Probabilistic Resource-Aware Session Types

ANKUSH DAS*, Amazon, USA
DI WANG, Carnegie Mellon University, USA
JAN HOFFMANN, Carnegie Mellon University, USA

Session types guarantee that message-passing processes adhere to predefined communication protocols. Prior work on session types has focused on deterministic languages but many message-passing systems, such as Markov chains and randomized distributed algorithms, are probabilistic. To implement and analyze such systems, this article develops the meta theory of probabilistic session types with an application focus on automatic expected resource analysis. Probabilistic session types describe probability distributions over messages and are a conservative extension of intuitionistic (binary) session types. To send on a probabilistic channel, processes have to utilize internal randomness from a probabilistic branching or external randomness from receiving on a probabilistic channel. The analysis for expected resource bounds is smoothly integrated with the type system and is a variant of automatic amortized resource analysis. Type inference relies on linear constraint solving to automatically derive symbolic bounds for various cost metrics. The technical contributions include the meta theory that is based on a novel nested multiverse semantics and a type-reconstruction algorithm that allows flexible mixing of different sources of randomness without burdening the programmer with complex type annotations. The type system has been implemented in the language NomosPro with linear-time type checking. Experiments demonstrate that NomosPro is applicable in different domains such as cost analysis of randomized distributed algorithms, analysis of Markov chains, probabilistic analysis of amortized data structures and digital contracts. NomosPro is also shown to be scalable by *(i)* implementing two broadcast and a bounded retransmission protocol where messages are dropped with a fixed probability, and *(ii)* verifying the limiting distribution of a Markov chain with 64 states and 420 transitions.

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

Additional Key Words and Phrases: Session Types, Resource Analysis, Probabilistic Concurrency, Nested Multiverse Semantics

ACM Reference Format:
Ankush Das, Di Wang, and Jan Hoffmann. 2023. Probabilistic Resource-Aware Session Types. *Proc. ACM Program. Lang.* 7, POPL, Article 66 (January 2023), 32 pages. https://doi.org/10.1145/3571259

1 INTRODUCTION

Session types were introduced by Honda [1993] to statically prescribe binary communication protocols for message-passing processes. As an example, the session type

$$\mathsf{coins} \triangleq \oplus\{\textbf{heads} : \mathsf{coins}, \textbf{tails} : \mathsf{coins}\}$$

*work completed prior to joining Amazon

Authors' addresses: Ankush Das, Amazon, USA, daankus@amazon.com; Di Wang, Carnegie Mellon University, USA, diw3@alumni.cmu.edu; Jan Hoffmann, Carnegie Mellon University, USA, jhoffmann@cmu.edu.

2475-1421/2023/1-ART66
https://doi.org/10.1145/3571259

describes a protocol whose provider process generates a stream of coin flips: the process internally decides a label (either **heads** or **tails**) to send out (specified by the type former $\oplus$), and then recurses to generate further coin flips. Session-typed systems often enjoy type safety properties such as *absence of deadlocks* (global progress) and *session fidelity* (type preservation) and guarantee adherence to protocols at runtime [Caires and Pfenning 2010]. Existing work on session types has focused on deterministic languages [Balzer and Pfenning 2017; Das and Pfenning 2020a; Honda et al. 2008; Toninho et al. 2013] but many message-passing systems are naturally *probabilistic*. For instance, a desirable property of the coins protocol is that its provider process always generates *fair* coin flips, i.e., the process sends out **heads** and **tails** with the same probability.

In general, distributed algorithms often use randomization as a tool to overcome limitations of deterministic algorithms (e.g., Itai and Rodeh [1990] leader election protocol and Chaum [1988] dining cryptographers protocol). In other systems, probability distributions are used to model uncertainty of external events, e.g., messages being dropped in distributed protocols such as Helmink et al. [1994] bounded retransmission and Bracha [1987] reliable broadcast. Markov chains can also be modeled as distributed systems of probabilistic message-passing processes.

This article presents NomosPro, a session-typed concurrent probabilistic programming language that can be used to analyze and implement probabilistic message-passing systems. In NomosPro, a process can utilize two sources of randomness: *(i)* an internal source from a new term former for probabilistic branching and *(ii)* an external source from receiving messages on a probabilistic channel according to some distribution. Our design of NomosPro has focused on two aspects:

- NomosPro is capable of *automatically deriving expected cost bounds for concurrent message-passing systems*, parameterized by a cost metric. Expected-cost analyses, also known as *average-case* cost analyses, are often precise and more useful than *worst-case* cost analyses, for probabilistic systems that exhibits program paths with high cost but low probability.
- NomosPro is a *conservative* extension of a practical session-typed language [Das et al. 2018] such that existing code of concurrent protocols can be *reused* and analyzed directly. In particular, NomosPro supports *both probabilistic and standard choices* and higher-order protocols (that involve channel passing). Supporting standard choice operators is especially crucial since symbolic expected bound analysis often involves concurrent data structures like lists, queues, etc. that are implemented via standard choice operators (as our examples will show later).

To design a compositional expected-cost analysis, we need to track the probability distributions of messages being exchanged among processes. Recent work has been integrating session types with probabilities [Aman and Ciobanu 2019; Inverso et al. 2020], where the key innovation was to replace choice types with probabilistic variants, e.g., the fcoins protocol below prescribes a protocol for generating fair coin flips, where each label is annotated with a probability.

$$\mathsf{fcoins} \triangleq \oplus_{\mathsf{P}}\{\mathbf{heads}^{0.5} : \mathsf{fcoins}, \mathbf{tails}^{0.5} : \mathsf{fcoins}\}$$

Despite their success in modeling probabilities, none of existing approaches are a *conservative* extension of their base non-probabilistic session-typed language, which hampers their ability of implementing and analyzing probabilistic systems. Aman and Ciobanu [2019] considered multiparty session types, but their system only supported probabilistic internal choices and standard external choices. Inverso et al. [2020] considered binary session types, but both internal and external choices in their system can only be probabilistic. In addition, none of them provide an *implementation* of their systems.

Supporting both probabilistic and standard choice types may look straightforward, but it turns out to cause substantial challenges in the development of the *meta theory* of NomosPro, which is one of our major contributions. State-of-the-art techniques for proving preservation of probabilistic programming languages [Avanzini et al. 2019; Inverso et al. 2020; Wang et al. 2020] do not directly

apply to NomosPro. From an operational point of view, the issue is that a probabilistic branch results in different universes; for example, a coin flip results in two universes, and the flip shows heads in one universe, but shows tails in the other. These universes cannot be considered in isolation because the process that performed a coin flip may yield diverged message distributions based on the result of the coin flip, but other processes may still assume an outdated protocol for message distributions, which can be inconsistent in the universes after the coin flip. Our solution is to develop a novel *nested-multiverse* semantics that manages the divergence of message distributions to control the impact of randomness in one process on other processes.

Based on the type information on probability distributions of messages, we develop a type-based expected-cost analysis based on automatic amortized resource analysis (AARA) [Hoffmann et al. 2017; Hofmann and Jost 2003]. Recent advances in AARA have covered expected-cost analysis for sequential probabilistic programs [Wang et al. 2020] and work analysis for concurrent deterministic systems [Das et al. 2018]. However, the development of a probabilistic variant of AARA has shown to be non-trivial due to the complexity in meta theory [Wang et al. 2020]. We address this issue by *flattening* the nested-multiverse semantics and proving its soundness with respect to a cost model using Markov-chain-based reasoning. Similar to other AARA techniques, our implementation of NomosPro reduces expected-cost bound inference to linear programming (LP).

NomosPro's expected-cost analysis provides several unique advantages over state-of-the-art techniques for automated cost analysis for concurrent probabilistic systems [Dehnert et al. 2017; Kwiatkowska et al. 2011]. First, integrating the probability information with session types enables a more compositional approach in which different processes can be analyzed in isolation (discussed in Section 2). Second, amortization allows storing potential within data structures which ultimately result in symbolic bounds. Such bounds are not simply functions of data-structure sizes but of *interactions* between communicating processes, a feature exclusive to NomosPro. Finally, amortization can also interact with probability in non-trivial ways. For instance, we implemented a concurrent queue (using two lists) receiving insert and delete requests with fixed probabilities and, using amortization, inferred a linear expected cost bound. In comparison, a naïve analysis would report a quadratic bound since deletion can be linear-time in the worst case. We are not aware of other tools that can perform such automated amortized analyses for probabilistic concurrent programs.

Another exclusive benefit of NomosPro's expected-cost analysis is that it improves the *precision* for the analysis of probability distribution of messages. A probabilistic choice type specifies a probability distribution where the probability annotations of messages sum up to one, but existing systems (e.g., [Inverso et al. 2020]) can only interpret the annotations as *upper bounds* on the actual probabilities. The reason is that a process might not be terminating at all, thus some portions of the message distribution cannot be reachable. NomosPro can prove that a probabilistic concurrent program terminates with probability one by showing that the expected total work of the system is finite ($\mathbb{E}[X] < \infty$ implies $\mathbb{P}[X < \infty] = 1$ where X is the expected work). In this case, NomosPro can prove the probabilistic choice types in a program are precise, rather than merely upper bounds.

In addition to the language design, the meta theory, and the expected-cost analysis, a major technical contribution is a practical implementation of NomosPro. At the core of our implementation is an efficient *linear-time* type-reconstruction algorithm that computes the probability distribution of labels sent on a probabilistic channel. The probabilistic type rules provide a high degree of flexibility allowing programmers to arbitrarily nest internal and external sources of randomness. We do not want to burden programmers with providing intermediate type annotations for probabilistic channels but this flexibility complicates type-reconstruction. We solve this challenge by employing a bi-directional type checker [Pierce and Turner 2000] that applies the typing rules to reconstruct the intermediate probabilities for each channel from its initial type. In some situations, NomosPro

can also *infer the probability distribution* on a given channel. For e.g., if the input probability distribution for a process is given, the output probability distribution can be inferred automatically.

We have extensively evaluated NomosPro using a diverse probabilistic systems. NomosPro can infer expected cost bounds on probabilistic distributed protocols, e.g., Itai and Rodeh [1990]'s leader election protocol. It can verify correctness and limiting distributions of Markov chains, and infer the expected number of state transitions they incur. We verified the correctness of dice programs [Knuth and Yao 1976] that model dice from coin flips, and limiting distribution of a Markov chain modeling the position of a king on a chessboard. NomosPro can also be used to study the probabilistic behavior of digital contracts such as lotteries and slot machines. The scalability of NomosPro is evaluated using two broadcast protocols: Bracha [1987] and Srikanth and Toueg [1987], and a bounded retransmission protocol [Helmink et al. 1994] where messages are dropped with a fixed probability. Our linear-time type checking and reconstruction algorithm, and the use of efficient LP solving for bound inference show that NomosPro scales to programs larger than 800 LOC. However, since the inference engine in our implementation is based on LP solving, we can only compute linear amortized bounds. But linear bounds are only a limitation of the implementation, not the formal framework or the type system.

In summary, we make the following contributions.

- The design of NomosPro, a language with probabilistic session types which conservatively extend deterministic session types. (§4)
- An AARA for deriving symbolic bounds on the expected cost of NomosPro programs and verifying that probabilistic distributions are precise (not upper bounds). (§4)
- The soundness proof of the type system with a novel nested-multiverse semantics establishing type preservation, global progress, and probability consistency. (§5)
- An implementation of NomosPro with linear-time type checking and extensive evaluation. (§6 and §7)

2 OVERVIEW OF NOMOSPRO

We follow the approach and syntax of the Rast language [Das and Pfenning 2020a,b,c], which implements recursively defined session types and processes. In this approach, every channel has a unique provider and client. We view the session type as describing the communication from the provider's point of view, with the client having to perform dual actions.

Consider the session type bool defined as

$$\text{bool} \triangleq \oplus\{\textbf{true} : \mathbf{1}, \textbf{false} : \mathbf{1}\}$$

The *internal choice* type constructor $\oplus$ dictates that the provider must send either **true** or **false** followed by *termination* as indicated by the continuation type **1**.

Probabilistic Processes. Suppose we wish to define a process TF that outputs **true** with probability $p \in [0, 1]$ and **false** with probability $1 - p$. We introduce a probabilistic term flip p ($\mathsf{H} \Rightarrow P_1 \mid \mathsf{T} \Rightarrow P_2$), operationally interpreted as flipping a coin that outputs heads (H) with probability p and tails (T) with probability $1 - p$. If the coin outputs heads, we execute P_1, otherwise we execute P_2. We employ this term to define process TF with $p = 0.6$ (we only allow constant probabilities).

```
decl TF : . ⊢ (b : bool)
proc b ← TF = flip 0.6 ( H ⇒ b.true ; close b | T ⇒ b.false ; close b )
```

On the first line, we declare the TF process showing that it uses an empty context (dot before the turnstile) and offers the channel b of type bool. On the next line, we define the process expression: term $b \leftarrow \mathsf{TF}$ is the syntax for defining (or spawning) process TF offering on b and using no channels. The TF process first flips a coin with probability of H being 0.6. If the coin flips to H (resp., T), the process sends the label **true** (resp., **false**) using the term b.**true** (resp., b.**false**) and

terminates by closing the channel b using term close b. Since the probability of H is 0.6, the process TF outputs **true** with probability 0.6, and **false** with probability $1 - 0.6 = 0.4$.

