Type-Based Complexity Analysis of Probabilistic Functional Programs

Martin Avanzini, Ugo Dal Lago, Alexis Ghyselen

Type-Based Complexity Analysis of Probabilistic Functional Programs

Martin Avanzini
INRIA Sophia Antipolis
Email: martin.avanzini@inria.fr

Ugo Dal Lago
University of Bologna & INRIA Sophia Antipolis
Email: ugo.dallago@unibo.it

Alexis Ghyselen
Univ Lyon, EnsL, UCBL, CNRS, LIP
Email: alexis.ghyselen@ens-lyon.fr

***Abstract*—We show that complexity analysis of probabilistic higher-order functional programs can be carried out compositionally by way of a type system. The introduced type system is a significant extension of refinement types. On the one hand, the presence of probabilistic effects requires adopting a form of *dynamic* distribution type, subject to a coupling-based subtyping discipline. On the other hand, recursive definitions are proved terminating by way of Lyapunov ranking functions. We prove not only that the obtained type system, called ℓRPCF, provides a *sound methodology* for average case complexity analysis, but also that it is *extensionally complete*, in the sense that any average case polytime Turing machines can be encoded as a term typable in ℓRPCF.**

I. Introduction

Probabilistic models are more and more pervasive in computer science [1]–[3]. Moreover, the concept of an algorithm, originally assuming determinism, has been relaxed so as to allow probabilistic evolution since the very early days of theoretical computer science [4]. All this has given impetus to research on probabilistic programming languages, which however have been studied at a large scale only in the last twenty years, following advances in randomized computation [5], cryptographic protocol verification [6], [7], and machine learning [8]. Probabilistic programs can be seen as ordinary programs in which specific instructions are provided to make the program evolve probabilistically rather than deterministically or to attribute a score to a probabilistic branch, this way allowing for a form of soft conditioning, the latter popularized by probabilistic programming languages like CHURCH [8] or ANGLICAN [9]. Quite remarkably, many of the proposed idioms are functional, some of them providing higher-order functions.

Among the many reasons why randomized algorithms have been introduced and studied, we should certainly mention their *efficiency*: in many cases, the most efficient algorithm for the problem at hand is randomized [5]. A relevant example is primality testing, for which Miller Rabin's testing [10] remains the algorithm of choice, even if a deterministic polynomial primality testing algorithm has been known for more than fifteen years now [11]. As a consequence, analyzing the performances of probabilistic programs is a particularly interesting, although intrinsically challenging, problem.

But there is more to that: being able to analyze the complexity of randomized algorithms has the potential of enabling the average case complexity analysis, e.g., of ordinary deterministic programs: if M is such a program, say from binary strings to binary strings, to be analyzed when inputs are distributed according to a distribution family $\mathcal{D}_n$, one could compose M with another program which, given any binary string of length n, sample a string from $\mathcal{D}_n$, call it N. This way, average case complexity analysis of M is reduced to the complexity analysis of $\lambda x.M(N\,x)$. Of course, this requires the underlying programming language to be flexible enough to support the aforementioned construction, but functional languages are simply *built around* the notion of function composition, and invariably offer such a feature.

The programming language research community has indeed devoted quite some effort to this research problem, with many interesting results coming out in recent years, ranging from ranking-function based methodologies [12], to Hoare-Logic based ones [13], through amortized analysis [14] and the interpretation method [15]. Remarkably, higher-order functional programming languages have been left out of the picture, despite type-based techniques for them are well-known to be useful in the complexity analysis of deterministic programs [16], [17], but also of a problem deeply related to *complexity* analysis, namely that of *termination* analysis [18], [19].

In this paper, we introduce a system of affine refinement types for a probabilistic λ-calculus with recursion, called ℓRPCF. We then show that precise time complexity upper bounds on typed programs can be read from type derivations. Interestingly, the ℓRPCF type system can perform precise evaluation of the average complexity of some nontrivial randomized algorithms. As an example, ℓRPCF is expressive enough to type the following programs:

- A form of **biased random-walk** which, when starting at the natural number n, is proved to reach zero in an expected number of steps linear in n.
- An adaption of the **coupon collector** scheme [20], which can be proved to evaluate in $O(n\log(n))$ iterations. This scheme, although relatively simple, is already challenging to analyze: the probability of choosing a good coupon, namely one that the player still does not have, progressively decreases.
- **Randomized quicksort** [21], which can be proved by our type system to have $O(n\log n)$ average complexity. This is the first example of a precise type-based analysis of this algorithm, which represents one of the milestones of

randomized algorithmics.

The "type flexibility" one needs when typing these examples, the last one in particular, is provided by *distribution types*, by which the underlying probabilistic monad is lifted from the realm of programs to that of types. This idea, already used in [18], is brought to extreme consequences here, by making distributions types *dependent*, i.e. dynamic, themselves. As we will explain in Section II below, this is a necessary ingredient: the distribution to which a program rewrites could depend on the input, and abstracting these distributions by finite ones represents too much of an information loss, which we cannot afford when dealing with nontrivial examples such as randomized quicksort.

Of course, expressivity by itself is pointless if the type system does not guarantee any dynamical property, i.e., if it were not for so-called *type soundness*. Indeed, the two main technical contributions of this paper are as follows:

- We first of all prove a **type soundness** theorem, which goes well beyond the usual type preservation and progress template: not only are types preserved along reduction, but an expression labeling any type derivation is proved to represent an upper bound to the average complexity of evaluation for the typed term. This is in Section V-A.
- We then prove a form of **extensional completeness**, stating that for every Turing machine M working in average polynomial time, there is a term typable in ℓRPCF simulating M. This is in Section V-B below.

Compared with previous work on refinement types, our type system is on the one hand designed to be as simple as possible, and on the other specifically tailored for complexity analysis. This is different from the kind of properties traditional refinement type systems are able to enforce, which are extensional, and sometimes relational. As we will see in Section II below, complexity analysis requires a fine-grain inspection into the more *intentional* aspects of normalization, something which requires some novel ingredients. On the one hand, duplication must be properly taken into account, and in this paper we consider an *affine* type system in which, however, multiple calls to a recursively defined function are allowed. On the other hand, the outcome of any function being possibly probabilistic, it is absolutely necessary to keep track of the possible outputs at a type theoretic level, by way of *monadic*, or *distribution* types. Finally, the use of refinements is kept to a minimum, being limited to base types, and to a form of bounded universal quantification over types.

A. Related Work

This is definitely not the first contribution on type-based complexity analysis of functional programs *nor* the first work on complexity analysis of probabilistic programs. It is however the first one applying the former techniques, namely type systems, to probabilistic higher-order programs.

An extensive literature is available about type-based *termination* analysis of functional programs by way, e.g. of sized-types [22], dependent types [17] or intersection types [23]. Some of these ideas were later applied to the *complexity* analysis of deterministic functional programs [17], [24], or to *probabilistic* functional programs [18], [19], but keeping termination as the underlying problem. The only attempts at giving type-based complexity analysis of probabilistic programs were concerned with *worst-case* complexity, and not with *average-case* complexity, thus being conceptually closer to the deterministic case [25], and ultimately much easier.

In probabilistic programs, any terminating computation path is attributed a probability, and thus termination becomes a *quantitative* property. It is therefore natural to consider a program terminating when its terminating paths form a set of measure one or, equivalently, when it terminates with maximal probability. This is dubbed "almost sure termination" (AST for short) in the literature [26], and many techniques for automatically and semi-automatically checking programs for AST have been introduced in the last years [27]–[30]. Some of these techniques have been generalized to complexity analysis [31]. Other techniques, like amortized analysis, have recently been applied to imperative programs, obtaining some promising results [32], while it is not known whether these scale to (higher-order) functional programs.

Finally, a few words deserve to be spent on refinement types which, starting from the pioneering work by Freeman and Pfenning [33], have been applied to a variety of higher-order languages and verification problems, including termination, and including λ-calculi with probabilistic features [34]. Again, none of the cited works deal with the complexity analysis of probabilistic programs. Moreover, the framework we develop in this paper is simpler than the one from previous works along these lines, and can be seen as defining a *minimal* system in which, however, significant examples can be caught: indeed, we are not aware of any refinement type system in which average case complexity analysis can be carried out.

II. On Average Complexity Analysis by way of Types: a Non-Trivial Example

In this section, we will introduce the problem we are interested in tackling by way of an example, at the same time justifying some of the key and novel concepts we introduce in this paper. The example we have chosen is one of the simplest, yet very interesting, randomized algorithms, namely a probabilistic version of Hoare's Quicksort [35], sometimes called Randomized Quicksort [21],

Randomized Quicksort can be seen as being obtained from deterministic quicksort by choosing the pivot *at random* over all the elements of the input array, rather than choosing it as a fixed one (e.g. the first one). An ML-style implementation of Randomized Quicksort is in Figure 1. Here, function `quicksort` is defined in terms of a randomized procedure `ppartition`, which chooses a pivot and returns a partitioning of the given list together with that pivot element. While partitioning itself is performed by the auxiliary, pure function `partition`, the pivot is chosen at random via a primitive `Unif` : $\mathsf{Nat} \to \mathsf{Nat}$. This procedure, given a number n, samples a value uniformly from the interval $[0, n]$.

```
let rec partition (p,xs) =
  match xs with
  | []     ↦ ([],[])
  | x:xs′ ↦ let (l,r) = partition (p,xs′)
             in if x < p then (x:l,r) else (l,x:r)
let ppartition xs =
  let i = Unif (length xs − 1)
      p = nth xs i
      (l,r) = partition (p,drop p xs)
  in (l,p,r)
let rec quicksort xs =
  match xs with
  | [] ↦ []
  | _  ↦ let (l,p,r) = ppartition xs
         in quicksort l @ (p : quicksort r)
```

Fig. 1: Randomized Quicksort.

Reasoning about the average runtime of `quicksort` makes it necessary to prove some functional properties, concerning the length of the lists produced in output by `partition` in terms of the input list's length. To this end, we make use of *refinement types*. Types may thus be equipped with constraints that inhabited values satisfy. Concerning lists for instance, the type $\mathsf{List}(l \mid \Phi_l)$ captures those lists l that satisfy the constraint Φ_l, or simply $\mathsf{List}(l)$ to capture exactly the list l.

To carry on with the example, for the sake of simplicity let us assume that input lists contain no duplicates. Consider the constraint $\mathbb{L}_I^K(l) \triangleq \mathsf{Dist}(l) \wedge \mathsf{Len}(l) = I \wedge \mathsf{Max}(l) \leq K$, for integer valued indices K and I, where $\mathsf{Dist}(l)$ holds if l consists only of elements that are pairwise different, and where $\mathsf{Len}(l)$ and $\mathsf{Max}(l)$ give the length and maximal element of a list l, respectively. Based on this constraint, a suitable type for `partition` derivable in our system is drawn in Figure 2a. The depicted type depends on two further auxiliary index functions, $\mathsf{CntLe}(l,n)$ and $\mathsf{CntGt}(l,n)$, denoting the number of elements in l strictly less and greater than n, respectively. The type of `partition` demonstrates that our system permits polymorphism over indices in the form of bounded quantification $\forall a : \Phi_a.\sigma$ over indices a. Such a type can be instantiated with any index a satisfying the constraint Φ_a. As one can easily realize, the refinement type of `partition` is significantly more informative than the simple type $\mathsf{Nat} \otimes \mathsf{List} \multimap \mathsf{List} \otimes \mathsf{Nat} \otimes \mathsf{List}$. In particular, it provides some quite precise information about the *length* of the two sublists produced in output.

