Probabilistic Refinement Session Types

QIANCHENG FU, Boston University, USA
ANKUSH DAS, Boston University, USA
MARCO GABOARDI, Boston University, USA

Session types provide a formal type system to define and verify communication protocols between message-passing processes. In order to analyze randomized systems, recent works have extended session types with probabilistic type constructors. Unfortunately, all the proposed extensions only support constant probabilities which limits their applicability to real-world systems. Our work addresses this limitation by introducing probabilistic refinement session types which enable symbolic reasoning for concurrent probabilistic systems in a core calculus we call PReST. The type system is carefully designed to be a *conservative extension* of refinement session types and supports both probabilistic and regular choice type operators. We also implement PReST in a prototype which we use for validating probabilistic concurrent programs. The added expressive power leads to significant challenges, in both the meta theory and implementation of PReST, particularly with type checking: it requires reconstructing intermediate types for channels when type checking probabilistic branching expressions. The theory handles this by semantically quantifying refinement variables in probabilistic typing rules, a deviation from standard refinement type systems. The implementation relies on a bi-directional type checker that uses an SMT solver to reconstruct the intermediate types minimizing annotation overhead and increasing usability. To guarantee that probabilistic processes are almost-surely terminating, we integrate cost analysis into our type system to obtain expected upper bounds on recursion depth. We evaluate PReST on a wide variety of benchmarks from 4 categories: *(i)* randomized distributed protocols such as Itai and Rodeh's leader election, bounded retransmission, etc., *(ii)* parametric Markov chains such as random walks, *(iii)* probabilistic analysis of concurrent data structures such as queues, and *(iv)* distributions obtained by composing uniform distributions using operators like max and sum. Our experiments show that the PReST type checker scales to large programs with sophisticated probabilistic distributions.

CCS Concepts: • **Theory of computation** → **Distributed computing models**; **Probabilistic computation**; **Type theory**; **Operational semantics**.

Additional Key Words and Phrases: Session Types, Refinement Types, Probabilistic Reasoning, Message-passing Concurrency, Resource Analysis

ACM Reference Format:
Qiancheng Fu, Ankush Das, and Marco Gaboardi. 2025. Probabilistic Refinement Session Types. *Proc. ACM Program. Lang.* 9, PLDI, Article 214 (June 2025), 26 pages. https://doi.org/10.1145/3729317

1 Introduction

Session types [14, 43–45] provide a formal discipline for message-passing systems where types specify communication protocols that processes must adhere to. Type checking then ensures error-free communication and also guarantees deadlock freedom. Recent works [2, 27, 46] have introduced *probabilistic session types* which enhance session types with annotations that capture probability distributions over interactions in communication protocols. For example, in the NomosPro

Authors' Contact Information: Qiancheng Fu, Boston University, Boston, USA, qcfu@bu.edu; Ankush Das, Boston University, Boston, USA, ankushd@bu.edu; Marco Gaboardi, Boston University, Boston, USA, gaboardi@bu.edu.

ACM 2475-1421/2025/6-ART214
https://doi.org/10.1145/3729317

system [27], the following type bools

$$\mathsf{bools} \triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{0.6} : \mathsf{bools},\ \mathbf{false}^{0.4} : \mathsf{bools}\}$$

describes a protocol where the provider *sends* (indicated by type constructor $\oplus_{\mathsf{P}}$) an infinite stream of *biased booleans* to its client with the probability of true being 0.6 and false being 0.4.

Probabilistic session types can help analyze simple randomized protocols and Markov chains but are severely limited in their applicability to model real-world systems because they can only express constant probability distributions. For instance, consider a queue running on a resource constrained system. The probability of insertion failure dynamically increases as the queue grows. To model this with constants, one would need to define a new queue type for each length of interest. The following types model queues with an exponentially decreasing rate of insertion success.

$$\begin{aligned}
\mathsf{queue}_0 &\triangleq \&\{\mathbf{ins} : A \multimap \oplus_{\mathsf{P}}\{\mathbf{ok}^{0.8} : \mathsf{queue}_1,\ \mathbf{fail}^{0.2} : \mathsf{queue}_0\}, \ldots\}\\
\mathsf{queue}_1 &\triangleq \&\{\mathbf{ins} : A \multimap \oplus_{\mathsf{P}}\{\mathbf{ok}^{0.4} : \mathsf{queue}_2,\ \mathbf{fail}^{0.6} : \mathsf{queue}_1\}, \ldots\}\\
\mathsf{queue}_2 &\triangleq \&\{\mathbf{ins} : A \multimap \oplus_{\mathsf{P}}\{\mathbf{ok}^{0.2} : \mathsf{queue}_3,\ \mathbf{fail}^{0.8} : \mathsf{queue}_2\}, \ldots\} \quad \ldots\ldots
\end{aligned}$$

The queue_0 type represents queues of size 0. After a queue_0 provider receives an insertion message (indicated by & constructor) and data of type A (using $\multimap$), it may report success (**ok**) with probability 0.8 and continue as queue_1 (a queue of size 1) or it may report failure (**fail**) with probability 0.2 and remain as queue_0. Now from the definition of queue_1, we can observe that it behaves the exact same way, except the success rate is halved. And the same holds for queue_2 which, in turn, requires type queue_3. Clearly, this encoding scheme requires an infinite series of queue types where queue_{i+1} has half the success rate of queue_i. This means, in practice, that it is impossible to type and implement queues of arbitrary length. In general, lack of *symbolic probabilities* prohibits the specification of many k-way protocols such as leader election [48] and crowd forwarding [64] where probability distributions are dependent on the number of participants.

This paper introduces PReST, a novel *probabilistic refinement session-typed language* that parameterizes probabilities using refinement variables. The introduction of refinements allows programmers to specify parameterized probabilistic protocols and verify lightweight properties about probabilistic concurrent message-passing programs. Although prior works [24–26, 37, 71, 72] have used refinements to express sizes and values of data structures (e.g., $\mathsf{nat}[n]$ for natural numbers of value n), such refinements were never applied to probabilistic reasoning. Using PReST, we can replace all the queue types above with a single indexed $\mathsf{queue}[n, p, c]$ type as follows

$$\begin{aligned}
&\mathsf{queue}[n, p, c \mid 0 \le n \wedge 0 \le p \le 1 \wedge 0 < c < 1] \triangleq\\
&\quad \&\{\mathbf{ins} : A \multimap \oplus_{\mathsf{P}}\{\mathbf{ok}^{p} : \mathsf{queue}[n+1, p \cdot c, c],\ \mathbf{fail}^{1-p} : \mathsf{queue}[n, p, c]\}, \ldots\}
\end{aligned}$$

Our new $\mathsf{queue}[n, p, c]$ type is indexed by refinement variables n, p and c which intuitively represent the length of the queue, the current probability of success and the coefficient of success rate change respectively. From the body of the type we can see that the probability p of success and the probability $1 - p$ of failure are *dependent* on the variable p. In the success case, the continuation has type $\mathsf{queue}[n + 1, p \cdot c, c]$ with updated size and probability indices. The logical constraint $0 \le n \wedge 0 \le p \le 1 \wedge 0 < c < 1$ on the type definition tells us that the new probability of success $p \cdot c$ is indeed a valid probability (in the interval [0,1]). Furthermore, as the queue grows longer, the probability of insertion success decreases exponentially.

Mixing probabilities and refinements leads to significant technical challenges that we need to overcome, in both theory and implementation. This is especially apparent when probabilities and refinements *interact*. For instance, our type system allows defining a type $\mathsf{unat}[n]$ that produces a

natural number between 0 and n *uniformly* at random.

$$\mathsf{unat}[n \mid 0 \leq n] \triangleq \oplus_{\mathsf{P}}\{\mathbf{succ}^{\frac{n}{n+1}} : ?\{1 \leq n\}.\, \mathsf{unat}[n-1],\ \mathbf{zero}^{\frac{1}{n+1}} : \mathbf{1}\}$$

Here, n is used both to describe the support of the distribution as well as to describe the actual probabilities. Checking the validity of types and processes that mix probabilities and constraints turns out to be subtle and requires care. We show that a naïve generalization of probabilistic typing rules can lead to paradoxical situations that result in invalid negative probabilities. Our first major contribution is developing the theory of probabilistic refinement session types. To avoid paradoxes encountered by the naïve approach, we develop a novel *semantic quantifier* formulation of typing rules for probabilistic branch expressions. This is crucial for providing the type system with the flexibility to support distribution-encoding types like $\mathsf{unat}[n]$ and k-way protocols that rely on them. Additionally, we develop a novel *probing process* technique that utilizes PReST itself to automatically verify that types like $\mathsf{unat}[n]$ indeed encode the correct distribution.

Our next contribution is the type safety proof of PReST. As is standard for session-typed languages, type safety is realized by session fidelity (i.e., preservation) and deadlock freedom (i.e., progress). We prove them by generalizing the nested multiverse semantics from prior work [27] to refinement types. Session fidelity implies a *probability consistency* property: the distribution of messages sent on a probabilistic channel at runtime will exactly match the distribution expressed by the type of that channel. A caveat to probability consistency is that trivially looping processes will not output anything on a probabilistic channel. To address this limitation, PReST features a *potential-based* [23] type system that allows one to reason about the almost-sure termination of programs and, by extension, ensure probability consistency.

Our final contribution is a practical implementation for PReST that comes equipped with an efficient type checking algorithm. Probabilistic behavior in PReST manifests at the process level via two process expressions: *(i)* a flip expression that flips a coin and *(ii)* a pcase expression that receives messages on a probabilistically typed channel. As Section 4 will explain, the type of each channel can potentially be different in each branch of these expressions. Since we allow *arbitrary nesting* of such expressions, requiring user-provided type annotations on every channel in each branch would add a significant burden on programmers. Recognizing these challenges, our implementation relies on an SMT solver (we support cvc5 [7] and z3 [28]) to reconstruct the intermediate types of channels using constraints collected during type checking. Surprisingly, even though these constraints are complicated and non-linear, the z3 solver is able to succeed for all of our examples.

We evaluate PReST extensively on a wide variety of benchmarks. Our benchmarks come from 4 main categories: *(i)* randomized distributed protocols, *(ii)* parametric Markov chains, *(iii)* probabilistic analysis of concurrent data structures, and *(iv)* distributions obtained by composing uniform distributions using operators like max and sum. A unique feature of PReST is that for many benchmarks, the overall structure of probabilistic process definitions is no different from deterministic ones, thus reducing programming overhead even further.

To summarize, our contributions include:

- Theory of PReST that introduces probabilistic refinements, including the development of semantically quantified typing rules and probing processes that facilitate the definition and reasoning about types that encode probability distributions.
- Type soundness proof ensuring that the probability distributions and cost bounds specified by our type system are respected at runtime.
- An efficient implementation that relies on SMT solvers to minimize programmer burden and user-provided annotations.
- An evaluation on a challenging set of benchmarks that include non-trivial interactions between probabilities and refinements.

2 Overview

Session types are in a Curry-Howard isomorphism [13] with intuitionistic linear logic [35] providing an operational interpretation to all linear logic connectives. This interpretation establishes a *provider-client* relationship between processes on either end of a channel. In this section, we will mainly focus on the internal choice $\oplus\{\ell : A_\ell\}_{\ell \in L}$ (generalization of $A \oplus B$) and external choice $\&\{\ell : A_\ell\}_{\ell \in L}$ (generalization of $A\&B$) connectives that are ubiquitous in describing communication protocols that involve branching. Intuitively, the provider of a channel of type $\oplus\{\ell : A_\ell\}_{\ell \in L}$ sends a label ℓ in set L and continues communication as a channel of type A_ℓ. On the other hand, a provider of type $\&\{\ell : A_\ell\}_{\ell \in L}$ receives a label ℓ and continues to provide type A_ℓ.

For instance, one can define a simple protocol encoding a stream of boolean choices.

$$\mathsf{bools} \triangleq \oplus\{\mathbf{true} : \mathsf{bools},\ \mathbf{false} : \mathsf{bools}\}$$

A process *providing* a channel of this type must send the label **true** or send the label **false**. After sending the label, the process recurses back to the same type, thus repeating the protocol. Dually, a process *consuming* a bools channel will receive a label (either **true** or **false**) on it and wait for the next label to arrive. While these standard choice connectives are sufficient for expressing many non-deterministic protocols, they are ill-suited for expressing *probabilistic protocols*. They offer no information regarding the probability that a particular label will be transmitted so one cannot specify or enforce the distribution of labels that get communicated at runtime.

Probabilistic Session Types. To express probabilistic protocols, prior work [27] introduced two connectives: probabilistic internal choice $\oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}$ and probabilistic external choice $\&_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}$ that generalize the non-deterministic variants. Each label ℓ is annotated with a probability p_ℓ (a real number) which describes the likelihood that ℓ gets transmitted. With probabilistic choice, one can define types such as the following

$$\mathsf{coins} \triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{0.6} : \mathsf{coins},\ \mathbf{false}^{0.4} : \mathsf{coins}\}$$

representing a stream of coins that lands on **true** with 0.6 probability or **false** with 0.4 probability.

However, constant probability annotations are only effective at specifying protocols with fixed probabilities; they do not allow for protocols that are *parametric* over probabilities. This limitation leads to large amounts of code duplication as programmers must manually write variations of the same protocol at the probabilities they would need. More importantly, the lack of parameterized probabilities precludes constructing processes that operate *universally* for all probabilities (e.g., debias) or where probabilities evolve during execution (e.g., center-biased random walk). This hampers code reuse, modularity and compositionality.

