Formalisation of Probabilistic Testing Semantics in Coq*

Yuxin Deng[1] and Jean-Francois Monin[2]

[1] Shanghai Key Laboratory of Trustworthy Computing,
MOE International Joint Lab of Trustworthy Software, East China Normal University
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France

Abstract. Van Breugel et al. [F. van Breugel et al, *Theor. Comput. Sci.* 333(1-2):171-197, 2005] have given an elegant testing framework that can characterise probabilistic bisimulation, but its completeness proof is highly involved. Deng and Feng [Y. Deng and Y. Feng, *Inf. Comput.* 257:58-64, 2017] have simplified that result for finite-state processes. The crucial part in the latter work is an algorithm that can construct enhanced tests. We formalise the algorithm and prove its correctness by maintaining a number of subtle invariants in Coq. To support the formalisation, we develop a reusable library for manipulating finite sets. This sets an early example of formalising probabilistic concurrency theory or quantitative aspects of concurrency theory at large, which is a rich field to be pursued.

1 Introduction

One of the central concepts in concurrency theory is bisimulation [24,25]. Its generalisation in probabilistic concurrency theory is put forward by Larsen and Skou in [22]. Various characterisations of the largest probabilistic bisimulation (aka bisimilarity) by probabilistic extensions of Hennessy-Milner logic [17] have appeared in the literature [22,11,12,26,8,18,16,10,6]. For example, it is shown in [22] that probabilistic bisimilarity can be characterised by a very simple testing framework for reactive probabilistic processes [22,28] with a minimal probability assumption. In [27] van Breugel et al. generalise the testing characterisation of [22] to labelled Markov processes, i.e., reactive probabilistic processes [22,28] with continuous state spaces, and surprisingly, with an even simpler testing language. Generally speaking, the simpler the logical or testing characterisation, the more difficult the completeness proof of the chararacterisation. The reason is that this kind of proofs usually involve constructing distinguishing formulae or tests for non-bisimilar states, which is more challenging if there are fewer modalities to use. Van Breugel et al. succeeded in proving such an elegant result by making use of advanced machinery such as the Lawson topology on probabilistic powerdomains [19] and Banach algebras. In [7], Deng and Feng consider finite-state reactive probabilistic processes and give an extremely elementary proof of the coincidence of bisimilarity with the aforementioned testing equivalence while avoiding all the advanced machinery used in [27]. The core of that proof is to construct a sort of enhanced tests from basic tests by a tricky algorithm. Therefore, the correctness of the algorithm is crucial for the validity of the testing characterisation of probabilistic bisimilarity. A manual proof is given in [7], but to increase our confidence, a machine-checkable proof would be preferable. This is worthwhile because, as far as we know, among all the modal or testing characterisations of probabilistic bisimilarity, the one in [27] is the simplest, and the completeness proof presented in [7] is the most elementary and thus accessible.

In the current work, we formalise the algorithm of [7] in Coq [5] and prove its termination and correctness. We choose Coq because it is one of the mainstream proof assistants that has a large number of users in both industry and research communities. It has been successfully used for formal specifications of the X86 and LLVM instruction sets and programming languages such as C

* Supported by the National Natural Science Foundation of China (61672229, 61832015), the French national research organization ANR (grant ANR-15-CE25-0008), and the Inria-CAS joint project Quasar.

[20,29,21]. It has also been used to build CompCert [23], a fully-verified optimizing C compiler, and CertiKos [15], a fully verified hypervisor, for proving the correctness of many algorithms. Moreover, some important results in mathematics, such as the four-color theorem [13] and the Feit-Thompson theorem [14] are formally proved with Coq. We also claim that some features of the type system of Coq are very important in our formalisation in the presence of nested loops (more on this in Section 4.5).

The algorithm that we formalise in the current work involves two nested loops that render termination and correctness proofs highly non-trivial. We carefully design a number of invariants and show that they are preserved by appropriate loops. Sometimes, the invariants are not mutually independent. As we will see in Section 5.3, there are scenarios where we have two invariants (a) and (b), and after some steps of execution, invariant (b) holds as a post-condition because in the precondition both (a) and (b) are required to hold. What is more, as invariants are predicates on program states, they rely on another kind of invariants on program states. This happens because we heavily use finite sets. On one hand, if a set contains elements of the form (i, j) where i and j are natural numbers with $i < j$, we must make sure that no matter how the set expands or shrinks, its elements always keep that form. This is a particular invariant originated from program states. On the other hand, since we represent finite sets as lists, a general invariant to be maintained is that there is no duplicated elements in the lists no matter how the lists evolve under different set operations. The outcome of our development is not only a formal proof of the correctness of the non-trivial algorithm for constructing enhanced tests, but also a convenient library for manipulating finite sets.

Although there are efforts on the mechanisation of reasoning about randomised algorithms [3] and on applications to cryptography [4] there exists little work on formalising probabilistic concurrency theory, or quantitative aspects of concurrency theory at large. The current work sets an example towards this direction. Undoubtedly, much more can be done in the future.

The rest of the paper is structured as follows. In Section 2 we recall the background on probabilistic testing semantics and the algorithm that we will consider in Section 4. In Section 3 we outline our use of Coq. In Section 4 we formalise the algorithm in Coq. In Section 5 we introduce the main invariants used in proving the correctness of the algorithm. Finally, we conclude in Section 6.

The Coq scripts are available at the following link

http://www-verimag.imag.fr/~monin/Proof/ProbaTesting/.

2 Preliminaries

In this section, we recall the probabilistic testing semantics and the algorithm for computing enhanced tests introduced in [7].

Let S be a finite set. A *(discrete) probability distribution* over S is a function $\Delta : S \to [0, 1]$ with $\sum_{s \in S} \Delta(s) = 1$. Its *support*, written $\lceil \Delta \rceil$, is the set $\{s \in S \mid \Delta(s) > 0\}$. Let $\mathcal{D}(S)$ denote the set of all distributions over S. We write $\overline{s}$ for the point distribution satisfying $\overline{s}(t) = 1$ if $t = s$, and 0 otherwise. If $p_i \geq 0$ and Δ_i is a distribution for each i in some finite index set I, then $\sum_{i \in I} p_i \cdot \Delta_i$ is the function given by

$$(\sum_{i \in I} p_i \cdot \Delta_i)(s) \quad = \quad \sum_{i \in I} p_i \cdot \Delta_i(s) \ .$$

If $\sum_{i \in I} p_i = 1$ then this is easily seen to be a distribution in $\mathcal{D}(S)$.

Definition 1. *A* reactive probabilistic labelled transition system *(rpLTS) is a triple* $(S, A, \rightarrow)$*, where* S *is a finite set of states,* A *is a finite set of actions, and the transition relation* $\rightarrow$ *is a partial function from* $S \times A$ *to* $\mathcal{D}(S)$*.*

We write $s \xrightarrow{a} \Delta$ for $\rightarrow (s, a) = \Delta$.