Negation. Suppose we consider a negation process neg defined below that takes a channel b : bool as input and negates it (output **false** if input is **true** and vice-versa).

```
decl neg : (b : bool) ⊢ (c : bool)
proc c ← neg b = case b ( true ⇒ c.false ; wait b ; close c
                        | false ⇒ c.true ; wait b ; close c )
```

The declaration describes that the neg process uses channel b : bool and provides c : bool. This is similarly denoted in the definition as $c \leftarrow$ neg b. The definition branches on the label received on channel b: if the process receives **true**, it sends **false** on c and vice-versa. Then, in either case, the process waits for the channel b to close using the term wait b and then closes channel c.

Probabilistic Session Types. Although processes can exhibit probabilistic behavior, this information is not visible in their session types. Therefore, in this article, we use probabilistic session types (e.g., [Inverso et al. 2020]) that assign probabilities to the labels in a session type. We introduce a *probabilistic internal choice type operator* $\oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}$ prescribing that the provider sends label ℓ with probability p_ℓ and continues to provide type A_ℓ. The dual type is $\&_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}$ where the provider is guaranteed to receive label ℓ with probability p_ℓ. Below presents some examples.

$$\mathsf{pbool} \triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{0.6} : \mathbf{1}, \mathbf{false}^{0.4} : \mathbf{1}\} \qquad \mathsf{npbool} \triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{0.4} : \mathbf{1}, \mathbf{false}^{0.6} : \mathbf{1}\}$$

The type pbool outputs **true** with probability 0.6 and **false** otherwise. Its negation type npbool outputs **true** with probability 0.4 and **false** otherwise. With these types, *without changing the process definitions*,[1] we obtain the following types for the aforementioned processes.

```
decl TF : . ⊢ (b : pbool)          decl neg : (b : pbool) ⊢ (c : npbool)
```

The soundness theorem ensures that the distribution of the labels sent on a probabilistic channel at runtime is *upper-bounded* by the distribution of the labels in the choice types. More importantly, if a concurrent system is almost-surely terminating (i.e., terminates with probability one), the distribution of labels on a probabilistic channel *exactly matches* the type of the channel. We show later how our resource-aware type system can be used to prove almost-sure termination.

To send on a probabilistic channel, a process can use two sources of randomness: a flip (like in TF) or case on labels received on another probabilistic channel according to a known distribution (like in neg). These sources of randomness can be *combined and nested* as long as the resulting distributions are valid. For instance, we can define a process debias as follows that uses a biased coin (b : pbool) and produces an unbiased coin (c : ubool $\triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{0.5} : \mathbf{1}, \mathbf{false}^{0.5} : \mathbf{1}\}$). The process first branches on the biased coin b. In the first branch, it flips a biased coin to decide whether to negate the input or not. In the second branch it just sends **false**.

```
decl debias : (b : pbool) ⊢ (c : ubool)
proc c ← debias b = case b ( true ⇒ flip 0.166667 ( H ⇒ c.false; wait b; close c
                                                  | T ⇒ c.true; wait b; close c )
                           | false ⇒ c.false; wait b; close c )
```

A contribution of the article is an efficient *linear-time* type checking algorithm that validates that implementations produce the distributions defined in the types.

Probabilistic session types are naturally *compositional*. We can define a negneg process that calls neg twice to obtain an identity process:

```
decl negneg : (b : pbool) ⊢ (d : pbool)
proc d ← negneg b = c ← neg b ; d ← neg c
```

[1] In this article, we actually distinguish between standard and probabilistic send and case analysis for clarity. However, it is not necessary to make this distinction in the surface syntax.

Since $1-(1-p)=p$, we obtain that the input and output types for negneg are equal.

Inference of Probabilities. When the generated constraints are linear, NomosPro can automatically infer the output probabilities of a process given the input probabilities. Recall the TF process that outputs **true** with probability 0.6 and **false** otherwise. We allow the programmer to define a *starred* boolean type as sbool $\triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^* : \mathbf{1}, \mathbf{false}^* : \mathbf{1}\}$, where $*$ denotes probability values that need to be inferred. The type checker internally replaces $*$ with probability variables, e.g. $\oplus_{\mathsf{P}}\{\mathbf{true}^{p_1} : \mathbf{1}, \mathbf{false}^{p_2} : \mathbf{1}\}$ with the constraint $p_1 + p_2 = 1$. Then, the typing rules of NomosPro are applied which intuitively compute the probability of outputting each label. In the H branch, the probability of outputting **true** is 1, while in the T branch, the probability of outputting **true** is 0. Computing the overall probability of outputting **true**, the type checker thereby generates the constraint $p_1 = 0.6 \times 1 + (1-0.6) \times 0$. Similarly, computing the probability of outputting **false** generates the constraint $p_2 = 0.6 \times 0 + (1-0.6) \times 1$. The LP solver takes these constraints as input and produces the satisfying assignment $p_1 = 0.6, p_2 = 0.4$ which is then substituted back in the program. In a similar fashion, NomosPro can infer the output probabilities for the neg and debias processes if the input probabilities are given. Internally, NomosPro employs a linear programming (LP) solver to compute these probability annotations, and can only be used if the constraints generated by the type checker are linear. We will explain this further when discussing the probabilistic typing rules in Section 4.

Recursion. A core feature of NomosPro is its support for recursion, both at the type and process level. For instance, consider a recursive debiasing process that implements the von Neumann algorithm to take a stream of biased booleans and produce an unbiased boolean. First, we use co-inductively interpreted [Das et al. 2021b] types to define an infinite stream as follows, where the $\&$ type constructor stands for *external choice*, i.e., the client can send either **req** or **done**.

$$\text{pbools} \triangleq \&\{\mathbf{req} : \oplus_{\mathsf{P}}\{\mathbf{true}^{0.6} : \text{pbools}, \mathbf{false}^{0.4} : \text{pbools}\}, \mathbf{done} : \mathbf{1}\}$$

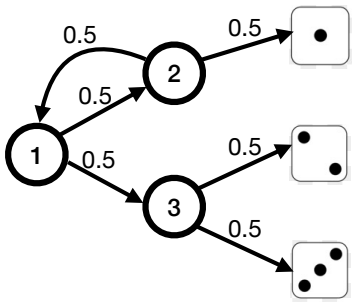
We define ubool $\triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^* : \mathbf{1}, \mathbf{false}^* : \mathbf{1}\}$ so that NomosPro can infer that the output is an unbiased boolean. Next, we define a recursive process rec_debias:

```
decl rec_debias : (b : pbools) ⊢ (c : ubool)
proc c ← rec_debias b =
  b.req; case b ( true ⇒ b.req; case b ( true ⇒ c ← rec_debias b
                                       | false ⇒ c.true; b.done; wait b; close c )
                | false ⇒ b.req; case b ( true ⇒ c.false; b.done; wait b; close c
                                       | false ⇒ c ← rec_debias b) )
```

The process branches on b twice and outputs **true** when input is (**true**, **false**), outputs **false** when input is (**false**, **true**), and recurses in the other two cases. NomosPro can automatically infer that the output probability annotations of **true** and **false** is 0.5 each. *This example combines most of NomosPro's powerful features: type and process level recursion, nesting of case analysis, probabilistic and regular choice operators, automatic inference of probabilities, and almost-sure termination verification* (will be discussed later in this section).

Application to Markov Chains. Probabilistic session types are a natural fit for *implementing and analyzing* Markov chains. An application of Markov chains are *dice programs* [Knuth and Yao 1976] that use a fair coin to model a die. The Markov chain for one such program is described in Figure 1(a). For simplicity of exposition, we consider a 3-faced die, although we have implemented the complete 6-faced die program (see Section 7).

The Markov chain initiates in state 1, and transitions to states 2 and 3 with probability 0.5 each. In state 2, with probability 0.5, the chain outputs face 1 and with probability 0.5, it transitions to back to state 1. State 3 outputs face 2 and 3 with probability 0.5 each.




```
decl P1 : . ⊢ (p : T1)
decl P2 : . ⊢ (p : T2)
decl P3 : . ⊢ (p : T3)

proc p ← P1 = flip 0.5 ( H ⇒ p ← P2
                       | T ⇒ p ← P3 )
proc p ← P2 = flip 0.5 ( H ⇒ p.one ; close p
                       | T ⇒ p ← P1 )
proc p ← P3 = flip 0.5 ( H ⇒ p.two ; close p
                       | T ⇒ p.three ; close p )
```

Fig. 1. (a) A Markov chain associated with the 3-faced die program, and (b) its corresponding program.

Probabilistic session types can prove the functional correctness of this die program, i.e., the modeled die produces each face with equal probability. To this end, we implement the program corresponding this chain and define three probabilistic types.

$$T_1 \triangleq \oplus_{\mathsf{P}}\{\mathbf{one}^{p_1} : \mathbf{1}, \mathbf{two}^{p_2} : \mathbf{1}, \mathbf{three}^{p_3} : \mathbf{1}\} \qquad T_2 \triangleq \oplus_{\mathsf{P}}\{\mathbf{one}^{p_4} : \mathbf{1}, \mathbf{two}^{p_5} : \mathbf{1}, \mathbf{three}^{p_6} : \mathbf{1}\}$$
$$T_3 \triangleq \oplus_{\mathsf{P}}\{\mathbf{one}^{p_7} : \mathbf{1}, \mathbf{two}^{p_8} : \mathbf{1}, \mathbf{three}^{p_9} : \mathbf{1}\}$$

Type T_i denotes the conditional probability of outputting each label from state i. Next, we define process P_i corresponding to state i. Each process P_i offers type T_i. Figure 1(b) outlines the declaration and definition of each process. Process P_1 flips with probability 0.5 and calls P_2 in the H branch, (corresponds to transition to state 2), and calls P_3 in the T branch. Process P_2 flips with probability 0.5 and outputs **one** in the H branch, and calls P_1 in the T branch. Finally, P_3 flips with probability 0.5 and outputs **two** in the H branch and **three** in the T branch, respectively.

Since this Markov chain is mutually recursive, computing the conditional probability of sending each label from each state is challenging. As a first illustration, consider process P_3. The probability of sending **one** for P_3 in either branch is 0. Therefore, the total probability of outputting **one** is $0.5 \times 0 + (1 - 0.5) \times 0 = 0$ (weighted sum of both branches). Similarly, the probability of outputting **two** and **three** is $0.5 \times 1 + (1 - 0.5) \times 0 = 0.5$ and $0.5 \times 0 + (1 - 0.5) \times 1 = 0.5$, respectively. Substituting these values into type T_3, we infer $p_7 = 0$, $p_8 = 0.5$, and $p_9 = 0.5$.

Next, consider the probability of outputting **one** for process P_1. In the H branch, it calls process P_2 which outputs **one** with probability p_4. In the T branch, it calls process P_3 which outputs **one** with probability p_7. Therefore, the total probability that P_1 outputs **one** is $0.5 \cdot p_4 + 0.5 \cdot p_7$. Applying a similar argument for labels **two** and **three**, we obtain the constraints shown below.

$$p_1 = 0.5 \cdot p_4 + 0.5 \cdot p_7 \qquad p_2 = 0.5 \cdot p_5 + 0.5 \cdot p_8 \qquad p_3 = 0.5 \cdot p_6 + 0.5 \cdot p_9 \qquad p_1 + p_2 + p_3 = 1$$

We obtain similar linear constraints by equating the probabilities for process P_2 as well.

Practically, the programmer only need define types T_i with $*$ probability annotations and implement the program in Figure 1(b). The type checker replaces the $*$ annotations with variables, generates linear constraints, and ships them to the LP solver which returns a satisfying assignment, which is then substituted back into the program. For the dice program, we obtain the solution:

$$T_1 \triangleq \oplus_{\mathsf{P}}\{\mathbf{one}^{1/3} : \mathbf{1}, \mathbf{two}^{1/3} : \mathbf{1}, \mathbf{three}^{1/3} : \mathbf{1}\} \qquad T_2 \triangleq \oplus_{\mathsf{P}}\{\mathbf{one}^{2/3} : \mathbf{1}, \mathbf{two}^{1/6} : \mathbf{1}, \mathbf{three}^{1/6} : \mathbf{1}\}$$
$$T_3 \triangleq \oplus_{\mathsf{P}}\{\mathbf{one}^{0} : \mathbf{1}, \mathbf{two}^{1/2} : \mathbf{1}, \mathbf{three}^{1/2} : \mathbf{1}\}$$

As inferred by the LP solver, the probabilities for each label in type T_1 (and therefore, state 1) are equal, thus proving that the program models a fair 3-faced die.

Automated Expected Cost Analysis. A distinguishing feature of NomosPro is the type-guided analysis of the expected cost of distributed protocols and Markov chains. For instance, we would

like to automatically compute a bound on the expected number of coin flips when we initiate the previously defined Markov chain modeling a 3-faced die from state 1 (i.e. process P_1). To perform the expected cost analysis, we integrate probabilistic session types with existing techniques for automatic work analysis with session types [Das et al. 2018], which can be seen as an instantiation of *automatic amortized resource analysis* (AARA) [Hoffmann et al. 2017; Hofmann and Jost 2003]. The core idea is to statically associate a *potential* with each process that is used to pay for the (expected) work performed by it. A key feature is that this potential can also be *transferred to other processes* to cover the work incurred by them, thus allowing *amortization*. Importantly, the amount of potential transferred with a message or associated with a process can be efficiently inferred by linear-constraint solving. This technique is parametric in the cost model and can, for e.g., bound the expected number of spawned processes, messages, flips, or other user-defined quantities.