To address these limitations, we integrate probabilistic session types with refinements [31, 70] which allows types to be indexed by arithmetic expressions. Logical constraints on indices can then be used to *refine* types to better characterize their inhabiting objects. The ability to index types with varying parameters is what allows us to generalize probabilistic session types. In particular, we allow the probabilistic choice connectives $\oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}$ and $\&_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}$ to take arithmetic expressions for p_ℓ instead of just constants. These expressions may contain quantified variables representing real numbers (thus, generalizing refinements from naturals). We refer to these new types as probabilistic refinement session types (PReST). The following example illustrates how the coins protocol can be generalized using probabilistic refinement session types.

$$\mathsf{pcoins}[p \mid 0 \leq p \leq 1] \triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{p} : \mathsf{pcoins}[p],\ \mathbf{false}^{1-p} : \mathsf{pcoins}[p]\}$$

Here, the pcoins$[p]$ protocol is indexed by variable p ranging over real numbers between 0 and 1. For a process providing a channel of type pcoin$[p]$, it sends **true** with probability p and **false**

with probability $1 - p$. With pcoins[p], one can easily define coins at specific probabilities by instantiating p. For instance, pcoins[0.6] is equivalent to our original coins protocol.

Probabilistic Processes. Suppose we wish to define a `debias` process that takes a stream of arbitrarily biased coins and produces a stream of fair coins. To accomplish this, `debias` receives labels from `x` twice. If the labels received are **true** and **false**, then **true** is output on channel `y`. If the labels received are **false** and **true**, then **false** is output instead. In all other cases, `debias` recurses to obtain fresh labels from `x`. Notice the probability of receiving **true** followed by **false** is $p \cdot (1 - p)$ and the probability of receiving **false** followed by **true** is $(1 - p) \cdot p$. So the probability of `debias` sending **true** or **false** is exactly the same. Note the usage of constraint $0 < p < 1$ in the definition of `debias`. If we allow $p = 0$ or $p = 1$, then only **false** or only **true** can be received from `x`. In both cases, `debias` will loop forever without outputting any labels on `y` as the **true**-**false** and **false**-**true** branches never get executed.

```
proc debias[p | 0 < p < 1] (x : pcoins[p]) ⊢ (y : pcoins[0.5]) =
  match x { true  ⇒ match x { true  ⇒ y ← debias[p] x,
                              false ⇒ y.true; y ← debias[p] x },
            false ⇒ match x { true  ⇒ y.false; y ← debias[p] x,
                              false ⇒ y ← debias[p] x } }
```

The expressive power of probabilistic refinement session types goes beyond just facilitating parametric probabilities. For instance, one can refine probabilities using logical constraints. The following type presents an encoded probabilistic approximation type.

$$\text{approx}[x, y \mid 0 \le x < y \le 1] \triangleq \exists p.\ ?\{x < p < y\}.\ \oplus_{\mathsf{P}}\{\textbf{true}^{p} : \mathbf{1},\ \textbf{false}^{1-p} : \mathbf{1}\}$$

Intuitively, approx$[x, y]$ describes a coin whose probability of outputting **true** lies between x and y. The precise probability p of outputting **true**, however, is statically unknown. The uncertainty in p's value is encoded through existential quantification ($\exists p$). The provider of approx$[x, y]$ may choose any arbitrary value for p at runtime so long as the provider can statically prove that p satisfies $x < p < y$. Existential quantification and constraining parameters have been used in prior work [26] but never used in the context of probabilities.

Expected Cost Analysis. A unique feature of PReST is to allow one to reason about the expected cost of programs using resource-aware session types [22, 23]. Due to the fact that processes are probabilistic, resource bounds in PReST may depend on probability variables. Consider the following `debias1` program. It is a variation of the previous `debias` example that makes requests to an arbitrarily biased stream of coins in order to produce a single fair coin.

```
type coin1[p | 0 ≤ p ≤ 1] =
  &{ req : ⊕P{ true^p : coin1[p]
             , false^(1−p) : coin1[p] }
   , done : 1 }

proc debias1[p | 0 < p < 1] (x : coin1[p])
⊢^(2·p²−2·p+1)/(2·p·(1−p))
  (y : ⊕P{ true^0.5 : 1, false^0.5 : 1 }) =
  ...
  // recurse if true-true or false-false
  work 1; y ←debias1[p] x
  ...
```

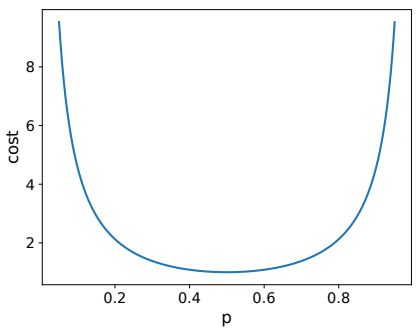

Fig. 1. Relationship between p and cost

The operational principle of `debias1` is virtually the same as `debias`. The only difference is that `debias1` terminates once a single fair coin has been provided. Let us assume a cost model where spawning a new process costs 1 unit of potential. This is accomplished by inserting the construct `work 1` before each spawn in `debias1`. PReST is able to automatically check that the potential bound $\frac{2\cdot p^2-2\cdot p+1}{2\cdot p\cdot(1-p)}$ of `debias1` (annotated on the turnstile) is valid. As shown in Figure 1, this bound is well-defined, finite and positive only for p in the open interval $(0, 1)$. The expected cost bound theorem (Theorem 5.6) tells us that the expected number of iterations required of `debias1` to terminate is finite. So our original `debias` program is able to productively generate fair coins given $0 < p < 1$ because the debiasing of each individual output coin is *almost surely terminating*.

Indistinguishability. Labels communicated probabilistically respect the distributions specified on probabilistic session types as shown in Theorem 5.4. In other words, if two channels have equal probabilistic session types, then the distributions of labels communicated over these channels at runtime (assuming productivity) are *indistinguishable*. To illustrate this, consider the following program which XORs together bit `b1` with arbitrary bias p and a fair bit `b2`.

```
type bit[p | 0 ≤ p ≤ 1] = ⊕P{ x0^p : 1, x1^(1−p) : 1 }
proc xor[p | 0 ≤ p ≤ 1] (b1 : bit[p]) (b2 : bit[0.5]) ⊢ (b3 : bit[0.5]) =
match b1 {
  x0 ⇒ match b2 { x0 ⇒ { b3.x0; ... } x1 ⇒ { b3.x1; ... } },
  x1 ⇒ match b2 { x0 ⇒ { b3.x1; ... } x1 ⇒ { b3.x0; ... } } }
```

PReST correctly recognizes that the output channel `b3` must be a fair bit regardless of `b1`'s bias. Any information that one gains from `b3` is indistinguishable from a fresh coin toss.

3 Mixing Probabilities and Refinements

Uniform Distribution. The most interesting probabilistic behavior exhibited by our type system arises when size and probability annotations interact in a non-trivial fashion. For instance, consider the $\mathsf{unat}[n]$ type defined as follows

$$\mathsf{unat}[n \mid 0 \le n] \triangleq \oplus_{\mathsf{P}}\{\mathbf{succ}^{\frac{n}{n+1}} : ?\{1 \le n\}.\, \mathsf{unat}[n-1],\ \mathbf{zero}^{\frac{1}{n+1}} :\ \mathbf{1}\}$$

In contrast to the usual $\mathsf{nat}[n]$ type, a channel of type $\mathsf{unat}[n]$ does not send exactly n **succ** labels. Instead, the number of **succ** labels that it sends forms a *discrete uniform distribution* over the $(n+1)$-sized set $\{0, \ldots, n\}$. To see this in action, let the random variable X denote the total number of **succ** labels sent by a provider of this type. First, we immediately have $\Pr[X = 0] = \frac{1}{n+1}$ from the annotation on the **zero** label. To determine $\Pr[X = k]$, we need to compute the probability of sending exactly k **succ** labels followed by a **zero** label. The first **succ** label is sent with probability $\frac{n}{n+1}$ (as indicated by the annotation on **succ**) after which the type transforms to $\mathsf{unat}[n-1]$. Thus, the next **succ** label is sent with probability $\frac{n-1}{n}$. Applying this intuitive argument inductively and conjoining the probabilities, we deduce that the k **succ** labels are sent with probability $\frac{n}{n+1}\cdot\frac{n-1}{n}\cdot\frac{n-2}{n-1}\cdots\frac{n-k+1}{n-k} = \frac{n-k+1}{n+1}$ and the new type of the channel would be $\mathsf{unat}[n-k]$. With this type, the **zero** label is sent with probability $\frac{1}{n-k+1}$. Taking the product with the earlier probability, we get that $\Pr[X = k] = \frac{n-k+1}{n+1}\cdot\frac{1}{n-k+1} = \frac{1}{n+1}$. Note also that if $k > n$, then $\Pr[X = k] = 0$ since type $\mathsf{unat}[0]$ constrains the provider to only send the **zero** label with probability 1 (label **succ** has 0 probability for $n = 0$). We can hence conclude that the value of X is uniformly distributed over $\{0, \ldots, n\}$.

To generate such a `unat`, we define the following `gen_unat[n]` process (again omitting code for waiting and closing channels). Given an arbitrary natural number channel `nat[n]` as input, `gen_unat` generates a uniform distribution over $\{0, \ldots, n\}$.

```
proc gen_unat[n | 0 ≤ n] (x : nat[n]) ⊢ n/2 (y : unat[n]) =
  match x {
    zero ⇒ y.zero; ...,
    succ ⇒ flip n/(n+1) {
      H ⇒ work 1; y.succ; y ←gen_unat[n − 1] x,
      T ⇒ y.zero; ... } }
```

Basically, `gen_unat[n]` invokes the coin flip primitive at the n-th recursive call with probability $\frac{n}{n+1}$. If the coin lands on heads, then `gen_unat` outputs `succ` on `y` and recurses with index $n-1$. If the coin lands on tails, then `gen_unat` outputs `zero` on `y` and terminates. On each call to `gen_unat`, the probability that the type outputs `succ` keeps decreasing until $n = 0$ is reached when the only possible outcome of the type is `zero`.

We now describe how automated expected cost analysis works for `gen_unat`. Let us begin by assuming a cost model where sending a `succ` label requires 1 unit of potential. Next, we write annotation $\frac{n}{2}$ on the turnstile as the assumed potential required (i.e. cost) to execute this process with some argument of type `nat[n]`. Our analysis will essentially justify this assumption through induction. The `gen_unat` process begins with a non-deterministic `match` expression. For non-deterministic `match` expressions, it suffices to consider the branch with the largest cost (`succ` in this case) as its execution imposes an upper bound on the cost of the entire expression. Now for the `flip` expression, in the heads branch we use `work 1` to perform 1 unit of work immediately before sending a `succ` label. Now by induction, the recursively spawned `gen_unat` process will require $\frac{n-1}{2}$ units of potential to execute. The tails branch costs 0 units of potential as it does not send `succ`. Since the heads branch is executed with $\frac{n}{n+1}$ probability, we can compute the expected cost of the overall `flip` as $\frac{n}{n+1} \cdot (1 + \frac{n-1}{2}) + (1 - \frac{n}{n+1}) \cdot 0 = \frac{n}{2}$. This proves that our original assumption that $\frac{n}{2}$ is the expected cost of running `gen_unat`. Moreover, $\frac{n}{2}$ neatly coincides with the expected value of a number uniformly drawn from the set $\{0,\ldots,n\}$.

Probing Processes. A unique feature of PReST is that the meta-level inductive reasoning to prove that `unat[n]` is uniform can be carried out automatically by the type checker. We can construct a *probing process* to automatically check that `unat[n]` has the correct probability mass function (PMF). Consider the process that checks if x : `unat[n]` = y : `nat[k]` and outputs true or false based on this comparison (we omit expressions for waiting and closing channels).

```
proc probe_unat [n,k | 0 ≤ k ≤ n] (x : unat[n]) (y : nat[k])
⊢ n/2 (z : ⊕P{ true 1/(n+1) : 1, false n/(n+1) : 1 }) =
  match y {
    zero ⇒ match x { zero ⇒ z.true; ...        // x = 0, y = 0
                     succ ⇒ z.false; ... },     // x > 0, y = 0
    succ ⇒ match x { zero ⇒ z.false; ...,       // x = 0, y > 0
                     succ ⇒ work 1; z ←probe_unat[n − 1, k − 1] x y } }
```

The `probe_unat` process is implemented exactly as a standard comparison process for two natural numbers. It is indexed by variables n and k representing the upper bound of the uniform distribution and an arbitrary natural number between 0 and n. In the definition, if both `x` and `y` send label **zero**, we output **true** on `z`. Otherwise if one sends **succ** and the other sends **zero**, we output **false** on `z`. If both send **succ** labels, we simply recurse with indices $n-1$ and $k-1$. Crucially, `probe_unat` allows us to learn about the `unat[n]` type even without executing it. The well-typedness of `probe_unat` establishes that $\Pr[X = k] = \frac{1}{n+1}$ and $\Pr[X \neq k] = \frac{n}{n+1}$. Essentially, `probe_unat` becomes a constructive proof of `unat[n]`'s correctness. Probing processes of this kind

are impossible to construct in prior state-of-the-art languages [27] as they lack the ability to reason about probabilities symbolically. The expected cost $\frac{n}{2}$ of `probe_unat` again coincides with the expected value of `unat[n]`.

Non-Uniform Distributions. This technique of probing processes also works for automatic verification of protocols that encode non-uniform distributions. Consider the following `umax[n]` type. Its purpose is to encode the distribution of numbers that one would obtain after taking the maximum of two samples drawn from the uniform distribution over $\{0, .., n\}$. This distribution is well-known: $\Pr[\max(X, Y) \leq k] = \Pr[X \leq k] \cdot \Pr[Y \leq k] = \frac{k+1}{n+1} \cdot \frac{k+1}{n+1} = \frac{(k+1)^2}{(n+1)^2}$.

