Safe Couplings: Coupled Refinement Types

LISA VASILENKO, IMDEA Software Institute, Spain
NIKI VAZOU, IMDEA Software Institute, Spain
GILLES BARTHE, MPI-SP, Germany IMDEA Software Institute, Spain

We enhance refinement types with mechanisms to reason about relational properties of probabilistic computations. Our mechanisms, which are inspired from probabilistic couplings, are applicable to a rich set of probabilistic properties, including expected sensitivity, which ensures that the distance between outputs of two probabilistic computations can be controlled from the distance between their inputs. We implement our mechanisms in the type system of Liquid Haskell and we use them to formally verify Haskell implementations of two classic machine learning algorithms: Temporal Difference (TD) reinforcement learning and stochastic gradient descent (SGD). We formalize a fragment of our system for discrete distributions and we prove soundness with respect to a set-theoretical semantics.

Additional Key Words and Phrases: refinement types, relational types, program verification, Haskell

1 INTRODUCTION

Refinement types provide an appealing mechanism for proving program properties in executable programming languages (including Haskell [Vazou et al. 2014b], Scala [Hamza et al. 2019], and F* [Swamy et al. 2016]). They have been used to good effect for reasoning about functional correctness and termination [Vazou et al. 2014a], resource analysis [Handley et al. 2019], security policies [Lehmann et al. 2021], and other properties of large developments.

However, refinement types do not provide support for reasoning about relational and hyper properties. The main difference between trace properties, which are the usual target of refinement type systems, and relational and hyper properties is that the latter reason about pairs of traces and sets of traces. This generalization allows to account for a wide variety of security, privacy, and robustness properties.

One natural approach to support relational reasoning is to use relational type systems, as proposed for instance in [Barthe et al. 2014; Maillard et al. 2020]. These type systems are similar to classic refinement type systems, but reason about relational assertions. The latter are interpreted over pairs of (typed) values, and therefore capture relational properties in a natural way. Relatonial refinement type systems retain the feel of refinement types and are particularly effective when reasoning about two executions of the same program, or two programs that follow the same control-flow.

Unfortunately, relational refinement types offer limited support to reason about programs with diverging control-flow. This is due to the fact that relational refinement types are syntax-directed, whereas many examples of relational program verification benefit from or even require non-syntax-directed reasoning. This is in particular the case for reasoning about program optimizations that restructure the control flow of the program, and about probabilistic programs, since their correctness or security proofs often use mathematical arguments that is not reflected in their syntax. Moreover, it remains a challenge to make relational refinement types practical, even in settings where syntax-directed reasoning suffices. This difficulty is perhaps best witnessed by prior work on relational cost [Çiçek et al. 2019]. In this work, Çiçek et al. [2019] develop BiRelCost, a state-of-the-art bi-directional type checker that compares the cost of two programs, or two program executions. In this setting, relational syntax-directed reasoning alternates with non-relational syntax-directed reasoning, in a way that the latter takes over whenever the two program executions

Authors' addresses: Lisa Vasilenko, elizaveta.vasilenko@imdea.org, IMDEA Software Institute, Madrid, Spain; Niki Vazou, niki.vazou@imdea.org, IMDEA Software Institute, Madrid, Spain; Gilles Barthe, gbarthe@mpi-sp.org, gilles.barthe@imdea.org, MPI-SP, Bochum, Germany and IMDEA Software Institute, Madrid, Spain.

no longer have the same control-flow. Unfortunately, controlling such alternations automatically by typing ultimately relies on intricate and partial heuristics. As a consequence, BiRelCost sacrifices predictability and generality, which are some key advantages of refinement types.

A principled approach to overcome the limitations of relational refinement types is to impose a separation between types and relational assertions. This approach, which is realized by Relational Higher-Order Logic (RHOL) [Aguirre et al. 2017], ensures maximal flexibility and expressiveness. However, the approach is not implemented, and as a consequence it remains an open question if Relational Higher-Order Logic can be made practical.

In this paper, we explore a middle ground approach that retains key benefits of refinement types. The crux of our approach is a carefully crafted interface for supporting relational reasoning *within* a unary refinement type system. Our approach is implemented atop Liquid Haskell [Vazou et al. 2014b] and inherits many of its essential features: first, our formal guarantees hold for Haskell programs and these programs can be executed using the existing runtime system and optimized libraries of Haskell. Second, verification is carried using a mature refinement type checker and should be familiar for users of Liquid Haskell. Third, the known techniques of Vazou et al. [2018] for encoding proofs manually remain applicable. Naturally, these benefits come at a cost: concretely, our proofs are less automated than proofs in classic Liquid Haskell. However, proofs remain reasonably short, even for relatively complex examples, demonstrating that our middle ground approach achieves predictable and practical verification, a combination that has not been achieved by any prior relational verification tool.

We realize our approach not only for classic higher-order programs, but also for probabilistic programs, an important class of programs that is used pervasively in cryptography, privacy, machine learning and many other areas. In addition to their numerous applications, probabilistic algorithms are an interesting class of programs to consider in their own right, because they often have intricate specifications and complex proofs. In particular, many properties of interest of probabilistic programs are quantitative, i.e. they reason about probabilities or expectations — or in a relational setting, about differences between probabilities or expectations. Although such forms of quantitative reasoning are *seemingly* out of reach of SMT-based verification, prior work has shown that relational verification of probabilistic programs can be achieved using *probabilistic couplings* [Barthe and Hsu 2020; Lindvall 2002; Thorisson 2000; Villani 2008]. We formalize the main tools from coupling-based reasoning in our framework, and illustrate how these tools can be used to verify two classic examples of probabilistic programs from machine learning. The first example is Temporal Difference (TD) reinforcement learning, for which we show rapid convergence to a stationary distribution independently of its initial input. The second example is Stochastic Gradient Descent (SGD), for which we show algorithmic stability— a classic machine learning property ensuring that a supervised machine learning algorithm generalizes well and does not overfit with respect to its training set. Both properties are captured in our system as instances of *expected sensitivity*, i.e. they upper bound the expected distance between two outputs of the program as a function of the distance between the corresponding inputs. However, both examples use distinct proof tools: the first example is verified using classic techniques from probabilistic couplings. In contrast, this second example uses a more elaborate, quantitative, form of probabilistic couplings which embeds reasoning about the Kantorovich distance between two distributions. Thus, the two examples showcase the different components of our system.

Summary of contributions. Our contributions are the following:

- We define a probabilistic relational refinement type system and encode it into the unary types of Liquid Haskell (§ 3). We choose Liquid Haskell as a mature refinement type checker, but our methodology can be used to encode any relational to any unary refinement type system.

- We use our system to prove two case studies from the literature: TD (§ 4.1) and SGD (§ 4.2).
- We prove soundness of our type system with respect to a denotational semantics (§ 5).

We start with an overview of our approach (§ 2) and conclude with related work (§ 6) and future work (§ 7).

2 OVERVIEW

We start with an overview of our system that uses (unary) refinement types to machine check relational properties of probabilistic, executable programs. First (§ 2.1) we introduce the `PrM` probabilistic monad and our `bins` running example. Next, we encode (§ 2.2) and formally prove (§ 2.3) a relational specification for the returned values of `bins` by axiomatizing probabilistic relational logic as refinement type assumptions. Finally, we follow a similar methodology to encode (§ 2.4) and prove (§ 2.5) a relational property about `bins` distance.

2.1 The Probability Distribution

The common way to implement probability distributions in Haskell is to use a probability monad, see for instance [Ramsey and Pfeffer 2002]. Therefore, our framework is set up as a verification wrapper around any Haskell library that supports a monadic implementation of probabilities. In order to execute our implementations, we wrapped the `probability` library[1]; however, our proofs are independent on the choice of the library, and only requires the existence of some type `PrM` that implements the standard interface for a probabilistic monad. This includes `pure` and `bind` and constructors for bernoulli, choice, and uniform distributions.

```
type PrM a = ... -- defined in Sec 3 using an existing probabilistic Haskell library

{-@ type Prob = {p:Double | 0 ≤ p ≤ 1}                           @-}
{-@ pure      :: a → PrM a                                       @-}
{-@ (>>=)     :: PrM a → (a → PrM b) → PrM b                     @-}

{-@ bernoulli :: Prob → PrM {v:Integer | v == 0 || v == 1}  @-}
{-@ choice    :: Prob → PrM a → PrM a → PrM a                    @-}
{-@ unif      :: {xs:[a] | 0 < len xs} → PrM a                   @-}
```

We define the Haskell probability monad `PrM` using an interface of an existing library (§ 3). We use the notation `{-@ ... @-}` to define refinement types and refinement type specifications. That is, the `Prob` type is a Haskell `Double`, refined to be between `0` and `1`. The specifications for the monadic `pure` and `>>=` are standard. The `bernoulli` function takes an input a probability `p`, *i.e.* a `Double` between `0` and `1`, and returns `1` with probability `p`, otherwise `0`. The function `choice p d1 d2` returns the distribution `d1` with probability `p`, otherwise `d2`. Finally, `unif` takes an input a non-empty list and returns one of its elements uniformly at random. All these functions are executed using the underlying Haskell implementation, but are left as uninterpreted (later § 3.4 and § 3.5 axiomatized) in our logic.

Probabilistic Programming: The Bins Example. Using the above interface, we can define (and execute) probabilistic programs. For example below we define the `bins` program that models a simple balls and bins process. In this process, `n` balls are thrown into a bin; in each throw, there is a probability that the ball lands outside the bin. The throws are independent and the probability to send any ball in the bin is `p`. The result of the process is the number of balls that lands into the bin. We model the process using the `bernoulli` distribution.

[1] We used the probabilistic functional programming library https://hackage.haskell.org/package/probability-0.2.7

```
{-@ type Nat  = {n:Integer | 0 ≤ n }           @-}

{-@ bins :: n:Nat → Prob → PrM {r:Nat | r ≤ n} @-}
bins 0 _ = pure 0
bins n p = do x ← bins (n - 1) p
              y ← bernoulli p
              pure (x + y)
```

We can use standard refinement types to verify various *unary* properties of the `bins` function. In particular, Liquid Haskell will use SMT automation to easily verify that `bins` terminates (because the recursive call occurs on smaller `n`). One can also prove that the result is always a natural number (as specified by the refined signature) and that it is not greater than `n`.

Using the theorem proving capabilities of Liquid Haskell [Vazou et al. 2017], we can construct extrinsic proofs that validate probabilistic, unary properties of `bins`. For example, we can define `expect f e` to be the expected value of `e`, for some function `f` that turns the values of `e` to booleans:

```
{-@ expect :: (a → Double) → PrM a → Double    @-}
{-@ natToD :: n:Nat → {d:Double | d = to_real n} @-}
```

We can extrinsically prove that `expect natToD (bins n p) = n * p`, assuming that the expectation is linear and `expect naToD (bernoulli p) = p`. Note that the Haskell `Double` is represented, by Liquid Haskell, as `real` in SMT, so the function `natToD` converts natural to double numbers while its specification ensures that the value is not changed. Next, we see how to construct extrinsic proofs that establish relational properties.