We build intuition for the type system by revisiting the previously discussed examples. Consider again the processes TF and neg and assume a cost model in which the cost of sending the label **true** is 1 and the cost of sending the label **false** is 2. Since $p = 0.6$, we can derive the typing

$$\cdot \vdash^{1.4} \mathsf{TF} :: (b : \mathsf{bool})$$

where the number on the turnstile reflects the expected cost. Since we incur cost 1 with probability 0.6, and cost 2 with probability 0.4, we deduce that the expected cost of executing TF is $0.6 \times 1 + 0.4 \times 2 = 1.4$. To infer the expected potential q of a probabilistic expression flip p $(\mathsf{H} \Rightarrow P_H \mid \mathsf{T} \Rightarrow P_T)$, we use the second key idea expressed as the equation

$$q = p \cdot q_H + (1 - p) \cdot q_T \ ,$$

where q_H and q_T are the expected potentials for processes P_H and P_T respectively.

Since the expected cost is inextricably connected to the probabilistic behavior of processes, expressing them in the type system allows a compositional and tight bound analysis. For example, we are able to derive the following typing for the neg process (cost of 2 for **false**, and 1 for **true**).

$$(b : \mathsf{pbool}) \vdash^{2 \times 0.6 + 1 \times (1-0.6)} \mathsf{neg} :: (c : \mathsf{npbool})$$

It states that the expected cost of the process is bounded by $2 \times 0.6 + 1 \times (1{-}0.6) = 1.6$. In the typing derivation, it is essential to have access to the distribution of messages on the channel b. If this information is not available then we have to assume the worst case—the label **false** is sent on channel c—and derive the bound 2:

$$(b : \mathsf{bool}) \vdash^{2} \mathsf{neg} :: (c : \mathsf{bool})$$

Thus, probabilistic session types help us infer *precise* expected cost, instead of worst-case cost.

For the 3-faced die program, we are interested in the expected cost of the process P_1. For illustration purposes, it is convenient to consider a cost metric that counts the number of evaluated flips. Then the expected cost of P_1 is $\frac{8}{3}$. To infer this (tight) bound, the type system assigns potential q_i to process P_i. Process P_3 requires only 1 unit of potential, since it performs *only one* flip. Taking the p-weighted sum of the expected cost in both branches, we get

$$q_1 = 1 + 0.5 \cdot q_2 + 0.5 \cdot q_3 = 1.5 + 0.5 \cdot q_2 \qquad q_2 = 1 + 0.5 \cdot 0 + 0.5 \cdot q_1 = 1 + 0.5 \cdot q_1$$

For each equation, the summand 1 accounts for the flip at the start of each process. For q_1, the potential in the H branch is q_2 since we call process P_2 and q_3 in the T branch since we call process P_3. For q_2, the potential in H branch is 0 since it does not involve any flips, and q_1 in the T branch since we call P_1. Solving these equations leads to the solution $q_1 = \frac{8}{3}, q_2 = \frac{7}{3}, q_3 = 1$.

Our implementation (Section 7) automatically generates and solves these linear equations. In Section 6, we show how we can automatically infer expected cost of randomized distributed protocols and derive symbolic bounds that depend on the numbers of processes in the network by employing amortization. For example, NomosPro infers a *linear bound* for the expected number of file chunks transmitted in the Helmink et al. [1994] bounded retransmission protocol. NomosPro

also infers a linear bound on the expected cost of probabilistic insert and delete operations on a concurrent queue even though naïve approaches would produce a quadratic bound. To the best of our knowledge, NomosPro is the only tool that can compute such amortized bounds on the number of interactions for probabilistic concurrent programs.

Section 6 also demonstrates the importance of supporting *both* probabilistic and regular choice types. Symbolic bounds in bounded retransmission [Helmink et al. 1994] were obtained by representing file chunks as a list using *regular* internal choice and an unreliable channel using *probabilistic* external choice. Similarly, concurrent data structures from Section 6.2 and the slot machine benchmark from Section 6.4 require both choice types. In conclusion, symbolic resource analysis requires NomosPro to be a *conservative* extension of resource-aware session types.

Almost-Sure Termination. Recall that in NomosPro, a probabilistic session type describes *upper bounds* on the actual distribution of labels sent on a channel, but if we can prove that a concurrent system is almost-surely terminating, i.e., terminates with probability one, the probabilities in the session types are indeed *precise*. A program implemented in NomosPro can be non-terminating, in general, since it supports recursive process spawning. *Therefore, we can rely on NomosPro to carry out an expected-cost analysis with respect to a cost metric that counts the number of spawned processes, and a finite expected total number of spawned processes implies almost-sure termination.*[2] For example, NomosPro derives the following type for the previously discussed rec_debias process:

$$(b : \mathsf{pbools}) \vdash^{13/12} \mathsf{rec\_debias} :: (c : \mathsf{ubool})$$

Thus although the process involves recursion, NomosPro verifies that the expected total number of spawned process is $\frac{13}{12}$, which is finite and implies almost-sure termination. As we discussed earlier, probabilistic session types enable us to infer precise expected cost, and here we see that the expected-cost analysis enables us infer precise probabilities. Thus the two fragments (probability and resource) are not orthogonal and indeed reciprocal in our design of NomosPro.

Standard and Probabilistic Choices. At first, it may seem redundant to support standard choice operators in NomosPro since we already provide probabilistic choice operators, which might seem more expressive. However, supporting standard choices is crucial for the practicality of NomosPro. Most data structures like lists, stacks, queues, trees, unary and binary natural numbers that are commonly used in concurrent protocols need standard choices and cannot be directly expressed using only probabilistic choice operators. Intuitively, the beauty of the standard choice operators is that they allow arbitrary probability distribution over the labels that are sent. In contrast, probabilistic choice operators only allow a fixed probability over the labels sent.

As an example, recall the coins type from Section 1.

$$\mathsf{coins} \triangleq \oplus\{\mathbf{heads} : \mathsf{coins}, \mathbf{tails} : \mathsf{coins}\}$$

Now, consider the heads – tails process defined as

```
decl heads-tails : . ⊢ (c : coins)
proc c ← heads-tails = c.heads ; c.tails ; c ← heads-tails
```

The process simply alternates between sending **heads** and **tails** on offered channel c. This process would typecheck successfully if c has type coins. However, if c has type fcoins $\triangleq \oplus_{\mathsf{P}}\{\mathbf{heads}^{0.5} : \mathsf{fcoins}, \mathbf{tails}^{0.5} : \mathsf{fcoins}\}$, then the process would not typecheck. The reason is that fcoins imposes a strong restriction on a channel: the probability of sending **heads** or **tails** has to be 0.5 *in each recursive step*. The heads – tails process does not achieve this probability distribution at each step, only over a sequence of steps. Using the following type,

[2]Let random variable T represent total number of spawned processes. Then $\mathbb{E}[T]<\infty$ implies $\mathbb{P}[T<\infty] = 1$ [Ferrer Fioriti and Hermanns 2015], i.e., the event 'total number of spawned processes is finite' happens with probability one. Since non-termination can only be achieved by an infinite number of process spawns, $\mathbb{P}[T<\infty] = 1$ implies almost-sure termination.

$$\mathsf{hcoins} = \oplus_{\mathsf{P}}\{\mathbf{heads}^{1.0} : \mathsf{tcoins}, \mathbf{tails}^{0.0} : \mathsf{tcoins}\}$$
$$\mathsf{tcoins} = \oplus_{\mathsf{P}}\{\mathbf{heads}^{0.0} : \mathsf{hcoins}, \mathbf{tails}^{1.0} : \mathsf{hcoins}\}$$

we can type the process `heads-tails : . ⊢ (c : hcoins)` since the hcoins type indeed alternates between **heads** and **tails**. Thus, even typing simple processes would mean introducing more and more complex types if NomosPro only supported probabilistic choice operators.

This issue becomes even more involved when we introduce traditional data structures into the language. Computing the exact probability distribution of choices in each recursive step can be complicated even with simple data structures. For example, consider the type nat.

$$\mathsf{nat} = \oplus\{\mathbf{zero} : \mathbf{1}, \mathbf{succ} : \mathsf{nat}\}$$

A process two that produces **succ**, **succ**, and **zero** would typecheck with the type nat.

```
decl two : . ⊢ (n : nat)
proc n ← two = n.succ ; n.succ ; n.zero ; close n
```

However, a probabilistic session type that typechecks with the process two would need to produce the first two **succ** messages with probability 1.0, and then the **zero** message with probability 1.0. Such a type called nat2 would be defined as

$$\mathsf{nat2} = \oplus_{\mathsf{P}}\{\mathbf{zero}^{0.0} : \mathbf{1}, \mathbf{succ}^{1.0} : \oplus_{\mathsf{P}}\{\mathbf{zero}^{0.0} : \mathbf{1}, \mathbf{succ}^{1.0} : \oplus_{\mathsf{P}}\{\mathbf{zero}^{1.0} : \mathbf{1}, \mathbf{succ}^{0.0} : \mathbf{1}\}\}\}$$

Defining standard functions like add that would add two numbers would become even more complex since we need to reason about the probability annotations on every **succ** / **zero** message produced. For the same reason, defining standard list functions would also become complicated. Thus, supporting standard choices are critical for practical programming scenarios. In fact, several examples from Section 6 like bounded retransmission, amortized queues and slot machines require using a type called potlist (a list that stores potential) to obtain symbolic bounds which can only be expressed using standard branching operators.

3 BACKGROUND ON RESOURCE-AWARE SESSION TYPES

We begin with the deterministic fragment of NomosPro [Das et al. 2018]. The types and expressions of NomosPro are defined by the grammars in Figure 2 (novel probabilistic extension marked in blue). The symbol ℓ stands for a label (like in a sum type) and the symbols x and y stand for variables, which range over channels. The annotations r and r_ℓ are non-negative rational numbers, and denote potential annotations and probabilities.

The base system of session types is derived from a Curry-Howard interpretation [Caires and Pfenning 2010] of intuitionistic linear logic [Girard and Lafont 1987] which defines the following judgment for typing processes

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash P :: (z : C)$$

This states that process P *provides* a service of session type C along channel z, while *using* the services of session types $A_1, \ldots, A_n$ provided along channels $x_1, \ldots, x_n$, respectively. All these channels must be distinct. We usually abbreviate the antecedent of the sequent by Δ for brevity.

The typing judgment of NomosPro adds a non-negative rational number q (to cover expected evaluation cost) and a fixed global signature Σ containing type and process definitions.

$$\Delta \vdash^{q}_{\Sigma} P :: (x : A)$$

Type definitions have the form $V = A$ and can be (mutually) recursive. We require A to be *contractive* [Gay and Hole 2005] meaning A should not itself be a type name. Our type definitions are *equirecursive* so we can silently replace type names V by A during type checking, and do not need explicit rules for recursive types. Process definitions have the form $f = (\Delta, q, P, x, A)$, where f is the name of the process and P its defining expression, with Δ being the channels used by f and $x : A$ being the offered channel, and q its potential. For a *well-formed signature*, we require that

$$\begin{aligned}\text{Proc}\quad P, Q ::= {} & x.k\ ;\ P \mid \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} \mid \mathsf{send}\ x\ y\ ;\ P \mid y \leftarrow \mathsf{recv}\ x\ ;\ P \mid \mathsf{close}\ x \mid \mathsf{wait}\ x\ ;\ P \\ & \mid x \leftrightarrow y \mid y \leftarrow f\ \overline{x}\ ;\ P \mid \mathsf{get}\ x\ \{r\} \mid \mathsf{pay}\ x\ \{r\} \mid \mathsf{work}\ \{r\}\ ;\ P \\ & \mid \mathsf{flip}\ p\ (\mathsf{H} \Rightarrow P_H \mid \mathsf{T} \Rightarrow P_T) \mid x..k\ ;\ P \mid \mathsf{pcase}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L}\end{aligned}$$

$$\begin{aligned}\text{Type}\quad A, B ::= {} & \oplus\{\ell : A_\ell\}_{\ell \in L} \mid \&\{\ell : A_\ell\}_{\ell \in L} \mid A \otimes B \mid A \multimap B \mid \mathbf{1} \mid V \mid \triangleright^r A \mid \triangleleft^r A \\ & \mid \oplus_{\mathsf{P}}\{\ell^{r_\ell} : A_\ell\}_{\ell \in L} \mid \&_{\mathsf{P}}\{\ell^{r_\ell} : A_\ell\}_{\ell \in L}\end{aligned}$$

Fig. 2. Process expressions and session types in NomosPro.

$$\frac{(k \in L) \quad \Delta \overset{g}{\vdash} P :: (x : A_k)}{\Delta \overset{g}{\vdash} x.k\ ;\ P :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R \qquad \frac{(\forall \ell \in L) \quad \Delta, (x : A_\ell) \overset{g}{\vdash} Q_\ell :: (z : C)}{\Delta, (x : \oplus\{\ell : A_\ell\}_{\ell \in L}) \overset{g}{\vdash} \mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)}\ \oplus L$$

$$\frac{f = (\overline{y' : B}, p, P, x', A) \in \Sigma \qquad r = p + q \qquad \Delta' = \overline{(y : B)} \qquad \Delta, (x : A) \overset{g}{\vdash} Q :: (z : C)}{\Delta, \Delta' \overset{r}{\vdash} (x \leftarrow f\ \overline{y}\ ;\ Q) :: (z : C)}\ \mathsf{spawn}$$

$$\frac{q \geq r \quad \Delta \overset{g-r}{\vdash} P :: (x : A)}{\Delta \overset{g}{\vdash} \mathsf{pay}\ x\ \{r\}\ ;\ P :: (x : \triangleright^r A)}\ \triangleright R \qquad \frac{q \geq r \quad \Delta \overset{g-r}{\vdash} P :: (x : A)}{\Delta \overset{g}{\vdash} \mathsf{work}\ \{r\}\ ;\ P :: (x : A)}\ \mathsf{work} \qquad \frac{\Delta \overset{g+r}{\vdash} P :: (x : A)}{\Delta \overset{g}{\vdash} P :: (x : A)}\ \mathsf{weak}$$

$$\frac{\Delta, (x : A) \overset{g+r}{\vdash} Q :: (z : C)}{\Delta, (x : \triangleright^r A) \overset{g}{\vdash} \mathsf{get}\ x\ \{r\}\ ;\ Q :: (z : C)}\ \triangleright L \qquad \frac{V = A_V \in \Sigma \qquad \Delta \overset{g}{\vdash} P :: (x : A_V)}{\Delta \overset{g}{\vdash} P :: (x : V)}\ \mu R$$

Fig. 3. Selected type rules for resource-aware session types.