In the probabilistic setting, we often need to compare distributions. There is a way of lifting relations on states to relations on distributions [9].

Definition 2. *Given two sets S, T and a relation $\mathcal{R} \subseteq S \times T$, the lifted relation $\mathcal{R}^\dagger \subseteq \mathcal{D}(S) \times \mathcal{D}(T)$ is the smallest relation that satisfies:*

(i) $s \mathcal{R} t$ *implies* $\overline{s} \mathcal{R}^\dagger \overline{t}$;
(ii) $\Delta_i \mathcal{R}^\dagger \Theta_i$ *for all* $i \in I$ *implies* $(\sum_{i\in I} p_i \cdot \Delta_i) \mathcal{R}^\dagger (\sum_{i\in I} p_i \cdot \Theta_i)$, *where* I *is a finite index set and* $\sum_{i\in I} p_i = 1$.

The above lifting operation is used to define probabilistic bisimulation.

Definition 3. *A binary relation $\mathcal{R} \subseteq S \times S$ is a* probabilistic simulation *if $s \mathcal{R} t$ and $s \xrightarrow{a} \Delta$ implies the existence of some transition $t \xrightarrow{a} \Theta$ with $\Delta \mathcal{R}^\dagger \Theta$.*

If both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are probabilistic simulations, then $\mathcal{R}$ is a probabilistic bisimulation. *The largest probabilistic bisimulation is* probabilistic bisimilarity.

Let us fix a rpLTS $(S, A, \rightarrow)$ and recall the simple testing framework proposed in [27].

Definition 4. *Let $\mathcal{T}$ be a testing language given by the grammar*

$$t \ ::= \ \omega \mid a \cdot t \mid \langle t, t \rangle$$

where a ranges over the set of labels A of our rpLTS. The function $Pr : S \times \mathcal{T} \mapsto [0, 1]$ prescribes the probability of applying a test to a state as follows:

$$\begin{aligned} Pr(s, \omega) &= 1 \\ Pr(s, a \cdot t) &= \begin{cases} \sum_{s' \in S} \Delta(s') \cdot Pr(s', t) & \text{if } s \xrightarrow{a} \Delta \\ 0 & \text{otherwise.} \end{cases} \\ Pr(s, \langle t_1, t_2 \rangle) &= Pr(s, t_1) \cdot Pr(s, t_2) \end{aligned}$$

We call $\langle t_1, t_2 \rangle$ a *conjunction* of two tests, which models the copying capacity of probabilistic testing. Here, conjunction is given the arithmetic interpretation as multiplication, which differs from other logical characterisations of probabilistic bisimilarity. We often write t^2 for $\langle t, t \rangle$, and t^{m+1} for $\langle t, t^m \rangle$, where $m \geq 2$. That is, t^m is the conjunction of m copies of t. It is obvious that

$$Pr(s, t^m) = (Pr(s, t))^m \tag{1}$$

for any state s and test t.

Definition 5. *The testing language $\mathcal{T}$ induces a testing equivalence relation, written $=_\mathcal{T}$, by letting $s_1 =_\mathcal{T} s_2$ if $Pr(s_1, t) = Pr(s_2, t)$ for any $t \in \mathcal{T}$.*

It is shown in [7] that $=_\mathcal{T}$ is a probabilistic bisimulation [22]. The key ingredient of the proof is to introduce a notion of enhanced tests. By making use of Algorithm 1, we can construct enhanced tests that satisfy the conditions in Lemma 1. From that lemma, it is not difficult to prove that $=_\mathcal{T}$ is a probabilistic bisimulation. Let us first set up the scenario where Algorithm 1 applies. Observe that $=_\mathcal{T}$ is an equivalence relation. Hence, we can partition the state space S according to $=_\mathcal{T}$. Let $C_1, \cdots, C_n$ be the equivalence classes induced by $=_\mathcal{T}$, where $n \geq 1$. Within each equivalence class C_i, the states are testing equivalent. So we can write $Pr(C_i, t)$ for $Pr(s_{ij}, t)$, where s_{ij} is any state in C_i and t is any test. Nevertheless, for any two states in different equivalence classes, there exist some tests that can tell them apart. For any i, j with $1 \leq i < j \leq n$, let t_{ij} be a test that distinguishes C_i from C_j; that is, $Pr(C_i, t_{ij}) \neq Pr(C_j, t_{ij})$. Notice that here t_{ij} is only a distinguishing test for C_i and C_j, and in general it says nothing about a third equivalence class C_k when $k \neq i, j$. For

example, applying t_{ij} to C_i and then to C_k might yield the same outcome. This is normal because t_{ij} is not necessarily a distinguishing test for C_i and C_k. The surprising fact discovered in [7] is that it is possible to construct an *enhanced test* that sharpens testing outcomes to distinguish many equivalence classes. More precisely, applying the enhanced test to some equivalence classes gives either 0 or distinct positive values.

Lemma 1. *For any $I \subseteq \{1, \cdots, n\}$ with $I \neq \emptyset$, there exist a nonempty $I' \subseteq I$ and an enhanced test t such that*

(i) for all $k \in I$, $Pr(C_k, t) > 0$ iff $k \in I'$;
(ii) for any $i \neq j \in I'$, $Pr(C_i, t) \neq Pr(C_j, t)$.

Algorithm 1: Compute an enhanced test

input : A nonempty set $I = \{1, \cdots, n\}$ with the distinguishing tests t_{ij} for all $i \neq j$.
output: A nonempty $I' \subseteq I$ and an enhanced test t satisfying (i) and (ii) in Lemma 1.

```
1  begin
2      I_pass ← ∅;
3      I_rem ← {(i, j) ∈ I × I : i < j};
4      I' ← I;
5      t ← ω;
6      while I_rem ≠ ∅ do
7          Choose arbitrarily (i, j) ∈ I_rem;
8          I' ← {k ∈ I' : Pr(C_k, t_ij) > 0};
9          I_dis ← {(k, l) ∈ I_rem ∩ I' × I' : Pr(C_k, t_ij) ≠ Pr(C_l, t_ij)};
10         I_rem ← (I_rem ∩ I' × I')\I_dis;
11         I_pass ← (I_pass ∩ I' × I') ∪ I_dis;
12         t ← ⟨t, t_ij⟩;
13         I_tem ← ∅;
14         I ← I_pass;
15         while I ≠ ∅ do
16             I ← {(k, l) ∈ I_pass\I_tem : Pr(C_k, t) = Pr(C_l, t)};
17             if I ≠ ∅ then
18                 t ← ⟨t, t_ij⟩;
19                 I_tem ← I_tem ∪ I;
20             end
21         end
22     end
23     return I', t;
24 end
```