The main novel aspects of our type system, however, manifest themselves in the typing of randomized procedures such as `ppartition`. It is clear that the length of the two lists `ppartition` obtains from `partition` depend on the outcome of the probabilistic choice `ppartition` performs internally. ℓRPCF models this form of uncertainty by a new kind of type construction which lifts probability distributions to types, called *dynamic distribution types* (*DDT*), in which a distribution having type τ_a with probability P_a (for every $0 \leq a \leq J$) is attributed the type $\{P_a : \tau_a \mid a \leq J\}$. From the type of `partition`, it is straightforward to assign `ppartition` the type drawn in Figure 2b, assuming suitable refinement types for the auxiliary functions `nth` and `drop`. Notice that the index J is directly related to the length of the first argument. This gives a dynamic flavor to distribution types, as not only type refinements of *outputs* but also the shape of *distributions* can depend on input annotations. This demonstrates a novel aspect of our system that is absolutely essential for the runtime analysis of `quicksort`, which we carry out in a moment.

So far, the type assigned to `ppartition` reflects the information provided by the type of `partition`, coupled with the probability of choosing the pivot's index i that is uniformly sampled in the interval $[0, J]$. Towards the complexity analysis of `quicksort` itself, it is helpful to express the result of the computation of `ppartition` in terms of the length of its argument *alone*. Our system thus includes a form of *subtyping*. Subtyping ℓRPCF allows one to ascribe the type given in Figure 2c to `ppartition` based on the type from Figure 2b. The correspondence between the two types relies on several facts, such as that the sets $\{\mathsf{CntLe}(l, \mathsf{Nth}(l,i)) \mid i \leq J\}$ and $\{0, \ldots, J\}$ are in bijective correspondence, and that $\mathsf{CntLe}(l,n) + \mathsf{CntGt}(l,n) = J$. To support this kind of reasoning, subtyping not only permits to relax refinements, but it incorporates logical entailment as well as reasoning on probabilistic types via *probabilistic coupling* [36], a proof technique stemming from the analysis of stochastic processes. The revised type of `ppartition` not only tells us the length of the two generated sublists in terms of the length of its argument, it also gives us the probability with which they were sampled, crucial for the complexity analysis of `quicksort`.

To reason about expected runtimes, our type system assigns to every typeable term t a *weight*. Slightly simplifying the exposition, typing judgments have the form

$$\Phi; \Gamma \vdash_R t : \mu \,,$$

denoting that under the logical context Φ and typing context Γ, the term t receives the DDT μ and a weight of R. Inspired by *Lyapunov* or *probabilistic ranking functions* [15], also referred to as *ranking supermartingales* [37], the type system ensures that weights reduce in expectation along reductions, and consequently, relate tightly to expected runtimes. With this intuition in mind the majority of the typing rules are fairly standard. Noteworthy, assuming (for now) that no recursion is involved, the rule for sequencing probabilistic computations, viz that of let-expressions, becomes

$$\frac{\Phi; \Gamma \vdash_R t : \{P_a : \sigma_a \mid a \leq I\} \qquad \Phi, a \leq I, P_a \neq 0; \Delta, x : \sigma_a \vdash_{R_a} u : \mu_a}{\Phi; \Gamma, \Delta \vdash_{1+R+\sum_{a \leq I} P_a \cdot R_a} \texttt{let } x = t \texttt{ in } u : \sum_{a \leq I} P_a \cdot \mu_a}$$

The second premise should be read as a family of typings for u, spanning over all $a \leq I$ for which the probability P_a is non-zero. For each such a, we thus receive a type μ_a of u as well as a weight R_a, conditioned to the event that a value of type σ_a is bound to x in u after evaluating t. When u results in a value of type τ_b with probability Q_b, overall the let-expression will thus evaluate to a value of type τ_b with probability $P_a \cdot Q_b$.

$$\forall l : \mathbb{L}^K_J(l).\forall n : n \leq K \wedge \mathsf{notElem}(n,l).\ \mathsf{Nat}(n) \otimes \mathsf{List}(l) \multimap \mathsf{List}(g \mid \mathbb{L}^K_{\mathsf{CntLe}(l,n)}(g)) \otimes \mathsf{List}(g \mid \mathbb{L}^K_{\mathsf{CntGt}(l,n)}(g))$$

(a) Type of `partition`.

$$\forall l : \mathbb{L}^K_{J+1}(l).\ \mathsf{List}(l) \multimap \left\{ \tfrac{1}{J+1} : \mathsf{List}(g \mid \mathbb{L}^K_{\mathsf{CntLe}(l,\mathsf{Nth}(l,i))}(g)) \otimes \mathsf{Nat}(b \mid b \leq K) \otimes \mathsf{List}(g \mid \mathbb{L}^K_{\mathsf{CntGt}(l,\mathsf{Nth}(l,i))}(g)) \mid i \leq J \right\}$$

(b) Type of `ppartition`.

$$\mathsf{List}(l \mid \mathbb{L}^K_{J+1}(l)).\ \multimap \left\{ \tfrac{1}{J+1} : \mathsf{List}(g \mid \mathbb{L}^K_c(g)) \otimes \mathsf{Nat}(b \mid b \leq K) \otimes \mathsf{List}(g \mid \mathbb{L}^K_{J-c}(g)) \mid c \leq J \right\}$$

(c) Subtype of type `ppartition`.

Fig. 2: Refinement types for (a) `partition` and (b) `ppartition` as well as (c) subtype of `ppartitions` type.

This is precisely what the *convolution* $\sum_{a \leq I} P_a \cdot \mu_a$ computes. Concerning the ascribed overall weight, it accounts for the rank R of t, the expectation $\sum_{a \leq I} P_a \cdot R_a$ of the weight of u wrt. the distribution computed by t, and one, accounting for the let-expression itself.

While for non-recursive terms weights can be computed from leafs to roots, the situation is less clear for recursively defined functions. Which weights should be ascribed to recursive calls? To overcome this dilemma, we postulate a weight $Q(b)$ for b indexing the different recursive calls. The typing rule for fixed-points, handling recursive definitions such as `quicksort`, is then in charge of witnessing that Q gives a suitable weight. To this end, typing contexts keep track on the expected number of (immediate) recursive calls. More precisely, recursive functions y are mapped to types of the form $\{R_a : \sigma_a \mid a \leq I\}$ in the context. Unlike for DDTs, R_a can take values greater than one in case y is called multiple times. All the rules of our system then enforce that indeed recursion can be performed at most R_a times at type σ_a, for each $a \leq I$. Restricted to base arguments and ground terms for brevity, the fixed-point rule has the form

$$\frac{y : \{R_a : \mathsf{Nat}(J_a) \multimap \mu\{J_a/b\} \mid a \leq I\} \vdash_R v : \mathsf{Nat}(b) \multimap \mu \qquad \Phi \vDash Q(b) \geq 1 + R + \sum_{a \leq I} R_a \cdot Q(J_a)}{\vdash_{Q(J)} \mathtt{fix}\ y.v : \mathsf{Nat}(J) \multimap \mu\{J/b\}}$$

for b a fresh index variable. Here, $\mu\{K/b\}$ denotes the DDTs obtained from μ by substituting indices K for b. While the first premise enforces that v takes any $\mathsf{Nat}(b)$ to μ assuming R_a recursive calls at the specialized type $\mathsf{Nat}(J_a) \multimap \mu\{J_a/b\}$ ($a \leq I$), the recurrence enforces that $Q(b)$ covers the unfolding of the fixed-point, the weight R of the function itself, and the expected weights of recursive calls. For a specific argument in $\mathsf{Nat}(J)$, this then gives an overall weight $Q(J)$ and type $\mathsf{Nat}(J) \multimap \mu\{J/b\}$.

Coming back to the analysis of `quicksort`, typing its recursive definition yields the recurrence

$$Q(J+1) \geq L(J) + \textstyle\sum_{c \leq J} \frac{1}{J+1} \cdot (Q(c) + Q(J-c))\ ,$$

where $L(J)$ is a linear expression, accounting for running the body of `quicksort` at a single iteration J. This recurrence is not too difficult to solve with $Q \in \mathsf{O}(n \cdot \log(n))$, witnessing that the expected runtime of `quicksort` lies within this class.

In conclusion, ℓRPCF combines a variety of ingredients that are necessary in the analysis of non-trivial randomized algorithms such as `quicksort`, most notably (i) refinement types to establish functional properties, (ii) dynamic distribution types to accurately account for the probabilistic behaviour of programs, (iii) an expressive subtyping relation enabling proofs via probabilistic coupling, and (iv) the integration of a randomized complexity analysis within the type system, heavily inspired by probabilistic ranking functions.

III. A Probabilistic Affine Functional Language

In this section we introduce the programming language ℓRPCF that we consider throughout this work. It is basically an affine version of Plotkin's PCF [38], extended with operations for sampling from (discrete) distributions. For the sake of brevity, we consider a single operator `Unif`, for sampling from *uniform distributions*. Nevertheless, since our language permits recursion and case analysis, it is expressive enough to define on top of `Unif` a variety of sampling operations, such as sampling from *Bernoulli*, *geometric*, or any computable distribution with finite support and rational probabilities. As in the case of PCF, we consider a single data type Nat for now. Extensions to the system, so as to encompass more interesting (ground) types such as the ones for lists, are discussed later in Section VI.

A. Statics

The sets of *terms*, *values* and *types* are generated by the following grammars:

$$\begin{array}{lll} \textit{Terms} & t,u \triangleq v \mid v\ w \mid \mathtt{let}\ x = t\ \mathtt{in}\ u \\ & \qquad \mid \mathtt{match}\ v\ \mathtt{with}\ \{\ \mathtt{0} \mapsto t \mid \mathtt{s} \mapsto w\ \} \\ & \qquad \mid \mathtt{let}\ \langle x,y\rangle = v\ \mathtt{in}\ t \\ \textit{Values} & v,w \triangleq x \mid \mathtt{0} \mid \mathtt{s}(v) \mid \mathtt{Unif} \mid \lambda x.t \mid \mathtt{fix}\ x.v \mid \langle v,w\rangle \\ \textit{Types} & T,U \triangleq \mathsf{Nat} \mid T \multimap U \mid T \otimes U \end{array}$$

Terms are restricted to A-normal form [39]. In this setting, the unrestricted applications can be recovered by rewriting $t\ u$ to $\mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ y = u\ \mathtt{in}\ x\ y$, for x and y fresh variables. Apart from our sampling operator `Unif`, the constructors are standard. We follow the usual convention where application $t\ u$ binds to the left, whereas λ-abstraction $\lambda x.t$ binds to the right. Terms which are not values are called *active*. For an integer

$n \in \mathbb{N}$, we denote with $\underline{n}$ the value $\mathtt{s}(\dots,\mathtt{s}(0),\dots)$, with n occurrences of $\mathtt{s}$. The capture free substitution of variable x by value v in t is denoted $t[x := v]$. We impose a linear typing regime on terms. The typing rules are presented in Figure 3. The judgment $\Gamma \mid \Theta \vdash t : T$ means that under the *(linear) typing context* Γ and the *global typing context* Θ, the term t receives type T. Here, a typing context Γ is a non-ordered sequence of the form $x_1 : T_1 \dots x_n : T_n$. The union of two linear type contexts Γ and Δ, denoted Γ, Δ, is defined only if for each variable x with $(x : T) \in \Gamma$ and $(x : U) \in \Delta$, it holds that $T = U = \mathsf{Nat}$. Global type contexts Θ, used to treat recursive functions, are either empty or consist of a unique hypothesis $x : T \multimap U$. In the rule for fixpoints, $\ell\Gamma$ denotes a typing context where all variables are given the type Nat. For closed terms t, we abbreviate $\varnothing \mid \varnothing \vdash t : T$ by $\vdash t : T$. Notice that this affine type system permits duplication of values of base type, whereas duplication of values of functional types is prohibited. However, the system permits several recursive calls. Finally, we remark that this type system does not guarantee any complexity propriety.