$\Delta \overset{q}{\vdash}_\Sigma P :: (x : A)$ for every process definition $f = (\Delta, q, P, x, A)$ in Σ. Like type definitions, process definitions are mutually recursive. Since Σ is fixed, we often elide it from the typing rules.

Basic Session Types. Type constructors in session types ($\oplus$, $\&$, $\mathbf{1}$, $\otimes$, $\multimap$) are derived from assigning an operational interpretation to connectives in intuitionistic linear logic (see Figure 2). We focus on the *choice operators*—since they set up the development of probabilistic typing rules—and provide only a brief description for the remaining operators. Our technical report [Das et al. 2021c] contains the complete set of statics and semantics rules.

The *internal choice* type constructor $\oplus\{\ell : A_\ell\}_{\ell \in L}$ is an n-ary labeled generalization of the additive disjunction $A \oplus B$. Operationally, the provider of $x : \oplus\{\ell : A_\ell\}_{\ell \in L}$ is required to send a label $k \in L$ and then continue to provide A_k. The corresponding process expression is $(x.k\ ;\ P)$ where P is the continuation. Dually, the client must branch based on the label $k \in L$ received from the provider using the expression $(\mathsf{case}\ x\ (\ell \Rightarrow Q)_{\ell \in L})$. The corresponding typing rules are $\oplus R$ and $\oplus L$ in Figure 3. The process potential is unaffected and will be equal in the premise and conclusion for all the basic rules. The *external choice* constructor $\&\{\ell : A_\ell\}_{\ell \in L}$ is the dual of internal choice requiring the provider to branch on one of the labels received from the client. Dual constructors, like this one, reverse the role of the provider and client.

The remaining constructors are directly adopted from the Rast language [Das and Pfenning 2020a,b,c]. Type $\mathbf{1}$ indicates *termination* requiring that the provider send a *close* message followed by terminating the communication. Dually, the client waits for the close message using $(\mathsf{wait}\ x\ ;\ Q)$ and continues to execute Q. A process can also terminate via a *forwarding expression* $(x \leftrightarrow y)$ unifying the channels x and y. The *tensor* operator $A \otimes B$ prescribes that the provider of $x : A \otimes B$ sends a channel y of type A and continues to provide type B. The corresponding process expression is $(\mathsf{send}\ x\ y\ ;\ P)$ where P is the continuation. Correspondingly, its client must receive a channel using the expression $(y \leftarrow \mathsf{recv}\ x\ ;\ Q)$, binding it to variable y and continuing to execute Q. The dual operator $A \multimap B$ allows the provider to receive a channel of type A (sent by its client) and continue to provide type B.

A new instance of a defined process f can be spawned with the expression $x \leftarrow f\ \overline{y}\ ;\ Q$ (the rule spawn in Figure 3). The newly spawned process will use all variables in $\overline{y}$ and provide x to the continuation Q. The potential r of the parent process must be equal to the sum of the potential p of the spawned process and q of the continuation. If a process invocation is a tail call, i.e. invocation without a continuation, it is written as $x \leftarrow f\ \overline{y}$.

Type-level recursion in NomosPro is handled in the same fashion as Rast [Das and Pfenning 2020b] (rule μR in Figure 3). On encountering a type name V, it is silently replaced by its definition A_V. A similar rule μL operates on a channel in the context Δ has been omitted for brevity.

Resource-Aware Types. To describe resource contracts for inter-process communication, the type system further supports amortized resource analysis [Tarjan 1985] in the same way as resource aware session types [Das et al. 2018]. The key idea is that *processes store potential* and *messages carry potential*. This potential can either be consumed to perform *work* or exchanged using special messages. Selected resource-aware type rules are presented in Figure 3.

The type system provides the programmer with the flexibility to specify what constitutes work. We use the expression work $\{r\}\ ;\ P$ to define cost r. The type rule work requires that the potential q is sufficient to pay for the cost r and the remaining potential $q - r\ \geq\ 0$. Our type system can automatically instrument the analyzed program with work $\{r\}$ expressions based on the user-specified cost model (e.g., number of messages sent and number of spawned processes). For example, to count the total number of messages sent, we insert work $\{1\}$ just before sending every message.

Two dual type constructors $\triangleright^r A$ and $\triangleleft^r A$ are used to transfer potential to allow amortization. The provider of $x : \triangleright^r A$ must *pay* r units of potential along x using process expression $(\text{pay}\ x\ \{r\}\ ;\ P)$, and continue to provide A by executing P. These r units are deducted from the potential stored inside the sender process. When sending potential, we ensure that the sender has sufficient potential to pay (premise $q \geq r$). Dually, the client must receive the r units of potential using the expression $(\text{get}\ x\ \{r\}\ ;\ Q)$ and add this to its internal stored potential. This is reflected in the type rules $\triangleright R$ and $\triangleright L$. The dual operator $\triangleleft^r A$ allows the provider to receive potential sent by its client. The complete set of typing rules is skipped for brevity and provided in the technical report [Das et al. 2021c].

The previously-discussed rules treat potential as a linear resource. As a result, the potential reflects the exact cost of programs. However, we are most often interested in upper bounds on the resource usage. For instance, if we treat potential linearly we cannot type a process that has different work cost in different branches. To treat potential in an affine way, we have to provide the ability to throw away potential. This can be achieved by the rule weak in Figure 3.

Shared Session Types. Session types were recently extended with sharing constructors [Balzer and Pfenning 2017] to provide multi-client support imposing an *acquire-release* discipline on shared processes. Although our meta theory and formal soundness theorem have been focused on the linear fragment [Das et al. 2018] of session types, the NomosPro implementation support shared session types which is crucial for some distributed and contract protocols. We believe our results will extend to the shared fragment, and the formal development is future work. Sharing in session types is largely orthogonal to probabilistic behavior since the two novel type formers of NomosPro ($\oplus_{\mathsf{P}}$ and $\&_{\mathsf{P}}$) only exist in the linear fragment with no interaction with the shared fragment.

4 PROBABILISTIC RESOURCE-AWARE SESSION TYPES

We now discuss the static semantics of the novel aspects of NomosPro. Validating whether messages sent on a channel follow the distribution prescribed in the type is an interesting aspect of the type system. Although we could combine probabilistic branching and sending in one atomic approach, we take a more *flexible* approach. We *decouple* the sending of labels from probabilistic branching by altering the types of the channels in each probabilistic branch. This enables us to combine multiple

$$\frac{\Delta = p \cdot \Delta_H +^{\mathsf{L}} (1-p) \cdot \Delta_T \qquad A = p \cdot A_H +^{\mathsf{R}} (1-p) \cdot A_T \qquad q = p \cdot q_H + (1-p) \cdot q_T \qquad \Delta_H \overset{q_H}{\vdash} P_H :: (x : A_H) \qquad \Delta_T \overset{q_T}{\vdash} P_T :: (x : A_T)}{\Delta \overset{q}{\vdash} \mathsf{flip}\ p\ (\mathsf{H} \Rightarrow P_H \mid \mathsf{T} \Rightarrow P_T) :: (x : A)}\ \mathsf{flip}$$

$$\frac{(\forall \ell \in L)\ \ \Delta_\ell, (x : A_\ell) \overset{q_\ell}{\vdash} Q_\ell :: (z : C_\ell) \qquad q = \Sigma_{\ell \in L}\, p_\ell \cdot q_\ell \qquad \Delta = \Sigma^{\mathsf{L}}_{\ell \in L}\, p_\ell \cdot \Delta_\ell \qquad C = \Sigma^{\mathsf{R}}_{\ell \in L}\, p_\ell \cdot C_\ell}{\Delta, (x : \oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}) \overset{q}{\vdash} \mathsf{pcase}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)}\ \oplus_{\mathsf{P}} L$$

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

$$\frac{(\forall \ell \in L)\ \ \Delta_\ell \overset{q_\ell}{\vdash} Q_\ell :: (x : A_\ell) \qquad q = \Sigma_{\ell \in L}\, p_\ell \cdot q_\ell \qquad \Delta = \Sigma^{\mathsf{L}}_{\ell \in L}\, p_\ell \cdot \Delta_\ell}{\Delta \overset{q}{\vdash} \mathsf{pcase}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (x : \&_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L})}\ \&_{\mathsf{P}} R$$

Fig. 4. Type rules for probabilistic resource-aware session types.

sources of randomness (local and from communication) or to use a single source of randomness for multiple sends. The type rules for the novel fragment are given in Figure 4.

Probabilistic Branching. The expression $\mathsf{flip}\ p\ (\mathsf{H} \Rightarrow P_H \mid \mathsf{T} \Rightarrow P_T)$ operationally corresponds to flipping a coin with probability p (of outputting H, and T otherwise) and executing P_H if the coin flips to H and executing P_T otherwise. The corresponding typing rule is flip in Figure 4. This expression together with the deterministic fragment of NomosPro from Section 3 results in a probabilistic session-typed language where the probabilistic split of the types (in blue) can be ignored. As a simplification, consider the following rule simple-flip that is a special case and identical to flip in the deterministically-typed fragment of NomosPro, where both branches P_H and P_T of the probabilistic branching, are typed with the initial context Δ and have to offer on the same channel x of type A.

$$\frac{q = p \cdot q_H + (1-p) \cdot q_T \qquad \Delta \overset{q_H}{\vdash} P_H :: (x : A) \qquad \Delta \overset{q_T}{\vdash} P_T :: (x : A)}{\Delta \overset{q}{\vdash} \mathsf{flip}\ p\ (\mathsf{H} \Rightarrow P_H \mid \mathsf{T} \Rightarrow P_T) :: (x : A)}\ \text{simple-flip}$$

Notably, the probabilistic behavior of a process is not visible in its type. The interesting aspect of rule is the treatment of potential. The initial potential q is in general not identical to q_H and q_T (unlike rule $\oplus L$ or rule $\&R$ in Figure 3). Instead, q is the weighted sum $p \cdot q_H + (1-p) \cdot q_T$ which corresponds to the potential needed to cover the *expected cost* of the probabilistic branch. The rule simple-flip can already be used in conjunction with the deterministic rules to derive interesting and non-trivial bounds on the expected cost.

Probabilistic Choices. Using probabilistic branching, processes can send labels according to a certain probability distribution. To reflect such a distribution in session types, we introduce the type formers $\oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}$ and $\&_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}$ for probabilistic internal and external choice, respectively. The types are similar to their deterministic versions but labels are annotated with probabilities p_ℓ. In a well-formed type, we have $p_\ell \in [0, 1]$ and $\sum_{\ell \in L} p_\ell = 1$. The internal choice $\oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}$ (resp., the external choice $\&_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}$) requires the provider (resp., the client) to send label $k \in L$ with probability p_k.

It is instructive to first discuss the type rule $\oplus_{\mathsf{P}} R$ for probabilistic send. The sender uses expression $(x..k)$ to send label k on channel x. We require that the probability p_k of the label k be 1. For example, in the derivation of the process TF (introduced in Section 2), channel b has type $\oplus_{\mathsf{P}}\{\mathbf{true}^1 : \mathbf{1}, \mathbf{false}^0 : \mathbf{1}\}$ in the H branch of the flip and type $\oplus_{\mathsf{P}}\{\mathbf{true}^0 : \mathbf{1}, \mathbf{false}^1 : \mathbf{1}\}$ in the T branch. To arrive at such a trivial distribution, we need to apply probabilistic branching to alter the probabilities on the

channel. The probability distribution on **true** and **false** labels on channel b implemented by the process is $\mathbb{P}$ where $\mathbb{P}(\textbf{true}) = 0.6$ and $\mathbb{P}(\textbf{false}) = 0.4$.