The lemma is valid because we can use Algorithm 1 to construct such an enhanced test. The current work proposes (and proves the correctness of) a functional version of this algorithm. However, its design was driven by the original imperative version given in [7]. Proving that the functional version fits in with the imperative one could be performed using standard transformation techniques, but this is not needed: what matters is the existence of an algorithm satisfying the required specification. The algorithm initially sets I' to be I and t to be ω, then it gradually updates the test t to equipe it with more and more discriminating power. The construction of the new tests involves removing the indices k with $Pr(C_k, t) = 0$ from I' and keeping the indices i, j such that $Pr(C_i, t)$ and $Pr(C_j, t)$ are distinct and both positive. The outer loop uses four auxiliary sets: $\mathcal{I}_{pass}$, $\mathcal{I}_{rem}$, $\mathcal{I}_{dis}$, and $\mathcal{I}_{tem}$. Among them, $\mathcal{I}_{pass}$ and $\mathcal{I}_{rem}$ form a partition of the set $\{(i, j) \in I' \times I' : i < j\}$. The

subset $\mathcal{I}_{pass}$ contains the pairs (i, j) such that the current test t can distinguish C_i from C_j, while $\mathcal{I}_{rem}$ contains the remaining pairs to be processed. Each iteration of the outer loop picks up any pair (i, j) in $\mathcal{I}_{rem}$, uses the distinguishing test t_{ij} to form $\mathcal{I}_{dis}$, which is a subset of $\mathcal{I}_{rem}$, and moves it from $\mathcal{I}_{rem}$ to $\mathcal{I}_{pass}$. Each pair being moved, e.g., $(k, l) \in \mathcal{I}_{dis}$ indicates that C_k and C_l can be differed by t_{ij}. However, just expanding $\mathcal{I}_{pass}$ with $\mathcal{I}_{dis}$ is insufficient. A newly added index k might conflict with another index l, which occurs either already in the old $\mathcal{I}_{pass}$ or in the set $\mathcal{I}_{dis}$, in the sense that C_k and C_l cannot be distinguished by t. To solve this problem, the inner loop tries to update t by padding it with enough copies of t_{ij} until it can distinguish all the equivalence classes indicated by the pairs in $\mathcal{I}_{pass}$. Interestingly, the padding procedure only involves conjunctions of tests and this suffices for our purpose! The auxiliary set $\mathcal{I}_{tem}$ is introduced to facilitate this procedure and contains all the pairs indicating the equivalence classes distinguishable by the final enhanced test. When all the pairs in $\mathcal{I}_{rem}$ are explored, the whole procedure terminates.

Two main properties that interest us are the termination and the correctness of the algorithm. Let us give a more detailed analysis of the termination property. We first look at the inner **while** loop. In each iteration, $\mathcal{I}$ is assigned a new value, which is a subset of $\mathcal{I}_{pass} \backslash \mathcal{I}_{term}$. If it becomes empty, the loop terminates immediately. Otherwise, the set $\mathcal{I}_{tem}$ is enlarged to include $\mathcal{I}$. Since the set $\mathcal{I}_{pass}$ does not change in the inner loop, in the next iteration the set $\mathcal{I}_{pass} \backslash \mathcal{I}_{term}$ becomes smaller and so does $\mathcal{I}$. Eventually, $\mathcal{I}$ must become empty and the loop terminates. For the outer **while** loop, in each iteration we choose a pair, say (i, j), from $\mathcal{I}_{rem}$, and then update $\mathcal{I}_{dis}$ and $\mathcal{I}_{rem}$. Since t_{ij} is a distinguishing test for (i, j), the two values $Pr(C_i, t_{ij})$ and $Pr(C_j, t_{ij})$ cannot be 0 at the same time. If both of them are positive, then $\mathcal{I}_{dis}$ contains at least the pair (i, j) and is not empty. If exactly one of them, say $Pr(C_i, t_{ij})$, is 0, then the corresponding index i is removed from I', which causes I' to shrink. In both cases, the assignment of $\mathcal{I}_{rem}$ by $(\mathcal{I}_{rem} \cap I' \times I') \backslash \mathcal{I}_{dis}$ makes $\mathcal{I}_{rem}$ strictly smaller. Eventually, $\mathcal{I}_{rem}$ becomes empty and the outer loop terminates. The number of iterations for the inner loop depends on the size of the set $\mathcal{I}_{pass}$, and for the outer loop depends on the size of $\mathcal{I}_{rem}$.

The correctness of the algorithm relies on the following four invariants, namely, at the beginning of each run of the outer **while** loop,

(a) $\mathcal{I}_{pass} \cup \mathcal{I}_{rem} = \{(i, j) \in I' \times I' : i < j\}$;
(b) $I' \neq \emptyset$;
(c) for all $k \in I$, $Pr(C_k, t) > 0$ iff $k \in I'$;
(d) for any $(i, j) \in \mathcal{I}_{pass}$, $Pr(C_i, t) \neq Pr(C_j, t)$.

Statement (a) is easy to show because each time $\mathcal{I}_{dis}$ is obtained, it is moved from $\mathcal{I}_{rem}$ to $\mathcal{I}_{pass}$. Statement (b) is also simple. Since t_{ij} is a distinguishing test for (i, j), at least one of the two values $Pr(C_i, t_{ij})$ and $Pr(C_j, t_{ij})$ is positive. Therefore, I' cannot be empty. Statement (c) can be proved by induction on the number of iterations of the outer **while** loop. Statement (d) can also be proved inductively. A manual proof is provided in [7]. It is involved as it mixes logical reasoning with quantitative computation. Therefore, a machine-checkable proof is needed.

The above analysis of the algorithm serves as a guide for our formal proof of the correctness in Coq. However, we do not faithfully follow the manual proofs in [7]. In particular, for statement (d) we provide a proof that deviates a lot from the original one and is easier to implement inductively.

3 Our use of Coq

In order to complete our case study, several issues need to be addressed:

1. Representing an (originally imperative) algorithm in Coq;
2. Dealing with finite sets;
3. Formalising and proving assertions and invariants;
4. Proving the termination property.

The previous issues are not independent from each other. For instance, in the strongly normalising purely functional setting of Coq, only terminating functions can be defined. In practice, it means that termination has to be taken into account at the definition time of a function, making issues 1 and 4 interact. Moreover, part of the invariants can be relevant to termination. Finally, design choices related to issue 2 interact with all other issues.

About issue 1, an important design decision has to be taken from the very beginning. If we stick to imperative programs, there are at least two distinct techniques for representing them: by a function from states to states – typically, a while loop will be encoded as a tail-recursive function – or, by an inductive relation between input and output states. An extension of the last option is even to consider an IMP-like toy imperative language (including ad-hoc operations on sets), with its abstract syntax and a big-step or small-step operational semantics [1].