B. Dynamics

Following [18], we can give $\ell\mathsf{RPCF}$ an operational semantics in terms of a binary relation $\Rightarrow$ on distributions of terms. On the non-probabilistic fragment of our language, i.e., on terms without any occurrences of `Unif`, the semantics can be seen isomorphic to the usual *weak call-by-value* reduction relation.

A *(discrete) valuation* on a countable set X is a function $v : X \to [0, +\infty]$. The *support* of v is given by $Supp(v) \triangleq \{x \in X \mid v(x) > 0\}$. The valuation v is called *finite* if its support is finite. Scalar multiplication $p \cdot v$ and finite sum $\sum_{i\in I} v_i$ are defined point-wise. We use $\{p_i : t_i \mid i \in I\}$ to denote valuations v with support $\{t_i \mid i \in I\}$ and $v(t_i) = p_i$; in the case where $I = \{1, \dots, n\}$ is finite, we may also write $\{p_1 : t_1; \dots, p_n : t_n\}$. A *(discrete) distribution* on X is a valuation $\mathcal{D} : X \to [0, 1]$ such that $\sum \mathcal{D} \triangleq \sum_{x\in X} \mathcal{D}(x) \leq 1$. It is called *proper* if $\sum \mathcal{D} = 1$. The set of distributions is closed under convex combinations. We define the relation $\leq$ on distributions with $\mathcal{D} \leq \mathcal{E}$ if $\mathcal{D}(x) \leq \mathcal{E}(x)$ for all $x \in X$.

The reduction relation $\Rightarrow$ is itself based on an auxiliary relation $\to$, depicted in Figure 4, which maps active terms to distributions. If $t \to \{p_i : t_i \mid i \in \mathcal{I}\}$, then t_i should be understood as a one-step reduct of t with probability p_i. The rules follow the standard operational semantics of PCF. Concerning sampling, `Unif` applied to a natural number $\underline{n}$ reduces to a natural number distributed uniformly in the interval $[0, n]$.

Let $\mathcal{D}_a +_{a/v} \mathcal{D}_v$ indicate the decomposition of a distribution $\mathcal{D}$ on terms into active and value part, i.e., $\mathcal{D} = \mathcal{D}_a + \mathcal{D}_v$, so that the supports of $\mathcal{D}_a$ and $\mathcal{D}_v$ consist only of active terms and values, respectively. Based on the auxiliary relation $\to$, the relation $\Rightarrow$ on distributions of terms is given by the following inference rule.

$$\frac{\mathcal{D} = \{p_i : t_i \mid i \in \mathcal{I}\} +_{a/v} \mathcal{D}_v \quad t_i \to \mathcal{E}_i \text{ for all } i \in \mathcal{I}}{\mathcal{D} \Rightarrow (\sum_{i\in\mathcal{I}} p_i \cdot \mathcal{E}_i) + \mathcal{D}_v}$$

We denote with $\Rightarrow^*$ the reflexive and transitive closure of $\Rightarrow$, and we denote with $\Rightarrow^n$ the n^{th} iteration of the reduction relation $\Rightarrow$. Notice that $\to$ is non-overlapping, and as a consequence, $\Rightarrow$ is deterministic. Moreover, distributions form an ω**CPO**. This justifies that we define the semantics $[\![t]\!]$ of a term t as the following distribution on values $[\![t]\!] \triangleq \sup\{\mathcal{D}_v \mid t \Rightarrow^* \mathcal{D}_a +_{a/v} \mathcal{D}_v\}$.

We remark that the affine type system from Figure 3 is coherent with the reduction rules, in particular, the type system enjoys subject reduction and progress in the following sense.

Proposition III.1 (Subject Reduction and Progress)**.** *Let t be such that $\vdash t : T$ holds. Then:*

1. *If $t \to \{p_i : t_i \mid i \in \mathcal{I}\}$ then $\vdash t_i : T$ for all $i \in \mathcal{I}$.*
2. *The term t is in normal form for $\to$ if and only if t is a value.*

Subject reduction implies in particular that distributions over well-typed terms are closed under reduction.

Example III.1 (Biased Random Walk). Consider the term `rwalk` depicted in Figure 5. This term performs a random walk over $\mathbb{N}$, stopping at zero. More precisely, in the case $n > 0$, `rwalk` $\underline{n}$ reduces to `rwalk` $\underline{n+1}$ with probability $\frac{1}{3}$ (the probability of sampling $p = 0$), and to `rwalk` $\underline{n-1}$ with probability $\frac{2}{3}$ (the probability of sampling $p = 1$ or $p = 2$). It can be shown that for any $n \in \mathbb{N}$, `rwalk` $\underline{n}$ reduces to `0` almost surely. Consequently, $[\![\texttt{rwalk}\ \underline{n}]\!] = \{1 : \texttt{0}\}$.

In this work, we are interested in average case complexity analysis, in terms of reduction steps. In a probabilistic setting, the reduction length from a term t can be understood as a random variable $\mathcal{S}_t$ on $\mathbb{N} \cup \{\infty\}$, with $\mathbb{P}(\mathcal{S}_t = n)$ being the probability that t evaluates to normal form in n steps, or diverges in the case $n = \infty$. The expected runtime of a term t is then defined in terms of its expectation

$$\mathbb{E}(\mathcal{S}_t) \triangleq \sum_{n=1}^{\infty} n \cdot \mathbb{P}(\mathcal{S}_t = n) = \sum_{n=0}^{\infty} \mathbb{P}(\mathcal{S}_t > n)\ .$$

Here, $\mathbb{P}(\mathcal{S}_t > n)$ gives the probability that a reduction takes strictly more than n steps. In our setting, this probability is expressed as $\sum \mathcal{D}_a^n$, for $\mathcal{D}_a^n$ the distribution of active terms reachable in n steps from t, i.e., $t \Rightarrow^n \mathcal{D}_a^n +_{a/v} \mathcal{D}_v^n$. This motivates the following definition, compare [15] for further justification of this definition.

Definition III.1 (Expected Runtime)**.** For a distribution $\mathcal{D} = \mathcal{D}_a +_{a/v} \mathcal{D}_v$, let us denote by $|\mathcal{D}|_a \triangleq \sum \mathcal{D}_a$ the total mass on active terms. The *expected runtime* of a term t with

$$\{1 : t\} = \mathcal{D}_0 \Rightarrow \mathcal{D}_1 \Rightarrow \mathcal{D}_2 \Rightarrow \mathcal{D}_3 \Rightarrow \dots$$

is defined by $\mathsf{ert}(t) \triangleq \sum_{n=0}^{\infty} |\mathcal{D}_n|_a$.

This definition is well-defined as $\Rightarrow$ is deterministic. We remark that $\mathsf{ert}(v) = 0$ for any value v, and if $t \to \{p_i : t_i \mid i \in \mathcal{I}\}$ then $\mathsf{ert}(t) = 1 + \sum_{i\in\mathcal{I}} p_i \cdot \mathsf{ert}(t_i)$. Notice that $\mathsf{ert}(t)$ can be infinite, even if t is terminating almost surely.

$$\frac{}{\Gamma, x : T \mid \Theta \vdash x : T}\,[\textsc{Var}] \qquad \frac{}{\Gamma \mid x : T \vdash x : T}\,[\textsc{RVar}] \qquad \frac{\Gamma, x : T \mid \Theta \vdash t : U}{\Gamma \mid \Theta \vdash \lambda x.t : T \multimap U}\,[\textsc{Abs}] \qquad \frac{\Gamma \mid \Theta \vdash v : T \multimap U \quad \Delta \mid \Theta \vdash w : T}{\Gamma, \Delta \mid \Theta \vdash v\ w : U}\,[\textsc{App}]$$

$$\frac{\ell\Gamma \mid x : T \multimap U \vdash v : T \multimap U}{\Gamma, \ell\Gamma \mid \Theta \vdash \mathtt{fix}\ x.v : T \multimap U}\,[\textsc{Fix}] \qquad \frac{\Gamma \mid \Theta \vdash t : T \quad \Delta, x : T \mid \Theta \vdash u : U}{\Gamma, \Delta \mid \Theta \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u : U}\,[\textsc{Let}] \qquad \frac{}{\Gamma \mid \Theta \vdash \mathtt{Unif} : \mathsf{Nat} \multimap \mathsf{Nat}}\,[\textsc{Unif}]$$

$$\frac{}{\Gamma \mid \Theta \vdash \mathtt{0} : \mathsf{Nat}}\,[\textsc{Zero}] \qquad \frac{\Gamma \mid \Theta \vdash v : \mathsf{Nat}}{\Gamma \mid \Theta \vdash \mathtt{s}(v) : \mathsf{Nat}}\,[\textsc{Succ}] \qquad \frac{\Gamma \mid \Theta \vdash v : \mathsf{Nat} \quad \Delta \mid \Theta \vdash t : T \quad \Delta \mid \Theta \vdash w : \mathsf{Nat} \multimap T}{\Gamma, \Delta \mid \Theta \vdash \mathtt{match}\ v\ \mathtt{with}\ \{\ \mathtt{0} \mapsto t \mid \mathtt{s} \mapsto w\ \} : T}\,[\textsc{Match}]$$

$$\frac{\Gamma \mid \Theta \vdash v : T \quad \Delta \mid \Theta \vdash w : U}{\Gamma, \Delta \mid \Theta \vdash \langle v, w \rangle : T \otimes U}\,[\otimes_i] \qquad \frac{\Gamma \mid \Theta \vdash v : T \otimes T' \quad \Delta, x : T, y : T' \mid \Theta \vdash t : U}{\Gamma, \Delta \mid \Theta \vdash \mathtt{let}\ \langle x, y \rangle = v\ \mathtt{in}\ t : U}\,[\otimes_e]$$

Fig. 3: Affine type system for probabilistic PCF.