However, we provide a different encoding of this distribution, which we claim is a *constructive representation* of this maximum and more intuitive from a programmatic perspective. To see why, we present the type `umax[n]` and the corresponding process for providing this type.

$$\begin{aligned}\texttt{umax}[n \mid 0 \leq n] \triangleq \oplus_{\mathsf{P}}\{&\mathbf{next}^{\frac{n^2}{(n+1)^2}} : \oplus\{\mathbf{succ} : ?\{1 \leq n\}.\, \texttt{umax}[n-1]\},\\ &\mathbf{unif}^{\frac{2n}{(n+1)^2}} : \oplus\{\mathbf{succ} : ?\{1 \leq n\}.\, \texttt{unat}[n-1]\},\ \mathbf{zero}^{\frac{1}{(n+1)^2}} : \mathbf{1}\}\end{aligned}$$

The `unat_max` process takes two inputs `x` and `y` both of type `unat[n]`. To provide channel `z` of type `umax[n]`, this process will receive labels from `x` and `y` and compare them. If **zero** is received on both `x` and `y`, then the maximum of these two samples is equal to **zero**. So **zero** is sent on `z` and the process terminates successfully. Since `unat[n]` sends **zero** with probability $\frac{1}{n+1}$, the probability of receiving **zero** on both input channels is $\frac{1}{(n+1)^2}$. This is confirmed in the `umax` type as the probability of **zero** branch is $\frac{1}{(n+1)^2}$.

```
proc unat_max[n | 0 ≤ n] (x : unat[n]) (y : unat[n]) ⊢ (z : umax[n]) =
  match x { zero ⇒ match y { zero ⇒ wait x; wait y; z.zero; close z,
                             succ ⇒ wait x; z.unif; z.succ; z ↔ y },
            succ ⇒ match y { zero ⇒ wait y; z.unif; z.succ; z ↔ x,
                             succ ⇒ z.next; z.succ; z ← unat_max[n − 1] x y}}
```

The interesting cases are when one input sends **succ** and the other one sends **zero** (second and third sub-branches). In these cases, the maximum behaves *exactly* as a uniform distribution, i.e., type `unat`. For instance, in the second sub-branch where `x` sends **zero** and `y` sends **succ**, `z` is completely determined by `y`. We need to send **succ** on `z` and just output `y` on `z`, indicated by $z \leftrightarrow y$ that provides the remaining **succ** labels. This intuition is precisely captured in the **unif** branch in the `umax` type. The third sub-branch is analogous; `z` is completely determined by `x`. The probability of entering each of these two sub-branches is the product of the probabilities that one `unat` channel sends **succ**, i.e., $\frac{n}{n+1}$ and the other channel sends **zero**, i.e., $\frac{1}{n+1}$. Hence, the probability of the **unif** branch in `umax` type combines these two, i.e., $\frac{2n}{(n+1)^2}$, and since the max behaves like `unat` in these cases, the continuation type is `unat`$[n-1]$. Finally, the probability of receiving **succ** from both channels is $\frac{n^2}{(n+1)^2}$, which is captured in the **next** branch of type `umax` where we send the **succ** label on `z` followed by a recursive call to `unat_max` at index $n-1$. In this branch, the continuation type is the same as the output type of `unat_max`, i.e., `umax`$[n-1]$. The types of `x` and `y` after receiving **succ** both become `unat[n − 1]` which correctly matches the signature of `unat_max[n − 1]`. Looking back at the `umax`$[n]$ type, this constructive representation captures the essence of taking the maximum and facilitates a natural definition of the `unat_max` process that does not even need to mention probabilities explicitly.

Can we still recover the standard declarative description of `umax` from this constructive representation? Yes, we can use the same probing mechanism to prove that `umax[n]` correctly characterizes

its intended distribution! Below is the signature of a `probe_umax` process which encodes the cumulative distribution function (CDF).

$$\texttt{proc probe\_umax}[n, k \mid 0 \le k \le n]\ \texttt{(x : umax}[n]\texttt{)}\ \texttt{(y : nat}[k]\texttt{)}$$
$$\vdash^{\frac{n\cdot(4\cdot n+5)}{6\cdot(n+1)}}\ \texttt{(z : }\oplus_{\mathsf{P}}\texttt{\{ true}^{\frac{(k+1)^2}{(n+1)^2}}\texttt{ : 1, false}^{1-\frac{(k+1)^2}{(n+1)^2}}\texttt{ : 1 \})}$$

Given `x` of type `umax[`n`]` and y of type `nat[`k`]` where k is an arbitrary value between 0 and n, `probe_umax` intuitively checks if the value of `x` is less than or equal to k. The successful type checking of the process verifies that the event max $\le k$ occurs (when `true` is sent on `z`) with probability $(k+1)^2/(n+1)^2$. Furthermore, the expected cost $\frac{n\cdot(4\cdot n+5)}{6\cdot(n+1)}$ is exactly the expected value of the maximum of two numbers uniformly drawn from $\{0, \ldots, n\}$. Attempting to modify any of the constants in the cost expression will result in rejection by the constraint solver backend. This indicates that our bound is close to being tight.

Arbitrary Distributions. PReST also provides an encoding of arbitrary discrete distributions. Suppose, for a random variable X whose domain is $\{0, \ldots, n\}$, the probability $\Pr[X = k] = p_k$. We encode this distribution with the following type

$$\mathsf{dist}[n \mid 0 \le n] \triangleq \oplus_{\mathsf{P}}\{\mathbf{succ}^{q(n)} : ?\{1 \le n\}.\, \mathsf{dist}[n-1],\ \mathbf{zero}^{1-q(n)} :\ \mathbf{1}\}$$

where q is a function $q : \{0, \ldots, n\} \to [0, 1]$. With this distribution, $\Pr[X = k]$ is the probability of sending k **succ** messages followed by a **zero** message. Following the $\mathsf{dist}[n]$ type recursively, we get $p_k = \Pr[X = k] = q(n) \cdot q(n-1) \cdots q(n-k+1) \cdot (1 - q(n-k))$. Thus, we provide a recipe for encoding an arbitrary distribution which reduces to solving the recurrence relation above. It is easy to see that $q(n) = \frac{n}{n+1}$ leads to a uniform distribution. In fact, if $q(n)$ is a constant function, this leads to an *exponential distribution.*

4 Formal Type System

With all features of the language introduced, we turn our attention to the formal aspects of PReST. This section presents the type system of PReST and the following section connects it to the semantics and proves type safety. The typing rules are all mutually inductively defined together.

4.1 Type Grammar and Validity

$$\begin{array}{lll}
\textit{Types} & A, B & ::= \oplus\{\ell : A_\ell\}_{\ell\in L} \mid \&\{\ell : A_\ell\}_{\ell\in L} \mid A \otimes B \mid A \multimap B \mid \mathbf{1} \mid X[\overline{e}] \mid \triangleright^{w} A \mid \triangleleft^{w} A \\
& & \quad \mid \oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell\in L} \mid \&_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell\in L} \mid \exists r.A \mid \forall r.A \mid ?\{\phi\}.A \mid !\{\phi\}.A \\
\textit{Exprs} & e, p, w & ::= v \mid e + e \mid e - e \mid e \times e \mid e \div e \mid r \\
\textit{Props} & \phi & ::= e = e \mid e < e \mid \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \exists r.\phi \mid \forall r.\phi
\end{array}$$

Fig. 2. Type Grammar for PReST

The types in PReST follow the grammar in Figure 2 where v is a real valued constant, and r is a refinement variable. We use p and w to range over expressions that are used as probabilities and cost/work respectively. The interpretation of each type comes from the behavior of the *provider* (process that has this type on the right of $\vdash$ in its declaration) and *client* (process that has this type on the left of $\vdash$) of the type. We have already described the standard and probabilistic choice types. In addition, type $A \otimes B$ represents the multiplicative conjunction where the provider sends a channel of type A and continues with type B. Its dual is the linear implication $A \multimap B$ which possesses the opposite operational interpretation: the provider receives a channel of type A and continues with type B. The $\mathbf{1}$ type is the linear unit and denotes the end of protocol. The type $X[\overline{e}]$ is used to represent user-defined type names (e.g., $\mathsf{nat}[n]$). Refinement variables can be introduced

into a session type using $\exists r.A$ and $\forall r.A$. The former requires the provider to send a real value whereas the latter ensures the provider receives a value from the client. The type $?\{\phi\}.A$ asserts and $!\{\phi\}.A$ assumes logical proposition ϕ to constrain refinement variables. Finally ergometric session types $\triangleright^w A$ and $\triangleleft^w A$ facilitate the exchange of w units of potential between processes.

Type Definitions. These type definitions (like the ones introduced in Section 2 and 3) provided by the user are collected in a *global signature* denoted by Σ. Type definitions have the form $X[\overline{r} \mid \phi] \triangleq A$ where X is a type name indexed by variables $\overline{r}$ that satisfy constraint ϕ and A is a type expression that serves as the definition of $X[\overline{r}]$. All types in PReST are implicitly *equi-recursive*: a type name is equal to its definition; there are no explicit folds and unfolds for recursive types.

In PReST, validity of a type A is expressed as the judgment $\mathsf{V}\ ;\ \mathsf{C} \vdash_\Sigma A\ \mathit{valid}$ (Figure 3 presents an excerpt of these rules). The V here is a context of refinement variables that may occur in A. Constraint context C is a proposition that is assumed to be true for all instantiations of V. This is central for exchanging proofs, e.g., checking that $\mathsf{nat}[e]$ is indeed a valid type requires proving $\mathsf{C} \vDash e \geq 0$. Intuitively, the judgment $\mathsf{V}\ ;\ \mathsf{C} \vdash_\Sigma A\ \mathit{valid}$ means that for all substitutions $\sigma : \mathsf{V} \to \mathbb{R}$ such that $\mathsf{C}[\sigma]$ is satisfied, $A[\sigma]$ is a valid type.

$$\frac{\mathsf{V}\ ;\ \mathsf{C} \vdash_\Sigma A\ \mathit{valid} \qquad \mathsf{V}\ ;\ \mathsf{C} \vDash 0 \leq w}{\mathsf{V}\ ;\ \mathsf{C} \vdash_\Sigma \triangleright^w A\ \mathit{valid}}\ \triangleright V \qquad \frac{(\forall \ell \in L) \quad \mathsf{V}\ ;\ \mathsf{C} \vdash_\Sigma A_\ell\ \mathit{valid} \qquad \mathsf{V}\ ;\ \mathsf{C} \vDash 0 \leq p_\ell \leq 1 \qquad \mathsf{V}\ ;\ \mathsf{C} \vDash \Sigma_{\ell \in L} p_\ell = 1}{\mathsf{V}\ ;\ \mathsf{C} \vdash_\Sigma \oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}\ \mathit{valid}}\ \oplus_{\mathsf{P}} V$$

Fig. 3. Selected Rules for Checking Validity of Types in PReST

As explained in Section 2, PReST allows each label ℓ in probabilistic choice types to be annotated with a probability expression p_ℓ. To ensure that types represent valid probability distributions, we generalize the validity check performed for refinement types. The validity rule $\oplus_{\mathsf{P}} V$ for probabilistic internal choice requires two constraints to be satisfied: $\mathsf{V}\ ;\ \mathsf{C} \vDash 0 \leq p_\ell \leq 1$ and $\mathsf{V}\ ;\ \mathsf{C} \vDash \Sigma_{\ell \in L} p_\ell = 1$. This statically guarantees that p_ℓ's represent a valid distribution.

We would like to point out that standard choices can in fact be represented using refinements and probabilistic choices. The type $\oplus\{\ell : A_\ell\}_{\ell \in L}$ can be encoded as

$$\exists p_\ell.\ ?\{0 \leq p_\ell \leq 1 \wedge \Sigma_{\ell \in L} p_\ell = 1\}.\oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}$$

and we can do a similar encoding for external choice type. However we choose to include the standard choice types as primitives for numerous reasons. First, the above type adds significant annotation burden on the programmer who needs to determine the probability of sending each label in a process definition. Second, this adds significant complexity to the type checker that has numerous additional refinement variables to reason about while establishing type validity. Keeping both choice types provides a much needed flexibility to programmers who need only annotate choice types they wish to reason about with no additional burden.

4.2 Typing Rules

$$\begin{aligned}
\textit{Processes}\quad P, Q \ ::=\ & x.k\ ;\ P \mid \mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} \mid x..k\ ;\ P \mid \mathsf{pcase}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} \mid x \leftarrow f\ [\overline{e}]\ \overline{y}\ ;\ P \\
\mid\ & \mathsf{send}\ x\ t\ ;\ P \mid y \leftarrow \mathsf{recv}\ x\ ;\ Q \mid \mathsf{close}\ x \mid \mathsf{wait}\ x\ ;\ Q \mid \mathsf{flip}\ p\ (\mathsf{H} \Rightarrow P_H \mid \mathsf{T} \Rightarrow P_T) \\
\mid\ & \mathsf{work}\ w\ ;\ P \mid \mathsf{pay}\ c\ w\ ;\ P \mid \mathsf{get}\ c\ w\ ;\ P \mid \mathsf{send}\ x\ [e]\ ;\ P \mid [r] \leftarrow \mathsf{recv}\ x\ ;\ Q \\
\mid\ & \mathsf{assert}\ x\ \{\phi\}\ ;\ P \mid \mathsf{assume}\ x\ \{\phi\}\ ;\ Q \mid x \leftrightarrow y
\end{aligned}$$

Fig. 4. Process Grammar for PReST

Session types have a strong logical foundation in linear logic [14, 34]. An intuitionistic linear sequent $A_1, A_2, \ldots, A_n \vdash A$ can be interpreted as the offer of a session A by a process P using the sessions

$A_1, A_2, \ldots, A_n$. In our notation, we write this as $(x_1 : A_1), (x_2 : A_2), \ldots, (x_n : A_n) \vdash P :: (x : A)$. The x_i's correspond to channels used by P, and x is the channel provided by P. As is standard, we use the linear context Δ to combine multiple assumptions. To account for refinements and resource analysis, we extend the typing judgment for PReST and write it as $\mathsf{V} \,;\, \mathsf{C} \,;\, \Delta \vdash^{w} P :: (x : A)$. Figure 4 presents the syntax of process P. Context V represents the free refinement variables that can occur in C, Δ, w, P and A. Constraint context C is a proposition that all instantiations of V are assumed to satisfy. The w expression here is the expected cost bound of evaluating P. Similarly to validity, this judgment states that for any valid substitution $\sigma : \mathsf{V} \to \mathbb{R}$, $\mathsf{C}[\sigma]$ implies that $P[\sigma]$ is expected to consume no more than $w[\sigma]$ units of potential, uses the channels in $\Delta[\sigma]$ and offers channel x of type $A[\sigma]$. The typing rules of PReST can be organized into four categories: basic rules, refinement rules, resource rules and probabilistic rules. We omit detailed presentation of some dual types in the descriptions below and present them in our companion report [33].