For our purpose here, we only need to prove that there exists an effective terminating algorithm satisfying the expected input-output relation. The imperative or functional nature of the presentation of the algorithm does not matter. Therefore we just provide a Coq function returning the desired final state from the initial state and the above considerations about imperative programming just vanish. Nevertheless, the manual proof given before in an imperative setting is an important guide in the Coq development. In particular, the given invariants remain very important and provide the structure of the proof. Some amount of dependent typing is needed, in particular in order to deal with the issue of termination. For clarity, at the end, a simple executable functional program can be extracted and it is easy (but not crucial) to manually check that this program corresponds to the imperative one given in the previous section.

Issue 2 is not fundamental by itself: models and basic results for finite sets are available in the standard library of Coq [2]. However, for the management of our proofs, the introduction of concise and familiar set notations on top of it turned out to be very important: our very first formalization, based only on available libraries, resulted in statements very hard to reread and follow, and completing the last steps of the proofs became a discouraging hassle. Another choice had to be made: the representation of finite sets. Several options are possible and available in existing librairies: lists, with or without duplicates, ordered or unordered; binary search trees; AVL, to quote a few. Here we only need to prove the correctness of an abstract algorithm, which does not depend on a specific representation of finite sets. Efficiency of computation of the various operations is an orthogonal issue which is not relevant here. In summary, we choose a very simple representation of finite sets based on lists without duplicates, already available in the Coq standard library, completed with convenient notations and suitable basic lemmas.

Most of our efforts are spent on issue (3) because we are interested in the correctness of the algorithm, which follows from a non-trivial combination of invariants. In the presence of nested loops, the design of invariants is subtle. We should pay attention to not only (i) the internal logic of the algorithm but also (ii) the program states. Both (i) and (ii) give rise to different invariants and furthermore one invariant may depend on another.

3.1 A library on finite sets

The algorithm deals with two kinds of finite sets: on `nat` (also denoted by $\mathbb{N}$) and on $\mathbb{N}^2$. Basic operations on sets ($\cap$, $\cup$, $\subseteq$, $\in$, etc.) correspond to algorithms ultimately depending on a test for equality on the elements. Therefore, the actual operations are parameterised not only by the underlying types, such as $\mathbb{N}$ or $\mathbb{N}^2$, but also by an actual decision algorithm for equality on this type. For instance, the complete Coq term for an expression such as $A \cap B$, where A and B are two sets of natural numbers, would be `intersection` $\mathbb{N}$ `dec_eq_nat A B`. Therefore, even initially quite short expressions like the ones used in the algorithm described in the previous section quickly become very cumbersome: hard to read and to reason about. In order to recover short expressions, three mechanisms are used:

– *Infix notations*, available in Coq for a long time, e.g., + for addition on nats. But binary operators can only be used on expressions of the form $f\ x_1\ x_2$. The issue is then to hide the underlying type T and the equality test of type $T \to T \to Prop$ to be used on the significant arguments x_1 and x_2.
– *Implicit arguments*: in our example, the argument $\mathbb{N}$ can be automatically inferred from `A` (or `B`).
– *Type classes*, allowing us to provide a default value having a given type; in our case, the intended value is an equality test of type $T \to T \to Prop$, where T was inferred from x_1 and x_2. Note that in general, there is no automatic way to provide such a value. For instance, $\mathbb{N} \to \mathbb{N}$ has no decidable equality, whereas there are several possible algorithms for an equality test on $\mathbb{N}$. Type classes are an advanced mechanism of Coq, introduced much more recently than implicit arguments[3].

In terms of the data structure for modeling finite sets, we simply use lists. We rely on the Coq library `ListSet` and represent a finite set by a list without duplicates.

Notations
The membership and inclusion relations are just parameterised by the implicit types of the elements. Corresponding operations exist in the standard library, thus we just add suitable notations for them. Here is the code for membership.

```
Notation "x ∈ y" := (set_In x y) (at level 70, no associativity).
```

Then, `x ∉ y` and `u ⊆ v` (set inclusion) are defined in a similar way. Non-inclusion (`u ⊄ v`) is not the negation of inclusion, but a constructive version providing a witness.

```
Inductive nincl {A} (u v : set A) : Prop :=
  nincl_intro : forall a, a ∈ u -> a ∉ v -> nincl u v.
```

Strict inclusion (`u ⊂ v`) is defined as `u ⊆ v ∧ ∃a, a ∈ v ∧ a ∉ u`.

The usual binary algebraic operations on sets assume a decidable equality on the implicit type of elements. For instance, in the standard library, the union of two sets u and v of elements of type A can be written (`set_union` $t_A\ u\ v$), where t_A is a test for equality on A. In order to hide t_A, we define a suitable type class `EqDec` as follows.

```
Definition eq_dec (A: Type) := ∀ (x y: A), {x = y}+{x <> y}.

Class EqDec (A : Type) := {Aeq_dec : eq_dec A}.
```

This makes it possible to redefine `union` as follows:

```
Definition union {A}{ED: EqDec A} u v := set_union Aeq_dec u v.
Notation "u ∪ v" := (union u v) (at level 50, left associativity).
```

Thanks to the mechanism of Coq classes, `ED` can be given as an implicit argument and the equality test `Aeq_dec` is guessed from the type of `ED`. At this stage, `union` is presented as a binary relation and can be provided a binary infix operator. Coq offers an additional mechanism called *generalisable variables*, allowing us to shorten sequences of implicit arguments. Here, A can be inferred from the type of `ED`. Therefore, one uses the following equivalent definition for `union`.

```
Definition union `{ED: EqDec A} u v := set_union Aeq_dec u v.
```

[3] Alternatively, we could use *canonical structures*, but more complex interactions from the user would be required.

The usual operations `u ∩ v`, `u \ v` (set difference) and `u × v` are defined along the same lines, as well as `a ⊕ u` (resp. `a † u`) for adding (resp. removing) an element `a` to a set `u`.

Introduction and elimination principles
The Coq library on finite sets provides lemmas suitable for rewriting-style proofs. For instance, `set_union_iff` states that `a ∈ u ∪ v` can be rewritten as `a ∈ u \/ a ∈ v` and conversely. However, Coq is much better at unifying propositions than at recognising that the subterm of a proposition matches the left (or right) member of a proven congruence, modulo folding or unfolding of definitions. Moreover, it is often the case that only an inclusion between two sets is available, instead of an equality. Rather than insisting on algebraic-style reasoning, we prefer to go back to basic logical principles, that is, to the obvious introduction and elimination principles dedicated to each set operation. This turns out to be suitable and efficient for our needs. For instance, we provide the following lemmas dedicated to union.

```
Lemma union_intro1 ‘{ED: EqDec A} {a u v} : a ∈ u -> a ∈ u ∪ v.
Lemma union_intro2 ‘{ED: EqDec A} {a u v} : a ∈ v -> a ∈ u ∪ v.
Lemma union_elim ‘{ED: EqDec A} {a u v} : a ∈ u ∪ v -> a ∈ u \/ a ∈ v.
```

Similar lemmas are provided for the empty set, intersection, set difference, Cartesian product, addition and removal of an element. It is then easy to prove additional general lemmas, such as `(u ∩ v) ⊆ u`, or more specific lemmas like `(u ∩ w) ∪ (v ∩ w) ⊆ ((u ∪ v) ∩ w)`.