$$\overline{(\lambda x.t)\ v \to \{1 : t[x := v]\}} \qquad \overline{(\mathtt{fix}\ x.w)\ v \to \{1 : w[x := \mathtt{fix}\ x.w]\ v\}} \qquad \overline{\mathtt{Unif}\ \underline{n} \to \{\tfrac{1}{n+1} : \underline{m} \mid 0 \le m \le n\}}$$

$$\overline{\mathtt{match}\ \mathtt{0}\ \mathtt{with}\ \{\ \mathtt{0} \mapsto t \mid \mathtt{s} \mapsto w\ \} \to \{1 : t\}} \qquad \overline{\mathtt{match}\ \mathtt{s}(v)\ \mathtt{with}\ \{\ \mathtt{0} \mapsto t \mid \mathtt{s} \mapsto w\ \} \to \{1 : w\ v\}}$$

$$\overline{\mathtt{let}\ x = v\ \mathtt{in}\ t \to \{1 : t[x := v]\}} \qquad \overline{\mathtt{let}\ \langle x, y \rangle = \langle v, w \rangle\ \mathtt{in}\ t \to \{1 : t[x := v][y := w]\}}$$

$$\frac{t \to \{p_i : t_i \mid i \in \mathcal{I}\}}{\mathtt{let}\ x = t\ \mathtt{in}\ u \to \{p_i : \mathtt{let}\ x = t_i\ \mathtt{in}\ u \mid i \in \mathcal{I}\}}$$

Fig. 4: Single step reduction relation $\to$ from terms to distributions.

$$\begin{array}{l}\mathtt{rwalk} \equiv \mathtt{fix}\ rw.\lambda n.\mathtt{match}\ n\ \mathtt{with}\ \{\ \mathtt{0} \mapsto \mathtt{0} \mid \mathtt{s} \mapsto \mathtt{step}\ \} \\ \quad \textit{where}\ \mathtt{step} \equiv \lambda m.\mathtt{let}\ p = \mathtt{Unif}\ \underline{2} \\ \qquad\qquad \mathtt{in\ match}\ p\ \mathtt{with}\ \{ \\ \qquad\qquad\quad \mathtt{0} \mapsto rw\ \mathtt{s}(\mathtt{s}(m)) \\ \qquad\qquad \mid\ \mathtt{s} \mapsto \lambda\ \mathtt{o}.rw\ m \\ \qquad\qquad \}\end{array}$$

Fig. 5: Biased Random Walk.

IV. The Type System

This section is devoted to introducing the main object of study of this paper, ℓRPCF, a monadic, linear refinement type system for reasoning about expected runtimes.

A. Indices and Types

As in the case of linear dependent types, base types are annotated with refinement constraints. Constraints are formed over *index terms*, i.e., first-order terms generated freely from a set of *index symbols* $\mathcal{I}$ and *index variables* $\mathcal{V}$, denoted by $f, g, \ldots$ and $a, b, \ldots$ respectively. Each symbol $f \in \mathcal{I}$ is associated with a natural number $\mathsf{ar}(f)$, its *arity*.

Definition IV.1 (Indices)**.** We consider natural and rational indices formed according to the following grammar from index symbols $f \in \mathcal{I}$ and index variables $a \in \mathcal{V}$:

$$\begin{array}{lll} \textit{natural indices} & I, J \triangleq a \mid f(I_1, \ldots, I_{\mathsf{ar}(f)}) & \\ & \qquad \mid \textstyle\sum_{a \le I} J \mid \max_{a \le I} J & \\ \textit{rational indices} & P, Q, R \triangleq \dfrac{I}{J}\,. & \end{array}$$

We use $A, B, \ldots$ to denote both natural and rational indices. In indices A of the form $\sum_{a \le I} J$ and $\max_{A \le I} J$, the variable a may occur free in J but not in I. It is considered bound in A. The set of free variables occurring in A is denoted by $\mathsf{FV}(A)$. We frequently write J_a to emphasise that the index variable occurs free in J_a, e.g., we may write $\sum_{a \le I} J_a$. As a is an index variable, no confusion can arise from this.

We assume that each function symbol $f \in \mathcal{I}$ comes equipped with an interpretation $[\![f]\!] : \mathbb{N}^{\mathsf{ar}(f)} \to \mathbb{N}$. We do not put any constraints on these functions, apart from computability. Given an *index valuation* $\vartheta : \mathcal{V} \to \mathbb{N}$, the interpretation of symbols is extended in the natural way to indices A. In particular $[\![\sum_{a \in I} J_a]\!]_\vartheta \triangleq \sum_{n \le [\![I]\!]_\vartheta} [\![J_a]\!]_{\vartheta[a \mapsto n]}$ where the index substitution $\vartheta[a \mapsto n]$ maps a to n and $b \neq a$ to $\vartheta(b)$. Similar, the interpretation of the maximum operator is defined in terms of max. In the case $[\![J]\!]_\vartheta = 0$, $[\![\frac{I}{J}]\!]_\vartheta$ is undefined.

We suppose that $\mathcal{I}$ contains for each $n \in \mathbb{N}$ a constant n interpreted by $[\![\mathsf{n}]\!] \triangleq n$ as well as symbols $+$ and $\cdot$ interpreted as addition and multiplication, respectively. For an index A, we define the capture-avoiding substitution of a in A by a natural index J, that we note $A\{J/a\}$, in the obvious way.

Definition IV.2 (Constraints on Indices)**.** Let $\phi \subseteq \mathcal{V}$ be a set of index variables. A *constraint* C on ϕ is an expression of the form $A \bowtie B$ where A, B are indices with free variables in ϕ and $\bowtie$ denotes a binary relation on integers or rationals.

Usually, we use relations in the set $\{\leq, <, =, \neq\}$ but $\bowtie$ may stand for any computable relation. Finite sets of constraints are denoted by Φ. An index valuation $\vartheta : \phi \to \mathbb{N}$ *satisfies* a constraint $A \bowtie B$, in notation $\vartheta \vDash A \bowtie B$, if $[\![A]\!]_\vartheta$ and $[\![B]\!]_\vartheta$ are defined and $[\![A]\!]_\vartheta \bowtie [\![B]\!]_\vartheta$ holds. Likewise, $\vartheta \vDash \Phi$ if $\vartheta \vDash A \bowtie B$ holds for all $(A \bowtie B) \in \Phi$. Similarly, we define $\phi; \Phi \vDash A \bowtie B$ to hold when $\vartheta \vDash A \bowtie B$ holds for all $\vartheta : \phi \to \mathbb{N}$ with $\vartheta \vDash \Phi$.

Note that every relation $\bowtie$ can be lifted to a zero/one valued index function. To avoid confusion, we make use of *Iverson's bracket* $[\cdot]$ so that $[A \bowtie B]$ is interpreted as 1 under ϕ if $\vartheta \vDash A \bowtie B$ holds, and zero otherwise. For instance, $[\![[a > 0]]\!]_\vartheta = 1$ if $\vartheta(a) > 0$, and $[\![[a > 0]]\!]_\vartheta = 0$ otherwise.

Definition IV.3 (Linear Refinement Types)**.** *Linear refinement types* σ, τ and *dynamic distribution types* (DDTs) μ, ν are defined as follows:

$$\begin{array}{rr} \textit{linear refinement types} & \sigma, \tau \triangleq \mathsf{Nat}(a \mid \Phi_a) \mid \sigma \otimes \tau \mid \rho \\ \textit{arrow types} & \rho \triangleq \sigma \multimap \mu \mid \forall a : \Phi_a . \rho_a \\ \textit{dynamic distribution types} & \mu, \nu \triangleq \{P_a : \sigma_a \mid a \leq I\} \end{array}$$

A type $\mathsf{Nat}(a \mid \Phi_a)$ represents the set of naturals n for which $\Phi\{\mathsf{n}/a\}$ is true. It should thus be understood as an existential type binding a, with a occurring free in Φ. For instance, $\mathsf{Nat}(a \mid a \leq I)$ represents natural numbers bounded by I. We may write $\mathsf{Nat}(I)$ for $\mathsf{Nat}(a \mid a = I)$.

Our type system admits polymorphism over indices in the form of *bounded universal quantification* over function types. The variable a in a type $\forall a : \Phi_a . \rho_a$ can be free in Φ_a and ρ_a, whereas it is bound in $\forall a : \Phi_a . \rho_a$.

Finally, the DDT $\{P_a : \sigma_a \mid a \leq I\}$ should be understood as a monadic type for probabilistic computations which yield with probability P_a an element of type σ_a $(a \leq I)$. Here, a can be free in P_a and σ_a but not in I, and it will be considered bound in $\{P_a : \sigma_a \mid a \leq I\}$. For instance, the DDT $\left\{\frac{1}{I+1} : \mathsf{Nat}(b) \mid b \leq I\right\}$ represents a probabilistic computation that evaluates to a natural number uniformly distributed in the interval from 0 to I. This is indeed the type that our system will assign to the term $\texttt{Unif}\ t$, where t is of type $\mathsf{Nat}(I)$. Having a more general form of DDT $\mu = \{P : \tau \mid a_1 \leq I_1, \ldots a_n \leq I_n\}$, with the intended meaning that μ represents an element of type τ with probability P for all a_i between zero and I_i (where a_i may occur free in all I_j for $j > i$), does not improve upon the expressiveness of DDTs, as to μ we can give a DDT (quantified over a single variable) with equal meaning. This justifies that we use the extended notation in informal contexts. When I is interpreted as a natural number n, we may also write the DDT $\{P : \sigma \mid a \leq I\}$ as $\{P_0 : \sigma_0, \ldots, P_n : \sigma_n\}$, where P_i and σ_i are obtained from P and σ by substituting i for a. For instance, $\{\frac{1}{2} : \sigma, \frac{1}{2} : \tau\}$ denotes a term that evaluates with probability half to a term typable as σ, and with probability half to a term typable as τ. We may assign to terms of function type that exhibit no probabilistic behaviour the usual type $\tau \multimap \sigma$ as an abbreviation for $\tau \multimap \{1 : \sigma\}$. DDTs are closed under the *convolution* $\sum_{a \leq I} P_a \cdot \mu_a$. For $\mu_a = \{Q_{a,b} : \tau_{a,b} \mid b \leq J_a\}$, this convolution assigns for all $a \leq I$ and $b \leq J_a$ the probability $P_a \cdot Q_{a,b}$ to $\tau_{a,b}$. Informally, $\sum_{a \leq I} P_a \cdot \mu_a$ is defined as $\{P_a \cdot Q_{a,b} : \tau_{a,b} \mid a \leq I, b \leq J_a\}$, see [40] for a formal definition.

Types in general are indicated by $\zeta, \xi, \ldots$. We consider types equal modulo renaming of bound variables and denote by $\zeta\{I/a\}$ the capture-avoiding substitution of the index variable a by I in the type ζ. All types are defined under some restrictions, such as the fact that the sum of probabilities in a distribution must be equal to one and that denominator in rational indices are not zero, and consequently the interpretation of indices is always well-defined. These restrictions are captured in our notion of *valid type*, specified in Figure 6. Note that validity on arrow types $\forall a : \Phi_a . \rho$ under ϕ, Φ asserts that a is bounded by some index term I, for all valuations satisfying Φ and Φ_a. In the following sections, in particular in the definition of subtyping and typing, we assume that all types are valid under the corresponding contexts ϕ, Φ.

B. Typing Rules

We arrive at the formal definition of the typing relation. As in the case of the affine type system from Section III, we distinguish between two kinds of variable contexts, viz, *linear contexts* Γ and *valuation contexts* Θ used to treat recursive definitions.