4.3 Basic Rules

The basic rules of PReST correspond to the standard rules of intuitionistic session types [13]. Figure 5 presents an excerpt of the basic rules.

$$\frac{(k \in L) \quad \mathsf{V} \,;\, \mathsf{C} \,;\, \Delta \vdash^{w} P :: (x : A_k)}{\mathsf{V} \,;\, \mathsf{C} \,;\, \Delta \vdash^{w} (x.k \,;\, P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})} \oplus R \qquad \frac{(\forall \ell \in L) \quad \mathsf{V} \,;\, \mathsf{C} \,;\, \Delta \vdash^{w} P_\ell :: (x : A_\ell)}{\mathsf{V} \,;\, \mathsf{C} \,;\, \Delta \vdash^{w} \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})} \&R$$

Fig. 5. Selected Basic Typing Rules

The rules above govern the sending and receiving of labels on provided channels. In rule $\oplus R$, the construct $x.k \,;\, P$ sends label k on channel x and continues as process P. On the other hand, the case expression in rule $\&R$ receives a label from x and executes its associated branch. Note that the dot notation `x.k` and `match` syntax of our implementation are overloaded. Depending on the whether the type of `x` is a standard choice or a probabilistic choice, `x.k` resolves to either $x.k$ (standard) or $x..k$ (probabilistic). Similarly, `match` resolves to either case (standard) or pcase (probabilistic). The probabilistic rules for $x..k$ and pcase are formally described in Section 4.6.

4.4 Refinement Rules

Figure 6 presents an excerpt of the refinement typing rules. The $\exists R$ rule governs the sending of refinement expressions. A provider of $x : \exists r.A$ must send (the value of) a refinement expression e to a client using the syntax send $x\ [e] \,;\, P$ and continue executing P. The continuation P will provide x of type $A[e/r]$. The dual of $\exists r.A$ is $\forall r.A$ which just reverses the roles of the provider and client; the provider receives a refinement expression that is sent by the client.

$$\frac{\mathsf{V} \,;\, \mathsf{C} \,;\, \Delta \vdash^{w} P :: (x : A[e/r])}{\mathsf{V} \,;\, \mathsf{C} \,;\, \Delta \vdash^{w} \mathsf{send}\ x\ [e] \,;\, P :: (x : \exists r.A)} \exists R \qquad \frac{\mathsf{V} \,;\, \mathsf{C} \vDash \phi \qquad \mathsf{V} \,;\, \mathsf{C} \,;\, \Delta \vdash^{w} P :: (x : A)}{\mathsf{V} \,;\, \mathsf{C} \,;\, \Delta \vdash^{w} \mathsf{assert}\ x\ \{\phi\} \,;\, P :: (x : ?\{\phi\}.A)} ?R$$

Fig. 6. Selected Refinement Typing Rules

In rule $?R$, the construct assert $x\ \{\phi\}$ is interpreted as sending a proof of ϕ from the provider to the client. To send this proof, the provider needs to establish the semantic entailment $\mathsf{V} \,;\, \mathsf{C} \vDash \phi$. This proof obligation is what *constrains* the possible messages that can be sent over channel x. Dually, the type $!\{\phi\}.A$ utilizes assume $x\ \{\phi\}$ to receive a proof of ϕ and strengthen its local constraint context C into $\mathsf{C} \wedge \phi$. In the concrete syntax of our implementation, $\{\phi\}$ is often elided in uses of assert x and assume x as ϕ can be inferred from the type of x (either $?\{\phi\}.A$ or $!\{\phi\}.A$).

4.5 Resource Rules

Resource-aware session types [22, 23] facilitate amortized cost reasoning using the potential method. For some process P satisfying judgment $\mathsf{V}\ ;\ \mathsf{C}\ ;\ \Delta \vdash^{w} P :: (x : B)$, it has access to w units of potential. As shown in rule $\triangleleft R$, if B is an ergometric session type of the form $\triangleleft^{w'} A$, then P receives w' units of potential from another process through channel x. Dually, the type $\triangleright^{w'} A$ requires a process to send w' units of potential using the pay construct.

$$\frac{\mathsf{V}\ ;\ \mathsf{C}\ ;\ \Delta \vdash^{w+w'} P :: (x : A)}{\mathsf{V}\ ;\ \mathsf{C}\ ;\ \Delta \vdash^{w} \mathsf{get}\ x\ w'\ ;\ P :: (x : \triangleleft^{w'} A)} \triangleleft R \qquad \frac{\mathsf{V}\ ;\ \mathsf{C}\ ;\ \Delta \vdash^{w-w'} P :: (x : A) \qquad \mathsf{V}\ ;\ \mathsf{C} \vDash 0 \le w' \le w}{\mathsf{V}\ ;\ \mathsf{C}\ ;\ \Delta \vdash^{w} \mathsf{work}\ w'\ ;\ P :: (x : A)}\ \mathit{Work}$$

Fig. 7. Selected Resource Typing Rules

The get and pay constructs do not alter the total amount of potential in a closed system of processes as they simply redistribute potential. Ultimately, potential is consumed by the `work` construct which corresponds to the cost model that one wishes to check. Potential in PReST is *affine* as we allow extraneous potential to be discarded.

4.6 Probabilistic Rules

Weighted Sum. Consider the expression flip p ($\mathsf{H} \Rightarrow c.\mathbf{true}; \ldots \mid \mathsf{T} \Rightarrow c.\mathbf{false}; \ldots$) where we send **true** on c in the H and **false** on c in the T branches. Intuitively, the type of c for the overall expression should be $\oplus_{\mathsf{P}}\{\mathbf{true}^{p} : \mathbf{1}, \mathbf{false}^{1-p} : \mathbf{1}\}$. We achieve this by giving c the type $\oplus_{\mathsf{P}}\{\mathbf{true}^{1} : \mathbf{1}, \mathbf{false}^{0} : \mathbf{1}\}$ in the H branch since we are only sending **true**. Similarly, we give the type $\oplus_{\mathsf{P}}\{\mathbf{true}^{0} : \mathbf{1}, \mathbf{false}^{1} : \mathbf{1}\}$ to c in the T branch. The overall type of c is obtained by taking a *weighted sum* of these two types. This sum relation allows a fairly simple rule for sending messages on a probabilistic channel.

$$\frac{(k \in L) \qquad \mathsf{V}\ ;\ \mathsf{C} \vDash p_k = 1 \qquad \mathsf{V}\ ;\ \mathsf{C} \vDash p_j = 0 \quad (j \neq k) \qquad \mathsf{V}\ ;\ \mathsf{C}\ ;\ \Delta \vdash^{w} P :: (x : A_k)}{\mathsf{V}\ ;\ \mathsf{C}\ ;\ \Delta \vdash^{w} x..k\ ;\ P :: (x : \oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L})} \oplus_{\mathsf{P}} R$$

To send label k on channel x, p_k must be 1 and the probability annotations for all other labels of the type must be 0. The intuition behind this rule is that since we are only sending label k, other labels must have zero probability.

To perform weighted sum for types in the context and provided channel, we have two operators: $+^{\mathsf{L}}$ for the context and $+^{\mathsf{R}}$ for the provided channel. These two operators reduce to identity for probabilistic types that receive. This is crucial for the type safety of the flip construct which is an internal computation. If a process P flips heads, processes that send messages to P are not notified of this internal step; they will still send messages to P with the original probability distribution. If weighted sum alters the type of the receiver end of a channel, the two ends of the channel will have inconsistent probability distributions, leading to unsoundness in the type system. Hence, the only non-trivial cases of weighted sum are:

$$p \cdot \&_{\mathsf{P}}\{\ell^{q_\ell} : A_\ell\}_{\ell \in L} +^{\mathsf{L}} (1-p) \cdot \&_{\mathsf{P}}\{\ell^{r_\ell} : A_\ell\}_{\ell \in L} = \&_{\mathsf{P}}\{\ell^{p \cdot q_\ell + (1-p) \cdot r_\ell} : A_\ell\}_{\ell \in L}$$

$$p \cdot \oplus_{\mathsf{P}}\{\ell^{q_\ell} : A_\ell\}_{\ell \in L} +^{\mathsf{R}} (1-p) \cdot \oplus_{\mathsf{P}}\{\ell^{r_\ell} : A_\ell\}_{\ell \in L} = \oplus_{\mathsf{P}}\{\ell^{p \cdot q_\ell + (1-p) \cdot r_\ell} : A_\ell\}_{\ell \in L}$$

These two cases correspond to probabilistic types in the context ($\&_{\mathsf{P}}$) and provided channel ($\oplus_{\mathsf{P}}$) that perform sending. For all other types, the weighted sum relation simply reduces to identity, i.e., $p \cdot A +^{\mathsf{R}} (1-p) \cdot A = A$ and $p \cdot A +^{\mathsf{L}} (1-p) \cdot A = A$. This relation is generalized to n-ary sums ($\sum^{L}_{i \in I} p_i \cdot A_i$ and $\sum^{R}_{i \in I} p_i \cdot A_i$) and also pointwise to contexts.

Naïve Probabilistic Branching. Due to non-trivial interactions between probabilities and refinements, the probabilistic typing rules significantly depart from existing probabilistic session type systems [3, 27] and other refinement type systems in general [25, 70]. Suppose we wish to type check a probabilistic case expression: pcase x $(\ell \Rightarrow Q_\ell)_{\ell \in L}$. To understand why a naïve

generalization of existing type systems is not sufficient, we first present what such a generalized typing rule would look like.

$$\frac{(\forall \ell \in L) \quad \mathsf{V}\,;\,\mathsf{C}\,;\,\Delta_\ell, (x : A_\ell) \vdash^{w_\ell} Q_\ell :: (z : B_\ell) \qquad \mathsf{V}\,;\,\mathsf{C} \vDash \Delta = \Sigma^{\mathsf{L}}_{\ell\in L} p_\ell \cdot \Delta_\ell \qquad \mathsf{V}\,;\,\mathsf{C} \vDash B = \Sigma^{\mathsf{R}}_{\ell\in L} p_\ell \cdot B_\ell \qquad \mathsf{V}\,;\,\mathsf{C} \vDash w = \Sigma_{\ell\in L} p_\ell \cdot w_\ell}{\mathsf{V}\,;\,\mathsf{C}\,;\,\Delta, (x : \oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell\in L}) \vdash^{w} \mathsf{pcase}\ x\ (\ell \Rightarrow Q_\ell)_{\ell\in L} :: (z : B)}\ \textsc{naïve-}\oplus_{\mathsf{P}}L$$

We type check the branches with possibly different contexts Δ_ℓ, offered types B_ℓ and cost bounds w_ℓ. Then we perform a weighted sums for all types in Δ_ℓ and B_ℓ to recover context Δ, offered type B and cost bound w.

Why is the pcase rule above naïve? The naïve version of the $\oplus_{\mathsf{P}}L$ rule above would require fixed probabilistic expressions to be chosen for $\Delta_\ell, B_\ell, w_\ell$. While these expressions can range over different values as they may contain universally quantified variables from V, the fundamental structure of these expressions will remain fixed. This can lead to paradoxical situations where certain premises can only be satisfied by negative probabilities. Consider the program for comparing two `unat[`n`]`'s, i.e. two samples of a uniform distribution.

```
proc compare[n | 0 ≤ n] (x : unat[n]) (y : unat[n])
⊢ (c : ⊕P{ gt^{n/(2(n+1))} : 1, lt^{n/(2(n+1))} : 1, eq^{1/(n+1)} : 1 }) = match x {
  zero^1 ⇒ ...,
  succ^2 ⇒ match y { zero^3 ⇒ c.gt; ..., succ^4 ⇒ c ←compare[n − 1] x y } }
```

Given channels `x` and `y` both of type `unat[`n`]`, the process compares the number of `succ` labels received from `x` and `y`. A label of the form `gt` (greater-than), `lt` (less-than) or `eq` (equal), representing the comparison result, is output on `c`.