2.2 Relational Specifications & Lifting

A first relational property of interest is stochastic dominance, a classic property that defines when a real-valued probabilistic process is better then another. Informally, a real-valued probabilistic process is better than another if it always outputs a "higher value" w.r.t. the usual order on real numbers. Interestingly, the intuitive notion of "higher value " is formally defined over two random variables, which makes the definition of stochastic dominance non-trivial. Fortunately, stochastic dominance can be characterized using probabilistic couplings, a classic tool to reason about Markov chains. Informally, couplings are probabilistic equivalent of cartesian products, and can be used to lift relational properties to distributions. Informally, the lifting of a relational property is defined as follows: two distributions satisfy the lifting of a property p if there exists a coupling of the two distributions such that p holds surely, i.e. with probability 1, in this coupling. The formal definition can be found e.g. in [Barthe and Hsu 2020]. For our purposes, it suffices to assume an operator $\diamond$ that transforms a relation over two types into a relation over probabilistic distributions over these two types:

```
(⋄) :: (a → b → Bool) → PrM a → PrM b → Bool
```

The operator `(⋄)` supports compositional reasoning via two axioms: `pureAxiom` states that Dirac distributions of elements related by `p` are related by the lifted relation `⋄ p` and `bindAxiom` states that lifted relations are preserved by monadic composition. These axioms, which are expressed below using refinement type signatures, conveniently eschew probabilistic reasoning, and open the possibility of carrying SMT-based verification:

```
{-@ assume pureAxiom :: p:(a → b → Bool) → x_l:a → x_r:b → {p x_l x_r}
                     → {⋄ p (pure x_l) (pure x_r)} @-}

{-@ assume bindAxiom :: p:(b → b → Bool) → q:(a → a → Bool)
```

```
    → e_l:PrM a → f_l:(a → PrM b) → e_r:PrM a → f_r:(a → PrM b)
    → {⋄ q e_l e_r}
    → (x_l:a → x_r:{a | q x_l x_r} → {⋄ p (f_l x_l) (f_r x_r)})
    → {⋄ p (e_l >>= f_l) (e_r >>= f_r)} @-}
```

Both `pureAxiom` and `bindAxiom` are defined as Haskell functions that return `()`, but their refinement type signatures encode the desired axioms. The **assume** keyword prevents refinement type checking, since the function definitions do not actually inhabit their type specifications. `pureAxiom` states that forall `p`, x_l x_r, if `p` x_l x_r, then ⋄ `p (pure` x_l`) (pure` x_r`)`. The type `{p` x_l x_r`}` is an abbreviation of `{v:() | p` x_l x_r`}`. In general, we can write `{q}` instead of `{v:a | q}`, when `v` does not appear in `q`. Similarly, the `bindAxiom` ensures ⋄ `p (`e_l `>>=` f_l`) (`e_r `>>=` f_r`)` when it is provided a proof that ⋄ `q` e_l e_r and a (higher-order) proof that forall x_l and x_r such that `q` x_l x_r, ⋄ `p (`f_l x_l`) (`f_r x_r`)` holds. In § 3 we discuss the implementation of our library that includes these two assumptions, while later (§ 5) we develop a formalisation that justifies these assumptions, concretly, the `pureAxiom` and `bindAxiom` are respectively encoded in the rules T-Ret and T-Bind of Figure 7.

In our `bins` running example, we are interested to show that for two competing throwers sending the same number of balls into bins, the more gifted thrower, i.e. the thrower with a higher probability to send balls into the bins, will have a higher count. More formally, our goal is to show that if `p ≤ q` then ⋄ `(≤) (bins n p) (bins n q)`, from which one can conclude that `expect naToD (bins n p) ≤ expect naToD (bins n q)` by a simple property of couplings. In our syntax, we formalize our goal as:

```
{-@ binsSpec :: p:Prob → {q:Prob|p≤q} → n:Nat → {⋄ (≤) (bins n p) (bins n q)} @-}
```

In the next paragraph, we use relational refinement types to establish this goal.

2.3 Relational Proofs

Following Handley et al. [2019], relational proofs are (Haskell) inhabitants of the refinement type specification that expresses the relational specification. Such proofs rely on assumptions about relational properties of the probabilistic primitives. For example, `bins` is using `bernoulli`, thus the proof of `binsSpec` relies on the assumption below, which captures `bernoulli`'s relational specification.

```
{-@ assume bernoulliAxiom :: p:Prob → {q:Prob | p ≤ q}
                           →  {⋄ (≤) (bernoulli p) (bernoulli q)} @-}
```

The specification (formalized in rule T-Bern of § 5; Figure 7) states that `bernoulli q` stochastically dominates `bernoulli p` if `p ≤ q`. In our current implementation, this specification is taken as an axiom, although it would possible to establish this specification from first principles, by making the definition of lifting explicit for finitely supported distributions.

Using the `bernoulliAxiom` we prove `binsSpec` following the structure of `bins` definition:

```
binsSpec p q 0
  = pureAxiom (≤) 0 0 ()
binsSpec p q n
  = bindAxiom (≤) (≤) (bernoulli p) (bins1 n p) (bernoulli q) (bins1 n q)
             (bernoulliAxiom p q) (\x_l x_r →
    bindAxiom (≤) (≤) (bins (n-1) p) (bins2 x_l) (bins (n-1) q) (bins2 x_r)
             (binsSpec p q (n-1)) (\y_l y_r →
    pureAxiom (≤) (y_l + x_l) (y_r + x_r) ()
  where
  bins1 n p x = bind (bins (n-1) p) (bins2 x)
```

```
  bins2 x y   = pure (y + x)
```

The proof, as `bins`, is inductively defined on `n`. In the base case, we call `pureAxiom (≤) 0 0 ()` to get `◇ (≤) (pure 0) (pure 0)`, which concludes the proof, since `bins 0 p == pure 0`. Such rewrite steps are automated by Liquid Haskell's logical evaluation strategy (namely PLE [Vazou et al. 2017]). Further, our proofs are automated by SMT arithmetic. For example, `pureAxiom`'s last argument needs to prove that `0 ≤ 0`, which is trivially shown by `()` and SMT automation. The inductive case starts with a call to `bindAxiom`, again following `bins` inductive definition. There are two interesting points here. First, the `bins` definition binds `bernoulli p` to a continuation. Since `bindAxiom`'s 4th and 6th arguments require to explicitely pass these continuations, we use a `where` clause to name the continuation `bins1`. Second, `bindAxiom` requires two proof terms. The first one should show that `◇ (≤) (bernoulli p) (bernoulli q)`, which is shown by the `bernoulliAxiom`. The second one, should show `◇ (≤) (bins1 n p` x_l`) (bins1 n q` x_r`)`, for all x_l and x_r that satisfy $x_l \le x_r$. We construct such a proof term using a lambda[2]. Since the `bins` definition is using another `bind`, the proof again calls `bindAxiom` with similar arguments. In the last step, `bins` calls `pure`, thus the proof calls `pureAxiom`, whose proof argument is again `()`, *i.e.* automated by rewriting and SMT.

2.4 Quantitative Specifications and Kantorovich lifting

So far, we have established that if `p ≤ q` then `bins n q` stochastically dominates `bins n p` and thus `expect naToD (bins n p) ≤ expect naToD (bins n q)`. However, our specification does not provide quantitative information on `expect naToD (bins n q) - expect naToD (bins n p)`. In fact, one can use simple properties of expectation to show that if `p ≤ q` then we have `expect natToD (bins n q) - expect natToD (bins n p) ≤ n * (q-p)`, where `natToD` is just the cast defined in § 2.1. Unfortunately, one cannot prove this fact using the previous approach based on lifting. The solution is to use a richer notion of lifting, that allows to reason about quantitative properties, and in particular about the expected distance between two distributions. In order to accommodate such reasoning, one considers a richer setting where each type is equipped with a distance `dist`. These distances are defined inductively on the structure of types; for distribution types, they use the so-called Kantorovich metric[Deng 2015; Villani 2009], which lifts a distance over some types to a distance over its corresponding distribution type:

```
{-@ dist :: Dist a → (a → a → {d:Double | 0 ≤ d}) @-}
{-@ kant :: Dist a → Dist (PrM a)                   @-}

{-@ kdist :: Dist a → PrM a → PrM a → {d:Double | 0 ≤ d} @-}
kdist d = dist (kant d)
```

The `kant` function turns a distance into Kantorovich and `kdist` simply composes `kant` with `dist`. The formal definition of the Kantorovich metric can be found for instance in Deng [2015]; for this work, it suffices that the Kantorovich metric is also based on couplings and also lends itself to compositional reasoning. For instance, the following axioms (corresponding to the rules T-Ret and T-Bind § 5; Fig. 7) are valid:

```
{-@ assume pureDist :: d:Dist a
                    → xl:a → xr:a
                    → { kdist d (pure xl) (pure xr) = dist d xl xr } @-}

{-@ assume bindDist :: d:Dist b
```

[2] The actual proof is using a named function with explicit type specification, since Liquid Haskell does not infer preconditions, but for space, here we use lambdas.

```
                    → m:Double → p:(a → a → Bool)
                    → f_l:(a → PrM b) → e_l:PrM a
                    → f_r:(a → PrM b) → e_r:{PrM a | ◇ p e_l e_r }
                    → (x_l:a→{x_r:a | p x_l x_r} → { kdist d (f_l x_l) (f_r x_r) ≤ m})
                    → { kdist d (e_l >>= f_l) (e_r >>= f_r) ≤ m }
```

The first axiom states that the Kantorovich distance of two Dirac distributions is the distance of their generating element. The second axiom upper bounds the Kantorovich distance between two monadic compositions by the maximal Kantorovich distance between f_l x_l and f_r x_r for all x_l and x_r related by an intermediate assertion p, such that e_l and e_r are related by the lifting of p. We emphasize that the bind rule for Kantorovich distance is more intricate than the corresponding rules for lifting, and that the "obvious" compositional rule that adds distance between the es and the fs would not be sound, as discussed in § 5.

Then, the distance spec of bins is:

```
{-@ binsDist :: p:Prob → {q:Prob| p ≤ q} → n:Nat
             → { kdist distNat (bins n p) (bins n q) ≤ n * (q - p) } @-}

{-@ distNat :: Dist Nat @-}
```

That is, for each probabilities p and q, so that p ≤ q and each natural number n the Kantorovich distance between bins n p and bins n q is bounded by n * (q - p). Since bins returns natural numbers, the distance is given by distNat that defines the distance metric on natural numbers (§ 3.2). Finally, Haskell's Doubles are represented in the SMT logic are SMT reals, *i.e.* there is no reasoning about overflows and precision loss. The binsDist specification is well sorted in Z3, since Z3 automatically converts between int and reals.

2.5 Distance Proofs

Unlike the proof of § 2.3, the proof of binsDist is not syntax directed. On the contrary, it requires the construction of a "ghost" probabilistic function that splits the distance between bins n p and bins n q. We call this function ghost bins (gbins) and define it as follows:

```
gbins :: Nat → Prob → Prob → PrM Nat
gbins n p q = do x ← bins (n-1) p
                 y ← bernoulli q
                 pure (x + y)
```

The ghost gbins n p q adds bins with probability argument p and bernoulli with probability argument q, thus connecting bins n p and bins n q.

Using mostly syntax-directed proofs, we establish the following two lemmata:

```
{-@ binsDistL :: p:Prob → {q:Prob| p ≤ q} → n:Nat
              → { kdist distNat (bins n p) (gbins n p q) ≤ q - p} @-}

{-@ binsDistR :: p:Prob → {q:Prob| p ≤ q} → n:Nat
              → { kdist distNat (gbins n p q) (bins n q) ≤ (n-1) * (q - p)} @-}
```

Both proofs use the distance axioms of § 2.4. The proof of binsDistR (inductively) calls binsDist, while the proof of binsDistL requires the below axiom for bernoulli's distance.

```
{-@ assume bernoulliDist :: d:Dist Nat
                         → p:Prob → {q:Prob ! p ≤ q}
                         → { kdist d (bernoulli p) (bernoulli q) ≤ p - q } @-}
```

	module name	LoC
1.	Data.Dist	155
2.	Monad.PrM	93
3.	Monad.PrM.TCB.Axioms	43
4.	Monad.PrM.TCB.Dist	62
5.	Monad.PrM.Theorems	100
	Total	453

Table 1. Summary of safe-coupling library. **LoC** is commented lines of Haskell Code.

That is, the expected distance of two bernoulli distributions, is bounded by the distance of the bernoulli's arguments (as formalized in rule T-Bern of § 5; Figure 7).