Definition IV.4 (Linear Contexts)**.** A *linear context* Γ is a non-ordered sequence $\Gamma = x_1 : \sigma_1, \ldots x_n : \sigma_n$.

With $\ell\Gamma$ we indicate a context over base types $\mathsf{Nat}(a|\Phi_a)$. For two type contexts Γ and Δ, we denote the concatenation of those contexts by Γ, Δ. This concatenation is defined if and only if for each variable x with $(x : \sigma) \in \Gamma$ and $(x : \tau) \in \Delta$, then $\sigma = \tau = \mathsf{Nat}(a \mid \Phi_a)$ for some Φ_a.

Definition IV.5 (Valuation Contexts)**.** A *valuation context* Θ is either empty or a context of the form $y : \{R_a : \sigma_a \mid a \leq I\}$. It is valid under $\phi; \Phi$ if I is valid under $\phi; \Phi$ and R_a, σ_a are valid under $\phi, a; \Phi, a \leq I$.

Notice that, in contrast to DDTs, we do not impose any condition on the sum of R_a in $\Theta = y : \{R_a : \sigma_a \mid a \leq I\}$: it could be zero or more than one in particular. Valuation context Θ refines the global context from affine type system from Section III by recording the expected number of recursive calls R_a with type σ_a $(a \leq I)$. We define the concatenation $\Theta + \Psi$ of two valuation contexts Θ and Ψ only if $\Theta = y : \{R_a : \sigma_a \mid a \leq I\}$, $\Psi = y : \{R'_a : \sigma_a \mid a \leq I\}$, in which case $\Theta + \Psi = y : \{R_a + R'_a : \sigma_a \mid a \leq I\}$.

Our linear refinement type system is given in Figure 7. Typing judgments have the form

$$\phi; \Phi; \Gamma \mid \Theta \vdash_R t : \mu \,,$$

Validity of indices and constraints:

$$\frac{\mathsf{FV}(I) \subseteq \phi}{\phi; \Phi \vdash I \; valid} \qquad \frac{\phi; \Phi \vdash I \; valid \quad \phi; \Phi \vdash J \; valid \quad \phi; \Phi \vDash J \neq 0}{\phi; \Phi \vdash \frac{I}{J} \; valid}$$

$$\frac{\phi; \Phi \vdash A \; valid \quad \phi; \Phi \vdash B \; valid}{\phi; \Phi \vdash A \bowtie B \; valid} \qquad \frac{\phi; \Phi \vdash A \bowtie B \; valid \text{ for all } (A \bowtie B) \in \Phi'}{\phi; \Phi \vdash \Phi' \; valid}$$

Validity of types:

$$\frac{a \notin \phi \quad (\phi, a); \Phi \vdash \Phi_a \; valid}{\phi; \Phi \vdash \mathsf{Nat}(a \mid \Phi_a) \; valid} \qquad \frac{\phi; \Phi \vdash \sigma \; valid \quad \phi; \Phi \vdash \tau \; valid}{\phi; \Phi \vdash \sigma \otimes \tau \; valid} \qquad \frac{\phi; \Phi \vdash \sigma \; valid \quad \phi; \Phi \vdash \tau \; valid}{\phi; \Phi \vdash \sigma \multimap \tau \; valid}$$

$$\frac{a \notin \phi \quad \phi, a; \Phi \vdash \Phi_a \; valid \quad \phi, a; \Phi, \Phi_a \vdash \rho \; valid \quad \phi, a; \Phi, \Phi_a \vDash a \leq I \text{ for some index } I}{\phi; \Phi \vdash \forall a : \Phi_a . \rho \; valid}$$

$$\frac{\phi; \Phi \vdash I \; valid \quad \phi, a; \Phi, a \leq I \vdash P_a \; valid \quad \phi, a; \Phi, a \leq I \vdash \sigma_a \; valid \quad \phi; \Phi \vDash \sum_{a \leq I} P_a = 1}{\phi; \Phi \vdash \{P_a : \sigma_a \mid a \leq I\} \; valid}$$

Fig. 6: Validity of indices, constraints and types.

with all types present in Γ, Θ and μ valid under $\phi; \Phi$. Active terms are typed with a distribution type μ and values are typed with a linear refinement type σ. The rational index R is called the *weight*. It is an indication of the expected runtime of a term and is used to extract a probabilistic ranking function when we speak about type soundness.

The system constitutes a refinement of the affine system depicted in Figure 3 in Section III. For brevity, let us give some intuition only concerning the more interesting typing rules. Rule (RVAR) matches the intuition on valuation contexts given previously. Under an evaluation context assigning $\Theta = \{R_a : \sigma_a \mid a \leq I\}$, the recursive variable variable y can be assign the J^{th} type $\sigma_a\{J/a\}$ $(J \leq I)$. The additional constraint $R_a\{J/a\} \geq 1$ ensures that the evaluation context permits at least one occurrence of a call to y with type $\sigma_a\{J/a\}$. Rules (ABS) and (APP) treat abstraction and application. Noteworthy, in the latter, the evaluation context is split among the two branches so as to properly account for the number of recursive calls performed in the two sub-terms.

The rule (FIX), governing the typing of recursive functions, follows the linearity requirements imposed by the corresponding rule from the affine type system. The body v is typed under an arrow type ρ, making use of an implicitly universally quantified variable b. The valuation type $\{R_a : \rho\{M_a/b\} \mid a \leq I\}$ expresses that v may perform on average R_a recursions at the specialized type $\rho\{M_a/b\}$ $(a \leq I)$. The side-condition on Q amounts to the recurrence relation informally introduced in Section II that establishes the probabilistic ranking condition for recursive functions. Here, the weight 1 accounts for the unfolding of the fixed point, the index R for the average runtime of v, and the sum for that of the recursive calls.

The rule (LET) for typing let-expressions $\texttt{let}\ x = t\ \texttt{in}\ u$ is on the surface fairly standard. If t is typeable as $\{P_a : \sigma_a \mid a \leq I\}$ and if u is typeable as μ_a assuming $x : \sigma_a$ the overall expression receives the type $\sum_{a \leq I} P \cdot \mu_a$. Note that the typing of u proceeds with the additional constraint $a \leq I$ stemming from the type of t. It should thus be read as a family of typings for u, with a ranging over values from zero to I for which the probability P_a is non-zero. In correspondence, the distribution context is not only split among the typing of t and u, but the latter is separated into a family of typing contexts for u, for each $a \leq I$. Concerning the weight of the proof, it is given by the weight of the proof for t plus the expected weight of the proofs for u.

Concerning the typing of values of base type, rule (ZERO) states that $\mathtt{0}$ may receive any type of the form $\mathsf{Nat}(a \mid \phi_a)$, as long as ϕ_a can be inferred to hold for $a = 0$ under the constraints Φ. Rule (SUCC) formalises that if v is a natural number a satisfying Φ_a, then $\mathtt{s}(v)$ is a non-zero natural number $b = a + 1$, i.e., $a = b - 1$, satisfying $\Phi_a\{b - 1/a\}$. Concerning the elimination rule for natural numbers, rule (MATCH), the constraints Φ_a inferred to hold for the deconstructed value, denoted by a in the typing, become available as instantiated premises for the typing of the branches. More precisely, any constraint set Φ' derivable from the global constraint Φ and $\Phi_a\{b + 1/a\}$, where b is a fresh variable denoting the deconstructed value, is available in a typing of the successor case.

Finally, let us thus detail the remain three rules dealing with polymorphism and subtyping. Rule (GEN) introduces bounded quantification over a free index variable a satisfying Φ_a. The weight, and in consequence average runtime bound, is over-approximated by the maximal value that the weight R_a of the sub-proof can take, assuming a satisfies Φ_a. It is worth remarking that the conclusion of the rule is valid only if a neither occurs in Φ nor in the variable contexts. Dual, the instantiation rule (INST) allows the instantiation of index-polymorphic type $\forall a : \Phi_a . \sigma$, with a being replaced by any index I satisfying Φ_a. Finally, the coercion rule (COERCE) introduces weakening through subtyping as well as weakening of weights in the

$$\frac{}{\phi;\Phi;\Gamma,x:\sigma \mid \Theta \vdash_0 x:\sigma}\ [\textsc{Var}] \qquad \frac{\phi;\Phi \vDash R_a\{J/a\} \geq 1 \quad \phi;\Phi \vDash J \leq I}{\phi;\Phi;\Gamma \mid y:\{R_a:\sigma_a \mid a \leq I\} \vdash_0 y:\sigma_a\{J/a\}}\ [\textsc{RVar}]$$

$$\frac{\phi;\Phi;\Gamma,x:\sigma \mid \Theta \vdash_R t:\mu}{\phi;\Phi;\Gamma \mid \Theta \vdash_{R+1} \lambda x.t:\sigma \multimap \mu}\ [\textsc{Abs}] \qquad \frac{\phi;\Phi;\Gamma \mid \Theta \vdash_R v:\sigma \multimap \mu \quad \phi;\Phi;\Delta \mid \Psi \vdash_{R'} w:\sigma}{\phi;\Phi;\Gamma,\Delta \mid \Theta+\Psi \vdash_{R+R'} v\ w:\mu}\ [\textsc{App}]$$

$$\frac{\phi,b;\Phi;\ell\Gamma \mid y:\{R_a:\rho\{M_a/b\} \mid a \leq I\} \vdash_R v:\rho \quad \phi,b;\Phi \vDash Q \geq 1+R+\sum_{a\leq I} R_a \cdot Q\{M_a/b\}}{\phi;\Phi;\Gamma,\ell\Gamma \mid \Theta \vdash_{Q\{J/b\}} \mathtt{fix}\ y.v:\rho\{J/b\}}\ [\textsc{Fix}]$$

$$\frac{\phi;\Phi;\Gamma \mid \Theta \vdash_R t:\{P_a:\sigma_a \mid a \leq I\} \quad \phi,a;\Phi,a\leq I,P_a\neq 0;\Delta,x:\sigma_a \mid y:\nu_a \vdash_{R_a} u:\mu_a}{\phi;\Phi;\Gamma,\Delta \mid \Theta+(y:\sum_{a\leq I} P_a\cdot\nu_a) \vdash_{1+R+\sum_{a\leq I} P_a\cdot R_a} \mathtt{let}\ x=t\ \mathtt{in}\ u:\sum_{a\leq I} P_a\cdot\mu_a}\ [\textsc{Let}]$$

$$\frac{}{\phi;\Phi;\Gamma \mid \Theta \vdash_1 \mathtt{Unif}:\mathsf{Nat}(I) \multimap \left\{\frac{1}{I+1}:\mathsf{Nat}(a) \mid a \leq I\right\}}\ [\textsc{Unif}]$$

$$\frac{\phi;\Phi \vDash \Phi_a\{0/a\}}{\phi;\Phi;\Gamma \mid \Theta \vdash_0 \mathtt{0}:\mathsf{Nat}(a \mid \Phi_a)}\ [\textsc{Zero}] \qquad \frac{\phi;\Phi;\Gamma \mid \Theta \vdash_R v:\mathsf{Nat}(a \mid \Phi_a)}{\phi;\Phi;\Gamma \mid \Theta \vdash_R \mathtt{s}(v):\mathsf{Nat}(b \mid \Phi_a\{b-1/a\}, b\neq 0)}\ [\textsc{Succ}]$$