It is easy to derive, using basic probability theory, that the probabilities of outputting `gt` and `lt` are both $\frac{n}{2(n+1)}$ and the probability of outputting `eq` is $\frac{1}{n+1}$. In the body of `compare`, we assign superscripts to the `match` branches. We will refer to these branches by their superscripts. In branch 3, since we are sending label `gt`, the type of `c` is $\oplus_{\mathsf{P}}\{\ \mathtt{gt}^1\mathtt{:1},\ \mathtt{lt}^0\mathtt{:1},\ \mathtt{eq}^0\mathtt{:1}\ \}$ as determined by rule $\oplus_{\mathsf{P}}R$ for sending probabilistic labels. In branch 4, we are making a recursive call to `compare[`$n-1$`]`, the type of `c` is $\oplus_{\mathsf{P}}\{\ \mathtt{gt}^{(n-1)/2n}\mathtt{:1},\ \mathtt{lt}^{(n-1)/2n}\mathtt{:1},\ \mathtt{eq}^{1/n}\mathtt{:1}\ \}$. Now to obtain the type of c in branch 2, we perform the following weighted sum computation.

$$\begin{aligned}&\frac{1}{n+1}\cdot\oplus_{\mathsf{P}}\{\mathtt{gt}^1 : \mathbf{1},\ \mathtt{lt}^0 : \mathbf{1},\ \mathtt{eq}^0 : \mathbf{1}\} +^{\mathsf{R}} \frac{n}{n+1}\cdot\oplus_{\mathsf{P}}\{\mathtt{gt}^{(n-1)/2n} : \mathbf{1},\ \mathtt{lt}^{(n-1)/2n} : \mathbf{1},\ \mathtt{eq}^{1/n} : \mathbf{1}\}\\ &= \oplus_{\mathsf{P}}\{\mathtt{gt}^{1/2} : \mathbf{1},\ \mathtt{lt}^{(n-1)/2(n+1)} : \mathbf{1},\ \mathtt{eq}^{1/(n+1)} : \mathbf{1}\}\end{aligned}$$

Let us focus on the types of `c` manifesting in branches 3 and 4 when using the naïve typing rules. Since they do not allow choosing different types for different values of n, the *only possible* type that would allow branches 3 and 4 to type check for all values of n is $\oplus_{\mathsf{P}}\{\ \mathtt{gt}^{\frac{1}{2}}\mathtt{:1},\ \mathtt{lt}^{\frac{n-1}{2(n+1)}}\mathtt{:1},\ \mathtt{eq}^{\frac{1}{n+1}}\mathtt{:1}\ \}$. This completely fixes the structure of the probabilistic expressions inside the type. However, prior to entering branch 2 and learning of $1 \le n$, the probability annotation $\frac{n-1}{2(n+1)}$ of `lt` does not strictly fall into the interval $[0, 1]$. So the only type that satisfies naïve typing is actually invalid without prior knowledge of n's value. This demonstrates that this typing rule imposes too strong a requirement and will reject reasonable programs due to purely technical reasons.

Semantic Probabilistic Branching. To solve the issues posed by naïve probabilistic branching, recall the meaning of our typing judgment $\mathsf{V}\,;\,\mathsf{C}\,;\,\Delta \vdash^{w} P :: (x : A)$: for all substitutions $\sigma : \mathsf{V} \to \mathbb{R}$, $P[\sigma]$ is well-typed. This meaning provides our language with an additional layer of flexibility to the type system by allowing Δ_ℓ, B_ℓ and w_ℓ to be reconstructed with fundamentally different probability

expressions depending on the values instantiating V. This is expressed as a more relaxed $\oplus_{\mathsf{P}} L$ rule:

$$\frac{\begin{array}{c}\forall \mathsf{V} \,.\, \exists \{\Delta_\ell, B_\ell, w_\ell\}_{\ell \in L} \,. \\ (\forall \ell \in L) \quad \cdot \,;\, \mathsf{C} \,;\, \Delta_\ell, (x : A_\ell) \vdash^{w_\ell} Q_\ell :: (z : B_\ell) \\ \cdot \,;\, \mathsf{C} \vDash \Delta = \Sigma^{\mathsf{L}}_{\ell \in L} p_\ell \cdot \Delta_\ell \qquad \cdot \,;\, \mathsf{C} \vDash B = \Sigma^{\mathsf{R}}_{\ell \in L} p_\ell \cdot B_\ell \qquad \cdot \,;\, \mathsf{C} \vDash w = \Sigma_{\ell \in L} p_\ell \cdot w_\ell\end{array}}{\mathsf{V} \,;\, \mathsf{C} \,;\, \Delta, (x : \oplus_{\mathsf{P}} \{\ell^{p_\ell} : A_\ell\}_{\ell \in L}) \vdash^{w} \mathsf{pcase}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : B)}\ \oplus_{\mathsf{P}} L$$

Note that the quantifiers $\forall \mathsf{V} \,.\, \exists \{\Delta_\ell, B_\ell, w_\ell\}_{\ell \in L}$ here are *semantic*: the choice of probability expressions may take into account semantic *assumptions* regarding the values that will instantiate V. The only requirement is that these semantic assumptions cover the entire range of possible values for V and that all the premises of the probabilistic rule are satisfied. For example, suppose there is some $r \in \mathsf{V}$ and we must reconstruct probability expression p. We can semantically assume that $r \leq 0$ and verify that $p = x + y$ satisfies all premises. Now assuming $0 < r$ we can choose $p = 2 \times z$ to satisfy all premises. Since we have verified all premises under assumptions that cover the entire range of r, type checking succeeds. Observe the different structures of p in these two cases. In contrast, the structure of p is fixed in the naïve-flip rule and cannot vary depending on r.

In our semantically quantified formulation of typing rules, programs like `compare` no longer pose an issue as the type checker can choose a valid set of contexts and types for weighted sum *after* making semantic assumptions about V. To type check `compare`, we proceed by making two semantic assumptions by case analysis on the instantiating value for n. If we have $1 \leq n$, then we can choose `c`'s type in branch 2 to be $\oplus_{\mathsf{P}}\{\ \mathtt{gt}^{\frac{1}{2}} : \mathbf{1},\ \mathtt{lt}^{\frac{n-1}{2(n+1)}} : \mathbf{1},\ \mathtt{eq}^{\frac{1}{n+1}} : \mathbf{1}\ \}$ as it is now a valid type. In the case of $n < 1$, we can choose `c`'s type in branch 2 to be $\oplus_{\mathsf{P}}\{\ \mathtt{gt}^{1} : \mathbf{1},\ \mathtt{lt}^{0} : \mathbf{1},\ \mathtt{eq}^{0} : \mathbf{1}\}$. At first this type does not appear to allow branches 3 and 4 to type check. However, our assumption $n < 1$ by case analysis contradicts the proposition $1 \leq n$ learned after entering branch 2. So the entirety of branch 2 is well-typed *vacuously* as it is deemed impossible.

From type checking `compare`, we can see that the actual rules are more semantic in nature than the naïve ones. To implement type checking for probabilistic branching, we first introduce existential variables as stand-ins for probability annotations in $\Delta_\ell, B_\ell, w_\ell$. We then gather all logical constraints synthesized during the type checking of the branches. A single large semantic entailment judgment is formed by taking the conjunction of all these constraints and putting them under forall-exists quantification. Finally, this semantic entailment is handed to our SMT backend for proving.

Due to the fact that the `flip` construct also introduces probabilistic branching, its corresponding FLIP rule utilizes semantic quantifiers to type check the H and T branches.

$$\frac{\begin{array}{c}\forall \mathsf{V} \,.\, \exists \Delta_H, \Delta_T, A_H, A_T, w_H, w_T \,. \\ \cdot \,;\, \mathsf{C} \,;\, \Delta_H \vdash^{w_H} P_H :: (x : A_H) \qquad \cdot \,;\, \mathsf{C} \,;\, \Delta_T \vdash^{w_T} P_T :: (x : A_T) \qquad \cdot \,;\, \mathsf{C} \vDash 0 \leq p \leq 1 \\ \cdot \,;\, \mathsf{C} \vDash \Delta = p \cdot \Delta_H +^{\mathsf{L}} (1 - p) \cdot \Delta_T \\ \cdot \,;\, \mathsf{C} \vDash A = p \cdot A_H +^{\mathsf{R}} (1 - p) \cdot A_T \qquad \cdot \,;\, \mathsf{C} \vDash w = p \cdot w_H +^{\mathsf{R}} (1 - p) \cdot w_T\end{array}}{\mathsf{V} \,;\, \mathsf{C} \,;\, \Delta \vdash^{w} \mathsf{flip}\ p\ (\mathsf{H} \Rightarrow P_H \mid \mathsf{T} \Rightarrow P_T) :: (x : A)}\ \textsc{flip}$$

5 Formal Semantics and Meta-Theory

At runtime, the state of a session-typed program manifests as a *configuration* of concurrent processes organized as a tree where each process offers the channel connecting to its parent and uses the channels offered by its children. To prove type preservation (Theorem 5.2) for PReST, we adopt the *nested-multiverse semantics* of Das el al. [27] which *localizes* the effects of probabilistic branching. We connect this nested-multiverse semantics to a distribution based semantics through a flattening procedure and simulation theorem (Theorem 5.5).

Nested-multiverse Semantics. In the nested-multiverse semantics, a configuration C is a sequence of *semantic objects* O in parallel composition $O_1 \parallel \ldots \parallel O_n$. Each semantic object takes on two forms: *(i)* a singleton process $\mathsf{proc}(c, w, P)$ where P is a process and w is its (non-negative)

work counter, or *(ii)* a nested distribution of configurations $\mathsf{proc}(c, \{C_i : p_i\}_{i \in \mathcal{I}})$ where C_i appears with probability p_i in the distribution. The c in both cases represents the provided channel. The nested-multiverse semantics is defined as a *multiset rewriting* [15] system featuring two kinds of transition relations $\mathcal{C} \mapsto \mathcal{C}'$ and $\mathcal{C} \overset{d,\kappa}{\Longmapsto} \mathcal{C}'$. Our companion report [33] presents their formal rules in detail. For $\mathcal{C} \mapsto \mathcal{C}'$, configuration $\mathcal{C}$ transitions to $\mathcal{C}'$ in a single step without communication between any of $\mathcal{C}$'s objects. The flip primitive induces this kind of transition when it evaluates:

$$\mathsf{proc}(c, w, \mathsf{flip}\ p\ (\mathsf{H} \Rightarrow P_H \mid \mathsf{T} \Rightarrow P_T)) \mapsto \mathsf{proc}(c, \{\mathsf{proc}(c, w, P_H) : p, \mathsf{proc}(c, w, P_T) : 1 - p\})$$

Other operations that induce $\mathcal{C} \mapsto \mathcal{C}'$ transitions include process spawning and the work primitive. Conversely, $\mathcal{C} \overset{d,\kappa}{\Longmapsto} \mathcal{C}'$ describes a step where two objects in $\mathcal{C}$ (synchronously) exchange a message on channel d and transitions to $\mathcal{C}'$. The *sort* $\kappa \in \{\mathsf{det}, \oplus_{\mathsf{P}}, \&_{\mathsf{P}}\}$ categorizes the communication taking place on d. Sorts $\oplus_{\mathsf{P}}$ and $\&_{\mathsf{P}}$ categorize communication originating from probabilistic internal choice and external choice respectively, and det is the category for all other sources of communication.

Distribution-based Semantics. In the distribution-based semantics, a configuration is a sequence of processes in parallel composition $\mathsf{proc}(c_1, w_1, P_1) \parallel \cdots \parallel \mathsf{proc}(c_n, w_n, P_n)$ where P_i is a process and w_i is its work counter. Note that the configurations here are non-nested. To differentiate the configurations of the two semantics, we will refer to configurations in the distribution-based semantics using variable $\mathcal{D}$ and (non-nested) distributions of these configurations using variable μ. The distribution-based semantics is defined in terms of transition relations $\mathcal{D} \overset{\mathsf{det}}{\longmapsto} \mathcal{D}'$, $\mathcal{D} \overset{\mathsf{prob}}{\longmapsto} \mu$ and $\mu \Longmapsto \mu'$. We present a detailed account of their formal rules in our companion report. Relations of the form $\mathcal{D} \overset{\mathsf{det}}{\longmapsto} \mathcal{D}'$ are deterministic configuration-to-configuration transitions. Examples include sending and receiving messages, paying and getting potential, etc. Next, $\mathcal{D} \overset{\mathsf{prob}}{\longmapsto} \mu$ represents configuration-to-distribution transitions. A $\overset{\mathsf{det}}{\longmapsto}$ relation can be viewed as a degenerative case of $\overset{\mathsf{prob}}{\longmapsto}$ where the resulting distribution has unique support. A non-degenerative example, as shown below, is again due to the evaluation of the flip.

$$\mathsf{proc}(c, w, \mathsf{flip}\ p\ (\mathsf{H} \Rightarrow P_H \mid \mathsf{T} \Rightarrow P_T)) \overset{\mathsf{prob}}{\longmapsto} \{\mathsf{proc}(c, w, P_H) : p, \mathsf{proc}(c, w, P_T) : 1 - p\}$$

Finally, $\overset{\mathsf{prob}}{\longmapsto}$ is lifted to the $\Longmapsto$ relation between distributions in a standard way.

Configuration Typing. A nested-multiverse configuration $\mathcal{C}$ is typed using the judgment $\Delta \Vdash^{w} \mathcal{C} :: \Gamma$ which states that $\mathcal{C}$ uses channels in context Δ and provides a context of channels Γ. The cost of running $\mathcal{C}$ is expected to be upper bounded by w.

$$\frac{\cdot\ ;\ \top\ ;\ \Delta \vdash^{w'} P :: (c : A)}{\Delta \Vdash^{w+w'} \mathsf{proc}(c, w, P) :: (c : A)}\ \text{T:Proc} \qquad \frac{\Delta_1, \Delta' \Vdash^{w_1} \mathcal{O} :: (c : A) \qquad \Delta_2 \Vdash^{w_2} \mathcal{C} :: (\Delta, \Delta')}{\Delta_1, \Delta_2 \Vdash^{w_1+w_2} (\mathcal{O} \parallel \mathcal{C}) :: (\Delta, (c : A))}\ \text{T:Compose}$$

$$\frac{(\forall i \in \mathcal{I}) \quad \Delta_i \Vdash^{w_i} \mathcal{C}_i :: (c : A_i) \quad 1 = \Sigma_{i \in \mathcal{I}} p_i \quad \Delta = \Sigma^{\mathsf{L}}_{i \in \mathcal{I}} p_i \cdot \Delta_i \quad A = \Sigma^{\mathsf{R}}_{i \in \mathcal{I}} p_i \cdot A_i \quad w = \Sigma_{i \in \mathcal{I}} w_i}{\Delta \Vdash^{w} \mathsf{proc}(c, \{\mathcal{C}_i : p_i\}_{i \in \mathcal{I}}) :: (c : A)}\ \text{T:Dist}$$

The rule T:Proc lifts a well-typed process P to a well-typed semantic object. At runtime, all refinement variables V of a process need to be fully instantiated with values that satisfy the refinement constraint C. So P here must be closed over refinement variables and have a trivial constraint $\top$. The T:Compose rule allows a semantic object $\mathcal{O}$ to connect to a configuration $\mathcal{C}$ by consuming some non-empty context Δ' of channels provided by $\mathcal{C}$. Finally, T:Dist types distributions based on the weighted sum of their underlying configurations' types. Note that configuration typing is not an explicitly checked judgment. It functions as meta-level invariant for reasoning about the runtime behavior of programs. Through type preservation we show that

starting from some well-typed process (accepted by the type checker), all configurations it can transition to must be well-typed according to configuration typing.