Receiving on a probabilistic channel can be seen as an external version of a probabilistic branch. We again first consider a simplified version that is a special case of the rule $\oplus_{\mathsf{P}} L$ in Figure 4.

$$\frac{(\forall \ell \in L)\ \ \Delta, (x : A_\ell) \overset{g_\ell}{\vdash} Q_\ell :: (z : C) \qquad q = \Sigma_{\ell \in L}\, p_\ell \cdot q_\ell}{\Delta, (x : \oplus_{\mathsf{P}}\{\ell^{p_\ell} : A_\ell\}_{\ell \in L}) \overset{g}{\vdash} \mathsf{pcase}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)}\ \text{simple-}\oplus_{\mathsf{P}} L$$

The rule simple-$\oplus_{\mathsf{P}} L$ is similar to the rule $\oplus L$ but takes the weighted sum $q = \Sigma_{\ell \in L}\, p_\ell \cdot q_\ell$ as initial potential instead of the maximum; such treatment of potential is similar to the rule flip. The additional premises in the rule $\oplus_{\mathsf{P}} L$ involve *weighted sums of session types*, which are used to enable probabilistic combination of the behavior for subsequent cases Q_ℓ $(\ell \in L)$.

Weighted Sums of Types. To type a probabilistic branch, we need to define a weighted-sum relation for session types. For example, the derivation of the process TF is sound because of the following relation on the type of channel b.

$$\oplus_{\mathsf{P}}\{\textbf{true}^{0.6} : \mathbf{1}, \textbf{false}^{0.4} : \mathbf{1}\} = 0.6 \cdot \oplus_{\mathsf{P}}\{\textbf{true}^{1} : \mathbf{1}, \textbf{false}^{0} : \mathbf{1}\} +^{\mathsf{R}} 0.4 \cdot \oplus_{\mathsf{P}}\{\textbf{true}^{0} : \mathbf{1}, \textbf{false}^{1} : \mathbf{1}\}$$

The types on the right side of the equation are the types of the channel b in the branches of the flip. The probabilities 0.6 and 0.4 are the probabilities of the branches. The operation $p \cdot A +^{\mathsf{R}} (1-p) \cdot B$ combines the label probabilities pointwise. Due to duality, we introduce two weighted-sum relations: $+^{\mathsf{L}}$ and $+^{\mathsf{R}}$, depending on if the type appears on the left or right of the turnstile.

We provide the two interesting cases of the definition. In both cases, we apply the weighted sum $p \cdot q_\ell + (1-p) \cdot r_\ell$ to the probabilities of the outermost labels. For all other possible types, we simply define $p \cdot A +^{\circ} (1-p) \cdot A = A$ for $\circ \in \{\mathsf{L}, \mathsf{R}\}$.

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

We generalize the notion of weighted sums to n-ary sums ($\Sigma^{\mathsf{R}}_{i \in \mathcal{I}}\, p_i \cdot A_i$ and $\Sigma^{\mathsf{L}}_{i \in \mathcal{I}}\, p_i \cdot A_i$) and also extend this relation pointwise to contexts.

Weighted sums are employed in rules $\oplus_{\mathsf{P}} L$, $\&_{\mathsf{P}} R$, and flip, adding a great degree of flexibility to the type system. For example, in the $\oplus_{\mathsf{P}} L$ rule, we allow arbitrary contexts Δ_ℓ and types C_ℓ for each branch. We only require that the weighted sum of the types in each branch matches the context Δ and type C (third and fourth premise). The potential also follows this weighted sum relation. To account for expected costs, the potential of a probabilistic branch is the sum of the potential in each branch weighted with its corresponding probability (instead of the maximum in the rule $\oplus L$ in Figure 3). Similarly, in the flip rule, we allow arbitrary contexts Δ_H and Δ_T and succedent types A_H and A_T in the H and T branches, respectively. Then we apply the weighted sum relation for the context (first premise), the succedent type (second premise), and the potential (third premise).

Importantly, note that the weighted sum relation does not uniquely determine the types of channels in each branch. More formally, given p and A and the constraint $A = p \cdot B +^{\circ} (1-p) \cdot C$, we cannot compute B and C deterministically. Instead, our novel type-reconstruction algorithm (Section 7) uses a bottom-up approach to resolve this non-determinism.

The Necessity of Non-Recursive Weighted Sums. At first, one might be tempted to recursively extend the weighted sum relation to the continuation types, e.g.,

(The rule is unsound) $p \cdot \oplus\{\ell{:}A_\ell\}_{\ell \in L} +^{\mathsf{R}} (1{-}p) \cdot \oplus\{\ell{:}B_\ell\}_{\ell \in L} = \oplus\{\ell{:}(p \cdot A_\ell +^{\mathsf{R}} (1{-}p) \cdot B_\ell)\}_{\ell \in L}$

However, a closer examination reveals that this generalization is generally *not compatible* with the intended semantics of probabilistic choice types. Distributions at deeper level of the types (e.g., A_ℓ, B_ℓ) should not be altered in different branches. Intuitively, the interaction of probabilistic choices with standard choices renders such a generalization unsound.

To demonstrate the issue, consider the following type.

$$B \triangleq \oplus\{\mathbf{true} : \oplus_{\mathsf{P}}\{\mathbf{H}^{0.5} : \mathbf{1}, \mathbf{T}^{0.5} : \mathbf{1}\}, \mathbf{false} : \oplus_{\mathsf{P}}\{\mathbf{H}^{0.5} : \mathbf{1}, \mathbf{T}^{0.5} : \mathbf{1}\}\}$$

A process that offers on a channel of type B should first send a Boolean and then provide a fair coin flip. However, if we would allow a nested weighted sum of probabilities, e.g.,

(The weighted sum is unsound)

$$B = 0.5 \cdot \oplus\{\mathbf{true} : \oplus_{\mathsf{P}}\{\mathbf{H}^{1} : \mathbf{1}, \mathbf{T}^{0} : \mathbf{1}\}, \mathbf{false} : \oplus_{\mathsf{P}}\{\mathbf{H}^{1} : \mathbf{1}, \mathbf{T}^{0} : \mathbf{1}\}\}$$
$$+^{\mathsf{R}}\ 0.5 \cdot \oplus\{\mathbf{true} : \oplus_{\mathsf{P}}\{\mathbf{H}^{0} : \mathbf{1}, \mathbf{T}^{1} : \mathbf{1}\}, \mathbf{false} : \oplus_{\mathsf{P}}\{\mathbf{H}^{0} : \mathbf{1}, \mathbf{T}^{1} : \mathbf{1}\}\}$$

the following undesirable implementation would type check. The problem is that a process that uses channel b does not receive a fair distribution after receiving the Boolean label.

```
decl bad : . ⊢ (b : B)
proc b ← bad = flip 0.5 (H ⇒ b.true ; b..H; close b | T ⇒ b.false ; b..T; close b)
```

The Necessity of Dual Weighted Sums. Our type system has *two* dual weighted sum relations $+^{\mathsf{L}}$ and $+^{\mathsf{R}}$. One might be tempted to think the two relations are equivalent, and allow rules like

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

However, changing the probabilities on internal choices which are provided by other processes would lead to unsound behavior. Consider for example the following process bad′. It offers a channel $x : \oplus_{\mathsf{P}}\{\mathbf{H}^{0.5} : \mathbf{1}, \mathbf{T}^{0.5} : \mathbf{1}\}$ with a fair probabilistic internal choice. However, we could justify the following unsound typing if we allowed the type $\oplus_{\mathsf{P}}\{\mathbf{H}^{0.5} : \mathbf{1}, \mathbf{T}^{0.5} : \mathbf{1}\}$ of y to be split as $0.5 \cdot \oplus_{\mathsf{P}}\{\mathbf{H}^{1} : \mathbf{1}, \mathbf{T}^{0} : \mathbf{1}\} +^{\mathsf{L}} 0.5 \cdot \oplus_{\mathsf{P}}\{\mathbf{H}^{0} : \mathbf{1}, \mathbf{T}^{1} : \mathbf{1}\}$ in the two branches of the flip; in this case, the type of channel x says the process bad′ always sends $\mathbf{H}$, which is not correct.

```
decl bad' : (y : ⊕P{H^0.5 : 1, T^0.5 : 1}) ⊢ (x : ⊕P{H^1 : 1, T^0 : 1})
proc x ← bad' y = flip 0.5 ( H ⇒ pcase y ( H ⇒ x..H; x ↔ y | T ⇒ x..T; x ↔ y )
                           | T ⇒ pcase y ( H ⇒ x..T; x ↔ y | T ⇒ x..H; x ↔ y ) )
```

Probability-Polymorphic Typing. NomosPro only supports constant probabilities. This is not a severe limitation for implementing probabilistic systems because one can implement a program that simulates other distributions, which can possibly rely on runtime values. A non-constant probability distribution of labels can then be type-checked with respect to a standard choice type. Support for non-constant probabilities in the type system is beyond the scope of this article; nevertheless, NomosPro's type system allows *probability-polymorphic typing*. In the type judgment $\Delta \overset{q}{\vdash}_{\Sigma} P :: (x : A)$, the process signature Σ can include *multiple* possible process typings (instead of a single one) for a process definition. For example, the neg process defined in Section 2 can be assigned a set of typing judgments such that one can spawn neg with an arbitrary distribution on Booleans as input:

$$\{(b : \oplus_{\mathsf{P}}\{\mathbf{true}^{p} : \mathbf{1}, \mathbf{false}^{1-p} : \mathbf{1}\}) \overset{0}{\vdash} \mathsf{neg} :: (c : \oplus_{\mathsf{P}}\{\mathbf{true}^{1-p} : \mathbf{1}, \mathbf{false}^{p} : \mathbf{1}\}) \mid p \in [0,1]\}\,,$$

Note that efficient type checking as well as our methods for inference of expected-cost bounds and probabilities do not extend to probability-polymorphic typing. Such an extension is beyond the scope of this article but worth future research.

5 META THEORY

In this section, we introduce the novel probabilistic *nested-multiverse* semantics of NomosPro. Our main results are *type preservation*, *global progress*, *probability consistency*, and that NomosPro is a *conservative extension* of resource-aware session types [Das et al. 2018].

Difficulty in Using Existing Semantics. Operationally, a state of a concurrent system is a *configuration* of running processes in the system. One common semantic construction for probabilities is based on *distributions* (e.g., [Borgström et al. 2016; Kozen 1981]); in our setting, evaluation rules of the semantics should transition from configurations to distributions on configurations. The

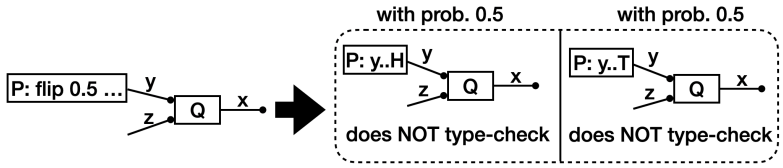


Fig. 5. A standard distribution-based semantics would fail to prove type preservation.

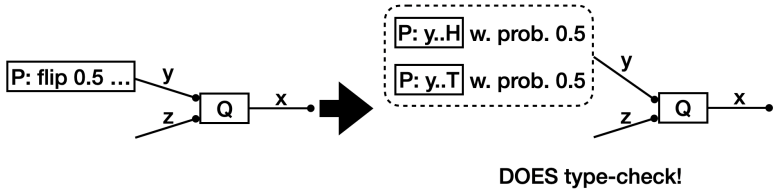


Fig. 6. The nested-multiverse semantics of a probabilistic flip.

distribution-based semantics has been used in the meta theory of a session-type concurrent system where choices can *only* be probabilistic [Inverso et al. 2020]. The idea behind that work is that proving type preservation for a probabilistic-flip expression requires to first construct new type derivations for all possible configurations after the probabilistic flip, and then combine the types via a "weighted sum" with respect to the result distribution on configurations. This approach is *global*, in the sense that after a *local* coin flip in a process, one has to re-type the *whole* configuration with other unchanged processes. However, this approach would fail already in proving preservation if one attempts to integrate standard choices ($\oplus$ and $\&$) with probabilistic choices ($\oplus_P$ and $\&_P$). Our technical report [Das et al. 2021c] includes a counterexample and Figure 5 demonstrates the difficulty: after a local flip in the process P, the type of channel y in one subsequent universe diverges from the original type of channel y, thus neither universes are guaranteed to be well-typed.

A Nested-Multiverse Semantics. To overcome the difficulty, our key innovation is to make *probabilistic branching as local as possible*. The influence of a coin flip should remain within the source process, *until* the process communicates with other processes. With this intuition, we devise *nested-multiverse* semantic objects $\mathcal{O}$, of the form (i) $\mathrm{proc}(c, w, P)$ for a process P that provides along channel c and has performed work w, or (ii) $\mathrm{proc}(c, \{\mathcal{C}_i : p_i\}_{i\in\mathcal{I}})$, for a distribution—resulted from flip expressions—of local configurations $\mathcal{C}_i$'s that are provided along channel c, where $\Sigma_{i\in\mathcal{I}}\, p_i = 1$. A *configuration* $\mathcal{C}$ is a sequence of semantic objects $\mathcal{O}_1 \parallel \cdots \parallel \mathcal{O}_n$. Intuitively, nested objects collect information from multiple *universes*, where each universe represents a possible outcome of executed probabilistic flips.