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma \mid \Theta \vdash_{R_1} v:\mathsf{Nat}(a \mid \Phi_a) \qquad \phi,b;\Phi,\Phi_a\{b+1/a\} \vDash \Phi' \\ \phi;\Phi,\Phi_a\{0/a\};\Delta \mid \Psi \vdash_{R_2} t:\mu \quad \phi;\Phi,\Phi';\Delta \mid \Psi \vdash_{R_2} w:\mathsf{Nat}(b \mid \Phi_a\{b+1/a\}) \multimap \mu\end{array}}{\phi;\Phi;\Gamma,\Delta \mid \Theta+\Psi \vdash_{1+R_1+R_2} \mathtt{match}\ v\ \mathtt{with}\ \{\ \mathtt{0}\mapsto t \mid \mathtt{s}\mapsto w\ \}:\mu}\ [\textsc{Match}]$$

$$\frac{\phi;\Phi;\Gamma \mid \Theta \vdash_R v:\sigma \quad \phi;\Phi;\Delta \mid \Psi \vdash_{R'} w:\tau}{\phi;\Phi;\Gamma,\Delta \mid \Theta+\Psi \vdash_{R+R'} \langle v,w\rangle:\sigma\otimes\tau}\ [\otimes_i] \qquad \frac{\phi;\Phi;\Gamma \mid \Theta \vdash_R v:\sigma\otimes\tau \quad \phi;\Phi;\Delta,x:\sigma,y:\tau \mid \Psi \vdash_{R'} t:\mu}{\phi;\Phi;\Gamma,\Delta \mid \Theta+\Psi \vdash_{1+R+R'} \mathtt{let}\ \langle x,y\rangle=v\ \mathtt{in}\ t:\mu}\ [\otimes_e]$$

$$\frac{\phi,a;\Phi,\Phi_a;\Gamma \mid \Theta \vdash_{R_a} v:\rho}{\phi;\Phi;\Gamma \mid \Theta \vdash_{\max_{a:\Phi_a} R_a} v:\forall a:\Phi_a.\rho}\ [\textsc{Gen}] \qquad \frac{\phi;\Phi \vDash \Phi_a\{I/a\} \quad \phi;\Phi;\Gamma \mid \Theta \vdash_R v:\forall a:\Phi_a.\rho_a}{\phi;\Phi;\Gamma \mid \Theta \vdash_R v:\rho_a\{I/a\}}\ [\textsc{Inst}]$$

$$\frac{\phi;\Phi;\Gamma \mid \Theta \vdash_R t:\nu \quad \phi;\Phi \vdash (\Delta \mid \Psi) \sqsubseteq (\Gamma \mid \Theta) \quad \phi;\Phi \vdash \nu \sqsubseteq \mu \quad \phi;\Phi \vDash R \leq R'}{\phi;\Phi;\Delta \mid \Psi \vdash_{R'} t:\mu}\ [\textsc{Coerce}]$$

Fig. 7: Type System for ℓRPCF.

expected way. Here, $\phi;\Phi \vdash (\Delta \mid \Psi) \sqsubseteq (\Gamma \mid \Theta)$ should be understood as the pointwise extension of subtyping to linear and evaluation contexts. The subtyping relation is explained in the next section.

C. Subtyping

Since our type annotations give a form of refinement, it should always be possible to relax a refinement to a more liberal one. This is the role of the subtyping relation, presented in Figure 8. The judgment

$$\phi;\Phi \vdash \zeta \sqsubseteq \xi\ ,$$

should be read as: ζ is a subtype of ξ assuming Φ holds, for all free variables ϕ occurring in the judgment. Informally, this relation is set up so that subtyping gives a subset inclusion on the semantic interpretation of types outlined informally above, i.e., ξ represents at least as many terms as ζ does, under the assumption that Φ holds for all ϕ.

The core of the subtyping rules is fairly standard. A type $\mathsf{Nat}(a \mid \Phi_1)$ is a subtype of $\mathsf{Nat}(a \mid \Phi_2)$ if Φ_1 entails Φ_2. Subtyping is extended to function types and product types in the standard way, taking polarities into account. Concerning bounded quantification, rule (∀-L) formalizes that $\forall a:\Phi_a.\sigma_a$ is a subtype of each of its instantiation $\sigma_a\{I/a\}$, for every substituent I for a satisfying Φ_a under the given context. Dual, with rule (∀-R) quantifiers on the right are internalized in the context. The rule (Conv) allows one to move existential quantifiers from arguments position outwards, by means of the auxiliary conversion relation $\triangleright$, also depicted in Figure 8. E.g., this rule establishes that

$$\phi,\Phi \vdash \forall a:\Phi_a.\mathsf{Nat}(a) \multimap \mu \sqsubseteq \mathsf{Nat}(a \mid \Phi_a) \multimap \mu\ ,$$

when a is not free in μ. This judgment could not be derived in the absence of this rule, while it is coherent with the intuition that subtyping amounts to a subset inclusion on the semantic interpretation of types.

Finally, rule (Coupling) extends the subtyping relation to DDTs. This draws inspirations from *probabilistic coupling*, a proof technique employed in the analysis of stochastic processes and probabilistic programs [41]. Proving that $\mu =$

Subtyping rules:

$$\frac{\phi, a; \Phi, \Phi_1 \vDash \Phi_2}{\phi; \Phi \vdash \mathsf{Nat}(a \mid \Phi_1) \sqsubseteq \mathsf{Nat}(a \mid \Phi_2)}\ [\textsc{Nat}] \qquad \frac{\phi; \Phi \vdash \tau \sqsubseteq \sigma \quad \phi; \Phi \vdash \mu \sqsubseteq \nu}{\phi; \Phi \vdash \sigma \multimap \mu \sqsubseteq \tau \multimap \nu}\ [\textsc{Arr}] \qquad \frac{\phi; \Phi \vdash \sigma_1 \sqsubseteq \tau_1 \quad \phi; \Phi \vdash \sigma_2 \sqsubseteq \tau_2}{\phi; \Phi \vdash \sigma_1 \otimes \sigma_2 \sqsubseteq \tau_1 \otimes \tau_2}\ [\otimes]$$

$$\frac{\phi; \Phi \vDash \Phi_a\{I/a\} \quad \phi; \Phi \vdash \sigma_a\{I/a\} \sqsubseteq \tau}{\phi; \Phi \vdash \forall a : \Phi_a.\sigma_a \sqsubseteq \tau}\ [\forall\text{-L}] \qquad \frac{\phi, a; \Phi, \Phi_a \vdash \sigma_a \sqsubseteq \tau}{\phi; \Phi \vdash \sigma_a \sqsubseteq \forall a : \Phi_a.\tau}\ [\forall\text{-R}]$$

$$\frac{\sigma \triangleright \forall a : \Phi_a.\sigma_a \quad a \notin \phi \quad \phi; \Phi \vdash \tau \sqsubseteq \forall a : \Phi_a.\sigma_a \multimap \mu}{\phi; \Phi \vdash \tau \sqsubseteq \sigma \multimap \mu}\ [\textsc{Conv}] \qquad \frac{\phi; \Phi \vdash S \lhd \langle \mu \& \nu \rangle \text{ for some index term } S}{\phi; \Phi \vdash \mu \sqsubseteq \nu}\ [\textsc{Coupling}]$$

Conversion rules:

$$\frac{}{\mathsf{Nat}(a \mid \Phi) \triangleright \forall a : \Phi.\mathsf{Nat}(a)} \qquad \frac{\sigma \triangleright \forall a : \Phi.\sigma'}{\sigma \otimes \tau \triangleright \forall a : \Phi.(\sigma' \otimes \tau)} \qquad \frac{\tau \triangleright \forall a : \Phi.\tau'}{\sigma \otimes \tau \triangleright \forall a : \Phi.(\sigma \otimes \tau')}$$

Coupling rule:

$$\frac{\begin{array}{c}\phi, a, b; \Phi, a \le I, b \le J \vdash S_{a,b} \textit{ valid} \\ \phi, b; \Phi, b \le J \vDash \sum_{a \le I} S_{a,b} = Q_b \end{array} \quad \phi, a, b; \Phi, a \le I, b \le J, S_{a,b} \neq 0 \vdash \sigma_a \sqsubseteq \tau_b}{\phi; \Phi \vdash S_{a,b} \lhd \langle \{P_a : \sigma_a \mid a \le I\} \,\&\, \{Q_b : \tau_b \mid b \le J\} \rangle} \quad \phi, b; \Phi, a \le I \vDash \textstyle\sum_{b \le J} S_{a,b} = P_a$$

Fig. 8: Subtyping, conversion and coupling rules.

$\{P_a : \sigma_a \mid a \le I\}$ is a subtype of $\nu = \{Q_b : \tau_b \mid b \le J\}$ amounts to establishing a coupling $S_{a,b} \lhd \langle \mu \& \nu \rangle$ for some rational index term $S_{a,b}$. This index term witnesses the existence of a distribution ϑ over pairs of types, ordered by the subtype relation, so that the marginal distributions of ϑ coincide with μ and ν. In consequence, each fraction of a type recorded in μ is covered by a fraction of a subtype recorded in ν. For instance, if $\phi; \Phi \vdash \sigma_1 \sqsubseteq \tau$ and $\phi; \Phi \vdash \sigma_2 \sqsubseteq \tau$ then

$$\phi; \Phi \vdash \{\tfrac{1}{2} : \sigma_1, \tfrac{1}{2} : \sigma_2\} \sqsubseteq \{\tfrac{1}{4} : \sigma_1, \tfrac{1}{4} : \sigma_2, \tfrac{1}{2} : \tau\} .$$

Here, half of the probability $\frac{1}{2}$ of σ_1 in the left-hand side is covered by the probability of σ_1 on the right-hand side, and the remaining fraction by half of the probability $\frac{1}{2}$ of τ; similar for σ_2.