Type Safety. To prove type safety for PReST, we must account for the exchanging of proofs by assert and assume. The following lemma allows constraint context C to be strengthened to C′ using a proof of entailment V ; C′ ⊨ C. So if a process receives a proof (via assume), it can use this proof to strengthen its local constraint context and make stronger assertions.

Lemma 5.1. *If* $V\,;\,C\,;\,\Delta \vdash^{w} P :: (x : A)$*, then for all* C' *such that* $V\,;\,C' \vDash C$ *holds, the judgment* $V\,;\,C'\,;\,\Delta \vdash^{w} P :: (x : A)$ *holds.*

We can prove the type preservation theorem by induction and appealing to Lemma 5.1 to resolve assume-assert communication cases.

Theorem 5.2 (Preservation). *If* $\Delta \Vdash^{w} C :: \Gamma$*, and* $C \mapsto C'$ *or* $C \overset{d,\kappa}{\Longmapsto} C'$*, then* $\Delta \Vdash^{w} C' :: \Gamma$.

Notice that the configuration typing judgment $\Delta \Vdash^{w} C :: \Gamma$ in the preservation theorem requires all processes in C to be *refinement-closed*. This implies that the semantic quantifiers of probabilistic branching expressions are always fully instantiated prior to their evaluation. Crucially, the refinement-closed property is invariant under both $\mapsto$ and $\overset{d,\kappa}{\Longmapsto}$ transition relations, which in turn makes our semantically quantified typing rules compatible with preservation.

Global Progress. To prove global progress for well-type configurations, the following judgments describing the status of semantic objects are defined. The full derivation rules for these judgments are given in our companion report [33].

- C poised : all processes in C are ready to communicate on the channel they provide
- C live : there exists some process in C that can step without communication
- C (d,κ)-comm : a pair of processes in C are ready to κ-communicate over channel d

For some well-typed configuration $\cdot \Vdash^{w} C :: \Gamma$, it must be in either poised, live or (d,κ)-comm states. One can show that if there is C live, then there exists C' such that $C \mapsto C'$ holds. Similarly, if there is C (d,κ)-comm, then there exists C' such that $C \overset{d,\kappa}{\Longmapsto} C'$ holds. Note that C poised means, by definition, C is in *normal form*. Combining these facts, we obtain the global progress theorem.

Theorem 5.3 (Progress). *If* $\cdot \Vdash^{w} C :: \Gamma$*, then either* $C \mapsto C'$*,* $C \overset{d,\kappa}{\Longmapsto} C'$ *or* C *poised.*

Probability Consistency. Now we show that the distribution of communicated messages at runtime indeed respects the probabilities specified by session types. For some configuration C that provides a probabilistic channel, we compose it with a *monitor process* with the following syntax:

$$M ::= \mathsf{record}\ x \mid \mathsf{wait}\ x\ \mathsf{with}\ k \mid \mathsf{stop}\ k$$

The record x construct receives a label $k \in L$ over channel $x : \oplus_P\{\ell^{p_\ell} : \mathbf{1}\}_{\ell \in L}$ and transitions to wait x with k where communication with C is synchronized for closing. Once channel x is closed, wait x with k transitions to stop k. It is straightforward to extend the proofs of type soundness to encompass monitor processes. One can then show that after evaluation of the overall configuration (C + monitor) has terminated, the distribution of labels recorded in stop k across all possible ending configurations is $\{\ell : p_\ell\}_{\ell \in L}$. This proves that the distributions of labels communicated across a channel is consistent with the probability annotations in its type.

Theorem 5.4 (Consistency). *If* $\cdot \Vdash^{w} C :: (c : \oplus_P\{\ell^{p_\ell} : \mathbf{1}\}_{\ell \in L})$ *and* C *evaluates to a poised configuration, the distribution of messages communicated over* c *is exactly* $\{\ell : p_\ell\}_{\ell \in L}$.

Flattening and Simulation. Configurations C in nested-multiverse semantics can be flattened into distributions μ of configurations in the distribution-based semantics. The flattening relation $C \approx \mu$ is inductively defined as follows:

$$\frac{}{\mathsf{proc}(c, w, P) \approx \{\mathsf{proc}(c, w, P) : 1\}} \qquad \frac{\forall i \in \mathcal{I} : C_i \approx \mu_i}{\mathsf{proc}(c, \{C_i : p_i\}_{i \in \mathcal{I}}) \approx \Sigma_{i \in \mathcal{I}} p_i \cdot \mu_i}$$

$$\frac{O \approx \mu_1 \qquad C \approx \mu_2}{(O \parallel C) \approx \{(\mathcal{D}_1 \parallel \mathcal{D}_2) : \mu_1(\mathcal{D}_1) \cdot \mu_2(\mathcal{D}_2)\}_{\mathcal{D}_1 \in \mathsf{dom}(\mu_1), \mathcal{D}_2 \in \mathsf{dom}(\mu_2)}}$$

The nested-multiverse semantics and distribution-based semantics can now be connected through the simulation theorem. Essentially, every possible transition in the nested-multiverse semantics has a corresponding transition in the distribution-based semantics.

Theorem 5.5 (Simulation). *If $C \approx \mu$, $C' \approx \mu'$ and $C \mapsto C'$ or $C \overset{d,\kappa}{\Longrightarrow} C'$, then $\mu \Mapsto \mu'$.*

Expected Cost Bounds. Having established flattening and simulation, we will use the distribution-based semantics to reason about the expected work of programs. Given a non-nested configuration $\mathcal{D} = \overline{\mathsf{proc}(c_i, w_i, P_i)}$, we define the total work it has done as $\mathsf{work}(\mathcal{D}) := \Sigma_i w_i$. For some well-typed configuration $\mathcal{D}_0$, its *expected* total work is defined as $\mathsf{etw}(\mathcal{D}_0) = \lim_{n \to \infty} \mathbb{E}[\mathsf{work}(\mathcal{D}_n)]$ where $\{\mathcal{D}_n\}_{n \in \mathbb{N}}$ is a Markov chain constructed from $\mathcal{D}_0$ as the initial state and $\overset{\mathsf{prob}}{\longmapsto}$ as the kernel. The following theorem shows that the expected total work of $\mathcal{D}_0$ never exceeds its stated cost.

Theorem 5.6 (Expected Cost Bound). *Given some well-typed configuration $\cdot \Vdash^w \mathcal{D}_0 :: \Gamma$, there is upper bound $etw(\mathcal{D}_0) \leq w$ on the expected total work performed by $\mathcal{D}_0$.*

Undecidability of Type Checking. Das [25] shows that structural session type equality is undecidable in the presence of integer type indices via a simple reduction to 2-counter machines. This result remains valid for real indices as the operations performed by the simulated 2-counter machine maintain the invariant that indices are always in the integer subset.

Theorem 5.7 (Undecidability of Type Equality). *Type equality in PReST is undecidable.*

6 Implementation and Evaluation

We have implemented a prototype of PReST in OCaml containing a lexer, parser and type checker in OCaml. The implementation consists of 2590 lines of OCaml code and 422 lines of Menhir parser specification. When implementing the PReST type checker, the primary challenge lies in the type checking of probabilistic constructs. As explained in Section 4.6, probabilistic rules such as flip and $\oplus_{\mathsf{P}} L$ have existential and universal quantifiers over judgments in their premise. Suppose we wish to type check the following basic usage of flip.

$$p\ ;\ (0 \leq p \leq 1)\ ;\ \cdot \vdash^0 \mathsf{flip}\ p\ (\mathsf{H} \Rightarrow x..\mathsf{true}\ ;\ \mathsf{close}\ x \mid \mathsf{T} \Rightarrow x..\mathsf{false}\ ;\ \mathsf{close}\ x) :: (x : \oplus_{\mathsf{P}}\{\mathsf{true}^{p} : \mathbf{1},\ \mathsf{false}^{1-p} : \mathbf{1}\})$$

The flip rule requires us to show that for all p, there exists $\Delta_H, \Delta_T, A_H, A_T$ such that the premises below all hold true (trivially satisfied premises are omitted).

$$\cdot\ ;\ (0 \leq p \leq 1)\ ;\ \Delta_H \vdash^0 x..\mathsf{true}\ ;\ \mathsf{close}\ x :: (x : A_H) \tag{1}$$

$$\cdot\ ;\ (0 \leq p \leq 1)\ ;\ \Delta_T \vdash^0 x..\mathsf{false}\ ;\ \mathsf{close}\ x :: (x : A_T) \tag{2}$$

$$\cdot\ ;\ (0 \leq p \leq 1) \vDash \oplus_{\mathsf{P}}\{\mathsf{true}^{p} : \mathbf{1},\ \mathsf{false}^{1-p} : \mathbf{1}\} = p \cdot A_H +^{\mathsf{R}} (1-p) \cdot A_T \tag{3}$$

Our type checking algorithm exploits the fact that the weighted sum relations only change the probability annotations of types and do not alter their overall structure. To check the remaining premises, we introduce fresh existential variables $p^{\mathsf{H}}_{\mathsf{true}}, p^{\mathsf{H}}_{\mathsf{false}}, p^{\mathsf{T}}_{\mathsf{true}}, p^{\mathsf{T}}_{\mathsf{false}}$ and define

$$A_H \triangleq \oplus_{\mathsf{P}}\{\mathsf{true}^{p^{\mathsf{H}}_{\mathsf{true}}} : \mathbf{1},\ \mathsf{false}^{p^{\mathsf{H}}_{\mathsf{false}}} : \mathbf{1}\} \qquad A_T \triangleq \oplus_{\mathsf{P}}\{\mathsf{true}^{p^{\mathsf{T}}_{\mathsf{true}}} : \mathbf{1},\ \mathsf{false}^{p^{\mathsf{T}}_{\mathsf{false}}} : \mathbf{1}\}$$

The following constraints are generated from the type validity requirement of A_H, A_T and the weighted sum condition of premise (3).

$$0 \le p^{\mathsf{H}}_{\mathsf{true}} \le 1 \qquad 0 \le p^{\mathsf{T}}_{\mathsf{true}} \le 1 \qquad p^{\mathsf{H}}_{\mathsf{true}} + p^{\mathsf{H}}_{\mathsf{false}} = 1 \qquad p \cdot p^{\mathsf{H}}_{\mathsf{true}} + (1-p) \cdot p^{\mathsf{T}}_{\mathsf{true}} = p$$
$$0 \le p^{\mathsf{H}}_{\mathsf{false}} \le 1 \qquad 0 \le p^{\mathsf{T}}_{\mathsf{false}} \le 1 \qquad p^{\mathsf{T}}_{\mathsf{true}} + p^{\mathsf{T}}_{\mathsf{false}} = 1 \qquad p \cdot p^{\mathsf{H}}_{\mathsf{false}} + (1-p) \cdot p^{\mathsf{T}}_{\mathsf{false}} = 1 - p$$

Futhermore, type checking the individual branches produces constraints $p^{\mathsf{H}}_{\mathsf{true}} = 1, p^{\mathsf{H}}_{\mathsf{false}} = 0, p^{\mathsf{T}}_{\mathsf{true}} = 0, p^{\mathsf{T}}_{\mathsf{false}} = 1$. If some nonempty Δ context is used for type checking, we will introduce fresh existential variables for all of its probabilistic external choice types and generate similar constraints.

At this point, all constraints regarding the existential variables have been generated. A logical conjunction C comprised of these constraints is formed. The formula

$$\forall p.\, (0 \le p \le 1) \supset \exists p^{\mathsf{H}}_{\mathsf{true}}, p^{\mathsf{H}}_{\mathsf{false}}, p^{\mathsf{T}}_{\mathsf{true}}, p^{\mathsf{T}}_{\mathsf{false}}.\, \mathrm{C}$$

is given to the backend for verification. Recall that the weighted sum relation can generate non-linear constraints as is evident from the rightmost constraints above. This necessitates the need for an SMT backend. We support 2 SMT solvers, z3 [28] and cvc5 [7] through the SMTLib2 [19] specification language. Following standard practice, we ask the solvers to prove unsat for the negation of these formulas to achieve better performance. For all of our examples, z3 is able to solve all constraints automatically and does not need any manual intervention.

6.1 Evaluation

To thoroughly evaluate our PReST implementation, we construct an assortment of benchmarks that utilizes refinements and probabilities. We now present more examples of probing processes and randomized distributed protocols. Our implementation scales to large examples generating thousands of SMT constraints. The full definitions of all examples are included with the source code of our implementation. Table 1 compiles the results of type checking all benchmarks.

