andrew Butterfield, ed. *Unifying Theories of Programming, Second International Symposium, UTP 2008, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers*. Vol. 5713. Lecture Notes in Computer Science. Springer, 2010.
- [CS09] Yifeng Chen and Jeff W. Sanders. “Unifying Probability with Nondeterminism”. In: *FM 2009, LNCS 5850*. 2009, pp. 467–482.
- [Dem68] A. P. Dempster. “A generalization of Bayesian inference”. In: *Journal of the Royal Statistical Society* 30.B (1968), pp. 205–247.
- [Dij75] Edsger W. Dijkstra. “Guarded commands, nondeterminacy and formal derivation of programs”. In: *Commun. ACM* 18.8 (1975), pp. 453–457.
- [Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [DK03] L. De Raedt and K. Kersting. “Probabilistic Logic Learning”. In: *ACM-SIGKDD Explorations: Special issue on Multi-Relational Data Mining* 5.1 (2003). Ed. by S. Džeroski and L. De Raedt, pp. 31–48.
- [DMV04] Paul Hankes Drielsma, Sebastian Mödersheim, and Luca Viganò. “A Formalization of Off-Line Guessing for Security Protocol Analysis”. In: *LPAR*. 2004, pp. 363–379.
- [DS06] Steve Dunne and Bill Stoddart, eds. *Unifying Theories of Programming, First International Symposium, UTP 2006, Walworth Castle, County Durham, UK, February 5-7, 2006, Revised Selected Papers*. Vol. 4010. Lecture Notes in Computer Science. Springer, 2006.
- [DY83] Danny Dolev and Andrew C. Yao. “On the security of public-key protocols”. In: *IEEE Transaction on Information Theory* 2.29 (Mar. 1983), pp. 198–208.
- [Fdr] *Failures-Divergence Refinement, FDR2 User Manual*. 9th. Accessed on April 19th, 2012. Formal Systems (Europe) Ltd. and Oxford University Computing Laboratory. 2010.
- [FHM90] Ronald Fagin, Joseph Y. Halpern, and Nimrod Megiddo. “A Logic for Reasoning about Probabilities”. In: *Information and Computation* 87 (1990), pp. 78–128.
- [FWB08] L. Freitas, J. Woodcock, and A. Butterfield. “POSIX and the Verification Grand Challenge: A Roadmap”. In: *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on* (2008), pp. 153–162.
- [GB09] Paweł Gancarski and Andrew Butterfield. “The Denotational Semantics of *slotted-Circus*”. In: *FM2009: Formal Methods*. Ed. by Ana Cavalcanti and Dennis Dams. Vol. 5850. LNCS. Springer, 2009, pp. 451–466.
- [Hae+01] R. Haenni et al. “A Survey on Probabilistic Argumentation”. In: *ECSQARU’01, Toulouse. Workshop: Adventures in Argumentation*. 2001, pp. 19–25.

- [Hae⁺08] Rolf Haenni et al. “Possible Semantics for a Common Framework of Probabilistic Logics”. In: *Interval / Probabilistic Uncertainty and Non-Classical Logics*. 2008, pp. 268–279.
- [He10] Jifeng He. “A Probabilistic BPEL-Like Language”. In: *UTP*. Ed. by Shengchao Qin. Vol. 6445. Lecture Notes in Computer Science. Springer, 2010, pp. 74–100.
- [Heh04] Eric C. R. Hehner. “Probabilistic Predicative Programming”. In: *MPC*. 2004, pp. 169–185.
- [Heh11] Eric C. R. Hehner. “A probability perspective”. In: *Formal Asp. Comput.* 23.4 (2011), pp. 391–419.
- [Heh84] Eric C. R. Hehner. “Predicative programming Part I & II”. In: *Commun. ACM* 27.2 (Feb. 1984), pp. 134–151.
- [HH98] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science, 1998.
- [HMM05] Joe Hurd, Annabelle McIver, and Carroll Morgan. “Probabilistic guarded commands mechanized in HOL”. In: *Theor. Comput. Sci* 346.1 (2005), pp. 96–112.
- [Hoa85a] C. A. R. Hoare. “Programs are predicates”. In: *Proceedings of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*. Upper Saddle River, NJ, USA: Prentice-Hall, 1985, pp. 141–155.
- [Hoa85b] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [HP07] Joseph Y. Halpern and Riccardo Pucella. “Characterizing and reasoning about probabilistic and non-probabilistic expectation”. In: *J. ACM* 54.3 (2007), p. 15.
- [HS06] Jifeng He and Jeff W. Sanders. “Unifying Probability”. In: *UTP 2006, LNCS 4010*. Ed. by Steve Dunne and Bill Stoddart. Vol. 4010. Lecture Notes in Computer Science. Springer, 2006, pp. 173–199.