The subtyping relation verifies the conditions of a preorder in the following sense:

Lemma IV.1 (Subtyping and Preorder)**.** *Let ϕ be an set of index variables and Φ be a set of constraints. Let ζ, ξ, κ be valid types under $\phi; \Phi$. Then:*

1. *$\phi; \Phi \vdash \zeta \sqsubseteq \zeta$; and*
2. *if $\phi; \Phi \vdash \zeta \sqsubseteq \xi$ and $\phi; \Phi \vdash \xi \sqsubseteq \kappa$ then $\phi; \Phi \vdash \zeta \sqsubseteq \kappa$.*

D. An Example

Before we elaborate on the form of soundness and completeness results that can be derived from our system, we demonstrate its use on a simple example. Reconsider the random walk depicted in Figure 5. We show

$$\vdash_{Q\{J/b\}} \texttt{rwalk} : \mathsf{Nat}(J) \multimap \mathsf{Nat}(0) ,$$

for every (ground) index J and $Q = 24b + 8$. Typing proceeds as follows:

1. We first make use of rule (Fix). To this end, we type its body
$$\lambda n.\texttt{match}\ n\ \texttt{with}\ \{\ \texttt{0} \mapsto \texttt{0} \mid \texttt{s} \mapsto \texttt{step}\ \} ,$$
with type $\mathsf{Nat}(b) \multimap \mathsf{Nat}(0)$ under weight 7, using the valuation context assigning to rw the DDT
$$\nu \triangleq \{[b > 0] \cdot \frac{a+1}{3} : \mathsf{Nat}(b+1-2a) \multimap \mathsf{Nat}(0) \mid a \le 1\} .$$
Informally, the chosen valuation context states that in the base case $b = 0$, `rwalk` does not make use of rw, i.e., does not call itself recursively. In the case $b > 0$, the DDT ν simplifies to
$$\{\frac{1}{3} : \mathsf{Nat}(b+1) \multimap \mathsf{Nat}(0); \frac{2}{3} : \mathsf{Nat}(b-1) \multimap \mathsf{Nat}(0)\} ,$$
thus suitably modelling the recursive calls performed in the step case. Notice that the side condition of the fixed-point rule states
$$b; \top \vDash Q \ge 8 + \sum_{a \le 1} [b > 0] \cdot \frac{a+1}{3} \cdot Q\{b + 1 - 2a/b\} ,$$
which is easy to verify by case analysis on b.
2. To type the body of `rwalk`, we assume by rule (Abs) that the variable n has type $\mathsf{Nat}(b)$, and type the match-expression with rule (Match) and weight 7. By rule (Var) we have $b; \top, n : \mathsf{Nat}(b) \mid \cdot \vdash_0 n : \mathsf{Nat}(b)$. Concerning the typing of the zero-branch, by rule (Zero) we can assign `0` the type $\mathsf{Nat}(0)$, independent of contexts with weight 0, as desired. Concerning the typing of the successor-branch, recall that by convention $\mathsf{Nat}(b)$ abbreviates $\mathsf{Nat}(a \mid a = b)$. Observe that $\mathsf{Nat}(a \mid a + 1 = b)$ is equivalent to $\mathsf{Nat}(b-1)$, by means of rule (Coerce) it suffices thus to show
$$b; b \ge 1; \cdot; rw : \nu \vdash_5 \texttt{step} : \mathsf{Nat}(b-1) \multimap \mathsf{Nat}(0) .$$
The additional premise $b \ge 1$ expresses that we match against a non-zero number. Formally, we use $b \ge 1$ for Φ' in rule (Match) and verify $b, a; a + 1 = b \vDash b \ge 1$.

3. To type `step`, i.e., the term

   $\lambda m.\texttt{let}\ p = \texttt{Unif}\ \underline{2}\ \texttt{in}\ e$ *where*
   $e \triangleq \texttt{match}\ p\ \texttt{with}\ \{\ \texttt{0} \mapsto rw\ \texttt{s}(\texttt{s}(m)) \mid \texttt{s} \mapsto \lambda o.rw\ m\ \}$

   as indicated, we assume $m : \mathsf{Nat}(b-1)$ and type the let-expression as $\mathsf{Nat}(0)$ with weight 4 using rule (LET). To this end, it is not difficult to derive a typing with weight one assigning the DDT $\{\frac{1}{3} : \mathsf{Nat}(c) \mid c \leq 2\}$ to $\texttt{Unif}\ \underline{2}$. Let

   $$\xi \triangleq \{1 : \mathsf{Nat}(b+1-2[c>0]) \multimap \mathsf{Nat}(0)\}\ .$$

   Notice that under the assumption $b \geq 1$, the valuation ν assigned to rw coincides with $\sum_{c\leq 2} \frac{1}{3}\cdot\xi$. To conclude the typing, it is thus sufficient to show

   $$b, c; b \geq 1, 2 \geq c; \Gamma \mid rw : \xi \vdash_2 e : \mathsf{Nat}(b-1) \multimap \mathsf{Nat}(0)\ ,$$

   where $\Gamma \triangleq m : \mathsf{Nat}(b-1), p : \mathsf{Nat}(c)$.
4. This final judgment is derivable with rule (MATCH). Briefly, from the context we have $p : \mathsf{Nat}(c)$. To type the zero-branch, by rule (RVAR) we get a proof of

   $$rw : \mathsf{Nat}(b+1-2[c>0]) \multimap \mathsf{Nat}(0)\ .$$

   As we may assume $c = 0$ in this branch, one application of (COERCE) yields $rw : \mathsf{Nat}(b+1) \multimap \mathsf{Nat}(0)$ and $rw\ \texttt{s}(\texttt{s}(m)) : \mathsf{Nat}(0)$ follows using $m : \mathsf{Nat}(b-1)$. Similarly, in the successor-branch, we may assume $c > 0$ and derive $rw : \mathsf{Nat}(b-1) \multimap \mathsf{Nat}(0)$ from which a proof of the final premise of rule (MATCH) can be easily derived. One can verify that the overall weight of the typing of the match-expression e is two, as required.

V. Soundness and Extensional Completeness

In this section, we briefly elaborate on the form of soundness and completeness that can be obtained from our system. Detailed justification of these results are given in the technical report [40].

A. Soundness

Our type system gives a sound methodology for reasoning about the expected runtime of programs written in ℓRPCF. More precisely, if a term t is typeable with weight R, then R bounds its expected runtime. Towards this soundness result, we prove a form of *subject reduction*, i.e., that typeability is preserved under reductions. Crucially, our adapted subject reduction lemma shows that the weight of the typing derivation reduces in expectation, so as to obtain a bound on the overall expected runtime of t.

The crux of subject reduction for higher-order systems lies often within a *substitution lemma*. This is also the case here. The separation of variable contexts into linear contexts Γ and valuation contexts Θ gives rise to two separate lemmas. Since we are in a call-by-value setting, we restrict our attention to values.

Lemma V.1 (Substitution Lemma for Linear Contexts)**.** *For every term t and value v,*

$$\phi; \Phi; \Gamma, x : \sigma \mid \Theta \vdash_R t : \mu \text{ and } \phi; \Phi; \cdot \vdash_{R'} v : \sigma\ ,$$

implies

$$\phi; \Phi; \Gamma \mid \Theta \vdash_{R+R'} t[x := v] : \mu\ .$$

Thus the weight of the derivation for $t[x := v]$, and in consequence its expected runtime, depends only additively on the weight of the derivation for the substituent v. This is in essence a consequence of linearity of the type system. Although elements of type $\mathsf{Nat}(a \mid \Phi_a)$ may be duplicated, ground values of this type can, without loss of generality, always be type with weight zero.

In contrast, our substitution lemma for valuation contexts is slightly more involved as no linearity condition on recursion variables is introduced. However, duplication is controlled by the context.

Lemma V.2 (Substitution Lemma for Valuation Contexts)**.** *For every term t and value v,*

$$\phi; \Phi; \Gamma \mid y : \{R_a : \rho_a \mid a \leq I\} \vdash_R t : \mu\ ,$$

and

$$\phi, a; \Phi, a \leq I, R_a \neq 0 \vdash_{R'_a} v : \rho_a\ ,$$

implies

$$\phi; \Phi; \Gamma \mid \cdot \vdash_{R+\sum_{a\leq I} R_a \cdot R'_a} t[y := v] : \mu\ .$$

To get some intuition, recall that the valuation context $\{R_a : \rho_a \mid a \leq I\}$ expresses that t makes at most R_a uses of the recursive variable y with type ρ_a, while R'_a gives an indication on the average complexity of the function v ($a \leq I$). The overall weight $R + \sum_{a\leq I} R_a \cdot R'_a$ thus accounts for the complexity of evaluating t together with all the potential calls to the substituent v.

To state subject reduction in a concise way, let us extend the typing relation to distributions $\mathcal{D}$ by means of the convolution of the types of terms in the support of $\mathcal{D}$.

$$\frac{\phi; \Phi \vdash_{R_i} t_i : \zeta_i \text{ for all } i \in \mathcal{I} \quad \phi; \Phi \vDash \sum_{i\in\mathcal{I}} p_i \cdot \zeta_i \sqsubseteq \mu}{\phi; \Phi \vdash_{\sum_{i\in\mathcal{I}} p_i \cdot R_i} \sum_{i\in\mathcal{I}} p_i \cdot \{1 : t_i\} : \mu}$$

Notice that the weight of the derivation is given by the average weight of the sub-derivations. This weight, as well the involved convolutions, are well-defined for all countable sets $\mathcal{I}$ and rationals p_i. The following lemma is based on the single-step reduction relation $\to$ introduced in Figure 4.

Lemma V.3 (Subject Reduction)**.** *Let t be an active term with $\phi; \Phi \vdash_R t : \zeta$. If $t \to \mathcal{D}$ then $\phi; \Phi \vdash_{R'} \mathcal{D} : \zeta$ with $\phi; \Phi \vDash R \geq 1 + R'$.*

This lemma extends to the reduction relation $\Rightarrow$ on term distributions. The following gives an extension to many-step reductions.

$$\frac{\phi;\Phi \vDash \Phi_l\{\mathsf{nil}/l\}}{\phi;\Phi;\Gamma \mid \Theta \vdash_0 \mathtt{nil} : \mathsf{List}(l \mid \Phi_l)}\ [\textsc{Nil}] \qquad \frac{\phi;\Phi;\Gamma \mid \Theta \vdash_R v : \mathsf{Nat}(a \mid \Phi_a) \quad \phi;\Phi;\Delta \mid \Psi \vdash_{R'} w : \mathsf{List}(l \mid \Phi_l)}{\phi;\Phi;\Gamma,\Delta \mid \Theta + \Psi \vdash_{R+R'} \mathtt{cons}(v,w) : \mathsf{List}(l \mid \Phi_l\{\mathsf{tl}(l)/l\}, \Phi_a\{\mathsf{hd}(l)/a\}, l \neq nil)}\ [\textsc{Cons}]$$

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma \mid \Theta \vdash_R v : \mathsf{List}(l \mid \Phi_0) \qquad \phi,a,l;\Phi,\Phi_0\{\mathtt{cons}(a,l)/l\} \vDash \Phi_a,\Phi_l,\Phi' \\ \phi;\Phi,\Phi_0\{\mathsf{nil}/l\};\Delta \mid \Psi \vdash_R t : \mu \quad \phi;\Phi,\Phi';\Delta \mid \Psi \vdash_{R'} w : \mathsf{Nat}(a \mid \Phi_a) \otimes \mathsf{List}(l \mid \Phi_l) \multimap \mu\end{array}}{\phi;\Phi;\Gamma,\Delta \mid \Theta + \Psi \vdash_{1+R+R'} \mathtt{match}\ v\ \mathtt{with}\ \{\ \mathtt{nil} \mapsto t \mid \mathtt{cons} \mapsto w\}\ : \mu}\ [\textsc{MatchList}]$$

Fig. 9: Type system for lists of natural numbers.

Theorem V.4 (Many-step Subject Reduction)**.** *Let t be an active term with $\phi;\Phi \vdash_R t : \zeta$. If*

$$\{1 : t\} = \mathcal{D}_0 \Rightarrow \mathcal{D}_1 \Rightarrow \mathcal{D}_2 \Rightarrow \mathcal{D}_3 \Rightarrow \dots ,$$

then $\phi;\Phi \vdash_{R_n} \mathcal{D}_n : \zeta$ with $\phi,\Phi \vDash R \geq \sum_{0\leq j<n} |\mathcal{D}_j|_a + R_n$ for all $n \geq 1$.