Filtering
The algorithm makes crucial use of filtering, thus we need introduction and elimination principles completed by a small number of lemmas as follows:

- `a ∈ u -> f a = true -> a ∈ (filter f u)`,
- `a ∈ filter f u -> a ∈ u ∧ f a = true`,
- `a ∈ u -> filter f u = ∅ -> f a = false`,
- `a ∈ (u \ filter f v) -> a ∈ u \ v ∨ f a = false`,
- `filter f (u \ v) = ∅ -> all_false v f -> all_false u f`.

Duplication
Similarly, as sets are represented by lists without duplicates, we need the following lemmas:

- `x ∈ u -> x ∈ nodup u`,
- `x ∈ nodup u -> x ∈ u`,
- `NoDup l -> NoDup (l ∩ l')`,
- `NoDup l -> NoDup (filter f l)`.

3.2 Termination

The termination of the algorithm is based on a sequence of strictly decreasing sets, ordered by inclusion. In the current version, sets are represented by finite lists without duplicates. Therefore, a simple measure is the size of the corresponding list. Technically, it means that, in addition to the invariants related to the design of the algorithm, we need invariants keeping track of non-duplication. In some occasions, e.g., the inner loop of the algorithm, the termination property may be slightly more complicated and depend on two sets because we jump out of a loop because either one set is empty or the other shrinks. To deal with that case, we introduce the following measure on two sets.

```
Definition size (l1 l2 : set ℕ²) : ℕ :=
  match l1 with
  | nil => 0
  | _ :: _ => S (length l2)
  end.
```

4 Formalisation of the algorithm

4.1 Preliminaries and input of the algorithm

We first formalise the testing semantics. According to the algorithm, an enhanced test t is always in the form $\langle \omega, t_{(i_1,j_1)}, t_{(i_2,j_2)}, ..., t_{(i_m,j_m)} \rangle$ for some $m > 0$ with $t_{(i_l,j_l)}$ being a distinguishing test and i_l, j_l the indices for two different equivalence classes. In other words, as far as the correctness of Algorithm 1 is concerned, we do not need to consider tests of the form $a \cdot t$, where a is a label and t is a test, as previously given in Definition 4. Since the input of Algorithm 1 has distinguishing tests and the final enhanced test is constructed out of them, we make a distinction between the two types of tests. We formalise a distinguishing test as a *basic test* and the enhanced test is constructed as a list of basic tests.

```
Inductive basic_test : Set :=
  | mk_bt : ℕ -> ℕ -> basic_test.

Definition test : Set := list basic_test.
```

Now that we have two types of tests, it is natural to model the function Pr given in Definition 4 in two steps. We first define a variable `Prb` that prescribes the probability of applying a basic test to the states in an equivalence class. We use the hypothesis that all probabilities are non-negative. Then we define the function `Prtest` to calculate the probability of applying a general test to a state by multiplying the probabilities of applying `Prb` to basic tests and that state.

```
Variable Prb : ℕ -> basic_test -> Q.

Fixpoint Prtest (i : ℕ) (t : test) : Q :=
  match t with
  | nil => 1
  | bt :: t' => Qmult (Prb i bt) (Prtest i t')
  end.
```

We see from the first case of the above definition that `Pr i nil = 1`, which corresponds nicely to the fact that $Pr(C_i, \omega) = 1$. Therefore, the special test ω is modelled by the empty list and there is no need to define a special basic test.

The set I of Lemma 1 is represented by an nonempty finite subset of $\mathbb{N}$ called `I0` (`I` has a special meaning in Coq). We also assume a function `oracle` returning the distinguishing test t_{ij} for all equivalence class indices $i \neq j$ with $i, j \in I$.

```
Definition distinguish i j :=
   {t : basic_test | i <> j  ->  ¬ Prb i t == Prb j t}.

Variable I0 : set ℕ.
Hypothesis initially_non_empty : I0 <> ∅.

Variable oracle : ∀ i j : ℕ, (i, j) ∈ I0×I0 -> distinguish Prb i j.
```

4.2 States

Different parts of the algorithm involve different variables. States are represented by records constrained by a structural invariant. Transitions are described by functions between such records. In particular, loops are modelled by tail-recursive functions. Instead of using a notion of global states containing all variables, we use *inner states* and *outer states* to model the data flow of the inner loop and the outer loop of the algorithm, respectively.

The inner states are determined by the three sets $\mathcal{I}$, $\mathcal{I}_{tem}$, and $\mathcal{I}_{pass}$, together with the test t. However, $\mathcal{I}_{pass}$ is not modified in the inner loop. An advantage of the functional style is that we can consider this set as a constant parameter of the corresponding function. This way, we get the invariance of $\mathcal{I}_{pass}$ for free, without additional proof obligation. In Coq, we can conveniently use the scoping mechanism of sections as follows.

```
Section sec_with_Y_pass.
Variable Y_pass : set ℕ^2.
```

The other sets are then modelled by the three components `Y`, `Y_term` and `tt` of the following record type.

```
Record in_iter_data : Type := mk_in_iter_data {
  Y : set ℕ^2;
  Y_term : set ℕ^2;
  tt : test;
}.
```

The outer states are determined by the three sets `I'`, $\mathcal{I}_{rem}$, and $\mathcal{I}_{pass}$, together with the test t. They are modelled by the four components `I'`, `Y_rem`, `Y_pass` and `et` of the following record type.

```
Record out_iter_data : Type := mk_out_iter_data {
  I' : set ℕ;
  Y_rem : set ℕ^2;
  Y_pass : set ℕ^2;
  et : test; (* enhanced test *)
}.
```

An invariant maintained by the outer states is the following. It says that each pair (i, j) in $\mathcal{I}_{rem}$ consists of two different indices taken in I, and there are no duplicates in the three sets I', $\mathcal{I}_{rem}$, and $\mathcal{I}_{pass}$.

```
Inductive out_data_invariant0 (r: out_iter_data) : Prop :=
| out_data_invariant0_intro :
    Y_rem r ⊆ I0×I0 ->
    (∀ i j, (i,j) ∈ (Y_rem r) -> i <> j) ->
    NoDup (I' r) -> NoDup (Y_rem r) -> NoDup (Y_pass r) ->
    out_data_invariant0 r.
```

Lines 2-5 in Algorithm 1 are the initialisation step. The effect is to create the following initial state.

```
Definition init_data : out_iter_data :=
  {| I' := initial_I;
     Y_rem := initial_Y;
     Y_pass := ∅;
     et := ∅ |}.
```

The two auxiliary constants `initial_I` and `initial_Y` are introduced in order to create the sets I and $\{(i, j) \in I \times I : i < j\}$, respectively.

```
Definition initial_I : set ℕ := nodup I0.
Definition initial_Y : set ℕ^2 := filter_lt (nodup (I0 × I0)).
```

Here we see an example of using filters: from the set $I \times I$ (without duplicates) we filter out all pairs (i, j) not satisfying the condition $i < j$. This is a convenient means of creating subsets from a given set that we often use in our Coq development.