We prove `binsDist` combining `binsDistL` and `binsDistR` with triangular inequality, which, as explained in § 3.2 is a property (concretely a field) of the `Dist` type. The proof goes by induction:

```
binsDist p q 0
  =   pureDist distNat 0 0
binsDist p q n
  =   dist d (bins n p) (bins n q)                            -- Step 1
      ? trinequality d (bins n p) (gbins n p q) (bins n p)  -- Defined in Sec 3.2
  =<= dist d (bins n p) (gbins n p q) + dist d (gbins n p q) (bins n q) -- Step 2
      ? binsDistL n p q
      ? binsDistR n p q
  =<= (q - p) + (n-1) * (q - p)                               -- Step 3
  =<= n * (q - p)                                             -- Step 4
  *** QED
  where d = kant distNat
```

The base case is merely an application of the `pureDist` axiom. In the inductive case, we use the (in)equational reasoning proof combinators of [Handley et al. 2019]: `l ? j =<= r` ensures `l` is not greater than `r` using the justification `? j` which is optional and `*** QED` concludes the proof. The first step is to start from the distance between `bins n p` and `bins n q`. Applying triangular inequality, in the second step, we split the distance using `gbins`. Next, we use the two helper lemmata to bound each of the two distances. Finally, using trivial (SMT-automated) arithmetic, we get the desired bound. We note that the lemma `binsDistR` inductively calls `binsDist` on a smaller `n`, so our proof is inductive, while Liquid Haskell is ensuring (mutually recursive) termination.

The `binsDist` example showcases that our framework can be used to machine-check sophisticated proofs, that require ghost proof objects. To evaluate the expressiveness of our framework, we used it to prove two classic properties of machine learning, probabilistic programs: convergence of TD (§ 4.1) and stability of SGD (§ 4.2).

3 IMPLEMENTATION OF `safe-coupling`

In this section we present `safe-coupling`, a Haskell library that exports an interface for probabilistic (executable) programming and permits relational probabilistic verification using Liquid Haskell (§ 3.1). Table 1 summarizes the five main modules of `safe-coupling` that *define* distance (§ 3.2) and the probabilistic monad (§ 3.3), *assume* relational (§ 3.4) and distance (§ 3.5) axioms, and *prove* relational theorems (§ 3.6). In section (§ 5), we formally justify the assumptions made by `safe-coupling`.

3.1 Liquid Haskell Preliminaries

Verification with Refinement Types. Refinement types are used to do “light” verification. For example `max` of two probabilities (*i.e.* doubles between `0` and `1`) is also a probability:

```
max :: Prob → Prob → Prob
max x y = if x ≤ y then y else x
```

To achieve SMT decidable and automatic verification, a refinement type systems clearly separate the executable code (here, the Haskell definitions) from the logic (here, the predicates on the refinement types). In the `max` example, every caller of `max` knows the type signature (*i.e.* that the result is also `Prob`), but not its implementation (*i.e.* that it returns one of its arguments). This way verification is modular, but, by default, the Haskell function does not exist in the logical fragment.

User Defined Functions in the Logic. An attempt to refer to user defined definitions in the refinement predicates, will lead to an undefined error. For instance, below we define a proof that `∀ x y . x ≤ max x y` as a function whose arguments encode the quantified `x` and `y` and its result is a unit refined with the desired predicate. (Notation: we simplify `{v:()|p}` to `{p}`.)

```
{-@ not_found :: x:Prob → y:Prob → {x ≤ max x y} @-} -- ERROR: max is unknown
not_found _ _ = ()
```

Since refinement types clearly separate the executable code from the logic, the above specification leads to an error: `max` is unknown to the logic. There are two ways to lift executable definitions in the logic: 1) axiomatization and 2) reflection.

1) Axiomatization of `max` defines a logical *uninterpreted* function that has the same refinement type and returns the same result as the executable `max`, but `max`’s definition is not available in the logic. For example, one can use axiomatization to show `0 ≤ max x y ≤ 1` but not that `x ≤ max x y`:

```
{-@ axiomatize max @-}
{-@ ok    :: x:Prob → y:Prob → {0 ≤ max x y} @-} -- max's specification is known
{-@ error :: x:Prob → y:Prob → {x ≤ max x y} @-} -- max's definition is unknown
```

2) Reflection of `max` defines a `max` function in the logic and further makes its definition available:

```
{-@ reflect max @-}
{-@ theorem :: x:Prob → y:Prob → {x ≤ max x y} @-} -- max's definition is known
theorem _ _ = ()
```

Reflection of executable functions permits “deep verification”, *i.e.* reasoning about sophisticated properties like Kantorovich distance of two probabilistic runs. Yet, for decidable refinement type checking, this reasoning requires explicit (user-provided) proofs. Importantly (but not surprisingly) functions can only get reflected, when their definitions consist only of reflected or axiomatized functions. For example, in our setting, functions imported from an unverified Haskell library cannot get reflected.

Proof Terms. Liquid Haskell is using refinement types to encode theorems []. The definitions of these functions can be unit (then the theorem is trivially proved by SMT and existing automation) or can contain inductive calls and combine other proof terms using proof combinators. Usually, such definitions do not have runtime meaning: if executed they will not produce any interesting result. But, since Liquid Haskell is checking for totality and completeness of these definitions they encode mathematical proofs.

We call *theorem* a refinement type specification that has a proof term, like the `theorem` for `max` defined above. As an alternative, Liquid Haskell’s **`assume`** keyword let’s you assume *axioms* (refinement types) that one cannot prove. We use such axioms to encode properties of axiomatized functions. For example, when max is axiomatized we can define two axioms that describe its behavior.

```
{-@ axiomatize max @-}
{-@ assume max1 :: x:Prob → y:Prob → {x ≤ max x y} @-}
{-@ assume max2 :: x:Prob → y:Prob → {y ≤ max x y} @-}
```

Since max is axiomatized, its definition is not available in the logic, so none of the above axioms can be proved.

3.2 Data.Dist: Definition of Distance

We used Liquid Haskell to define the refined data type Dist a that encodes a metric, as follows.

```
{-@ data Dist a = Dist {
       dist         :: a → a → {v:Double | 0.0 ≤ v }
     , identity     :: x:a → {dist x x == 0}
     , symmetry     :: x:a → y:a → {dist x y = dist x y}
     , trinequality :: x:a → y:a → z:a → {dist x z ≤ dist x y + dist y z}
     } @-}
```

The first field of Dist contains the distance function dist on any expressions of type a: it is a function that given two arguments of type a returns a non negative Double. The next three fields capture the metric's axioms for identity of indiscernibles, symmetry, and triangle inequality.

In this module, we further defined the distance metric on doubles (distDouble) and natural numbers (distNat):

```
distDouble :: Dist Double
distNat    :: Dist Nat
```

These definitions contain both the definition of the function dist and the proofs of the metric axioms on the concrete distance. Further, we define a function that computes distance between two same-length lists of a given a Dist a:

```
{-@ dList :: Dist a → xs:[a] → ys:{[a] | len xs = len ys} → {v:Double | 0 ≤ v} @-}
```

We proved all the metric axioms of dList, yet, since there exists the same-length dependency it is not possible to define a (well-typed) Dist [a] function. Still, we use the above definitions to compute distance between natural numbers, doubles and their lists:

```
assert (dist  distDouble 42.0 40.0   == 2.0)
assert (dList distDouble [42] [40.0] == 2.0)
```

3.3 Monad.PrM: Definition of the Probabilistic Monad

The module Monad.PrM is essentially a wrapper around an executable Haskell probability monad. We chose the probabilistic functional library probability, due to its clear interface. Our development uses probability to execute (and test) our probabilistic programs, but our mechanized proofs do not depend on it and could use alternative libraries (*e.g.*monad-bayes [Scibior et al. 2015]).

The underlying probability library (here prefixed as PLib) exports the type T prop a, that essentially maps each a to a probability prop and defines the monadic and probabilistic primitives.

The data type. Our probability monad type instantiates prop to the probability type Prop.

```
type PrM a = PLib.T Prop a
```

```
{-@ assume unifDist :: d:Dist a → xsₗ:[a] → xsᵣ:{[a] | xsₗ == xsᵣ}
                    → { kdist d (unif xsₗ) (unif xsᵣ) == 0 } @-}

{-@ assume choiceDist :: d:Dist a → p:Prob → eₗ:PrM a → uₗ:PrM a
                      → q:{Prob | p = q } → eᵣ:PrM a → uᵣ:PrM a
                      → { kdist d (choice p eₗ uₗ) (choice q eᵣ uᵣ) ≤
                          p * (kdist d eₗ eᵣ) + (1.0 - p) * (kdist d uₗ uᵣ)} @-}

{-@ assume pureBindDist :: da:Dist a → db:Dist b → m:Double
                        → fₗ:(a → b) → eₗ:PrM a
                        → fᵣ:(a → b) → eᵣ:PrM a
                        → (xₗ:a→xᵣ:a→{dist db (fₗ xₗ) (fᵣ xᵣ)-dist da xₗ xᵣ ≤ m})
                        → { kdist db (eₗ >>= (ppure . fₗ)) (eᵣ >>= (ppure . fᵣ)) ≤
                            kdist da eₗ eᵣ + m } @-}
```

Fig. 1. Distance Axioms of safe-coupling. Encoding rules T-Unif, T-Choice, and T-Bind-Ret of fig. 7.

Axiomatized primitives. For each monadic (>>= and pure) and probabilistic primitive (bernoulli, uniform, and choice) operations we used the probability functions to define the Haskell (executable) function and axiomatized (as described in § 3.1) them in the logic. For example, pure is defined as follows

```
{-@ axiomatize pure :: x:a → PrM {v:a| v = x} @-}
pure x = PLib.pure x
```

We followed this encoding for practical reasons: since PLib is not itself verified with Liquid Haskell (which is a challenging future work) its definitions are not available in the logic. Yet, this encoding leaves us the flexibility to axiomatize the primitives as desired (§ 3.4 and 3.5).

Reflected functions. Using the axiomatized primitives we defined further probabilistic functions, such as mapM.

```
{-@ reflect mapM :: (a → PrM b) → [a] → PrM [b] @-}
```

Since mapM is reflected, its definition (which is standard) is available in the logic and used to prove (relational) theorems about mapM (§ 3.6).

3.4 TCB.Axioms: Assumption of Relational Axioms

The first trusted computing base (TCB) of safe-coupling is called Axioms and contains the relational specification of each axiomatized primitive. It provides the Haskell axiomatized function (⋄) and uses it to encode the relational axioms for pure, (>>=), and bernoulli, as presented in § 2.2.

3.5 TCB.Dist: Assumption of Distance Specifications

The second trusted computing base (TCB) of safe-coupling is called Dist and contains the distance specification of each axiomatized primitive. It provides the Haskell function kant that (like ⋄) is axiomatized in the logic and for each distance on a returns the kantorovich distance on distributions of a and (as defined in § 2.4) kdist that simply composes kant with dist:

```
{-@ axiomatize kant :: Dist a → Dist (PrM a)              @-}
{-@ kdist :: Dist a → PrM a → PrM a → {d:Double | 0 ≤ d} @-}
```

This module provides the distance axioms for the axiomatized primitives. In § 2.4 we presented the axiomatization for bind (`bindDist`), pure (`pureDist`), and bernoulli (`bernoulliDist`). We assume three more axioms presented in Figure 1. First, `unifDist`, the distance axiom of `uniform`, states that the kantorovich distance between two uniform distributions is zero, when the sampling input lists are equal. Second, `choiceDist`, the distance axiom for `choice`, states that the Kantorovich distance of two choice expressions `choice p` e_l u_l and `choice q` e_r u_r is `p` times the kantorovich distance of e_l e_r and `1 - p` times the kantorovich distance of u_l u_r, when `p = q`. Finally, `pureBindDist` is a distance axiom for bind. For soundness reasons discussed in § 5, the rule is stated only for `bind` expressions whose second argument is (the monadic lifting of) a pure function. The axiom requires that the pure function f_l and f_r make the distance between two values grow by at most `m`; in order words, the distance between f_l x_l and f_r x_r cannot exceed the distance between x_l and x_r and some fixed constant `m`. Under this assumption, the Kantorovich distance between `(`e_l `>>= (ppure .` f_l`))` and `(`e_r `>>= (ppure .` f_r`))` is bounded by the Kantorovich distance between e_l and e_r plus `m`. This specialized axiom provide a means to upper bound the Kantorovich distance between two `bind` expressions as a function of the Kantorovich distance of their first arguments, and is instrumental for our case studies.