For any index valuations ϑ on ϕ satisfying Φ, the theorem states $[\![R]\!]_\vartheta \geq \sum_{j=0}^{n-1} |\mathcal{D}_j|_a$ for all $n \in \mathbb{N}$. As $\mathsf{ert}(t) = \sum_{n=0}^{\infty} |\mathcal{D}_n|_a$ our main theorem, *type soundness*, is an immediate consequence.

Corollary V.5 (Type Soundness)**.** *For any active term t,*

$$\phi;\Phi \vdash_R t : \zeta \quad \Longrightarrow \quad \mathsf{ert}(t) \leq [\![R]\!]_\vartheta ,$$

for all index valuations ϑ on ϕ with $\vartheta \vDash \Phi$.

B. Extensional Completeness

The careful reader may have wondered about the completeness of ℓRPCF, given that similar type systems, and noticeably dℓPCF [17], have been proved to be relatively complete as tools for the complexity analysis of *deterministic* programs: the time complexity of any terminating PCF program can be approximated.

Unfortunately, proving such an *intensional* completeness result is very far from trivial in the presence of probabilistic choice. Any such completeness result is proved by way of some form of *subject expansion*, which allows to prove type preservation backward, from values (which are trivially typable) to any term. This proof methodology is not available here, because normal forms are distributions, and they can even have infinite support. So, even typing *values* is not possible.

In the spirit of implicit computational complexity, however, we can at least prove that any *function* computable within the prescribed resource bounds has a counterpart in our type system, as an immediate consequence of the following theorem:

Theorem V.6. *For every probabilistic Turing Machine $\mathcal{M}$ working in average polynomial time, there is a term $t_\mathcal{M}$ that encodes $\mathcal{M}$ such that $t_\mathcal{M}$ has type*

$$\forall w : |w| \leq K, \mathsf{Word}(w) \multimap \mathsf{Word}(x \mid \top)$$

for any natural index K, and such that the runtime inferred through ℓRPCF is polynomial in K.

This theorem is formulated in terms of *word* types Word, i.e., bit-strings over zero and one. The extension of the type system to words is straight forward while retaining soundness.

VI. Extensions

In this section, we indicate how ℓRPCF can be extended to (ground) base types beyond natural numbers. The extension for primitive types like words Word is straight forward, more interesting is the treatment of compound data types like lists or words. For brevity, we exemplify the case of lists over natural numbers only, which allows us to formalize the treatment of `quicksort` from Section II, see the technical report [40].

Concerning the language, we extend the grammar of values by `nil` and $\mathtt{cons}(v,w)$, and that of active terms by a new construct $\mathtt{match}\ v\ \mathtt{with}\ \{\ \mathtt{nil} \mapsto t \mid \mathtt{cons} \mapsto w\}$ for pattern matching on lists. The reduction semantics of the latter is as expected. Concerning the refinement type system, we add a $\mathsf{List}(l \mid \Phi_l)$ to the grammar of linear refinement types, for *list index variable* l. As for list indices, we assume at least the presence of indices $\mathsf{hd}(L), \mathsf{tl}(L)$ denoting the head and tail of a list L, respectively, as well as a new functions nil and $\mathsf{cons}(I,L)$ that constructing lists on the index level. We can also account for bounded quantification over lists, requiring that the length of the lists must be bounded and the elements of the list itself must all be bounded. Consequently, this permits quantification over finite sets of lists.

Finally, the extension of the type system is given in Figure 9. Crucially, this extension can be shown to still validate our central soundness theorem.

VII. Conclusion

In this work, we have introduced ℓRPCF, a type system for an affine version of Plotkin's PCF extended with operations for probabilistic sampling. This system presents a sound and extensionally complete method for reasoning about expected runtimes. Although type systems have been successfully applied in the context of type-based complexity analysis, to the authors best knowledge this work constitutes the first where type-based techniques are used to reason about the complexity of higher-order, probabilistic programs.

Acknowledgments

The three authors are partially supported by the ANR project ELICA, and by the INRIA/JSPS project CRECOGI. The third author is also partially supported by the LABEX MILYON/ANR-10-LABX-0070.

References

[1] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*. MIT Press, 2001.

[2] J. Pearl, *Probabilistic reasoning in intelligent systems - networks of plausible inference*, ser. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.

[3] S. Thrun, "Robotic mapping: A survey," *Exploring artificial intelligence in the new millennium*, vol. 1, no. 1-35, p. 1, 2002.

[4] K. De Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro, "Computability by probabilistic machines," *Automata Studies*, vol. 34, pp. 183–198, 1956.

[5] R. Motwani and P. Raghavan, *Randomized algorithms*. Cambridge University Press, 1995.

[6] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," in *Proc. of 36th POPL*. ACM, 2009, pp. 90–101.

[7] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin, "Computer-aided security proofs for the working cryptographer," in *Proc. of 31st CRYPTO*, ser. LNCS, vol. 6841. Springer, 2011, pp. 71–90.

[8] N. D. Goodman, V. K. Mansinghka, D. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: A language for generative models," in *Proc. of 24th UAI*. AUAI Press, 2008, pp. 220–229.

[9] D. Tolpin, J.-W. van de Meent, and F. Wood, "Probabilistic programming in Anglican," in *Machine Learning and Knowledge Discovery in Databases*, ser. LNCS, vol. 9286. Springer, 2015, pp. 308–311.

[10] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128 – 138, 1980.

[11] M. Agrawal, N. Kayal, and N. Saxena, "PRIMES is in P," *Annals of Mathematics*, vol. 160, no. 2, pp. 781–793, 2004.

[12] K. Chatterjee, P. Novotný, and D. Zikelic, "Stochastic invariants for probabilistic termination," in *Proc. of 44th POPL*. ACM, 2017, pp. 145–160.

[13] B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo, "Weakest precondition reasoning for expected run–times of probabilistic programs," in *Proc. of 25th ESOP*, ser. LNCS, vol. 9632. Springer, 2016, pp. 364–389.

[14] S. Jost, K. Hammond, H.-W. Loidl, and M.Hofmann, "Static determination of quantitative resource usage for higher-order programs," in *Proc. of 37th POPL*. ACM, 2010, pp. 223–236.

[15] M. Avanzini, U. D. Lago, and A. Yamada, "On probabilistic term rewriting," in *Proc. of 14th FLOPS*, ser. LNCS, vol. 10818. Springer, 2018, pp. 132–148.

[16] M. Hofmann and S. Jost, "Type-based amortised heap-space analysis," in *Proc. of 15th ESOP*, ser. LNCS, vol. 3924. Springer, 2006, pp. 22–37.

[17] U. Dal Lago and M. Gaboardi, "Linear dependent types and relative completeness," *LMCS*, vol. 8, no. 4, pp. 1–44, 2011.

[18] U. Dal Lago and C. Grellois, "Probabilistic termination by monadic affine sized typing," in *Proc. of 26th ESOP*, ser. LNCS, vol. 10201. Springer, 2017, pp. 393–419.

[19] F. Breuvart and U. Dal Lago, "On intersection types and probabilistic lambda calculi," in *Proc. of 20th PPDP*. ACM, 2018, pp. 8:1–8:13.

[20] F. Mosteller, *Fifty challenging problems in probability with solutions*. Courier Corporation, 1987.

[21] M. Mitzenmacher and E. Upfal, *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[22] J. Hughes, L. Pareto, and A. Sabry, "Proving the correctness of reactive systems using sized types," in *Proc. of 23rd POPL*. ACM, 1996, pp. 410–423.

[23] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini, "A filter lambda model and the completeness of type assignment," *The Journal of Symbolic Logic*, vol. 48, no. 4, pp. 931–940, 1983.

[24] D. De Carvalho, "Execution time of lambda-terms via denotational semantics and intersection types," *Mathematical Structures in Computer Science*, vol. 28, no. 7, pp. 1169–1203, 2018.

[25] U. Dal Lago and P. Parisen Toldin, "A higher-order characterization of probabilistic polynomial time," in *Revised Selected Papers of 2nd FOPARA*, ser. LNCS, vol. 7177. Springer, 2012, pp. 1–18.

[26] O. Bournez and C. Kirchner, "Probabilistic Rewrite Strategies. Applications to ELAN," in *Proc. of 13th RTA*, ser. LNCS, vol. 2378. Springer, 2002, pp. 252–266.

[27] J. Esparza, A. Gaiser, and S. Kiefer, "Proving termination of probabilistic programs using patterns," in *Proc. of 24th CAV*, ser. LNCS, vol. 7358. Springer, 2012, pp. 123–138.

[28] F. Ferrer, L. María, and H. Hermanns, "Probabilistic termination: Soundness, completeness, and compositionality," in *Proc. of 42nd POPL*. ACM, 2015, pp. 489–501.

[29] K. Chatterjee, H. Fu, P. Novotný, and R. Hasheminezhad, "Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs," in *Proc. of 43rd POPL*. ACM, 2016, pp. 327–342.

[30] K. Chatterjee, H. Fu, and A. K. Goharshady, "Termination analysis of probabilistic programs through Positivstellensatz's," in *Proc. of 28th CAV*, ser. LNCS, vol. 9779. Springer, 2016, pp. 3–22.

[31] K. Chatterjee, H. Fu, and A. Murhekar, "Automated recurrence analysis for almost-linear expected-runtime bounds," in *Proc. of 29th CAV*, ser. LNCS, vol. 10426. Springer, 2017, pp. 118–139.

[32] V. C. Ngo, Q. Carbonneaux, and J. Hoffmann, "Bounded expectations: Resource analysis for probabilistic programs," in *Proc. of 39th PLDI*. ACM, 2018, pp. 496–512.

[33] T. Freeman and F. Pfenning, "Refinement types for ML," in *Proc. of 12th PLDI*, 1991, pp. 268–277.

[34] G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub, "Higher-order approximate relational refinement types for mechanism design and differential privacy," in *Proc. of 42nd POPL*, vol. 50. ACM, 2015, pp. 55–68.

[35] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[36] T. Lindvall, *Lectures on the Coupling Method*, ser. Dover Books on Mathematics Series. Dover Publications, Incorporated, 2002.

[37] A. Chakarov and S. Sankaranarayanan, "Probabilistic Program Analysis with Martingales," in *Proc. of 25th CAV*, ser. LNCS, vol. 8044. Springer, 2013, pp. 511–526.

[38] G. D. Plotkin, "LCF Considered as a Programming Language," *TCS*, vol. 5, no. 3, pp. 223–255, 1977.

[39] A. Sabry and M. Felleisen, "Reasoning about programs in continuation-passing style," *Lisp and symbolic computation*, vol. 6, no. 3-4, pp. 289–360, 1993.

[40] A. G. M. Avanzini U. Dal Lago, "Type-Based Complexity Analysis of Probabilistic Functional Programs (Technical Report)," University of Bologna, INRIA Sophia Antipolis and ENS Lyon, Tech. Rep., 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02103943

[41] G. Barthe, B. Grégoire, J. Hsu, and P.-Y. Strub, "Coupling proofs are probabilistic product programs," in *Proc. of 44th POPL*. ACM, 2017, pp. 161–174.