The final output data of the algorithm is the set I' and the enhanced test t, as described by the type `final_data`.

```
Record final_data : Type := mk_final_data {
  I'f : set ℕ;
  etf : test;
}.
```

4.3 Inner loop

To formalise loops, our general strategy is to first define one round of iteration, which is a function from a state possibly with additional parameters to a new state, and then repeatedly apply the iteration until the state meets the termination condition.

The inner loop (lines 15-21 of Algorithm 1) makes use of an iteration step described below, where `next_Y_inner_loop` computes the new value to be assigned to $\mathcal{I}$, as stated in line 16.

```
Definition next_Y_inner_loop r := filter_eq (tt r) (Y_pass \ Y_term r).

Definition inner_loop_iter (r : in_iter_data) (tij : basic_test) : in_iter_data :=
  match Y r with
  | ∅ => r
  | _ :: _ => let Y' := next_Y_inner_loop r in
              match Y' with
              | ∅ => {| Y := Y';
                        Y_term := Y_term r;
                        tt := tt r |}
              | _ :: _ => {| Y := Y';
                             Y_term := Y_term r ∪ Y';
                             tt := tij :: tt r |}
              end
  end.
```

The structural invariant of the record maintained by the inner loop is:

```
Inductive in_data_invariant (d: in_iter_data) : Prop :=
| in_data_invariant_intro :
    Y d ⊆ Y_pass -> Y_term d ⊆ Y_pass ->
    NoDup (Y d) -> NoDup (Y_term d) -> in_data_invariant d.
```

It requires that for an inner state $(\mathcal{I}, \mathcal{I}_{term}, t)$ to be legal, we must have that $\mathcal{I} \subseteq \mathcal{I}_{pass}, \mathcal{I}_{term} \subseteq \mathcal{I}_{pass}$, and there is no duplicated element in $\mathcal{I}$, $\mathcal{I}_{term}$ and $\mathcal{I}_{pass}$.

The inner loop itself is formulated as a recursive function that repeatedly applies the iteration step `inner_loop_iter` . Note that each iteration step either makes $\mathcal{I}$ empty, or decreases the size of the set $\mathcal{I}_{pass} \backslash \mathcal{I}_{tem}$. So we define the measure `size_of_data` for inner states, which yields a strictly decreasing argument for the function `in_loop`.

```
Definition size_of_data r := size (Y r) (Y_pass \ Y_term r).

Function in_loop (r : in_iter_data) (di : in_data_invariant r) (tij : basic_test)
  {measure size_of_data r} : in_iter_data :=
  match Y r with
   | ∅ => r
   | _ :: _ => in_loop (inner_loop_iter r tij) (inner_loop_iter_invar di tij) tij
  end.
```

The previous definitions are written inside Section `sec_with_Y_pass` mentioned in Section 4.2. When refering to them outside of this section, namely when they are used in functions and proofs modelling the outer loop, an actual parameter for `Y_pass` has to be provided. Moreover, `in_loop` needs an additional parameter stating that `Y_pass` has no duplicate in order to ensure that `di` is maintained invariant. This assumption is stated at the beginning of Section `sec_with_Y_pass`.

4.4 Interface between inner and outer loops

Before entering the inner loop, we need to pass the information stored in the outer state to the inner state. The interface between the two loops is modelled by the function `mk_in_from_out` that creates an inner state from an outer state. It contains the formalisation of lines 13-14 in Algorithm 1.

```
Definition mk_in_from_out (r : out_iter_data) (bt : basic_test) : in_iter_data :=
  let rnew := outer_loop_iter1 r bt in
  {| Y := Y_pass rnew;
     Y_term := ∅;
     tt := et rnew |}.

Lemma out_data_in_data_invar r (dio : out_data_invariant0 r) bt :
  let rnew := outer_loop_iter1 r bt in
  in_data_invariant (Y_pass rnew) (mk_in_from_out r bt).
```

Note that the actual parameter given for `Y_pass` to `in_data_invariant` is `Y_pass rnew`.

4.5 Outer loop

As we did for the inner loop, we first specify one iteration step of the outer loop. The sequence of assignments before entering the inner loop (lines 7-12 of Algorithm 1) is described by the function `outer_loop_iter1`.

```
Definition outer_loop_iter1 (r : out_iter_data) (bt : basic_test) : out_iter_data :=
  let nI' := filter_pos bt (I' r) in
  let Ydis := filter_neq bt (Y_rem r ∩ (nI' × nI')) in
  {| I' := nI';
     Y_rem := (Y_rem r ∩ (nI' × nI')) \ Ydis;
     Y_pass := (Y_pass r ∩ (nI' × nI')) ∪ Ydis;
     et := bt :: et r |}.
```

Then the function `outer_loop_iter2` formalises one iteration of the outer loop by first calling `outer_loop_iter1`, then creating an inner state through the interface `mk_in_from_out`, and finally invoking `in_loop` to deal with the tasks required by the inner loop.

```
Definition outer_loop_iter2 r (dio : out_data_invariant0 r) bt : out_iter_data :=
  let rnew := outer_loop_iter1 r bt in
  let rin := mk_in_from_out r bt in
  let ndYp := nd_Y_pass r dio bt in
  let rin' := in_loop (Y_pass rnew) ndYp rin (out_data_in_data_invar r dio bt) bt in
  {| I' := I' rnew;
     Y_rem := Y_rem rnew;
     Y_pass := Y_pass rnew;
     et := tt rin' |}.
```

By repeatedly applying the iteration step `outer_loop_iter2`, we obtain a formalisation of the outer loop. The decreasing argument for the recursive function `out_loop` is the size of the set $\mathcal{I}_{rem}$. The auxilliary function `pick` checks if its argument is nonempty, that is, contains a pair (i, j); in that case, it returns a basic test distiguishing i and j, using the function `oracle`.

```
Inductive resu_pick r : Set :=
| P_empty : Y_rem r = ∅ -> resu_pick r
| P_nonempty : ∀ i j s,
    Y_rem r = (i, j) :: s -> distinguish Prb i j -> resu_pick r.

Definition pick {r} (di : out_data_invariant0 r) : resu_pick r.

Function out_loop r (di : out_data_invariant0 r)
         {measure size_of_out_data r} : out_iter_data :=
  match pick di with
  | P_empty _ e => r
  | P_nonempty _ i j s e dis => let (bt, _) := dis in
                                out_loop (outer_loop_iter2 r di bt)
                                         (outer_loop_iter_invar0 r di bt)
  end.
```