Sum of Uniform Distributions. A well known fact from probability theory is that the probability distribution of the sum of two uniform random variables is not uniform. For two discrete random variables uniformly drawn from the set $\{0, \ldots, n\}$, the probability mass function (PMF) of their sum takes the form depicted in Figure 8. We claim that the following `usum[n]` type encodes the distribution obtained by adding two `unat[n]` values.

```
type usum[n | 0 ≤ n] =
⊕P{ next^(n²/(n+1)²) :
      ⊕{ succ : ⊕{ succ : ?{1 ≤ n}. usum[n − 1] } }
  , unat^(2n/(n+1)²) :
      ⊕{ succ : ?{1 ≤ n}. unat[n − 1] }
  , zero^(1/(n+1)²) : 1 }
```

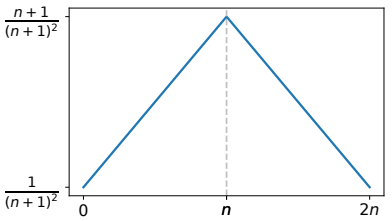

Fig. 8. Sum of 2 uniform values

Due to the kink in the PMF of the distribution of sum, verifying `usum[n]` with a single probing process will be challenging. Instead, we verify `usum[n]` piecewise using two probing processes. The following `probe_usum1` process checks if value `x : usum[n]` is equal to that of `y : unat[i]` where $0 \le i \le n$. Essentially, `probe_usum1` measures the PMF of sum in the range between 0 and n.

```
proc probe_usum1[n, i | 0 ≤ i ≤ n] (x : usum[n]) (y : nat[i])
⊢^n (z : ⊕P{ true^((i+1)/(n+1)²) : 1, false^((n²+2n−i)/(n+1)²) : 1 })
```

Next, the probing process `probe_usum2` is defined to measure the PMF between n and $2n$. It proceeds by subtracting `y : nat[i]` from `x : usum[n]` and comparing the remaining value to `n : nat[n]`.

```
proc probe_usum2[n, i | 0 ≤ i ≤ n] (x : usum[n]) (n : nat[n]) (y : nat[i])
⊢^n (z : ⊕P{ true^((n−i+1)/(n+1)^2) : 1, false^((n^2+n+i)/(n+1)^2) : 1 })
```

From the output types of these probing processes we can see that the probability first scales linearly (w.r.t. i) and reaches a maximum value when $i = n$. The probability then begins to drop off linearly.

Bounded Retransmission. The bounded retransmission protocol [41] is a commonly used protocol for sending chunks of data over a lossy channel. For c chunks of data and communication success rate p, the protocol will make b attempts at sending each chunk across the lossy channel. We define this protocol as the following `sender[c, b, i, p]` type. The extra i index here indicates that the sender is currently on its ith attempt for chunk c.

```
type sender[c, b, i, p  |  0 ≤ c ∧ 0 ≤ i ≤ b ∧ 0 ≤ p ≤ 1] =
  ⊕{ try : ?{1 ≤ i}.
     ⊕{ chunk : ?{1 ≤ c}. data_t ⊗
         &P{ ack^p : sender[c − 1, b, b, p], drop^(1−p) : data_t ⊸ sender[c, b, i − 1, p] }
       , done : ?{c = 0}. 1 }
   , fail : ?{0 = i}. 1 }
```

Compared to NomosPro which requires over 700 lines of code to encode a retransmission protocol with 5 chunks, a bound of 9 and a fixed success rate of 0.8, our implementation in PReST uses only 107 lines of code and operates universally of any number of chunks, bounds and success rates.

Randomized Dining Philosophers. A solution to the famous dining philosophers problem proposed by Lehmann and Rabin [56] utilizes randomness to break symmetry in the acquisition of forks. Initially, each philosopher attempts to acquire a left fork or a right fork based on the outcome of a fair coin flip. Once a fork has been acquired, each philosopher will attempt to acquire their missing fork. If the fork is available, the philosopher may eat. Otherwise, the philosopher will put back their fork and reset to the initial state. We model philosophers as the type `ph[i]`.

```
type ph[i | i = 0 ∨ i = 1 ∨ i = 2 ∨ i = 3] =
 ⊕{ has_none : ?{i = 0}.
     ⊕P{ request_left^0.5 : &{ acquire : fork ⊸ ph[1], none : ph[0] }
       , request_right^0.5 : &{ acquire : fork ⊸ ph[2], none : ph[0] } }
   , has_left  : ?{i = 1}. &{ done : fork ⊗ ph[3], reset : fork ⊗ ph[0] }
   , has_right : ?{i = 2}. &{ done : fork ⊗ ph[3], reset : fork ⊗ ph[0] }
   , done : ?{i = 3}. ph[3] }
```

The i variable in `ph[i]` indicates the state of the philosopher: 0 has no forks, 1 has left fork, 2 has right fork, 3 has eaten. The dining table of philosophers is modeled as a ring of interleaved `ph[i]` and `fork_opt[j]` processes where `fork_opt[j]` is an option type encoding fork availability. Using refinements, we enforce the states of adjacent philosophers and adjacent forks to always be consistent: philosopher can always put back their fork(s) into empty slots. The progress theorem (Theorem 5.3) ensures that no deadlocks occur during fork acquisition.

Randomized Consensus Shared Coin. The randomized consensus shared coin protocol of Aspnes and Herlihy [4] assigns preferences (1 or 2) to n processes using a global shared counter c. To begin the protocol, each process flips a fair coin and increments or decrements c based on the coin flip. After updating the shared counter, if $c \geq k \cdot n$, where k is a constant parameter greater

than 1, the process selects 1 as its preference. If $c \leq -k \cdot n$, the process selects 2 as its preference. Otherwise, the process coin flips again until c passes one of the barriers ($-k \cdot n$ or $k \cdot n$).

```
type node[k, n, x | 0 ≤ k ∧ 0 ≤ n ∧ (x = 0 ∨ x = 1 ∨ x = 2)] =
  ⊕{ try_flip : ?{x = 0}. ∀[c]. int[c] ⊸
       ⊕P{ H^0.5 : node_aux[k, n, (c + 1)], T^0.5 : node_aux[k, n, (c − 1)] }
   , done : ?{x = 1 ∨ x = 2}. nat[x] ⊗ node[k, n, x] }
type node_aux[k, n, c | 0 ≤ k ∧ 0 ≤ n] =
  ⊕{ prefer1 : ?{k · n ≤ c}. int[c] ⊗ node[k, n, 1]
   , prefer2 : ?{c ≤ −k · n}. int[c] ⊗ node[k, n, 2]
   , otherwise : ?{−k · n < c < k · n}. int[c] ⊗ node[k, n, 0] }
```

We give processes in the consensus shared coin protocol the `node[k, n, x]` type as presented above. The indices k and n are the protocol parameter and total number of processes respectively. The index x constrains the process to 3 modes: no preference ($x = 0$), prefers 1 ($x = 1$) and prefers 2 ($x = 2$). If a process is in modes $x = 1$ or $x = 2$, it will send the label `done` followed by a natural number corresponding to its preference. The process then repeats in the same mode. If a process is in mode $x = 0$, then it will receive the global counter c and transition to states `node_aux[k, n, (c + 1)]` or `node_aux[k, n, (c − 1)]` based on a fair coin flip. These two states correspond to incrementing or decrementing the counter respectively. In the `node_aux[k, n, c]` state, based on the value of c in relation to $k \cdot n$ and $-k \cdot n$, the process may prefer 1 and transition to state `node[k, n, 1]`, or prefer 2 and transition to state `node[k, n, 2]`, or reset back to the initial state `node[k, n, 0]`.

Evaluation Results. Table 1 presents additional benchmarks including (i) a center biased random walk, (ii) conjuction of approximate events, (iii) leader election of Itai and Rodeh [48], (iv) crowd forwarding of Reiter and Rubin [64], and (v) zeroconf IP selection [18].

Table 1. Evaluation of PReST. LOC = lines of code; Defs = #type and process definitions; z3(ms) = time to solve using z3; cvc5(ms) = time to solve using cvc5; TVars = total #quantified variables; PVars = #existential variables for probabilistic constructs; Cons = #constraints that need solving; Cost = resource upper bound.

Program	LOC	Defs	z3(ms)	cvc5(ms)	TVars	PVars	Cons	Cost
unat generator	52	5	14	31	8	4	145	$n/2$
unat (PMF)	63	5	20	timeout	14	8	219	$n/2$
unat (CDF)	64	5	22	timeout	14	8	219	$n/2$
umax (CDF)	192	9	278	timeout	59	38	664	$\frac{n\cdot(4\cdot n+5)}{6\cdot(n+1)}$
usum (PMF)	277	10	184	timeout	101	54	1120	n
unat compare	48	3	30	timeout	26	18	244	$n/2$
coin debias1	15	2	9	timeout	18	12	194	$\frac{2\cdot p^2-2\cdot p+1}{2\cdot p\cdot(1-p)}$
one time pad	93	12	32	26	27	12	478	n
approx conj	21	2	11	31	21	12	261	1
random walk	16	2	10	13	6	4	85	n
bounded retransmit	107	13	35	36	12	0	796	$i + c \cdot b$
leader election	257	22	68	timeout	81	16	1749	unbound
dining philosopher	297	14	51	56	14	4	1359	unbound
consensus coin	498	29	75	64	37	4	1932	unbound
crowd forwarding	260	23	62	65	48	4	1662	unbound
zeroconf	278	16	51	timeout	50	20	1064	unbound
unreliable queue	55	4	19	188	21	12	607	$n + 1$

Our experiments are performed on a laptop with an Apple M4 Pro CPU and 24 GB RAM. We compare the performance of using z3 (version 4.13.0) as the SMT backend against cvc5 (version 1.2.0). Table 1 presents the results of running our PReST type checker on various examples. We do not assign cost bounds to some k-way probabilistic distributed protocols because their bounds involve exponential constraints which cannot be solved by even state-of-the-art SMT solvers. From these results, we can see that z3 performs better than cvc5 as it is less prone to timeout and solves constraints within 0.3 seconds. Our results show that even for simple examples, we generate hundreds of constraints with multiple quantifiers, and solving them manually would be infeasible. Thus, a robust SMT backend is a necessity for the usability of PReST.

7 Related Work

Several works have explored combinations of probabilities and session types [11, 27]. As we already discussed extensively, Das et al. [27] proposed a probabilistic session type system but limited to constants with no support for symbolic reasoning. Aman and Ciobanu [3] designed a session type system based on with probabilistic intervals, probabilistic internal choice, and non-deterministic external choice [2]. Besides the differences in the design of the type system due to the fact that we use probabilities rather than probabilistic intervals, their type system does not support refinements, which is one of our main contributions. Dal Lago and Giusti [20] introduce a session typing discipline based on bounded linear logic aimed at capturing experiments in the computational model of cryptography and supports a version of refinement which is weaker than the one we present here: it includes refinements on the types for strings and replication. We consider instead a more general form of arithmetic refinements. Inverso et al. [47] consider a probabilistic session type system which can be seen as a simplified version of the one studied in [27]. This system does not consider refinements and thus is less expressive than the system we presented here.

Probabilistic versions of process calculi have been extensively studied, especially in the context of process equivalence [12, 49, 55]. Several variants of probabilistic processes have been studied [36, 42], where the differences often amount to the kind of probabilistic semantics that processes are equipped with, e.g. [66], which kind of probabilistic choices the calculi support, e.g. [63], and the relations between probabilistic choice and nondeterminism, e.g. [21]. The work by Bartels et al. [8] compare several different calculi accordingly to their expressive power and build a hierarchy out of some of them. Our type system assigns types to a probabilistic process calculus that combines internal and external probabilistic and non-deterministic choice constructors. The main difference from these works at the calculus level is that we embrace the nested multiverse semantics approach proposed by Das et al. [27].

Refinement types have been used extensively to support reasoning about probabilities, particularly in security, privacy and probabilistic programming. RF* [9] uses refinement types, à la F* [65], to enrich the expressivity of a relational calculus to reason about cryptographic primitives, which was subsequently combined with an indexed monad [10] to support reasoning about privacy. Grimm et al. [38] propose a technique to use F* proofs to directly support relational statements about probabilistic programs. Vasilenko et al. [67] use a similar approach to reason about probabilistic programs using Liquid Haskell. Their system supports quantitative specifications which abstract over the properties of the program. None of these works directly manages the exact probabilities within types in a way similar to what we do here nor do they support any form of concurrency.

Another related thread of research is probabilistic model checking tools such as PRISM [54] and Storm [29] that support analysis of both discrete- and continuous-time Markov chains [53], Markov decision processes [30], π-calculus [61], and randomized distributed algorithms [60]. Instead of using type systems, those model checkers provide a state-based language and a specification logic to specify the model and property to be checked. Unlike PRISM or Storm, PReST types serve as

compositional specifications that enable the analysis of different processes in isolation. Also, PReST doesn't just serve as a modeling language; it can be used to implement and generate distributions, not just analyze them.

Finally, there has been a large body of work on expected cost analysis of (sequential) probabilistic programs [16, 17, 39, 51, 57, 62], many of which build on foundational work on pre-expectation calculus by Kozen [52]. Absynth [59] uses the idea of amortized resource analysis for expected cost analysis of first-order probabilistic programs with loops. Unlike PReST, Absynth does not support symbolic probabilities, thus can infer bounds using LP solving. Wang et al. [69] develop a notion of polynomial cost martingales that can handle negative cost with bounded updates to variables. In contrast to these approaches that perform whole-program analysis, Avanzini et al. [6] generalize the expected runtime transformer of Kaminski et al. [51] to build a modular framework for expected cost analysis that allows sampling from dynamic distributions. Similarly to PReST, they generate non-linear constraints and use an SMT solver to find a satisfying assignment. Unlike PReST though, they do not support recursive functions. On a different thread, there have been recent innovations to support higher-order probabilistic programs using affine refinement types [5] and potential types [68]. Unlike PReST however, none of these works supports any form of concurrency. The main technical challenge in supporting message-passing concurrency is to decide how to divide the potential amongst the executing processes. This is precisely where resource-aware types [23] come in: special messages whose goal is to distribute potential amongst processes to pay for their execution cost. On the other hand, most of the aforementioned tools support automatic bound inference, while we only support type checking leaving inference to future work.

Also related are higher-order separation logic frameworks that enable cost analysis of probabilistic programs with mutable state. ExpIris [58] is a variant of Iris [50] that supports proving bounds on the expected cost of higher-order programs with mutable state. It enhances the standard weakest precondition of Iris with the notion of initial and final potential, with the cost being the difference between the two. However, since potential is not part of the logic, it is impossible to make the postcondition potential depend on the final state. Tachis [40] addresses this limitation by decoupling the weakest precondition and costs through the use of a credit resource [1]. With such a powerful logical framework, the bounds proved by these works are more expressive, but they lack automation. With a simple refinement system, we can reduce bound analysis to solving non-linear constraints that can be shipped to an SMT solver.