We develop a small-step operational semantics that manipulates the nested-multiverse objects. Figure 7 presents selected rules for the semantics, and all rules are presented in the technical report [Das et al. 2021c]. We distinguish two kinds of operational judgments, which we denote by $\mathcal{C} \mapsto \mathcal{C}'$ and $\mathcal{C} \overset{d,\kappa}{\Longmapsto} \mathcal{C}'$, for single-process evaluation and communication, respectively. The *single-process* judgment $\mathcal{C} \mapsto \mathcal{C}'$ means that a process in $\mathcal{C}$ makes a step *without* communicating with other processes resulting in $\mathcal{C}'$. To evaluate a coin flip $\mathsf{flip}\ p\ (\mathsf{H} \Rightarrow P_H \mid \mathsf{T} \Rightarrow P_T)$ while keeping the randomness local in the flipped process, the rule (E:Flip) creates a local distribution object, whose support contains two process objects: P_H with probability p and P_T with probability $1 - p$. Figure 6 illustrates the semantics of a coin flip in our system for the system shown in Figure 5. If the flipped process (P_H or P_T) can further evaluate probabilistic-flip expressions, the configuration will become nested naturally. We will later prove *type preservation* that ensures the nested configuration on

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

$$\text{(E:Work)}\quad \mathsf{proc}(c, w, \mathsf{work}\ \{r\}\ ;\ P) \mapsto \mathsf{proc}(c, w + r, P)$$

$$\text{(C:}\oplus_{\mathsf{P}}\text{)}\quad \mathsf{proc}(c, w_c, \mathsf{pcase}\ d\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \parallel \mathsf{proc}(d, w_d, d..k\ ;\ P) \overset{d,\oplus_{\mathsf{P}}}{\Longrightarrow} \mathsf{proc}(c, w_c, Q_k) \parallel \mathsf{proc}(d, w_d, P)$$

$$\text{(C:}\&_{\mathsf{P}}\text{)}\quad \mathsf{proc}(c, w_c, d..k\ ;\ Q) \parallel \mathsf{proc}(d, w_d, \mathsf{pcase}\ d\ (\ell \Rightarrow P_\ell)_{\ell \in L}) \overset{d,\&_{\mathsf{P}}}{\Longrightarrow} \mathsf{proc}(c, w_c, Q) \parallel \mathsf{proc}(d, w_d, P_k)$$

$$\text{(C:}\oplus\text{)}\quad \mathsf{proc}(c, w_c, \mathsf{case}\ d\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \parallel \mathsf{proc}(d, w_d, d.k\ ;\ P) \overset{d,\mathsf{det}}{\Longrightarrow} \mathsf{proc}(c, w_c, Q_k) \parallel \mathsf{proc}(d, w_d, P)$$

$$\text{(C:}\&\text{)}\quad \mathsf{proc}(c, w_c, d.k\ ;\ Q) \parallel \mathsf{proc}(d, w_d, \mathsf{case}\ d\ (\ell \Rightarrow P_\ell)_{\ell \in L}) \overset{d,\mathsf{det}}{\Longrightarrow} \mathsf{proc}(c, w_c, Q) \parallel \mathsf{proc}(d, w_d, P_k)$$

$$\frac{\mathsf{proc}(c, \{(\mathcal{C}_i \parallel \mathsf{proc}(d, \{\mathcal{C}'_j : p'_j\}_{j \in \mathcal{J}})) : p_i\}_{i \in \mathcal{I}}) \overset{d,\oplus_{\mathsf{P}}}{\Longrightarrow} \mathcal{C}''}{\mathsf{proc}(c, \{\mathcal{C}_i : p_i\}_{i \in \mathcal{I}}) \parallel \mathsf{proc}(d, \{\mathcal{C}'_j : p'_j\}_{j \in \mathcal{J}}) \overset{d,\oplus_{\mathsf{P}}}{\Longrightarrow} \mathcal{C}''}\ \text{(C:BDist:L)}$$

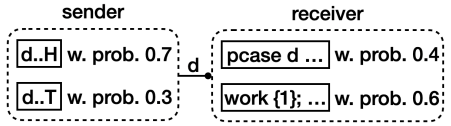
$$\frac{\mathsf{proc}(c, \{(\mathsf{proc}(c, \{\mathcal{C}_i : p_i\}_{i \in \mathcal{I}}) \parallel \mathcal{C}'_j) : p'_j\}_{j \in \mathcal{J}}) \overset{d,\&_{\mathsf{P}}}{\Longrightarrow} \mathcal{C}''}{\mathsf{proc}(c, \{\mathcal{C}_i : p_i\}_{i \in \mathcal{I}}) \parallel \mathsf{proc}(d, \{\mathcal{C}'_j : p'_j\}_{j \in \mathcal{J}}) \overset{d,\&_{\mathsf{P}}}{\Longrightarrow} \mathcal{C}''}\ \text{(C:BDist:R)}$$

$$\frac{\mathsf{proc}(c, \{(\mathcal{C}_i \parallel \mathcal{C}'_j) : p_i \cdot p'_j\}_{i \in \mathcal{I}, j \in \mathcal{J}}) \overset{d,\mathsf{det}}{\Longrightarrow} \mathcal{C}''}{\mathsf{proc}(c, \{\mathcal{C}_i : p_i\}_{i \in \mathcal{I}}) \parallel \mathsf{proc}(d, \{\mathcal{C}'_j : p'_j\}_{j \in \mathcal{J}}) \overset{d,\mathsf{det}}{\Longrightarrow} \mathcal{C}''}\ \text{(C:BDist:D)}$$

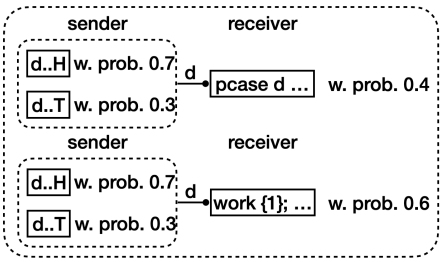
Fig. 7. Selected rules for the nested-multiverse semantics of NomosPro.

the right of the arrow in Figure 6 is well-typed. In the rule (E:Work) for work tracking, we simply increment the work counter of the process.

The *communication* judgment $\mathcal{C} \overset{d,\kappa}{\Longrightarrow} \mathcal{C}'$ means that two processes in $\mathcal{C}$ communicate on a channel d with sort κ. The *sort* $\kappa \in \{\mathsf{det}, \oplus_{\mathsf{P}}, \&_{\mathsf{P}}\}$ categorizes the communication on a channel: $\oplus_{\mathsf{P}}$ and $\&_{\mathsf{P}}$ stand for probabilistic internal choices (rule (C:$\oplus_{\mathsf{P}}$)) and external choices (rule (C:$\&_{\mathsf{P}}$)), whereas det represents all other communication (rules (C:$\oplus$) and (C:$\&$)). The communication is *synchronous*: the sender blocks until the message is delivered to the receiver. The sorts come into play when two communicating semantic objects are distribution objects. For example, in the rule (C:BDist:L), the underlying communication is of sort $\oplus_{\mathsf{P}}$, i.e., $\mathsf{proc}(d, \{\mathcal{C}'_j : p'_j\}_{j \in \mathcal{J}})$ sends probabilistically on channel d, and $\mathsf{proc}(c, \{\mathcal{C}_i : p_i\}_{i \in \mathcal{I}})$ receives on channel d. To make a step from a pair of distributions, we need to first *decompose* one of the distributions, i.e., extract out some inner configurations. Intuitively, $\mathsf{proc}(d, \{\mathcal{C}'_j : p'_j\}_{j \in \mathcal{J}})$ results from one or more probabilistic flips and describes processes from multiple universes that send labels on channel d. The receivers must group these senders from multiple universes together to obtain the probability distribution on the $\oplus_{\mathsf{P}}$-typed channel. Thus, in the premise, we keep $\mathsf{proc}(d, \{\mathcal{C}'_j : p'_j\}_{j \in \mathcal{J}})$ (sender) intact, but decompose $\mathsf{proc}(c, \{\mathcal{C}_i : p_i\}_{i \in \mathcal{I}})$ (receiver). Figure 8 demonstrates an example of this rule. Dually, the rule (C:BDist:R) works on $\&_{\mathsf{P}}$-sorted communications by decomposing the senders. For det-sorted



(a) (left) $\mathsf{proc}(d, \{\mathcal{C}'_1 : 0.7, \mathcal{C}'_2 : 0.3\})$; (right) $\mathsf{proc}(c, \{\mathcal{C}_1 : 0.4, \mathcal{C}_2 : 0.6\})$.



(b) Decompose the receiver: $\mathsf{proc}(c, \{(\mathcal{C}_1 \parallel \mathsf{proc}(d, \{\mathcal{C}'_1 : 0.7, \mathcal{C}'_2 : 0.3\})) : 0.4, (\mathcal{C}_2 \parallel \mathsf{proc}(d, \{\mathcal{C}'_1 : 0.7, \mathcal{C}'_2 : 0.3\})) : 0.6\})$.

Fig. 8. An example of (C:BDist:L).

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

$$\frac{\Delta \overset{g}{\vdash} P :: (c : A)}{\Delta \overset{q+w}{\Vdash} \mathsf{proc}(c, w, P) :: (c : A)}\ (\text{T:Proc}) \qquad \frac{\Delta_1, \Delta' \overset{q_1}{\Vdash} \mathcal{O} :: (c : A) \quad \Delta_2 \overset{q_2}{\Vdash} \mathcal{C} :: (\Delta, \Delta')}{\Delta_1, \Delta_2 \overset{q_1+q_2}{\Vdash} (\mathcal{O} \parallel \mathcal{C}) :: (\Delta, (c : A))}\ (\text{T:Compose})$$

Fig. 9. Type rules for configurations.

(i.e., non-probabilistic) communications, the rule (C:BDist:D) can decompose both the senders and the receivers, as the type for a non-probabilistic communication is also non-probabilistic.

Configuration Typing. We extend our type system to configurations (rules are given in Figure 9). The judgment $\Delta \overset{q}{\Vdash} \mathcal{C} :: \Gamma$ means that the configuration $\mathcal{C}$ uses the channels in the context Δ and provides the channels in Γ, and the nonnegative number q denotes the *expected* value of the sum of the total potential and work done by the system. The rule (T:Proc) connects configurations with the original NomosPro processes, e.g., if an initial process is given by $\Delta \overset{g}{\vdash} P :: (c : A)$, then an initial configuration is defined by $\mathsf{proc}(c, 0, P)$ with the configuration typing $\Delta \overset{q}{\Vdash} \mathsf{proc}(c, 0, P) :: (c : A)$.

Type Safety. We prove type preservation by induction on evaluation judgments.

Theorem 1 (Preservation). *If $\Delta \overset{q}{\Vdash} \mathcal{C} :: \Gamma$, $\mathcal{C} \mapsto \mathcal{C}'$ or $\mathcal{C} \overset{d,\kappa}{\Longrightarrow} \mathcal{C}'$ for some d, κ, then $\Delta \overset{q}{\Vdash} \mathcal{C}' :: \Gamma$.*

To aid the proof of progress, we define relations to characterize the *status* of semantic objects:

- $\mathcal{C}$ poised means that *all* processes in $\mathcal{C}$ are communicating along their providing channels.
- $\mathcal{C}$ live means that there exists a process in $\mathcal{C}$ that can make a step *without* communication.
- $\mathcal{C}$ (d, κ)-comm means that there exist two processes in $\mathcal{C}$ that are going to communicate along channel d of sort κ.

Formal definitions of the characterizations are included in the technical report [Das et al. 2021c]. We first prove that those characterizations are sufficient for a configuration to make a step in the semantics (Lem. 1(i)). We then prove that a well-typed configuration always admits a suitable characterization (Lem. 1(ii)).

Lemma 1.

(i) If $\mathcal{C}$ live*, then $\mathcal{C} \mapsto \mathcal{C}'$ for some $\mathcal{C}'$. If $(\cdot) \overset{q}{\Vdash} \mathcal{C} :: \Gamma$, $\mathcal{C}$ (d, κ)-*comm*, then $\mathcal{C} \overset{d,\kappa}{\Longrightarrow} \mathcal{C}'$ for some $\mathcal{C}'$.*

(ii) If $(\cdot) \overset{q}{\Vdash} \mathcal{C} :: \Gamma$, then either (a) $\mathcal{C}$ poised*; or (b) $\mathcal{C}$* live*, or $\mathcal{C}$ (d, κ)-*comm *for some d, κ.*

Theorem 2 (Progress). *If $(\cdot) \overset{q}{\Vdash} \mathcal{C} :: \Gamma$, then either (i) $\mathcal{C}$* poised*; or (ii) $\mathcal{C} \mapsto \mathcal{C}'$ for some $\mathcal{C}'$, or $\mathcal{C} \overset{d,\kappa}{\Longrightarrow} \mathcal{C}'$ for some $\mathcal{C}', d, \kappa$.*

We also formulate a result about *probability consistency*. Our approach of showing consistency is analogous to the common approach of proving the consistency of refinement types (e.g., [Knoth et al. 2019]): instead of showing intermediate values satisfy the refinement constraints, people usually prove a lemma for well-typed *closed values*. In our setting, this approach suggests reasoning about poised configurations that do not use any channels. Our technical report [Das et al. 2021c] includes the details.