- [HSM97] Jifeng He, K. Seidel, and A. McIver. “Probabilistic models for the guarded command language”. In: *Science of Computer Programming* 28.2-3 (1997). Formal Specifications: Foundations, Methods, Tools and Applications, pp. 171–192.
- [JKB07] Dominik Jain, Bernhard Kirchlechner, and Michael Beetz. “Extending Markov Logic to Model Probability Distributions in Relational Domains”. In: *KI '07: Proceedings of the 30th annual German conference on Advances in Artificial Intelligence*. Osnabrück, Germany: Springer-Verlag, 2007, pp. 129–143.
- [Jon90] C. Jones. “Probabilistic Non-determinism”. PhD Thesis — also published as Technical Report ECS-LFCS-90-105 or CST-63-90. University of Edinburgh, 1990.
- [JP89] C. Jones and Gordon D. Plotkin. “A Probabilistic Powerdomain of Evaluations”. In: *LICS*. 1989, pp. 186–195.
- [Jøs01] Audun Jøsang. “A logic for uncertain probabilities”. In: *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.* 9.3 (2001), pp. 279–311.
- [KJH08] R. Kohlas, J. Jonczyk, and R. Haenni. “A Trust Evaluation Method Based on Logic and Probability Theory”. In: *IFIPTM'08, 2nd Joint iTrust and PST Conferences on Privacy Trust Management and Security*. Ed. by Y. Karabulut et al. Vol. II. Trust Management. Trondheim, Norway, 2008, pp. 17–32.

- [Koh03] J. Kohlas. “Probabilistic Argumentation Systems: A New Way to Combine Logic With Probability”. In: *Journal of Applied Logic* 1.3-4 (2003), pp. 225–253.
- [Koz81] Dexter Kozen. “Semantics of Probabilistic Programs”. In: *J. Comput. Syst. Sci.* 22.3 (1981), pp. 328–350.
- [Koz85] Dexter Kozen. “A Probabilistic PDL”. In: *J. Comput. Syst. Sci.* 30.2 (1985), pp. 162–178.
- [Kra⁺95] Paul Krause et al. “A Logic of Argumentation for Reasoning under Uncertainty.” In: *Computational Intelligence* 11 (1995), pp. 113–131.
- [Low95] Gavin Lowe. “An attack on the Needham-Schroeder public-key authentication protocol”. In: *Inf. Process. Lett.* 56.3 (1995), pp. 131–133.
- [Low96] Gavin Lowe. “Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR”. In: *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*. London, UK: Springer-Verlag, 1996, pp. 147–166.
- [McI06] Annabelle McIver. “Quantitative Refinement and Model Checking for the Analysis of Probabilistic Systems”. In: *UTP 2006, LNCS 4010*. 2006, pp. 131–146.
- [MCM06] Annabelle McIver, E. Cohen, and Carroll Morgan. “Using Probabilistic Kleene Algebra for Protocol Verification”. In: *RelMiCS*. 2006, pp. 296–310.
- [Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.
- [Mis00] Michael W. Mislove. “Nondeterminism and Probabilistic Choice: Obeying the Laws”. In: *CONCUR 2000, LNCS 1877*. 2000, pp. 350–364.
- [Mit⁺01] John C. Mitchell et al. “A probabilistic polynomial-time calculus for analysis of cryptographic protocols”. In: *Electronic Notes in Theoretical Computer Science*. 2001.
- [MM02] Annabelle McIver and Carroll Morgan. “Games, Probability and the Quantitative μ -calculus $qM\mu$ ”. In: *LPAR*. 2002, pp. 292–310.
- [MM04] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. SpringerVerlag, 2004.
- [MM05] Annabelle McIver and Carroll Morgan. “Abstraction and refinement in probabilistic systems”. In: *SIGMETRICS Performance Evaluation Review* 32.4 (2005), pp. 41–47.
- [MM09] Annabelle McIver and Carroll C. Morgan. “*ums and Lovers*: Case Studies in Security, Compositionality and Refinement”. In: *FM 2009, LNCS 5850*. 2009, pp. 289–304.
- [MM97] Carroll Morgan and Annabelle McIver. *A Probabilistic Temporal Calculus Based on Expectations*. Tech. rep. PRG-TR-13-97. Oxford University Computing Laboratory, 1997.
- [MM98] A.K. McIver and Carroll Morgan. “Demonic, Angelic and Unbounded Probabilistic Choices in Sequential Programs”. In: *Acta Informatica* 37 (1998), p. 2001.