8 Conclusion

This paper presents PReST, a novel concurrent probabilistic language equipped with an expressive refinement type system. The main novelty is a combination of probabilities and refinements leading to an expressive type system that enables symbolic reasoning of randomized systems. We have also developed a prototype of the language that relies on an SMT solver backend to reconstruct intermediate types, thereby drastically reducing annotation overhead. The language is proven sound by generalizing the nested multiverse semantics to refinement types that establishes that probability distributions of interactions exactly match those prescribed by the type system. PReST is evaluated on a variety of randomized distributed protocols, Markov chains, and verified standard properties of discrete distributions (e.g. max, sum, double, etc.).

9 Data-Availability Statement

A self-contained version of the PReST implementation along with the benchmarks presented in Table 1 are available as an artifact [32].

10 Acknowledgments

Marco Gaboardi's work was partially supported by the National Science Foundation under Grants No. 2314324 and 2217679.

References

[1] Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2024. Error Credits: Resourceful Reasoning about Error Bounds for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 8, ICFP, Article 246 (Aug. 2024), 33 pages. https://doi.org/10.1145/3674635

[2] Bogdan Aman and Gabriel Ciobanu. 2019. Probabilities in Session Types. *Electronic Proceedings in Theoretical Computer Science* 303 (09 2019), 92–106. https://doi.org/10.4204/EPTCS.303.7

[3] Bogdan Aman and Gabriel Ciobanu. 2022. Interval Probability for Sessions Types. In *Logic, Language, Information, and Computation - 28th International Workshop, WoLLIC 2022, Iaşi, Romania, September 20-23, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13468)*, Agata Ciabattoni, Elaine Pimentel, and Ruy J. G. B. de Queiroz (Eds.). Springer, 123–140. https://doi.org/10.1007/978-3-031-15298-6_8

[4] James Aspnes and Maurice Herlihy. 1990. Fast randomized consensus using shared memory. *Journal of algorithms* 11, 3 (1990), 441–461.

[5] Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. 2019. Type-Based Complexity Analysis of Probabilistic Functional Programs. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. https://doi.org/10.1109/LICS.2019.8785725

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

[7] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Vol. 13243. Springer International Publishing, Cham, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24 Series Title: Lecture Notes in Computer Science.

[8] Falk Bartels, Ana Sokolova, and Erik de Vink. 2004. A hierarchy of probabilistic system types. *Theoretical Computer Science* 327, 1 (2004), 3–22. https://doi.org/10.1016/j.tcs.2004.07.019 Selected Papers of CMCS '03.

[9] 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

[10] 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

[11] Christian Bartolo Burlò, Adrian Francalanza, Alceste Scalas, Catia Trubiani, and Emilio Tuosto. 2021. Towards Probabilistic Session-Type Monitoring. In *Coordination Models and Languages*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer International Publishing, Cham, 106–120.

[12] Bard Bloom and Albert R. Meyer. 1989. A Remark on Bisimulation Between Probabilistic Processes. In *Logic at Botik '89, Symposium on Logical Foundations of Computer Science, Pereslav-Zalessky, USSR, July 3-8, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 363)*, Albert R. Meyer and Michael A. Taitslin (Eds.). Springer, 26–40. https://doi.org/10.1007/3-540-51237-3_4

[13] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Paul Gastin, and François Laroussinie (Eds.). Vol. 6269. Springer Berlin Heidelberg, Berlin, Heidelberg, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16 Series Title: Lecture Notes in Computer Science.

[14] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 760 (11 2014).

[15] Iliano Cervesato and Andre Scedrov. 2006. Relating State-Based and Process-Based Concurrency through Linear Logic. *Electronic Notes in Theoretical Computer Science* 165 (Nov. 2006), 145–176. https://doi.org/10.1016/j.entcs.2006.05.043

[16] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 3–22.

[17] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2018. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. *ACM Trans. Program. Lang. Syst.* 40, 2, Article 7 (May 2018), 45 pages. https://doi.org/10.1145/3174800

[18] S. Cheshire and Bernard Aboba. 2005. Dynamic Configuration of IPv4 Link-Local Addresses. *IETF Internet Draft* (01 2005).

[19] David R Cok et al. 2011. The smt-libv2 language and tools: A tutorial. *Language c* (2011), 2010–2011.

[20] Ugo Dal Lago and Giulia Giusti. 2022. On Session Typing, Probabilistic Polynomial Time, and Cryptographic Experiments. In *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland (LIPIcs, Vol. 243)*, Bartek Klin, Slawomir Lasota, and Anca Muscholl (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 37:1–37:18. https://doi.org/10.4230/LIPICS.CONCUR.2022.37

[21] Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. 1998. On Generative Parallel Composition. In *First International Workshop on Probabilistic Methods in Verification, PROBMIV 1998, Indianapolis, Indiana, USA, June 19-20, 1998 (Electronic Notes in Theoretical Computer Science, Vol. 22)*, Christel Baier, Michael Huth, Marta Z. Kwiatkowska, and Mark Ryan (Eds.). Elsevier, 30–54. https://doi.org/10.1016/S1571-0661(05)80596-1

[22] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2021. Resource-Aware Session Types for Digital Contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, Dubrovnik, Croatia, 1–16. https://doi.org/10.1109/CSF51468.2021.00004

[23] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 305–314. https://doi.org/10.1145/3209108.3209146

[24] Ankush Das and Frank Pfenning. 2020. Rast: Resource-Aware Session Types with Arithmetic Refinements (System Description). In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 33:1–33:17. https://doi.org/10.4230/LIPIcs.FSCD.2020.33

[25] Ankush Das and Frank Pfenning. 2020. Session Types with Arithmetic Refinements. (2020), 18 pages, 512384 bytes. https://doi.org/10.4230/LIPICS.CONCUR.2020.13

[26] Ankush Das and Frank Pfenning. 2020. Verified Linear Session-Typed Concurrent Programming. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*. ACM, Bologna Italy, 1–15. https://doi.org/10.1145/3414080.3414087

[27] Ankush Das, Di Wang, and Jan Hoffmann. 2023. Probabilistic Resource-Aware Session Types. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 1925–1956. https://doi.org/10.1145/3571259

[28] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[29] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A Storm is Coming: A Modern Probabilistic Model Checker. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 592–600.

[30] Vojtěch Forejt, Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. *Automated Verification Techniques for Probabilistic Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 53–113. https://doi.org/10.1007/978-3-642-21455-4_3

[31] Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. *SIGPLAN Not.* 26, 6 (may 1991), 268–277. https://doi.org/10.1145/113446.113468

[32] Qiancheng Fu, Ankush Das, and Marco Gaboardi. 2025. Probabilistic Refinement Session Types (Artifact). https://doi.org/10.5281/zenodo.15151161

[33] Qiancheng Fu, Ankush Das, and Marco Gaboardi. 2025. Probabilistic Refinement Session Types (Companion Report). https://doi.org/10.5281/zenodo.15185261

[34] Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

[35] J.-Y. Girard. 1995. Linear Logic: its syntax and semantics. In *Advances in Linear Logic* (1 ed.), Jean-Yves Girard, Yves Lafont, and Laurent Regnier (Eds.). Cambridge University Press, 1–42. https://doi.org/10.1017/CBO9780511629150.002

[36] Jean Goubault-Larrecq, Catuscia Palamidessi, and Angelo Troina. 2007. A Probabilistic Applied Pi–Calculus. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–190.

[37] Dennis Griffith and Elsa L. Gunter. 2013. LiquidPi: Inferrable Dependent Session Types. In *NASA Formal Methods*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 185–197.

[38] 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

[39] Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2019. Aiming low is harder: induction for lower bounds in probabilistic program verification. *Proc. ACM Program. Lang.* 4, POPL, Article 37 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371105

[40] Philipp G. Haselwarter, Kwing Hei Li, Markus de Medeiros, Simon Oddershede Gregersen, Alejandro Aguirre, Joseph Tassarotti, and Lars Birkedal. 2024. Tachis: Higher-Order Separation Logic with Credits for Expected Costs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 313 (Oct. 2024), 30 pages. https://doi.org/10.1145/3689753

[41] L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. 1994. Proof-checking a data link protocol. In *Types for Proofs and Programs*, Henk Barendregt and Tobias Nipkow (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 127–165.

[42] Oltea Mihaela Herescu and Catuscia Palamidessi. 2000. Probabilistic Asynchronous π-Calculus. In *Foundations of Software Science and Computation Structures*, Jerzy Tiuryn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–160.

[43] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, G. Goos, J. Hartmanis, and Eike Best (Eds.). Vol. 715. Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523. https://doi.org/10.1007/3-540-57208-2_35 Series Title: Lecture Notes in Computer Science.

[44] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

[45] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 273–284. https://doi.org/10.1145/1328438.1328472

[46] Omar Inverso, Hernán Melgratti, Luca Padovani, Catia Trubiani, and Emilio Tuosto. 2020. Probabilistic Analysis of Binary Sessions. In *31st International Conference on Concurrency Theory (CONCUR 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:21. https://doi.org/10.4230/LIPIcs.CONCUR.2020.14

[47] Omar Inverso, Hernán C. Melgratti, Luca Padovani, Catia Trubiani, and Emilio Tuosto. 2020. Probabilistic Analysis of Binary Sessions. In *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference) (LIPIcs, Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:21. https://doi.org/10.4230/LIPICS.CONCUR.2020.14

[48] Alon Itai and Michael Rodeh. 1990. Symmetry breaking in distributed networks. *Information and Computation* 88, 1 (1990), 60–87. https://doi.org/10.1016/0890-5401(90)90004-2

[49] Chi-Chang Jou and Scott A. Smolka. 1990. Equivalences, Congruences, and Complete Axiomatizations for Probabilistic Processes. In *CONCUR '90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 27-30, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 458)*, Jos C. M. Baeten and Jan Willem Klop (Eds.). Springer, 367–383. https://doi.org/10.1007/BFB0039071

[50] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 256–269. https://doi.org/10.1145/2951913.2951943

[51] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–389.

[52] Dexter Kozen. 1981. Semantics of probabilistic programs. *J. Comput. System Sci.* 22, 3 (1981), 328–350. https://doi.org/10.1016/0022-0000(81)90036-2

[53] Marta Kwiatkowska, Gethin Norman, and David Parker. 2007. *Stochastic Model Checking*. Springer Berlin Heidelberg, Berlin, Heidelberg, 220–270. https://doi.org/10.1007/978-3-540-72522-0_6

[54] Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 585–591.

[55] Kim G. Larsen and Arne Skou. 1989. Bisimulation Through Probabilistic Testing. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 344–352. https://doi.org/10.1145/75277.75307

[56] Daniel Lehmann and Michael O Rabin. 1981. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles*

of programming languages. 133–138.

[57] Lorenz Leutgeb, Georg Moser, and Florian Zuleger. 2022. Automated Expected Amortised Cost Analysis of Probabilistic Data Structures. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 70–91.

[58] Janine Lohse and Deepak Garg. 2024. An Iris for Expected Cost Analysis. arXiv:2406.00884 [cs.PL] https://arxiv.org/abs/2406.00884

[59] 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* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 496–512. https://doi.org/10.1145/3192366.3192394

[60] Gethin Norman. 2004. *Analysing Randomized Distributed Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 384–418. https://doi.org/10.1007/978-3-540-24611-4_11

[61] G. Norman, C. Palamidessi, D. Parker, and P. Wu. 2007. Model checking the probabilistic pi-calculus. In *Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007)*. 169–178.

[62] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 672–681. https://doi.org/10.1145/2933575.2935317

[63] Sylvain Pradalier and Catuscia Palamidessi. 2006. Expressiveness of Probabilistic pi. In *Proceedings of the 4th International Workshop on Quantitative Aspects of Programming Languages, QAPL 2006, Vienna, Austria, April 1-2, 2006 (Electronic Notes in Theoretical Computer Science, Vol. 164)*, Alessandra Di Pierro and Herbert Wiklicky (Eds.). Elsevier, 119–136. https://doi.org/10.1016/J.ENTCS.2006.07.015

[64] Michael K Reiter and Aviel D Rubin. 1999. Anonymous web transactions with crowds. *Commun. ACM* 42, 2 (1999), 32–48.

[65] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 256–270. https://doi.org/10.1145/2837614.2837655

[66] Rob J. van Glabbeek, Scott A. Smolka, and Bernhard Steffen. 1995. Reactive, Generative and Stratified Models of Probabilistic Processes. *Inf. Comput.* 121, 1 (1995), 59–80. https://doi.org/10.1006/INCO.1995.1123

[67] Elizaveta Vasilenko, Niki Vazou, and Gilles Barthe. 2022. Safe couplings: coupled refinement types. *Proc. ACM Program. Lang.* 6, ICFP (2022), 596–624. https://doi.org/10.1145/3547643

[68] Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising expectations: automating expected cost analysis with types. *Proc. ACM Program. Lang.* 4, ICFP, Article 110 (Aug. 2020), 31 pages. https://doi.org/10.1145/3408992

[69] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. 2019. Cost analysis of nondeterministic probabilistic programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 204–220. https://doi.org/10.1145/3314221.3314581

[70] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL 1999)*, A. Aiken (Ed.). ACM Press, San Antonio, Texas, USA, 214–227.

[71] Fangyi Zhou. 2019. *Refinement Session Types*. Master's thesis. Imperial College London.

[72] Fangyi Zhou, Francisco Ferreira, Rumyana Neykova, and Nobuko Yoshida. 2019. Fluid Types: Statically Verified Distributed Protocols with Refinements. In *11th Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software*.

Received 2024-11-14; accepted 2025-03-06