A technical difference from the inner loop is that the definition of the iteration step is defined only on states satisfying the loop invariant, entailing a more subtle use of dependent typing. More specifically, if we compare the definitions of `in_loop` and `out_loop`, both of them contain a tail recursive call with the first argument for the state reached after one iteration step then the second argument embedding the invariant satisfied by this state. This kind of of dependency, where properties depending on data are kept separated from them, is rather common in Coq developments: for instance, it avoids complications related to proof irrelevance that arise with dependent records (and any inductive type made of one constructor with dependencies between its arguments). However, in contrast to the definition of `in_loop`, where the state argument of the recursive call refers to data only, the corresponding argument for `out_loop`, namely `outer_loop_iter2 r di bt`, makes a crucial use of invariant `di`. We believe that this would be hard to express in a concise and accurate

way without dependent types. In other words, we benefit from the rich type system of Coq which includes dependent types, while avoiding issues related to dependent records.

Interestingly, the exact reason for this situation is that invariant `di` is needed in the body of `outer_loop_iter2` in order to allow for a call to `in_loop`. A similar situation can reasonably be expected with the formalisation of many algorithms containing nested loops, at least when non-trivial interactions occur between these loops.

4.6 The whole algorithm

Once the outer loop is formalised, the whole algorithm easily follows. We first initialise an outer state and then call `out_loop` before obtaining the final set I' and the enhanced test t.

```
Definition algo_compt_enhanced_test : final_data :=
  let r := init_data in
  let final_r := out_loop r init_data_invariant0 in
  {| I'f := I' final_r;
     etf := et final_r |}.
```

5 Formal proofs

In this section, we take a close look at some important invariants that finally entail the correctness of the algorithm.

5.1 Invariants of the inner loop

As mentioned earlier, the definition of `in_loop` embeds the structural invariant. The following invariants are used for invariant (c) of the outer loop.

```
Lemma inner_loop_iter_invar_c {k r bt} :
  0 < Pr k (tt r) -> (∃ t, tt r = bt :: t) -> Pr k (tt (inner_loop_iter r bt)) > 0.
Lemma inner_loop_iter_invar_c2 {k r} :
  Pr k (tt r) == 0 -> ∀ bt, Pr k (tt (inner_loop_iter r bt)) == 0.
```

The first one says that if $Pr(C_k, t) > 0$ and t is made from basic test bt then after running one iteration of the inner loop with bt, the test may evolve into some t', but we still have $Pr(C_k, t') > 0$. The second one states that if $Pr(C_k, t) = 0$ and t is changed into some t' then we always have $Pr(C_k, t') = 0$. The next two lemmas tell us that executing the whole inner loop preserves similar properties.

```
Lemma in_loop_invar_c {k r bt di} :
  0 < Pr k (tt r) -> (∃ t, (tt r) = bt :: t) -> 0 < Pr k (tt (in_loop r di bt)).
Lemma in_loop_invar_c2 {k r di} :
  Pr k (tt r) == 0 -> ∀ bt, Pr k (tt (in_loop r di bt)) == 0.
```

Many additional invariants and post-conditions are used for invariant (d) of the outer loop. For example, the lemma `in_loop_nextI` says that if the component $\mathcal{I}$ in an inner state is empty, then it remains empty after the execution of the inner loop.

```
Definition loop_body bt r := inner_loop_iter r bt.

Inductive ufp (iid: in_iter_data) r bt : Prop :=
  ufp_intro :
  ∀ n,
    iid = rditer _ n (loop_body bt) r ->
    (Y r <> ∅ -> (0 < n)%nat ∧
                 Y (rditer _ (pred n) (loop_body bt) r) <> ∅ ∧
```

```
                next_Y_inner_loop (rditer _ (pred n) (loop_body bt) r) = ∅) ->
    (Y r = ∅ -> (n = 0)%nat)
    -> ufp iid r bt.
Lemma unfold_fixed_point r (di : in_data_invariant r) bt :
  ufp (in_loop r di bt) r bt.

Lemma in_loop_nextI r di bt : Y r <> ∅ -> next_Y_inner_loop (in_loop r di bt) = ∅.
```

5.2 Invariants of the outer loop

The outer loop maintains four invariants as given in page 5. They are formalised as the predicates on outer states: `out_data_invariant_a`, `out_data_invariant_b`, `out_data_invariant_c`, and finally `out_data_invariant_d`. Althogether, they are used to form the global invariant `out_data_global_invariant`.

```
Inductive out_data_invariant_a (d: out_iter_data) : Prop :=
| out_data_invariant_a_intro :
    (Y_rem d ∪ Y_pass d) ⊆ (filter_lt (I' d × I' d)) ->
    (filter_lt (I' d × I' d)) ⊆ (Y_rem d ∪ Y_pass d) ->
    out_data_invariant_a d.

Inductive out_data_invariant_b (d: out_iter_data) : Prop :=
| out_data_invariant_b_intro : I' d <> ∅ -> out_data_invariant_b d.

Inductive out_data_invariant_c (d: out_iter_data)(I : set ℕ) : Prop :=
| out_data_invariant_c_intro :
    (∀ k, k ∈ I' d -> 0 < Pr k (et d)) ->
    (∀ k, k ∈ I \ I' d  -> Pr k (et d) == 0) ->
    out_data_invariant_c d I.

Inductive out_data_invariant_d (d: out_iter_data) : Prop :=
| out_data_invariant_d_intro :
    all_false (Y_pass d) (eq_prob (et d)) -> out_data_invariant_d d.

Inductive out_data_global_invariant (I : set ℕ) (r: out_iter_data) : Prop :=
  odgi_intro :
    out_data_invariant_a r ->
    out_data_invariant_b r ->
    out_data_invariant_c r I ->
    out_data_invariant_d r ->
    out_data_global_invariant I r.
```

As expected, the global invariant is established at the beginning of the outer loop.

```
Lemma init_data_invariant_all : out_data_global_invariant initial_I init_data.
```

5.3 Preservation lemmas for the outer loop

In order to show that the global invariant is preserved by the outer loop, we consider invariants (a)-(d) separately. Each of them is preserved after one iteration of the outer loop, and it is the same case for `out_data_invariant0` given in Section 4.2. Note that those invariants are not completely independent. For example, the preservation of invariant (b) depends on invariant (a), and the preservation of invariant (d) relies on invariant (c).

```
Lemma outer_loop_iter_invar0 :
  ∀ r (dio : out_data_invariant0 r) bt,
    out_data_invariant0 (outer_loop_iter2 r dio bt).
Lemma outer_loop_iter_invar_a {r dio bt} :
  out_data_invariant_a r -> out_data_invariant_a (outer_loop_iter2 r dio bt).
Lemma outer_loop_iter_invar_b {r i j dio bt} :
  out_data_invariant_a r ->
  (i,j) ∈ (Y_rem r) -> (i <> j -> ¬ Prb i bt == Prb j bt) ->
  out_data_invariant_b (outer_loop_iter2 r dio bt).
Lemma outer_loop_iter_invar_c {r bt dio I} :
  out_data_invariant_c r I -> out_data_invariant_c (outer_loop_iter2 r dio bt) I.
Lemma outer_loop_iter_invar_d {r bt dio I} :
  out_data_invariant_c r I -> out_data_invariant_d r ->
  out_data_invariant_d (outer_loop_iter2 r dio bt).
Lemma out_loop_invar I r (di : out_data_invariant0 r) :
  out_data_global_invariant I r -> out_data_global_invariant I (out_loop r di).
```

In the proofs of the lemmas `outer_loop_iter_invar_c` and `outer_loop_iter_invar_d`, we need to make an in-depth analysis of the inner loop and use a number of invariants discussed in Section 5.1.