3.6 Theorems: Proof of Relational Properties

This module proves common theorems using our assumed TCB and the defined functions of `Monad.PrM`. Concretely, it provides a simplified version of the `bindAxiom` when the two bind arguments form a bijectional coupling and the a relational specification for the monadic map.

All the properties on this module are proved. Next, we provide a formalism that justifies the assumptions of our two TCB modules.

4 CASE STUDIES

To evaluate `safe-coupling` we used it to verify two classic machine learning properties convergence of TD (§ 4.1) and stability of SGD (§ 4.2). § 4.3 summarizes our results.

4.1 Case Study I: Convergence of TD(0)

Our first case study proves convergence for TD(0), a classical algorithm for Reinforcement Learning.

4.1.1 Implementation of TD(0). In the standard reinforcement learning setting, an agent (*i.e.* the learning algorithm) repeatedly interacts with the environment, a Markov Decision Process (MDP) with state space `State` and set of actions `A`. At each step, the MDP reacts to the agent's action by drawing a new random state and a numeric reward according to a function `t :: State → PrM (State, Double)`. The current state `i` of the process is known to the learner, but the exact function `t` is not. Given black-box access to `t`, the goal of the learner is to find a policy map π `: State → A` from the state space to the best available action from `A` that maximizes the learner's expected reward over infinite time.

Figure 2 presents the implementation of TD(0), a Temporal Difference (TD) learning algorithm that estimates the value function `v :: State → Reward` of the MDP, *i.e.* the expected reward at each state if the agent were to repeatedly act according to some assumed policy π. For simplicity of verification, we defined `State` as `{s:Nat| s ≤ n}` and functions on `State` as lists of length `n`. The TD learner (*i.e.* `td0`) takes as input the number of iterations `n`, a transition function `t`, and an estimate of `v` and it `iterate`s through the `n` states. At each iteration `i`, the learner runs `sample` that draws a reward and transition `(j, r)` from the `i`th element of `t`. Then, the estimate `v j` is `update`d by incorporating the observed reward `r` and the estimated value `v j` of the new state. Estimated

```haskell
import Monad.PrM -- mapM defined here

td0 :: Int → [PrM (State, Reward)] → [Reward] → PrM [Reward]
td0 n t v = iterate n (act t) v

act :: [PrM (State, Reward)] → [Reward] → PrM [Reward]
act t v = mapM (sample t v) [0..length v]

sample :: [PrM (State, Reward)] → [Reward] → State → PrM Reward
sample t v i = do
  (j, r) ← t !! i
  pure (update v i j r)

iterate :: Int → (a → PrM a) → a → PrM a
iterate 0 _ x = pure x
iterate n f x = f x >>= iterate (n - 1) f

update :: [Reward] → State → State → Reward → Reward
update v i j r = (1 - α) * (v !! i) + α * (r + γ * v !! j)

type State  = Int
type Reward = Double
```

Fig. 2. Implementation of TD(0).

rewards in the future are reduced by a discount factor $\gamma \in [0, 1)$. Higher γ allows v to converge faster.

4.1.2 Convergence for TD(0). Our goal is to show that `td0` converges to a stationary distribution independently of its initial input. This can be achieved by proving that `td0` is contractive on v. One potential approach would be to prove that for `k = α * γ + (1-α)`,

```haskell
{-@ td0Goal :: n:Nat → t:[PrM (State, Reward)] → v_l:[Reward] → v_r:[Reward]
            → {kdist dList (td0 n v_l t) (td0 n v_r t) ≤ k^n * (dist dList v_l v_r)} @-}
```

Instead, wre prove a stronger property that `td0` is contractive for all possible outcomes (via lifting). To do so, first we defined a pure (to be lifted) predicate that bounds the distance (since Liquid Haskell does not permit lambdas in the refinements):

```haskell
bounded :: Dist a → Double → a → a → Bool
bounded d m v1 v2 = dist d v1 v2 ≤ m
```

Using `bounded` we define the `td0` specification as follows:

```haskell
{-@ td0Spec :: n:Nat → t:[PrM (State, Reward)] → v_l:[Reward] → v_r:[Reward] →
          {⋄ (bounded dList (k^n * (dist dList v_l v_r)) (td0 n v_l t) (td0 n v_r t)} @-}
```

The `td0Spec` specification implies our original `td0Goal`. This is because a bound property on all outcomes implies an average bound:

```haskell
{-@ bound :: d:Dist a → k:Double → e_l:PrM a → e_r:PrM a
          → {⋄ (\x_l x_r → dist d x_l x_r ≤ k) e_l e_r ⇒ kdist d e_l e_r ≤ k } @-}
```

```
type PDouble = {Double | 0 ≤ m}

{-@ iterateSpec :: m:PDouble → n:Nat → f:([Reward] → PrM [Reward])
      → (m:PDouble → xₗ:[Reward] → xᵣ:[Reward] →
         { bounded dList m xₗ xᵣ ⇒ ◇ (bounded dList (m * k)) (f xₗ) (f xᵣ)})
      → rₗ:[Reward] → rᵣ:[Reward]
      → { bounded dList m rₗ rᵣ ⇒ ◇ (bounded dList (m * kⁿ))
                                  (iterate n f rₗ) (iterate n f rᵣ)} @-}

{-@ actSpec :: m:PDouble → t:[PrM (State, Reward)] → vₗ:[Reward] → vᵣ:[Reward]
      → {bounded dList m vₗ vᵣ ⇒ ◇ (bounded dList (k * m)) (act t vₗ) (act t vᵣ)} @-}


{-@ sampleSpec :: m:PDouble → t:[PrM (State, Reward)] → vₗ:[Reward] → vᵣ:[Reward]
      → i:State
      → {bounded dList m vₗ vᵣ ⇒ ◇ (bounded distDouble (k * m))
                                 (sample t vₗ i) (sample t vᵣ i)} @-}

{-@ updateSpec :: vₗ:[Reward] → vᵣ:[Reward] → i:State → j:State → r:Reward
      → {distD (update vₗ i j r) (update vᵣ i j r) ≤
         k * max (distD (vₗ !! i) (vᵣ !! i)) (distD (vₗ !! j) (vᵣ !! j))} @-}
```

Fig. 3. Relational Lemmas for the `td0Spec` Proof; where `distD x y = dist distDouble x y`.

The reverse implication does not hold: two distributions can have a Kantorovich distance that is upper bounded by `k`, and do not satisfy the lifting of `bounded d k`. As a counterexample, assume e_l `= [(3,0.5), (77, 0.5)]` and e_r `= [(7,0.5), (75, 0.5)]`, then for `k = 3` and `d` the distance of natural numbers, the right side is true (*i.e.*`kdist distNat` e_l e_r ≤ 3), but the left side does not hold. Thus, and to our surprise, we could prove a stronger property than originally anticipated, and in a simpler system with plain, non-quantitative, liftings.

4.1.3 Proof of Convergence for TD(0). We proved `td0Spec` in 128 lines of (Liquid) Haskell code and, as summarized in table 2, we used five lemmas; one for each used function. We named each lemma by postfixing the name of the function with `Spec`. The specification `mapMSpec` comes from the `safe-coupling` library (§ 3.6), while the rest lemmas are presented in fig. 3.

The proof of each lemma, like `binsSpec` of § 2.3, is following the structure of the function definition. Concretely, `td0Spec` is proved by `iterateSpec`, using `actSpec` as the proof requirement; `iterateSpec` is proved by induction, using the pure and bind axioms; `actSpec` is proved by `mapMSpec`, using `sampleSpec` as the proof argument; `sampleSpec` is using the axioms and `updateSpec`, which is proved using the linearity and triangular inequality of the distance on doubles.

In short, the great challenge was to come up with the correct invariant for `td0Spec`, after which the proof follows the structure of the `td0`'s implementation.

4.2 Case Study II: Stability of SGD

Supervised machine learning algorithms are algorithms that aim to select the best fitting model from a class of parametric models by iteratively refining some initial parameter, based on some training set. A good measure of the quality of these algorithms is their, so called, generalization

```
import Monad.PrM
import Data.Derivative (grad)

{-@ sgd :: zs:{[DataPoint] | 2 ≤ length zs} → w0:Weight → αs:[StepSize]
        → f:(DataPoint → Weight → Double) → PrM Weight @-}
sgd _  w0 [] _ = pure w0
sgd zs w0 (α:αs) f
      = choice (1 / natToD (length zs))
          (pure (head zs) >>= sgdRec zs w0 αs f)
          (unif (tail zs) >>= sgdRec zs w0 αs f)
  where sgdRec zs w0 αs f z = do
          w ← sgd zs w0 αs f
          pure (update z α f w)

update :: DataPoint → StepSize → (DataPoint → Weight → Double)
       → Weight → Weight
update z α f w = w - α * grad (f z) w

{-@ type StepSize = {v:Double | 0 ≤ v} @-}
type StepSize     = Double
type DataPoint    = (Double, Double)
type Weight       = Double
```

Fig. 4. Implementation of SGD.

error, which measures how they perform on previously unseen data. One sufficient condition for an algorithm to have a controlled generalization error is to be algorithmically stable [Bousquet and Elisseeff 2002]. Informally, a supervised machine learning algorithm is algorithmically stable if the output of the algorithm is not overly dependent on any single element in the training set. More formally, a supervised machine learning algorithm is ϵ-stable if the Kantorovich distance between the parameters obtained by running the algorithm on the same initial parameter and two adjacent training sets (i.e. training sets that differ in a single element) is upper bounded by ϵ. In this case study, we show that Stochastic Gradient Descent, the *de facto* backpropagation algorithm for deep learning, is algorithmically stable. The proof follows the steps of [Hardt et al. 2016].

4.2.1 Implementation of SGD. Figure 4 presents our `sgd` implemention, a variant of Stochastic Gradient Descent. The algorithm takes as input a training set `zs`, modeled as a list of data points, an initial weight`w0`, and a list of learning step sizes αs and loss function `f`. In the general setting, the loss function `f` in the definition of SGD would be a vector function $Z \times \mathbb{R}^d \to \mathbb{R}$, where Z is the data set and $\mathbb{R}^d$ is the weight space. For the sake of simplicity, in our implementation `Weight` is defined as a single value of type `Double` which corresponds to $d = 1$.

The function `sgd` recursively computes a sequence of so-called weights (or classifiers) starting from the initial parameter `w0`, by updating at each step the current weight `w` into `update z α f w`, where `z` is sampled uniformly from data set `zs` and the learning step α represents the influence of each iteration in the final result.

In our implementation we unfold the definition of uniform sampling based on the operator `choice`. With probability `1 / length zs` we sample the head element of the data set. Otherwise, we sample z from the tail part. In both cases, we proceed by updating the result of a recursive call with respect to z.

To implement `update`, we assume a partial function `grad :: (Weight → Double) → Weight → Double` which computes the gradient of `f`. Our proof does not rely on `grad`'s definition, therefore it can be imported from any automatic differentiation library.