Theorem 3 (Consistency). *Suppose $(\cdot) \overset{q}{\Vdash} \mathcal{C} :: (c : \oplus_{\mathsf{P}}\{\ell^{p_\ell} : \mathbf{1}\}_{\ell\in L})$. By composing $\mathcal{C}$ with a monitor that collects the messages received on channel c, we show that if $\mathcal{C}$ evaluates to a poised configuration $\overline{\mathcal{C}}'$ then the monitored message distribution on c in $\overline{\mathcal{C}}'$ matches the type $\oplus_{\mathsf{P}}\{\ell^{p_\ell} : \mathbf{1}\}_{\ell\in L}$.*

$$\text{(FL:Proc)}\ \frac{}{\mathsf{proc}(c, w, P) \approx \{\mathsf{proc}(c, w, P) : 1\}} \qquad \text{(FL:Dist)}\ \frac{\forall i \in \mathcal{I} : \mathcal{C}_i \approx \mu_i}{\mathsf{proc}(c, \{\mathcal{C}_i : p_i\}_{i\in\mathcal{I}}) \approx \Sigma_{i\in\mathcal{I}}\, p_i \cdot \mu_i}$$

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

Fig. 10. Rules for the "flattening" procedure.

Connection with Distribution-based Semantics. To justify our design of the nested-multiverse semantics, we apply the standard distribution-based construction to developing a small-step operational semantics on *non-nested* configurations $\mathcal{D}$ of the form $\mathsf{proc}(c_1, w_1, P_1) \parallel \cdots \parallel \mathsf{proc}(c_n, w_n, P_n)$, and then prove that if a nested configuration can make an evaluation step, its corresponding distribution of non-nested configurations can also make a step. The non-nested semantics is based on a synchronous semantics for resource-aware session types [Balzer and Pfenning 2017; Das et al. 2018]. We introduce two kinds of operational judgments, which we denote by $\mathcal{D} \overset{\text{det}}{\longmapsto} \mathcal{D}'$ and $\mathcal{D} \overset{\text{prob}}{\longmapsto} \mu$, for deterministic and probabilistic evaluation, respectively, where μ is a distribution on configurations. Below presents selected rules for the semantics.

$$\begin{array}{ll}
\text{(S:Work)} & \mathsf{proc}(c, w, \mathsf{work}\,\{r\}\ ;\ P) \overset{\text{det}}{\longmapsto} \mathsf{proc}(c, w + r, P) \\
\text{(S:}\oplus\text{P)} & \mathsf{proc}(c, w_c, c..k\ ;\ P), \mathsf{proc}(d, w_d, \mathsf{pcase}\ c\ (\ell \Rightarrow Q_\ell)_{\ell\in L}) \overset{\text{det}}{\longmapsto} \mathsf{proc}(c, w_c, P), \mathsf{proc}(d, w_d, Q_k) \\
\text{(S:}\&\text{P)} & \mathsf{proc}(c, w_c, \mathsf{pcase}\ c\ (\ell \Rightarrow P_\ell)_{\ell\in L}), \mathsf{proc}(d, w_d, c..k\ ;\ Q) \overset{\text{det}}{\longmapsto} \mathsf{proc}(c, w_c, P_k), \mathsf{proc}(d, w_d, Q) \\
\text{(S:}\oplus\text{)} & \mathsf{proc}(c, w_c, c.k\ ;\ P), \mathsf{proc}(d, w_d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell\in L}) \mapsto \mathsf{proc}(c, w_c, P), \mathsf{proc}(d, w_d, Q_k) \\
\text{(S:}\&\text{)} & \mathsf{proc}(c, w_c, \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell\in L}), \mathsf{proc}(d, w_d, c.k\ ;\ Q) \mapsto \mathsf{proc}(c, w_c, P_k), \mathsf{proc}(d, w_d, Q) \\
\text{(SP:Det)} & \mathcal{D} \overset{\text{prob}}{\longmapsto} \{\mathcal{D}' : 1\} \\
\text{(SP:Flip)} & \mathsf{proc}(c, w_c, \mathsf{flip}\ p\ (\mathsf{H} \Rightarrow Q_H \mid \mathsf{T} \Rightarrow Q_T)) \overset{\text{prob}}{\longmapsto} \{\mathsf{proc}(c, w_c, Q_H)) : p, \mathsf{proc}(c, w_c, Q_T) : 1 - p\}
\end{array}$$

We lift the configuration-to-distribution relation $\overset{\text{prob}}{\longmapsto}$ to a distribution-to-distribution relation $\Mapsto$ in a standard way. Details of the distribution-based non-nested semantics are included in the technical report [Das et al. 2021c].

We then develop a "flattening" procedure that translates a nested configuration to a distribution of non-nested configurations. We formulate the translation as a *simulation* relation $\mathcal{C} \approx \mu$ and Figure 10 presents rules for the relation, where the weighted sum of distributions $\Sigma_{i\in\mathcal{I}}\, p_i \cdot \mu_i$ is defined as $\lambda\mathcal{D}.\ \Sigma_{i\in\mathcal{I}}(p_i \cdot \mu_i(\mathcal{D}))$. We can now formulate the connection between our nested-multiverse semantics and the standard distribution-based semantics as follows.

Theorem 4. *If $\mathcal{C} \approx \mu$, $\mathcal{C}' \approx \mu'$, and $\mathcal{C} \mapsto \mathcal{C}'$ or $\mathcal{C} \overset{d,\kappa}{\Longmapsto} \mathcal{C}'$, then $\mu \Mapsto \mu'$.*

Expected Work Analysis. To reason about the *expected* amount of work done by a probabilistic program, we construct a Markov chain that reflects NomosPro's non-nested distribution-based semantics, which, as we have shown above, simulates the original nested-multiverse semantics. Thus, the states of the Markov chain are the non-nested configurations. We define the expected total work with respect to the Markov chain and prove that our system derives a sound *upper* bound on the expected total work. We denote the total work done by a non-nested configuration $\mathcal{D} = \overline{\mathsf{proc}(c_i, w_i, P_i)}$ by $\mathsf{work}(\mathcal{D}) \coloneqq \Sigma_i\, w_i$. Details of the expected work analysis are included in the technical report [Das et al. 2021c].

Theorem 5. *Suppose that* $(\cdot) \Vdash^{q} \mathcal{C} :: \Gamma$. *Then we can construct a Markov chain* $\{\mathcal{D}_n\}_{n\in\mathbb{N}}$ *that reflects the computation with* $\mathcal{C}$ *as the initial configuration. The expected total work is defined as* $\mathsf{etw}(\mathcal{C}) \coloneqq \lim_{n\to\infty} \mathbb{E}[\mathsf{work}(\mathcal{D}_n)]$. *Then* $\mathsf{etw}(\mathcal{C}) \leq q$.

6 APPLICATIONS OF NOMOSPRO

We demonstrate applications from four categories: *(i)* implementating randomized distributed algorithms along with inferring linear bound for most benchmarks, *(ii)* computing expected operational cost of amortized data structures, *(iii)* verifying limiting distributions of Markov chains, and *(iv)* analyzing expected behavior of digital contracts. Further benchmarks are described in Section 7.

6.1 Randomized Distributed Algorithms

Synchronous Leader Election. A leader election protocol operates on a network of N processes that communicate with each other to designate a *unique* process among them as the leader. Itai and Rodeh [1990] proved that if the processes are indistinguishable, then there exists no *deterministic* protocol to elect a leader. They also proposed a randomized protocol for leader election in a *ring network* which proceeds in rounds. In each round, each process (independently) chooses a random number in the range $\{1, \ldots, K\}$ for some parameter K as an *id*. The processes then pass their ids around the ring. If there is a *unique maximum id*, then that process is elected as the leader. Otherwise, the processes initiate a new round. This protocol terminates with probability one because eventually the random numbers chosen by the processes will have a unique maximum.

We intuitively describe how NomosPro models the two main aspects of the protocol: choosing the process id, and then determining the leader. To model ids, we introduce the following session type (assume $K = 3$):

$$\mathsf{ids} \triangleq \oplus_{\mathsf{P}}\{\mathbf{one}^{1/3} : \mathbf{1}, \mathbf{two}^{1/3} : \mathbf{1}, \mathbf{three}^{1/3} : \mathbf{1}\}$$

This type produces one of the labels **one**, **two**, and **three** representing the three possible ids. Next, the following two processes are used to choose an id and determine the maximum id (for $N = 4$):

```
decl choose-id : . ⊢ (id : ids)
decl maximum : (id1 : ids) (id2 : ids) (id3 : ids) (id4 : ids) ⊢ (max : ids)
```

Each process calls the choose − id process, which produces one of the three labels with equal probability. Next, the maximum process case analyzes on the id value of each process; if a unique maximum is determined, it is returned, otherwise a new round is initiated.

We have implemented this protocol in NomosPro which automatically infers its expected cost. In this protocol, we have used a special cost model that counts the number of rounds. Figure 11(a) plots the expected number of rounds versus K. We have implemented this protocol for 2 ring networks with 3 and 4 processes, respectively. Intuitively, the expected number of rounds is directly proportional to N and inversely proportional to K. Both these observations are confirmed by Figure 11(a): for a given K, the number of rounds for $N = 4$ is greater than that for $N = 3$; and for a fixed N, the expected rounds decreases as K increases.

Bounded Retransmission Protocol. This protocol is used for delivery of a file over an unreliable channel [Helmink et al. 1994]. The sender splits the file into several small chunks and sends them over one-by-one in sequence. Upon getting a chunk, the receiver sends an acknowledgment with the chunk id. Until the sender receives this acknowledgment, they retransmit the same chunk up to a given bounded number of times (hence, the name bounded retransmission).

NomosPro automatically computes the expected number of chunks that need to be transmitted to ensure reliable delivery. We model this protocol for an unreliable channel with a 20% probability of dropping a chunk. Figure 11(b) plots the expected number of transmitted chunks for different numbers of allowed retries for $N = 4, 5$ where N is the number of file chunks to be transmitted. We

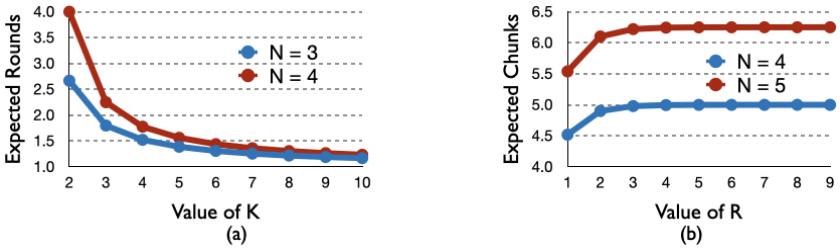


Fig. 11. (a) Expected number of rounds in leader election of Itai and Rodeh [1990] vs K, and (b) Expected number of chunks transmitted in Bounded Retransmission [Helmink et al. 1994] vs R (# retries).

observe that the expected value increases as the number of retries increases. Moreover, a standard result of this protocol is that the expected number of chunks converges to $(\frac{1}{1-p})N$ where p is the drop probability and N is the number of chunks. This is indeed confirmed by NomosPro for the case when $N \in \{4, 5\}$, since the $N = 4$ line converges at 5 and $N = 5$ line converges at 6.25 respectively.

NomosPro can also derive a symbolic upper bound by representing the file chunks as a list of arbitrary length and the retransmission bound with an argument R. To obtain a symbolic bound, we employ amortization by introducing a list type defined as

$$\text{potlist} \triangleq \oplus\{\mathbf{cons} : \triangleright^{*}\text{potlist}, \mathbf{nil} : \mathbf{1}\}$$

Each element of the list represents a file chunk and stores a fixed but unknown amount of potential. In each protocol iteration, an element is removed from potlist and transmitted. If the transmission is successful, we move to the next chunk, otherwise, we insert the chunk back into potlist and try again up to R times. For $p = 0.2$, NomosPro infers the value of 1.25 for the $*$ in the potlist type, thereby deriving the symbolic bound $1.25N$, which is tight since R is unbounded.

Byzantine Broadcast Protocols. Broadcast protocols like Srikanth and Toueg [1987] authenticated broadcast and Bracha [1987] reliable broadcast are commonly used to reach consensus among a leader and a set of followers, where a given number of parties are subject to Byzantine faults. They involve the leader broadcasting a message m to all its followers which are then echoed amongst each other by the followers. Once a follower receives a threshold number of echoes, it commits to m. Both these protocols have numerous applications in cryptography by ensuring secure broadcast even in the presence of failures. We implemented probabilistic variants of both protocols in NomosPro where messages are dropped with a fixed probability. We introduce the following optional message type omsg.

$$\text{omsg} \triangleq \oplus_{\mathrm{P}}\{\mathbf{none}^{p} : \mathbf{1}, \mathbf{some}^{1-p} : \text{msg}\}$$

The optional message produces none (equivalent to dropping) with probability p. We use this as the message type for all communication among all parties for both protocols. The two protocols amount to 397 and 732 lines of code, respectively. Owing to its bi-directional nature, our type checker is quite efficient in practice and checks the two programs in 1.03ms and 1.78ms respectively. We report more details in Table 1. We leave inferring cost bounds for these protocols as future work since the NomosPro implementation currently only support linear bounds while the message complexity for both these broadcast protocols is quadratic.

6.2 Concurrent Amortized Data Structures

Amortized Queues. A classical example of amortization is a concurrent queue implemented using two lists: an *inlist* and an *outlist*. Elements are inserted into the inlist and deleted from the outlist. For deletion, if the outlist is empty, all elements from the inlist are removed and inserted

into the outlist in reverse order. Though insertion is constant-time, deletion can be linear-time in the worst case.