Moreover, the emptyness of $\mathcal{I}\_rem$ is (trivially) ensured after running the outer loop, which implies the termination of the outer loop.

```
Lemma out_loop_ensures_empty_Yrem r di : Y_rem (out_loop r di) = ∅.
```

5.4 Main theorem

The desired post-condition is ensured after running the algorithm. It says that the final set I' and the enhanced test t satisfy the conditions required by Lemma 1. In other words, we have formally proved the correctness of the algorithm.

```
Theorem correctness:
  let (fI, ft) := algo_compt_enhanced_test in
  fI <> ∅ ∧
  (∀ k, k ∈ fI -> 0 < Pr k ft) ∧
  (∀ k, k ∈ (I0 \ fI) -> Pr k ft == 0) ∧
  (∀ i j, i ∈ fI -> j ∈ fI -> i <> j -> ¬ Pr i ft == Pr j ft).
```

As a corollary, we get a formal proof of Lemma 1. Note that the choice between a functional or an imperative style for the formalisation of Algorithm 1 is irrelevant here, since this algorithm is not part of the statement.

```
Corollary main_lemma :
  ∀ Prb : ℕ → basic_test → Q, (∀ i t, 0 <= Prb i t) →
  ∀ I0 : set ℕ, I0 ≠ ∅ →
  (∀ i j, (i, j) ∈ I0 × I0 → distinguish Prb i j) →
  ∃ (fI : set ℕ) (ft : test),
    let Pr := Prtest Prb in
    fI ≠ ∅ ∧
    (∀ k, k ∈ fI → 0 < Pr k ft) ∧
    (∀ k, k ∈ I0 \ fI → Pr k ft == 0) ∧
    (∀ i j, i ∈ fI → j ∈ fI → i ≠ j → ¬ Pr i ft == Pr j ft).
```

6 Conclusion

We have demonstrated a mechanisation of proofs in probabilistic testing semantics with Coq. Proving properties in this setting requires subtle reasoning both on algorithmic and quantitative aspects of program states. Our development includes more than 500 lines of specifications and definitions (but only 21 lines of them are needed for Lemma 1), and more than 900 lines of proof scripts. Along with a machine-checkable proof of the correctness of the non-trivial algorithm with nested loops for constructing enhanced tests, we obtain a convenient library for manipulating finite sets, which we believe will benefit future formalisation efforts such as formal reasoning with probabilistic bisimulations and testing equivalences.

Acknowledgment We would like to thank Yves Bertot for helpful discussion.

References

1. https://softwarefoundations.cis.upenn.edu/lf-current/index.html.
2. https://coq.inria.fr.
3. P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 74(8):568–589, 2009.
4. G. Barthe, J. M. Crespo, B. Grégoire, C. Kunz, and S. Zanella-Béguelin. Computer-Aided Cryptographic Proofs. In *ITP 2012*, volume 7406 of *LNCS*. Springer, 2012.
5. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
6. Y. Deng. *Semantics of Probabilistic Processes: An Operational Approach*. Springer, 2015.
7. Y. Deng and Y. Feng. Probabilistic bisimilarity as testing equivalence. *Information and Computation*, 257:58–64, 2017.
8. Y. Deng and R. van Glabbeek. Characterising probabilistic processes logically. In *Proc. LPAR 2010*, volume 6397 of *LNCS*, pages 278–293. Springer, 2010.
9. Y. Deng, R. van Glabbeek, M. Hennessy, and C. Morgan. Testing finitary probabilistic processes (extended abstract). In *Proc. CONCUR 2009*, volume 5710 of *LNCS*, pages 274–288. Springer, 2009.
10. Y. Deng and H. Wu. Modal characterisations of probabilistic and fuzzy bisimulations. In *Proc. ICFEM 2014*, volume 8829 of *LNCS*, pages 123–138. Springer, 2014.
11. J. Desharnais, A. Edalat, and P. Panangaden. Bisimulation for labelled Markov processes. *Information and Computation*, 179(2):163–193, 2002.
12. J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Approximating labelled Markov processes. *Information and Computation*, 184(1):160–200, 2003.
13. G. Gonthier. Formal proof — the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.
14. G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *Proc. ITP 2013*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013.
15. R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *Proc. OSDI 2016*, pages 653–669. USENIX Association, 2016.
16. M. Hennessy. Exploring probabilistic bisimulations, part I. *Formal Aspects of Computing*, 24(4-6):749–768, 2012.
17. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.
18. H. Hermanns, A. Parma, R. Segala, B. Wachter, and L. Zhang. Probabilistic logical characterization. *Information and Computation*, 209(2):154–172, 2011.
19. C. Jones. *Probabilistic nondeterminism*. PhD thesis, University of Edinburgh, 1990.
20. A. Kennedy, N. Benton, J. B. Jensen, and P. Dagand. Coq: the world's best macro assembler? In *Proc. PPDP 2013*, pages 13–24. ACM, 2013.

21. R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015.
22. K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94:1–28, 1991.
23. X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. Compcert – a formally verified optimizing compiler. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems*. SEE, 2016. https://hal.inria.fr/hal-01238879.
24. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
25. D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI Conference*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.
26. A. Parma and R. Segala. Logical characterizations of bisimulations for discrete probabilistic systems. In *Proc. FOSSACS 2007*, volume 4423 of *LNCS*, pages 287–301. Springer, 2007.
27. F. van Breugel, M. W. Mislove, J. Ouaknine, and J. Worrell. Domain theory, testing and simulation for labelled Markov processes. *Theoretical Computer Science*, 333(1-2):171–197, 2005.
28. R. J. van Glabbeek, S. A. Smolka, B. Steffen, and C. M. N. Tofts. Reactive, generative, and stratified models of probabilistic processes. In *Proc. LICS 1990*, pages 130–141. IEEE Computer Society, 1990.
29. J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proc. POPL 2012*, pages 427–440. ACM, 2012.