4.2.2 Stability of SGD. The stability statement bounds Kantorovich distance of two runs of `sgd` on data sets zs_l and zs_r which differ in exactly one element. Without loss of generality, we assume that the differing element is the first. We express the stability statement as a refinement type specification, as follows.

```
{-@ sgdDist :: d:Dist Double → αs:[StepSize] → f:(DataPoint → Weight → Double)
             → zsₗ:[DataPoint] → zsᵣ:{[DataPoint] | tail zsₗ = tail zsᵣ}
             → wsₗ:Weight → wsᵣ:Weight
             → { kdist d (sgd zsₗ wsₗ αs f) (sgd zsᵣ wsᵣ αs f)
                 ≤ dist d wsₗ wsᵣ + estab (length zsₗ) αs} @-}

estab :: Nat → [StepSize] → Double
estab l αs = 2.0 * lip / natToD l * sum αs
```

That is, the Kantorovich distance between two runs of `sgd` is bounded by the distance of the different weights plus an ϵ, defined by the helper function `estab`. Note that `estab` does not depend on the different inputs of `sgd`, but depends on `lip`, which represents the Lipschitz constant L and is axiomatized in our proof.

4.2.3 Assumptions. Proofs of algorithmic stability are traditionally based on strong assumptions on the loss function $f : Z \times \mathbb{R}^d \to \mathbb{R}$, namely:

(1) *L-Lipschitz:* $\|f(z, w_1) - f(z, w_2)\| \leq L\|w_1 - w_2\|$;
(2) *Convex:* $f(z, w_1) \geq f(z, w_2) + \langle \nabla f(z, w_2), w_1 - w_2 \rangle$; and
(3) *β-smooth:* $\|\nabla f(z, w_1) - \nabla f(z, w_2)\| \leq \beta\|w_1 - w_2\|$ and for all step sizes α, $\alpha\beta < 1$.

where $\|\cdot\|$ and $\langle\cdot\rangle$ are the norm and the scalar product on $\mathbb{R}^d$ respectively.

Instead of directly encoding these assumptions, which require advanced mathematical machinery that is not readily available in Liquid Haskell, we assume two properties of the `update` function. These properties follow from the assumptions on the loss function and are presented in fig. 5.

The contractiveness axiom for update `contractive` states that the distance of `update` on the same data point is equal to the distance of the weights. The boundedness of the difference of gradients `bounded` states that the distance of `update` on different data points is equal to the distance of the weights plus `2*lip*α`.

4.2.4 Proof of Stability of SGD. Using the assumptions of `update`, we proved `sgdDist` in 157 lines of Liquid Haskell code. The proof proceeds by induction on n. We use the structure of the implementation to distinguish between the case where the two algorithms sample the same element, for which we apply the contractiveness property of `update`, and the case where the two algorithms sample the element in which we datasets differ, for which we apply boundedness of the difference of gradients. The `choiceDist` axioms then guarantees that Kantorovich distance increases by `2*lip*α/n` at each iteration, from which we conclude by induction. Other than `choiceDist`, our proof is using the `pureDist` and `pureBindDist` axioms, respectively in the base and inductive case.

```
{-@ assume contractive :: d:Dist Double → α:StepSize
                       → f:(DataPoint → Weight → Double)
                       → z:DataPoint
                       → w_l:Weight → w_r:Weight
                       → {dist d (update z α f w_l) (update z α f w_r)
                         = dist d w w'} @-}

{-@ assume bounded     :: d:Dist Double → α:StepSize
                       → f:(DataPoint → Weight → Double)
                       → z_l:DataPoint → z_r:DataPoint
                       → w_l:Weight → w_r:Weight
                       → {dist d (update z_l α f w_l) (update z_r α f w_r)
                         = dist d w_l w_r + 2 * lip * α} @-}
```

Fig. 5. Assumptions of the sgdDist Proof.

Case study	LoC	Proof LoC	Lemmas	Axioms	Time (sec)
bins Spec (§ 2.3)	21	28	3	0	8
bins Dist (§ 2.5)		143	7	0	113
td0 (§ 4.1)	31	128	5	0	24
sgd (§ 4.2)	26	157	3	2	46
Total	78	456	18	2	191

Table 2. Summary of case studies. **LoC** is lines of executable Haskell code to define the implementations. **Proof LoC** is lines of commented Liquid Haskell code to define refinement type specifications and their proofs. **Lemmas** is the number of total lemmas proved. **Axioms** is the assumed specifications. **Time** is verification time in seconds on a 2,8 GHz Dual-Core Intel Core i5, RAM 8 GB 1600 MHz DDR3.

4.3 Quantitative Summary

Table 2 summarizes the effort required to verify the three examples that we presented through the paper: `bins`, `td0`, and `sgd`. In total, we used 456 lines of proof code, *i.e.* commented (Liquid) Haskell lines that express refinement types specifications and their proofs, to verify 78 lines of executable Haskell code, giving an executable-to proof-ration of almost 6, which is high but expected, given that our proofs are extrinsic. Most of our proofs directly follow the structure of the definitions, thus are easy, once the proper specification is set. The exception is the distance proof for `bins` (§ 2.5) which required the construction of a "ghost" proof distribution, providing evidence that such sophisticated proofs are feasible in our framework. The same proof is an outliner for our verification time: it requires almost 2 minutes, while the rest of the proofs need less than 1 minute. We note that for `td0` we distributed the proof over multiple Haskell modules to allow fast and interactive proof development, since Liquid Haskell verifies per-module and provides local proof-error messages. All our proofs are submitted as Anonymized Supplementary Material.

5 PROOF SYSTEM

To justify the axioms of our implementation, in this section, we define λ^{RP}, a core probabilistic λ-calculus, with a set of relational proof rules. Although each proof rule of the relational program

Types	t,s	::=	`nat` | `bool` | `list` t	*Discrete Types*
		|	`real` | $t \to t$ | `prM` t	*Continuous Types*
Constants	c	::=	`nil` | `cons` | $n \in \mathbb{N}$ | $a \in \mathbb{R}^+$ | $+\infty$ | `true` | `false`	
		|	$+ \mid - \mid \cdot \mid / \mid = \mid < \mid \wedge \mid \vee \mid \Rightarrow \mid \neg$	
Expressions	e, u	::=	`unif` e | `bern` x | `choice` $x\ e\ e$	*Probabilistic*
		|	`ret` e | `bind` $e\ e$	*Monadic*
		|	$\lambda x.e \mid e\ e \mid c \mid x$	*Pure*
		|	`case` e `of` $\{$`nil` $\to e;$ `cons` $x\ x \to e\}$	
		|	`let` $x = e$ `in` e	
Assertions	p	::=	$e \mid p[\wedge, \vee, \Rightarrow]p \mid \neg p$	*Boolean*
		|	$\forall x : t.\ p \mid \exists x : t.\ p$	*Quantifiers*
		|	$\diamond_e p\ e\ e$	*Lifting*
Environments				
Typing	Γ	::=	$\emptyset \mid x{:}t, \Gamma$	
Predicate	Φ		$\emptyset \mid p, \Phi$	

Fig. 6. Syntax of λ^{RP}. (Variables include r_l, r_r, and d.)

logic is encoded independently from others as an axiom in Liquid Haskell, we follow the same style of presentation as Aguirre et al. [2017] and treat our set of proof rules as a proof system. This treatment is primarily motivated by our desire to prove a crisp statement for soundness. We also note that for the purpose of establishing soundness, we only consider discrete distributions.

This section is organized as follows: we define the syntax (§ 5.1) and type system (§ 5.2) of λ^{RP}; then, the axioms (§ 5.3) and the proof rules (§ 5.4) of our logic; and finally, the denotational semantics of λ^{RP} (§ 5.5) which we use to show soundness (§ 5.6).

5.1 Syntax

We consider a typed probabilistic λ-calculus with algebraic datatypes and distributions (fig. 6). Types are built from base types using the usual function space constructor and type constructors for lists (`list`) and probability distributions (`prM`; which encodes the Haskell type `PrM` of § 3.3).

λ^{RP} features a rich set of *constants* that include natural (n) and real (a) numbers, the special constant $+\infty$, the `true` and `false` booleans, the `nil` and `cons` list constructors. Furthermore, λ^{RP} features constants for arithmetic operations, boolean operations, equality, and inequality.

Variables in λ^{RP} include three special variables d, r_l, and r_r that respectively model distance and the left and right result of computations.

Expressions are built from constants and variables using the standard constructions from λ-calculus and monadic constructions. The former include constructions for lambda abstraction ($\lambda x.e$), application ($e\ e$), case analysis (`case` e `of` $\{$`nil` $\to e;$ `cons` $x\ x \to e\}$), and structurally recursive definitions (`let` $x = t$ `in` e). The latter include the probabilistic primitives `unif` e that probabilistically returns an element of its input list e, with uniform distribution, `bern` x that returns 1 with probability x, otherwise 0, and `choice` $x\ e_1\ e_2$ that returns the distribution e_1 with probability x and e_2 with probability $1 - x$. There primitives model our implementation interface (§ 3.3).

Assertions include (arbitrary, but boolean typed) expressions of λ^{RP} and boolean operators. To allow reduction to RHOL, assertions also include quantifiers even though our system does not explicitly use them. Finally, λ^{RP} has the lift assertion $\diamond_k p\ e_l\ e_r$ that encodes the combination of `(⋄)` and `kdist` of our implementation (§ 3). Here k is a real typed expression, p is a relational assertion that depends two arguments of type t_l and t_l respectively, and e_l and e_r are probability distributions over t_l and t_l respectively. The assertion ensures that p holds for the distribution coupling, and that the Kantorovich distance between the two distributions is bounded by k.

We adopt standard conventions, *e.g.* $g \cdot f$ stands for $\lambda x.\ g\ (f\ x)$. By abuse of notation, we write $\texttt{ret} \cdot e$ as shorthand for $\lambda x.\ \texttt{ret}\ (f\ x)$ and $\diamond_k p$ as syntactic sugar for $\diamond_k(\lambda r_l\ r_r.p)\ r_l\ r_r$, *i.e.* when the lifting happens on the special variables r_l and r_r. By convention, we also write $\diamond p$ as a shorthand for $\diamond_{+\infty} p$. As usual, we also let $e[e_x/x]$ denote the capture-free substitution of e_x for x in e.

5.2 Type system

We equip our language with a simple type system which serves three purposes: first, it ensures that expressions respect the type signatures of operators; second, it ensures that recursive definitions are structurally terminating. Our logic is agnostic to the mechanism used to enforce structural termination, so we leave this mechanism abstract. Finally, our type system restricts the use of distribution types to discrete types, so that types and expressions of our language can be given a set-theoretic interpretation—we discuss the case of continuous distributions in the § 7.

5.3 Axioms

The special variable d encodes distance that we assume satisfies the axioms of an (extended) metric:

Definition 5.1 (Metric Axioms). For every type t and every $x, y, z : t$:

(1) *Identity:* $d(x, y) = 0 \Leftrightarrow x = y$.
(2) *Symmetry:* $d(x, y) = d(y, x)$.
(3) *Triangular Inequality:* $d(x, z) \leq d(x, y) + d(y, z)$.

In addition, we assume that the distance d is defined *for all* types of λ^{RP}: For `nat` and `real` is defined as $d(x, y) = |x - y|$ and further satisfies linearity. For booleans it is defined as $d(x, y) = 1$ if $x \neq y$, *i.e.* it coincides with the discrete distance on booleans. For lists of equal length, we assume the distance is the maximum of distances between elements at the same positions. When the length is different, the distance is infinite. For functions, it is defined as the maximum distance over all function domains and for distributions, as the Kantorovich distance (of Villani [2009]). These assumptions are required for soundness.

5.4 Proof System

Our proof system uses two judgments to decide logical implication and relational typing. The first judgment $\Gamma; \Phi \vdash p$ states that the assertion p is valid under the assumptions of Φ, where all the free variables of Φ and p appear in Γ. This first judgment is similar to [Aguirre et al. 2017] and the proof rules are thus omitted.

The second judgment $\Gamma; \Phi \vdash e_l : t_l \sim e_r : t_r \mid p$ states that the expressions e_l and e_r, respectively of types t_l and t_r under Γ, satisfy the predicate p, under the assumptions of Φ. To encode this property the predicate p might refer to two special variables r_l and r_r (*i.e.* not bound in the environment Γ) that respectively refer to the expressions e_l and e_r. For example $\emptyset; \emptyset \vdash 1 : \texttt{nat} \sim 2 : \texttt{nat} \mid r_l < r_r$ holds, since $1 < 2$. In general, the relational typing means that $\Gamma; \Phi \vdash p[e_l/r_l][e_r/r_r]$.