- [MMM09] Annabelle McIver, Larissa Meinicke, and Carroll Morgan. “Security, Probability and Nearly Fair Coins in the Cryptographers’ Café”. In: *FM 2009, LNCS 5850*. 2009, pp. 41–71.

- [MMS96] Carroll Morgan, Annabelle McIver, and Karen Seidel. “Probabilistic predicate transformers”. In: *ACM Transactions on Programming Languages and Systems* 18.3 (1996), pp. 325–353.
- [Mor04] Carroll Morgan. “Of Probabilistic Wp and SP-and Compositionality”. In: *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP*, in LNCS 3525. Springer, 2004, pp. 220–241.
- [Mor⁺95] Carrol Morgan et al. *Argument Duplication in Probabilistic CSP*. Tech. rep. PRG-TR-11-95. Oxford University, 1995.
- [Mor⁺96] Carroll Morgan et al. “Refinement-Oriented Probability for CSP”. In: *Formal Asp. Comput.* 8.6 (1996), pp. 617–647.
- [MV04] Pedro R. D’Argenio Miguel ViÑisquez Nicol is Wolovick. *Probabilistic Hoare-like Logics in Comparison*. Tech. rep. Universidad Nacional de C rdoba, 2004.
- [MW05] Annabelle McIver and Tjark Weber. “Towards Automated Proof Support for Probabilistic Distributed Systems”. In: *LPAR*. 2005, pp. 534–548.
- [NM10] Ukachukwu Ndukwu and Annabelle McIver. “An expectation transformer approach to predicate abstraction and data independence for probabilistic programs”. In: *CoRR* (2010).
- [NS09] Ukachukwu Ndukwu and J. W. Sanders. “Reasoning about a Distributed Probabilistic System”. In: *Fifteenth Computing: The Australasian Theory Symposium (CATS 2009)*. Ed. by Rod Downey and Prabhu Manyem. Vol. 94. CRPIT. Wellington, New Zealand: ACS, 2009, pp. 35–42.
- [OCW09] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. “A UTP semantics for Circus”. In: *Formal Asp. Comput* 21.1-2 (2009), pp. 3–32.
- [Pea88] Judea Pearl. *Probabilistic reasoning in intelligent systems*. San Francisco, CA: Morgan Kaufmann, 1988.
- [Pol05] Mario Poletti. *Distribuzioni*. Plus, 2005.
- [Qin10] Shengchao Qin, ed. *Unifying Theories of Programming - Third International Symposium, UTP 2010, Shanghai, China, November 15-16, 2010. Proceedings*. Vol. 6445. Lecture Notes in Computer Science. Springer, 2010.
- [SH03] Adnan Sherif and Jifeng He. “Towards a Time Model for Circus”. In: *Lecture Notes in Computer Science* 2495 (2003), pp. 613–624.
- [Sha76] Glenn Shafer. *A Mathematical Theory of Evidence*. Princeton, 1976.
- [SSL10] Jun Sun, Songzheng Song, and Yang Liu. “Model Checking Hierarchical Probabilistic Systems”. In: *ICFEM*. Ed. by Jin Song Dong and Huibiao Zhu. Vol. 6447. Lecture Notes in Computer Science. Springer, 2010, pp. 388–403.
- [SZ99] J. W. Sanders and P. Zuliani. “Quantum Programming”. In: *In Mathematics of Program Construction*. Springer-Verlag, 1999, pp. 80–99.
- [Tar55] Alfred Tarski. “A lattice-theoretical fixpoint theorem and its applications”. In: *Pacific J. Math.* 5.2 (1955), pp. 285–309.
- [Whe⁺11] Gregory Wheeler et al. *Probabilistic logic and probabilistic networks*. Springer, 2011.

-
- [Wll02] Jon Williamson. “Handbook of the Logic of Argument and Inference: the Turn Toward the Practical”. In: ed. by H. J. Ohlbach D. Gabbay R. Johnson and J. Woods. Elsevier, 2002. Chap. Probability Logic, pp. 397–424.
- [Yao77] Andrew Chi-Chih Yao. “Probabilistic Computations: Toward a Unified Measure of Complexity (Extended Abstract)”. In: *FOCS*. 1977.
- [Yin03] Mingsheng Ying. “Reasoning about probabilistic sequential programs in a probabilistic logic”. In: *Acta Inf.* 39.5 (2003), pp. 315–389.
- [ZD04] Roberto Zunino and Pierpaolo Degano. “A Note on the Perfect Encryption Assumption in a Process Calculus”. In: *FoSSaCS*. 2004, pp. 514–528.
- [ZD05] Roberto Zunino and Pierpaolo Degano. “Weakening the perfect encryption assumption in Dolev-Yao adversaries”. In: *Theor. Comput. Sci.* 340.1 (2005), pp. 154–178.