Contexts. Contexts are pairs consisting of a typing environment (Γ) that maps variables to their types and a logical environment (Φ) consisting of assertions.

Proof Rules. Figure 7 defines selected proof rules $\Gamma; \Phi \vdash e_l : \texttt{prM}\ t_l \sim e_r : \texttt{prM}\ t_r \mid p$. The proof rules are given for monadic expressions, in which case the relational assertions are of the form $\diamond_k p$. Informally, these assertions state that 1) the two related expressions satisfy the lifted predicate p and 2) the Kantorovich distance between the two expressions is upper bounded by k.

Relational Probabilistic Typing $\boxed{\Gamma; \Phi \vdash e_l : \mathsf{prM}\ t_l \sim e_r : \mathsf{prM}\ t_r \mid \diamond_k p}$

$$\frac{}{\Gamma; \Phi \vdash e_l : \mathsf{prM}\ t_l \sim e_r : \mathsf{prM}\ t_r \mid \diamond_{+\infty} \mathsf{true}}\ \textsc{T-Bot}$$

$$\frac{r_l{:}t_l, r_r{:}t_r, \Gamma; \Phi \vdash p \Rightarrow p_w \qquad r_l{:}t_l, r_r{:}t_r, \Gamma; \Phi \vdash k \leq k_w \qquad \Gamma; \Phi \vdash e_l : \mathsf{prM}\ t_l \sim e_r : \mathsf{prM}\ t_r \mid \diamond_k p}{\Gamma; \Phi \vdash e_l : \mathsf{prM}\ t_l \sim e_r : \mathsf{prM}\ t_r \mid \diamond_{k_w} p_w}\ \textsc{T-Weaken}$$

$$\frac{\Gamma; \Phi \vdash e_l : \mathsf{list}\ t \sim e_r : \mathsf{list}\ t \mid r_l = r_r}{\Gamma; \Phi \vdash \mathsf{unif}\ e_l : \mathsf{prM}\ t \sim \mathsf{unif}\ e_r : \mathsf{prM}\ t \mid \diamond_0 r_l = r_r}\ \textsc{T-Unif}$$

$$\frac{\Gamma; \Phi \vdash x_l : \mathsf{real} \sim x_r : \mathsf{real} \mid 0 \leq r_l \leq r_r \leq 1}{\Gamma; \Phi \vdash \mathsf{bern}\ x_l : \mathsf{prM}\ \mathsf{nat} \sim \mathsf{bern}\ x_r : \mathsf{prM}\ \mathsf{nat} \mid \diamond_{|x_r - x_l|} 0 \leq r_l \leq r_r \leq 1}\ \textsc{T-Bern}$$

$$\frac{\Gamma; \Phi \vdash x_l : \mathsf{real} \sim x_r : \mathsf{real} \mid r_l = r_r \qquad \Gamma; \Phi \vdash e_l : \mathsf{prM}\ t_l \sim e_r : \mathsf{prM}\ t_r \mid \diamond_{k_e} p \qquad \Gamma; \Phi \vdash u_l : \mathsf{prM}\ t_l \sim u_r : \mathsf{prM}\ t_r \mid \diamond_{k_u} p}{\Gamma; \Phi \vdash \mathsf{choice}\ x_l\ e_l\ u_l : \mathsf{prM}\ t_l \sim \mathsf{choice}\ x_r\ e_r\ u_r : \mathsf{prM}\ t_r \mid \diamond_{x_l \cdot k_e + (1 - x_l) \cdot k_u} p}\ \textsc{T-Choice}$$

$$\frac{\Gamma; \Phi \vdash e_l : t \sim e_r : t \mid p \wedge d(r_l, r_r) \leq k}{\Gamma; \Phi \vdash \mathsf{ret}\ e_l : \mathsf{prM}\ t \sim \mathsf{ret}\ e_r : \mathsf{prM}\ t \mid \diamond_k p}\ \textsc{T-Ret}$$

$$\frac{\Gamma; \Phi \vdash e_l : \mathsf{prM}\ s_l \sim e_r : \mathsf{prM}\ s_r \mid \diamond q \qquad x_l{:}s_l, x_r{:}s_r, \Gamma; q[x_l/r_l][x_r/r_r], \Phi \vdash u_l\ x_l : \mathsf{prM}\ t_l \sim u_r\ x_r : \mathsf{prM}\ t_r \mid \diamond_k p}{\Gamma; \Phi \vdash \mathsf{bind}\ e_l\ u_l : \mathsf{prM}\ t_l \sim \mathsf{bind}\ e_r\ u_r : \mathsf{prM}\ t_r \mid \diamond_k p}\ \textsc{T-Bind}$$

$$\frac{\Gamma; \Phi \vdash e_l : \mathsf{prM}\ s_l \sim e_r : \mathsf{prM}\ s_r \mid \diamond_{k_e} q \qquad x_l{:}s_l, x_r{:}s_r, \Gamma; q[x_l/r_l][x_r/r_r], \Phi \vdash f_l\ x_l : t_l \sim f_r\ x_r : t_r \mid p \wedge d(r_l, r_r) \leq m \cdot d(x_l, x_r) + k_u}{\Gamma; \Phi \vdash \mathsf{bind}\ e_l\ (\mathsf{ret}\ \cdot f_l) : \mathsf{prM}\ t_l \sim \mathsf{bind}\ e_r\ (\mathsf{ret}\ \cdot f_r) : \mathsf{prM}\ t_r \mid \diamond_{m \cdot k_e + k_u} p}\ \textsc{T-Bind-Ret}$$

Fig. 7. Typing of λ^{RP}, where k ranges over distance (real typed expressions). We use the syntactic sugar $g \cdot f \doteq \lambda x.g\,(f\ x)$ and $\diamond_k p \doteq \diamond_k(\lambda r_l\ r_r.p)\ r_l\ r_r$.

Rules T-Bot and T-Weaken are non-syntax-directed rules that can be applied to any probabilistic expressio. Rule T-Bot states that any two probabilistic expressions $e_l \sim e_r$ satisfy the (lifted) true predicate and have expected distance bounded by $+\infty$. Rule T-Weaken weakens the lifted predicate $\diamond_k p$ to $\diamond_{k_w} p_w$, provided $p \Rightarrow p_w$ and $k \leq k_w$. These two requirements are established by pure logic.

The rules T-Unif, T-Bern, and T-Choice are used to relate (the same) probabilistic primitives. Rule T-Unif states that the uniform distributions of two *semantically* equal lists produce equal values and have zero expected distance. To semantically equate two (pure) lists e_l and e_r we use (pure) relational typing. The lifted predicate $\diamond_0 r_l = r_r$ ensures that the two distributions are the same and accordingly their expected distance is zero. We note that more general rules exist, but are omitted here since they are not used in our examples.

The rule T-Bern states that the bernoulli distribution $\mathsf{bern}\ x_l$ dominates $\mathsf{bern}\ x_r$, when $x_l \leq x_r$. The condition is expressed by the pure predicate $r_l \leq r_r$ in the premise, while the conclusion of the rule lifts the same predicate to encode dominance. Finally, it ensures that the distance of the two distributions is bounded by $|x_r - x_l|$.

The rule T-Choice relates two choice expressions $\mathsf{choice}\ x_i\ e_i\ u_i$ (with i being l or r). To do so, it requires that x_l and x_r are equal, and that the pairs of distributions e_l and e_r and respectively u_l and u_r satisfy the same lifted predicate $\diamond p$ and respectively have distances k_e and k_u. It then ensures that choice will also satisfy the predicate $\diamond p$, while the distance is $x_l \cdot k_e + (1 - x_l) \cdot k_u$.

The rules T-Ret and T-Bind are used to relate (the same) monadic primitives. The rule T-Ret relates `ret` e_l with `ret` e_r. Using pure relational typing, it requires that e_l and e_r satisfy the predicates p (in which the relational variables can freely appear) and their distance is bounded by k. Note that bounding distance in the pure setting is encoded as a logical statement; while a weakening rule can bring the premise in the required syntactic form, potentially with infinite distance. The rule concludes that the return expressions satisfy the lifted p (since the relational variables are not monadic) and their distance is bounded by k. The rule T-Bind relates two expressions `bind` $e_i\ u_i$ (with i being l or r). In the first premise, it assumes that e_l and e_r satisfy some lifted predicate q. In the second premise, the predicate p is assumed in the predicate environment to check the application $u_i\ x_i$ where x_i is the value of the probabilistic argument e_i, *i.e.* satisfies the predicate q. The predicate and distance of the bind expressions are the same as these of the second premise.

The final rule T-Bind-Ret relates two `bind` expressions where the second argument is a pure function, composed with `ret` (we use the syntactic sugar for composition $g \cdot f \doteq \lambda x.g\ (f\ x)$). Such expressions could be typed by the rule T-Bind, but this special case permits more precise reasoning and is used critically by our case studies. The first premise of the rule is the same as of the T-Bind, but the second premise is now using pure relational typing to bound the distance of f's result as a function of the distance of it's input, *i.e.* $d(r_l, r_r) \leq m \cdot d(x_l, x_r) + k_u$, so the distance of the conclusion $(m \cdot k_e + k_u)$ depends on the distance of the e arguments (k_e).

We conclude this section with a brief, high-level, explanation for the need of having two rules for `bind`. In contrast with other existing quantitative relational logics, such as those used for differential privacy [Barthe et al. 2012], there is no "obvious" composition rule for `bind` when considering the Kantorovich metric. Specifically, given two distributions e_l and e_r whose Kantorovich distance is upper bounded by k, there is no obvious condition on continuations f_l and f_r, such that applying these continuations results in two distributions whose Kantorovich distance is upper bounded by k and some additional argument. At a high level, the problem arises from the fact that lifted assertions guarantee that the Kantorovich (i.e. average) distance between the two distributions is upper bounded by k, whereas assumptions in context would require that there exists a coupling such that the distance between e_l and e_r is bounded by k. Since they are complex and not needed for our purposes, we leave the study of more general rules for future work.

5.5 Denotational Semantics

Here we define a denotational semantics of λ^{RP} by defining the denotations of types and typing environments (§ 5.5.1); expressions (§ 5.5.2); and assertions and predicate environments (§ 5.5.3). Our denotational semantics only considers discrete distributions, so that expressions have a straightforward set-theoretic interpretation.

5.5.1 Denotations of Types and Typing Environments. Definition 5.3 inductively defines the denotations of types. The interesting case is `prM` that gives a probability distribution, as per definition 5.2.

Definition 5.2 (Discrete Probability Distribution). A probability distribution over a discrete set C is a function $\mu : C \to [0, 1]$ such that $\sum_{x \in \mathrm{supp}\, \mu} \mu(x) = 1$, where there support of μ is defined as $\mathrm{supp}\, \mu \doteq \{x \mid x \in C \wedge \mu(x) \neq 0\}$. We denote the set of discrete distributions over C as $D(C)$.

Definition 5.3 (Denotations of Types). For each type t, $[\![t]\!]$ is inductively defined as follows:

$$\begin{array}{lclclclclcl} [\![\texttt{bool}]\!] & \doteq & \mathbb{B} & [\![\texttt{nat}]\!] & \doteq & \mathbb{N} & [\![\texttt{list}\ t]\!] & \doteq & \texttt{list}_{[\![t]\!]} \\ [\![\texttt{prM}\ t]\!] & \doteq & D([\![t]\!]) & [\![\texttt{real}]\!] & \doteq & \mathbb{R}^+ & [\![t_x \to t]\!] & \doteq & [\![t_x]\!] \to [\![t]\!] \end{array}$$

Using the denotations of types, definition 5.4 defines the denotation of a typing environment Γ as a set of *models* ρ that map each binder $(x : t)$ in Γ to an element in the denotation of t.

Definition 5.4 (Denotation of Type Environments). $(\!|\Gamma|\!) \doteq \{\rho \mid \forall (x : t) \in \Gamma . \rho(x) \in [\![t]\!]\}$

5.5.2 Denotations of Expressions. The denotation of an expression e is defined for a fixed model ρ as $[\![e]\!]_\rho$ in definition 5.5. The denotations of the pure fragment are standard, where the model is used to define the definition of a variable as $\rho(x)$ and we skip the verbose definitions for `case` and `let`. The probability and return cases use standard distributions, while monadic bind maps to distribution composition, defined in definition 5.6.

Definition 5.5 (Denotations of Expressions). For each expression e and model ρ, $[\![e]\!]_\rho$ is defined as:

$$\begin{array}{rcl@{\qquad}rcl}
[\![\texttt{unif}\ e]\!]_\rho & \doteq & U_{[\![e]\!]_\rho} & [\![x]\!]_\rho & \doteq & \rho(x) \\
[\![\texttt{bern}\ x]\!]_\rho & \doteq & B_{[\![x]\!]_\rho} & [\![c]\!]_\rho & \doteq & c \\
[\![\texttt{choice}\ x\ e\ u]\!]_\rho & \doteq & [\![x]\!]_\rho \cdot [\![e]\!]_\rho + (1 - [\![x]\!]_\rho) \cdot [\![u]\!]_\rho & [\![\lambda x.e]\!]_\rho & \doteq & \lambda x.[\![e]\!]_\rho \\
[\![\texttt{bind}\ e\ u]\!]_\rho & \doteq & \text{scomp}\ [\![e]\!]_\rho\ [\![u]\!]_\rho & [\![e\ u]\!]_\rho & \doteq & [\![e]\!]_\rho\ [\![u]\!]_\rho \\
[\![\texttt{ret}\ e]\!]_\rho & \doteq & \delta_{[\![e]\!]_\rho} & & &
\end{array}$$

where δ_x represents the Dirac distribution at x, U_{xs} represents the uniform distribution over a non-repeating list xs, and B_p represents the p-biased Bernoulli distribution on $\{0, 1\}$.

Definition 5.6 (Sequential Composition Distribution). Let $\mu \in D(C)$ and $f : C \to D(C_2)$. Sequential composition distribution scomp $\mu\ f$ is defined as:

$$\text{scomp}\ \mu\ f(y) \doteq \sum_{x \in C} \mu(x) \cdot f(x)(y)$$

5.5.3 Denotations of Assertions and Predicate Environments. Finally, we inductively define denotations of assertions in Definition 5.10. Most of the cases are standard except from the case for lifting that relies on Kantorovich couplings (also known as expectation couplings), which we define in definition 5.9 and, in turn, relies on basic definitions of expectation and marginals.

Definition 5.7 (Expectation). For all $\mu \in D(C)$ and $f : C \to \mathbb{R}$, the expected value $E_{x \leftarrow \mu}[f(x)]$ (or $E_\mu[f]$) of f is a partial function defined as:

$$E_{x \leftarrow \mu}[f(x)] = \sum_{x \in C} \mu(x) \cdot f(x)$$

Definition 5.8 (Marginals). For all $\mu \in D(C_1 \times C_2)$, the first and second marginals of μ are respectively the distributions $\pi_1(\mu) \in D(C_1)$ and $\pi_2(\mu) \in D(C_2)$ defined as:

$$\pi_1(\mu)(x) \doteq \sum_{y \in C_2} \mu(x, y) \qquad \pi_2(\mu)(y) \doteq \sum_{x \in C_1} \mu(x, y)$$

Intuitively, the marginals are the "projections" of the couplings, *i.e.* "dependent products" over distributions. Using these, we define Kantorovich coupling, *i.e.* the conditions that render $\diamond_k R$ valid.

Definition 5.9 (Kantorovich coupling). For all $\mu_1 \in D(C_1)$, $\mu_2 \in D(C_2)$, $R \subseteq C_1 \times C_2$, $d : C_1 \times C_2 \to \mathbb{R}^+$, and $k \in \mathbb{R}^+$, $\models \diamond_k R\ (\mu_1, \mu_2)$ *iff* there *exists* $\mu \in D(A_1 \times A_2)$ such that:

$$(1)\ \pi_1(\mu) = \mu_1 \text{ and } \pi_2(\mu) = \mu_2 \qquad (2)\ \text{supp}(\mu) \subseteq R \qquad (3)\ E_{(x,y) \leftarrow \mu}[d(x, y)] \leq k$$

As usual, $\models p$, means logical validity of p. The first clause corresponds to the standard definition of coupling. The second clause corresponds to the definition of R-coupling, and is connected to lifting by the following equivalence: $\diamond R$ iff there exists an R-coupling. The last clause is specific to Kantorovich coupling, and states that the expected distance with respect to the definition μ is upper bounded by k. Since the Kantorovich distance corresponds to the minimum expected distance over all possible couplings, it follows that k is an upper bound for the Kantorovich distance.

We use the Kantorovich coupling to define the denotation of λ^{RP}'s lifted predicate, while the rest of the denotations are standard.

Definition 5.10 (Denotation of Assertions). For each assertion p and model ρ, $[\![p]\!]_\rho$ is defined as:

$$\begin{array}{rclrcl} [\![e]\!]_\rho & \doteq & [\![e]\!]_\rho & [\![p_1[\wedge,\vee,\Rightarrow]p_2]\!]_\rho & \doteq & [\![p_1]\!]_\rho[\wedge,\vee,\Rightarrow][\![p_2]\!]_\rho \\ [\![\neg p]\!]_\rho & \doteq & \neg[\![p]\!]_\rho & [\![\forall x:t.\,p]\!]_\rho & \doteq & \forall x.[\![p]\!]_\rho \\ [\![\diamond_k p\ e\ u]\!]_\rho & \doteq & \diamond_{[\![k]\!]_\rho}[\![p]\!]_\rho([\![e]\!]_\rho,[\![u]\!]_\rho) & [\![\exists x:t.\,p]\!]_\rho & \doteq & \exists x.[\![p]\!]_\rho \end{array}$$

The denotation of a predicate environment is the conjunction of the denotation of all its predicates.

Definition 5.11 (Denotation of Predicate Environment). $[\![\Phi]\!]_\rho \doteq \bigwedge_{p\in\Phi}[\![p]\!]_\rho$

5.6 Soundness

Our proof system is sound with respect to its denotational semantics. That is, for every model of the typing environment that renders the predicate environment valid, the denotation of the predicate, with the special variables r_l and r_r substitued by the typed expressions, is valid.

THEOREM 5.12 (SOUNDNESS). *If* $\Gamma;\Phi\vdash e_l:t_l\sim e_r:t_r\mid p$, *then for every* $\rho\in(\!|\Gamma|\!)$ *such that* $\models[\![\Phi]\!]_\rho$, *we have* $\models[\![p[[\![e_l]\!]_\rho/r_l][[\![e_r]\!]_\rho/r_r]]\!]_\rho$.

In the appendix we prove soundness of the monadic rules, while soundness for the pure fragment follows from Aguirre et al. [2017].

As a corollary of soundness and definition 5.9, we get soundness of the probabilistic fragment. Informally, our judgement relates e_l and e_r when there exists a coupling μ of e_l amd e_r such that predicate p holds for all samples with non-zero probability and the expected distance between samples is less than k.

COROLLARY 5.13 (SOUNDNESS OF PROB. FRAGMENT). *If* $\Gamma;\Phi\vdash e_l:\mathsf{prM}\,t_l\sim e_r:\mathsf{prM}\,t_r\mid\diamond_k p$, *then for every* $\rho\in(\!|\Gamma|\!)$ *such that* $\models[\![\Phi]\!]_\rho$, *there exists an* $\mu\in D([\![t_l]\!]\times[\![t_r]\!])$ *such that:*

(1) $\pi_1(\mu)=[\![e_l]\!]_\rho$ *and* $\pi_2(\mu)=[\![e_r]\!]_\rho$;
(2) $\mathrm{supp}(\mu)\subseteq\{(x_l,x_r)\mid x_l\in[\![t_l]\!],x_r\in[\![t_r]\!],\models[\![p]\!]_\rho\ x_l\ x_r\}$; *and*
(3) $E_{(x,y)\leftarrow\mu}[d(x,y)]\leq[\![k]\!]_\rho$.

5.7 Continuous distributions

Our denotational semantics and soundness claim are established for the fragment of the language with discrete distributions. We stress that, while it is possible to give a denotational semantics for continuous distributions using Quasi-Borel Spaces (QBS) [Heunen et al. 2017; Vákár et al. 2019]. However, prior work in [Aguirre et al. 2021a] suggests that it may be challenging to give a sound denotational semantics for relational preexpectations for a higher-order language.

6 RELATED WORK

First Order, Imperative, Probabilistic Languages. There is a large body of work that builds and applies program logics to reason about probabilistic programs. Many of these works are based on first-order imperative languages. Broadly speaking, there exists two main lines of work: the first line of work focuses on non-relational properties, and can be traced back to early work by Kozen [Kozen 1985] and Morgan and McIver [Morgan et al. 1996]. Many of these works focus on establishing sound foundations, and have not been implemented in practice. There are however, some noticeable exceptions [Hölzl 2016; Hurd 2003], including an active line of work in mechanizing or automating proofs of expected cost of probabilistic programs [Avanzini et al. 2020; Ngo et al. 2018; Tassarotti and Harper 2018].

The second line of work focuses on relational properties; this line of work has been initiated in [Barthe et al. 2009], and its mechanization in the Coq proof assistant or as a self-standing proof assistant [Barthe et al. 2011] have been used to verify formally concrete security of cryptographic constructions. Similar approaches have been developed by the FCF [Petcher and Morrisett 2015] and CryptHOL[Basin et al. 2020]. These logics have been further extended to reason about differential privacy [Barthe et al. 2012] and expected sensitivity [Barthe et al. 2018]. Our work is most closely related to the latter, which introduces the notion of expectation coupling and formally verifies algorithmic stability of SGD. More recently, Wang et al [Wang et al. 2020] develop an alternative formalism which supports a richer class of termination behaviors. Our work is also closely related to [Aguirre et al. 2021b] in which the authors develop a relational weakest precondition calculus to reason about expected sensitivity. Their calculus avoids some of the peculiarities of [Barthe et al. 2018], and is used to verify our two main examples, but is not implemented. We also note that their proof of convergence is significantly different and uses Kantorovich couplings rather than the simpler, vanilla couplings used in our proof.

Relational Reasoning for Higher Order Languages. Nanevski et al. [2011] were among the first to explore relational reasoning for higher-order languages. Their work defines Relational Hoare Type Theory (RHTT), a powerful program logic for proving relational properties of stateful higher-order programs. RHTT is implemented in the Coq proof assistant, and used to verify intricate information flow properties. The main difference with our work is that RHTT operates on a shallow embedding of programs and does not support probabilistic reasoning. BiRelCost [Çiçek et al. 2019] is a bidirectional type checker that automatically performs relational and unary cost analysis requiring minimal user annotations. However, the checker is incomplete and relies on example-driven heuristics. It is unlikely that the used heuristics would suffice to prove our case studies. Handley et al. [2019], like us, address this incompleteness by encoding the relational rules in Liquid Haskell and requiring explicit, user-provided, extrinsic proofs; sacrificing automation in the name of expressiveness and predictability. Our work applies the technique of Handley et al. [2019] to reason about probabilistic programs and quantitative specifications.

Verification of Higher Order Probabilistic Programs. There have been two lines of work that verify relational properties on executable probabilistic programs in F⋆ [Swamy et al. 2016]. First, rF⋆ [Barthe et al. 2014] is an extension of F⋆ that supports relational reasoning via relational refinement types. As with F⋆, type checking generates SMT queries that are discharged by Z3. Although rF⋆ supports probabilistic programs, reasoning is constrained by syntax-directed typing, for example the `binsDist` of § 2.5 cannot be type-checked. We overcome this limitation by supporting a richer set of axioms and extrinsic proofs. Second, Grimm et al. [2018] present an alternative approach to reason about relational properties in F⋆ that, similar to our work, uses extrinsic proofs to reason about programs. Their approach supports probabilistic reasoning, and is used to prove probabilistic non-interference of Shannon's classic cryptographic one-time pad. However, their support for probabilistic reasoning is limited and quantitative specifications are not supported.

Proof Systems of Higher Order Probabilistic Programs. Aguirre et al. [2021a] present several proof systems for higher-order stateful probabilistic programs. One key novelty of their proof systems is the support of interactive adversarial computations. Such rules can be used to prove that programs verify relational properties for all possible (well-typed) adversarial computations. Unfortunately, these rules are only proved sound for boolean relational properties and it is an open problem whether these rules remain sound for quantitative properties modeled using Kantorovich lifting. Their approach builds on Relational Higher-Order Logic (RHOL) [Aguirre et al. 2017], which achieves full expressiveness and bypasses limitations of syntax-directed reasoning via an embedding into HOL. However, there is no implementation of RHOL and of its extensions.

Differential privacy. Differential privacy is a mathematical notion of privacy that can be understood as a form of probabilistic sensitivity w.r.t. some pseudo-distance induced by a specific f-divergence. As such, differential privacy is directly related to our work.

Fuzz [Reed and Pierce 2010] was the first type-based, approach to verify distance of higher order programs, in the domain of differential privacy. Since introduced, Fuzz has been extended in various ways. DFuzz [Gaboardi et al. 2013] introduces recursion; Adaptive Fuzz [Winograd-Cort et al. 2017] supports dynamic data analysis; Fuzzi [Zhang et al. 2019] extend Fuzz with APRHL [Barthe et al. 2012] to prove trusted primitives (like Laplace) Duet [Near et al. 2019] extends Fuzz with support for advanced variants of differential privacy via a dual type system; HOARe2 [Barthe et al. 2015] provides a relational refinement system that embeds Fuzz; and Bunched Fuzz [june wunder et al. 2022] extends the system with bunches to, like us, reason about distances of probability distributions. Like our system, most of these approaches use typing rules to trace distance and have some support of probabilistic programs. However, they lack the flexibility to reason about expected sensitivity for most advanced examples such as those considered here.

In a work most closely related to ours, DPella [Lobo-Vesga et al. 2021] and Solo [Abuah et al. 2021] use Haskell's dependent types to encode differential privacy. Both these systems are similar to ours since they do verify executable Haskell code. The critical difference is that they use Haskell's dependent types while we use the refinement type extension of Liquid Haskell. One benefit of our approach is that we can delegate arithmetic reasoning to SMT. At a more general level, the two approaches are incomparable: we are able to reason finely about expected sensitivity, whereas they can reason about differential privacy using standard composition theorems, and about accuracy using a fine-grained approach that exploits a slick combination of information flow typing and concentration inequalities.

7 CONCLUSION & FUTURE WORK

We have enhanced Liquid Haskell with support for reasoning about relational properties of probabilistic computations. We have also demonstrated that the resulting framework is sufficiently expressive for proving formally convergence and algorithmic stability properties of classic machine learning algorithms from the literature. An important direction for future work is to verify the implementation of our library and complement it with mechanisms to reason about non-relational, expectation based properties. This would allow us to reason about convergence to the optimal parameters, which is another property of concern for machine learning algorithms. Armed with the mechanisms developed here and mechanisms for non-relational expectation-based reasoning, it would be possible to develop a formally verified library of machine-learning algorithms.

Another important direction for future work is to improve automation of relational proofs. The main challenge is to conceive a set of effective mechanisms to combine syntax-directed, synchronous, reasoning about the two programs, with syntax-directed, asynchronous reasoning about a single program (typically when the two programs do not follow the same control-flow), and non-syntax-directed reasoning, like our `binsDist` proof. One potential approach is to combine relational refinement type checking with the idea of "game-hopping" used in security proofs: informally, to help relational verification by introducing a sequence of intermediate programs, such that relational verification of two consecutive programs can be fully automated, and the overall verification goal follows directly from combining the results of these intermediate verifications.

ACKNOWLEDGMENTS

REFERENCES

Chike Abuah, David Darais, and Joseph P. Near. 2021. Solo: Enforcing Differential Privacy Without Fancy Types. *CoRR* abs/2105.01632 (2021). arXiv:2105.01632 https://arxiv.org/abs/2105.01632

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, Shin-ya Katsumata, and Tetsuya Sato. 2021a. Higher-order probabilistic adversarial computations: categorical semantics and program logics. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473598

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A relational logic for higher-order programs. *Proc. ACM Program. Lang.* 1, ICFP (2017), 21:1–21:29. https://doi.org/10.1145/3110265

Alejandro Aguirre, Gilles Barthe, Justin Hsu, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021b. A pre-expectation calculus for probabilistic sensitivity. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434333

Martin Avanzini, Georg Moser, and Michael Schaper. 2020. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 172:1–172:30. https://doi.org/10.1145/3428240

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. Proving expected sensitivity of probabilistic programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 57:1–57:29. https://doi.org/10.1145/3158145

Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. 2014. Probabilistic relational verification for cryptographic implementations. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 193–206. https://doi.org/10.1145/2535838.2535847

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 55–68. https://doi.org/10.1145/2676726.2677000

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 90–101. https://doi.org/10.1145/1480881.1480894

Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings (Lecture Notes in Computer Science)*, Phillip Rogaway (Ed.), Vol. 6841. Springer, 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

Gilles Barthe and Justin Hsu. 2020. Probabilistic couplings from program logics. *Foundations of Probabilistic Programming* (2020), 145.

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 97–110. https://doi.org/10.1145/2103656.2103670

David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. 2020. CryptHOL: Game-Based Proofs in Higher-Order Logic. *J. Cryptol.* 33, 2 (2020), 494–566. https://doi.org/10.1007/s00145-019-09341-z

Olivier Bousquet and André Elisseeff. 2002. Stability and Generalization. 2 (2002), 499–526. http://www.jmlr.org/papers/v2/bousquet02a.html

Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2019. Bidirectional type checking for relational properties. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 533–547. https://doi.org/10.1145/3314221.3314603

Yuxin Deng. 2015. *Semantics of Probabilistic Processes: An Operational Approach.* https://doi.org/10.1007/978-3-662-45198-4

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13).* https://doi.org/10.1145/2429069.2429113

Niklas Grimm, Kenji Maillard, Cédric Fournet, Catalin Hritcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella Béguelin. 2018. A monadic framework for relational verification: applied to information security, program equivalence, and optimizations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 130–145. https://doi.org/10.1145/3167090

Jad Hamza, Nicolas Voirol, and Viktor Kunčak. 2019. System FR: Formalized Foundations for the Stainless Verifier. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 166 (oct 2019), 30 pages. https://doi.org/10.1145/3360592

Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2019. Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell. *Principles of Programming Languages* (2019). https://doi.org/10.1145/3371092

Moritz Hardt, Ben Recht, and Yoram Singer. 2016. Train faster, generalize better: Stability of stochastic gradient descent, Vol. 48. JMLR.org, 1225–1234. http://jmlr.org/proceedings/papers/v48/hardt16.html

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005137

Johannes Hölzl. 2016. Formalising Semantics for Expected Running Time of Probabilistic Programs. In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings (Lecture Notes in Computer Science)*, Jasmin Christian Blanchette and Stephan Merz (Eds.), Vol. 9807. Springer, 475–482. https://doi.org/10.1007/978-3-319-43144-4_30

Joe Hurd. 2003. Verification of the Miller-Rabin probabilistic primality test. *J. Log. Algebraic Methods Program.* 56, 1-2 (2003), 3–21. https://doi.org/10.1016/S1567-8326(02)00065-6

june wunder, Arthur Azevedo de Amorim, Patrick Baillot, and Marco Gaboardi. 2022. Bunched Fuzz: Sensitivity for Vector Metrics. arXiv:cs.PL/2202.01901

Dexter Kozen. 1985. A Probabilistic PDL. 30, 2 (1985), 162–178.

Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. 2021. STORM: Refinement Types for Secure Web Applications. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 441–459. https://www.usenix.org/conference/osdi21/presentation/lehmann

Torgny Lindvall. 2002. *Lectures on the coupling method.* Courier Corporation.

Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. 2021. A Programming Language for Data Privacy with Accuracy Estimations. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 6 (jun 2021), 42 pages. https://doi.org/10.1145/3452096

Kenji Maillard, Catalin Hritcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 relational program logics. *Proc. ACM Program. Lang.* 4, POPL (2020), 4:1–4:33. https://doi.org/10.1145/3371072

Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic Predicate Transformers. 18, 3 (1996), 325–353.

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2011. Verification of Information Flow and Access Control Policies with Dependent Types. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. 165–179.

Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019. Duet: An Expressive Higher-Order Language and Linear Type System for Statically Enforcing Differential Privacy. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 172 (oct 2019), 30 pages. https://doi.org/10.1145/3360598

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 496–512. https://doi.org/10.1145/3192366.3192394

Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings (Lecture Notes in Computer Science)*, Riccardo Focardi and Andrew C. Myers (Eds.), Vol. 9036. Springer, 53–72. https://doi.org/10.1007/978-3-662-46666-7_4

Norman Ramsey and Avi Pfeffer. 2002. Stochastic lambda calculus and monads of probability distributions. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 154–165. https://doi.org/10.1145/503272.503288

Jason Reed and Benjamin C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. https://doi.org/10.1145/1863543.1863568

Adam Scibior, Zoubin Ghahramani, and Andrew D. Gordon. 2015. Practical Probabilistic Programming with Monads. *SIGPLAN Not.* 50, 12 (aug 2015), 165–176. https://doi.org/10.1145/2887747.2804317

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM. https://www.fstar-lang.org/papers/mumon/

Joseph Tassarotti and Robert Harper. 2018. Verified Tail Bounds for Randomized Programs. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings (Lecture Notes in Computer Science)*, Jeremy Avigad and Assia Mahboubi (Eds.), Vol. 10895. Springer, 560–578. https://doi.org/10.1007/978-3-319-94821-8_33

Hermann Thorisson. 2000. *Coupling, Stationarity, and Regeneration.*

Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. *Proc. ACM Program. Lang.* 3, POPL (2019), 36:1–36:29. https://doi.org/10.1145/3290349

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). *SIGPLAN Not.* 53, 7 (sep 2018), 132–144. https://doi.org/10.1145/3299711.3242756

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014a. LiquidHaskell: Experience with Refinement Types in the Real World. 49, 12 (sep 2014), 39–51. https://doi.org/10.1145/2775050.2633366

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014b. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* POPL (2017). https://doi.org/10.1145/3158141

Cédric Villani. 2008. *Optimal transport: old and new.*

Cédric Villani. 2009. *Optimal Transport, old and new.* Springer. https://link.springer.com/book/10.1007/978-3-540-71050-9

Peixin Wang, Hongfei Fu, Krishnendu Chatterjee, Yuxin Deng, and Ming Xu. 2020. Proving expected sensitivity of probabilistic programs with randomized variable-dependent termination time. *Proc. ACM Program. Lang.* 4, POPL (2020), 25:1–25:30. https://doi.org/10.1145/3371093

Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. 2017. A Framework for Adaptive Differential Privacy. *Proc. ACM Program. Lang.* ICFP, Article 10 (aug 2017), 29 pages. https://doi.org/10.1145/3110254

Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2019. Fuzzi: A Three-Level Logic for Differential Privacy. *Proc. ACM Program. Lang.* 3, ICFP, Article 93 (jul 2019), 28 pages. https://doi.org/10.1145/3341697