Notably, NomosPro can automatically infer that deletion has a *constant amortized expected cost*. We illustrate this by considering a probabilistic variant of a queue with the type

$$\mathsf{queue}_A \triangleq \&_{\mathrm{P}}\{\mathbf{ins}^{p} : \triangleleft^{*}(A \multimap \mathsf{queue}_A), \mathbf{del}^{1-p} : \oplus\{\mathbf{none} : \mathbf{1}, \mathbf{some} : A \otimes \mathsf{queue}_A\}\}$$

A queue offers a probabilistic choice (indicated by $\&_{\mathrm{P}}$) of either receiving an **ins** or **del** request with probabilities p and $1-p$ respectively. In the case of insert, the queue receives some fixed potential (denoted by $\triangleleft^{*}$) and a channel of type A (denoted by $\multimap$) to insert into the queue. In the latter case, the queue has a choice (indicated by $\oplus$) of either responding with the label **none** (if there is no element in the queue) and closing the channel (indicated by **1**), or the label some followed by an element of type A (denoted by $\otimes$) and recursing to await the next round of interactions.

To obtain the constant amortized bound, we allow the client to pay extra potential during insertion using the $\triangleleft^{*}$ operator which is stored in the inlist. During deletion, this extra potential is used to pay for the cost of reversal, thereby amortizing the linear cost.

Remarkably, the amortized cost of insertion is inversely proportional to the probability p of insert requests. As the expensive delete operations become more frequent, the cost of insertion rises. This is indeed validated by NomosPro. When we count all message exchanges: if $p = 0.1$, the cost is 58; if $p = 0.3$, the cost is 18; if $p = 0.8$, the cost is 5.5. To obtain a symbolic bound as a function of the number of operations, we employ amortization by relying on the potlist type introduced earlier. For each element of this list, a client flips a coin to decide whether to send an insert or delete request. The $*$ on the $\triangleright$ operator signals the inference engine to compute the expected potential.

Even more remarkably, the potential inferred by NomosPro for potlist is $\triangleright^{7}$, irrespective of the probability p! This is due to the fact that as p increases, the cost of insertion reduces but the total number of insert requests also increases, canceling out each other's effect. Thus, the expected cost for n operations on the concurrent queue is $7n$ independent of the value of p, which is indeed confirmed by NomosPro.

Unreliable Data Structures. Another scenario for expected amortized cost analysis is to reason about *unreliable* server processes that provide data-structure manipulation services. For example, suppose that a server process implements a non-probabilistic version of the concurrent queue described earlier in the section, but the server will fail to respond to a connection with some probability $p < 1$. Therefore, the client has to resend a request repeatedly until the server accepts the connection. A non-probabilistic amortized cost analysis would yield infinity, because it is possible for the server to refuse an unbound number of connections. On the other hand, a non-amortized expected cost analysis would indicate that a deletion can be linear-time, because the queue is implemented using an inlist and an outlist.

NomosPro can automatically infer that each operation has a *constant amortized expected cost*, and the *cost depends on the failure probability p*. The unreliable queue is implemented with the type

$$\mathsf{uqueue}_A \triangleq \oplus_{\mathrm{P}}\{\mathbf{fail}^{p} : \mathsf{uqueue}_A, \mathbf{succ}^{1-p} : \mathsf{queue}_A\}$$
$$\mathsf{queue}_A \triangleq \&\{\mathbf{ins} : \triangleleft^{*}(A \multimap \mathsf{uqueue}_A), \mathbf{del} : \triangleleft^{*} \oplus \{\mathbf{none} : \mathbf{1}, \mathbf{some} : A \otimes \mathsf{uqueue}_A\}\}$$

The actual non-probabilistic queue has type queue_A, and is wrapped in a probabilistic interface (the type uqueue_A) to model unreliability. To complete a request on the actual queue, we implement a recursive try-until-success process as follows.

```
decl try_request : (uq : uqueue) ⊢* (q : queue)
proc q ← try_request uq =
  work {1} ; pcase uq ( fail ⇒ q ← try_request uq | succ ⇒ q ↔ uq )
```

In essence, the process try_request simulates a geometric distribution whose success rate is $1 - p$; thus, the exact expected cost in terms of the failure probability p is proportional to $\frac{1}{1-p}$. Indeed, when we use a cost model specified by work expressions, NomosPro automatically derives that if $p = 0.8$, the expected cost of try_request is 5; and if $p = 0.2$, the expected cost is 1.25.

Similar to the analysis described earlier in the section, we can obtain a symbolic bound as a function of the number of operations, by introducing a list type of requests defined as

$$\mathsf{reqlist}_A \triangleq \oplus\{\mathbf{ins} : \triangleright^* A \otimes \mathsf{reqlist}_A, \mathbf{del} : \triangleright^* \mathsf{reqlist}_A, \mathbf{nil} : \mathbf{1}\}$$

The list stores a fixed but unknown amount of potential per request, and the $*$'s on the $\triangleright$ operators signal the type checker to compute the expected amortized costs. When we count all message exchanges, NomosPro automatically infers that if $p = 0.8$, the potential annotations for **ins** and **del** in type $\mathsf{reqlist}_A$ are $\triangleright^{11}$ and $\triangleright^{12}$, respectively; and if $p = 0.2$, the annotations are $\triangleright^{7.25}$ and $\triangleright^{8.25}$. Therefore, the expected cost for n operations on the unreliable concurrent queue is linear in n in both cases, and the expected cost gets larger when the failure probability p gets larger.

6.3 Markov Chains

Limiting Distributions of Markov Chains. One of the most common problems studied about Markov chains is their *asymptotic behavior*. Google's PageRank algorithm [Page et al. 1999], for instance, represents webpages as states in a giant Markov chain and utilizes its limiting distribution to rank the webpages in search results. The *limiting distribution* of a Markov chain M is a vector $\overline{\pi}$ where π_i denotes the fraction of time spent in state i. A chain is often described using a transition probability matrix $\mathbf{P}$ such that $\mathbf{P}_{ij}$ denotes the probability of transitioning from state i to state j. Then, the limiting distribution satisfies the equations $\overline{\pi} \cdot \mathbf{P} = \overline{\pi}$ and $\sum_i \pi_i = 1$.

We can implement a Markov chain in NomosPro and use the type system to verify its limiting distribution. A standard example of limiting distributions is the position of a king wandering in an otherwise empty 8-by-8 chessboard. At any step, the king can move from its current position to any of its neighboring squares with equal probability. It is known that the limiting positional distribution of this king is $\frac{3}{420}$ for a corner square, $\frac{5}{420}$ for an edge square, and $\frac{8}{420}$ for an internal square. To verify this distribution in NomosPro, we define a type limit abbreviated as follows

$$\mathsf{limit} \triangleq \oplus_{\mathsf{P}}\{\ldots, \mathbf{S_{ab}}^{3/420} : \mathbf{1}, \ldots, \mathbf{S_{cd}}^{5/420} : \mathbf{1}, \ldots, \mathbf{S_{ef}}^{8/420} : \mathbf{1}, \ldots\}$$

The type contains 64 branches, one for each square on the chessboard. Labels $\mathbf{S_{ab}}$, $\mathbf{S_{cd}}$, and $\mathbf{S_{ef}}$ denote one representative corner, edge, and internal square respectively. The probability annotation of each label exactly matches the limiting probability of the corresponding square. The transition probability matrix is represented using the following process

```
decl transition : (in : limit) ⊢ (out : limit)
proc out ← transition in = pcase in (
    S11 ⇒ u ← uniform3 ; pcase u ( one ⇒ out..S12 ; wait u ; out ↔ in
                                 | two ⇒ out..S21 ; wait u ; out ↔ in
                                 | three ⇒ out..S22 ; wait u ; out ↔ in) ... )
```

This transition process exactly implements the equation $\overline{\pi} \cdot \mathbf{P} = \overline{\pi}$. We showcase one such branch in the process (which exceeds 600 lines!) representing the corner square (1, 1). We call a process `uniform3` (not shown) that generates a uniform distribution with three possibilities: **one**, **two**, and **three**. Depending on the output of u, we transition to neigboring squares (1, 2), (2, 1), or (2, 2) respectively, guaranteeing that each neighboring transition is equally probable. The validity of type limit enforces that $\sum_i \pi_i = 1$. The successful type checking of transition verifies that limit indeed represents the limiting positional distribution of the king's position.

6.4 Digital Contracts

NomosPro can be used to inject probabilistic behavior into digital contracts [Das et al. 2021a]. To illustrate, consider the mechanism of slot machines where a player uses a *ticket* to play and can win with probability p to receive the *entire money* stored in the machine, or lose otherwise.

Notably, we can use potential to represent money in digital contracts: a player pays one unit of potential to play, and all the potential stored in the machine is paid to the winner.

$$\text{slot} \triangleq \uparrow^{\mathsf{S}}_{\mathsf{L}} \triangleleft^{1} \oplus_{\mathsf{P}} \{\mathbf{won}^{0.2} : \triangleright^{*} \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{slot}, \mathbf{lost}^{0.8} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{slot}\}$$

The session type slot represents the interface to a slot machine. To allow multiple players, the type is shared: the $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ represents that the channel must be acquired to play. Once acquired, the player must deposit the ticket by paying 1 unit of potential, as represented with $\triangleleft^{1}$. Then, the type transitions to a probabilistic internal choice, denoting that the player can win with probability 0.2. Thus, the player is *guaranteed a 20% chance of winning*. If the player wins, the type sends the **won** label followed by an unknown ($*$) amount of potential. The $*$ indicates that we would like the type system to infer how much potential the player wins. On the other hand, if the player loses, the type only sends the **lost** label. Then, in either case, the type detaches with the $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ operator issuing a release, and the session recurses to type slot.

NomosPro automatically computes the potential to be paid to the winner by inferring the type

$$\text{slot} \triangleq \uparrow^{\mathsf{S}}_{\mathsf{L}} \triangleleft^{1} \oplus_{\mathsf{P}} \{\mathbf{won}^{0.2} : \triangleright^{5} \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{slot}, \mathbf{lost}^{0.8} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{slot}\}$$

Thus, the player wins 5 times the price of a ticket. This matches our expectation as the winning probability is $0.2 = 1/5$. In general, NomosPro infers the win amount to be $1/p$ when the win probability is p for any fixed p.

The above analysis works when the slot machine itself makes no money; the winner obtains the entire pot stored in the slot. Interestingly, we can modify the slot session type to allow the machine to make money. For instance, suppose we modify the slot type to

$$\text{slot} \triangleq \uparrow^{\mathsf{S}}_{\mathsf{L}} \triangleleft^{1} \oplus_{\mathsf{P}} \{\mathbf{won}^{0.2} : \triangleright^{4} \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{slot}, \mathbf{lost}^{0.8} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{slot}\}$$

so that the winner obtains only 4 units. The remaining potential can be stored in an internal list inside the slot machine using the potlist type again.

$$\text{potlist} \triangleq \oplus\{\mathbf{cons} : \triangleright^{*}\text{potlist}, \mathbf{nil} : \mathbf{1}\}$$

NomosPro infers the $*$ annotation in potlist as $\triangleright^{1}$, confirming that the slot makes 1 unit of money in expectation. However, the winner cannot be paid more than 5 units. So, if we set the type for the slot machine as

$$\text{slot} \triangleq \uparrow^{\mathsf{S}}_{\mathsf{L}} \triangleleft^{1} \oplus_{\mathsf{P}} \{\mathbf{won}^{0.2} : \triangleright^{r} \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{slot}, \mathbf{lost}^{0.8} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{slot}\}$$

where $r > 5$, then the program fails to typecheck! Intuitively, this makes sense since money (or potential) cannot be generated out of thin air. The expected money going into the slot machine must be greater than or equal to the expected money coming out.

7 IMPLEMENTATION AND EVALUATION

We have implemented an open-source prototype for NomosPro in OCaml (7622 lines of code). In this section, we describe the main components of the implementation.

Type Reconstruction. We implemented a *bi-directional type checker* [Pierce and Turner 2000] for NomosPro specifically focusing on the *quality of error messages*. The programmer provides the initial type for each process in the declaration, and the intermediate types are reconstructed while type checking the corresponding definition. This aids in localizing the source of the error as the program location where type reconstruction fails.

The critical aspect of reconstruction is handling the *non-determinism* in the rules for flip and pcase. To typecheck a probabilistic branch, we need to *guess* the types of each channel in each branch (e.g., Δ_ℓ, C_ℓ in $\oplus_{\mathsf{P}} L$ and $\Delta_H, A_H, \Delta_T, A_T$ in flip). This problem is exacerbated further when such branches are arbitrarily nested. Consider the debias process again that involves such a nesting from Section 2, where pbool $\triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{0.6} : \mathbf{1}, \mathbf{false}^{0.4} : \mathbf{1}\}$ and ubool $\triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{0.5} : \mathbf{1}, \mathbf{false}^{0.5} : \mathbf{1}\}$.

$$\dfrac{\dfrac{(b : 1) \vdash c..\mathbf{false}\ ;\ \ldots :: (c : \mathbf{A_1}) \qquad (b : 1) \vdash c..\mathbf{true}\ ;\ \ldots :: (c : \mathbf{A_2}) \qquad \mathbf{A} = 0.166667 \cdot \mathbf{A_1} +^{\mathsf{R}} 0.833333 \cdot \mathbf{A_2}}{(b : 1) \vdash \mathsf{flip}\ 0.166667\ (\mathsf{H} \Rightarrow c..\mathbf{false}\ ;\ \ldots \mid \mathsf{T} \Rightarrow c..\mathbf{true}\ ;\ \ldots) :: (c : \mathbf{A})}}{(b : \mathrm{pbool}) \vdash \mathrm{pcase}\ b\ (\mathbf{true} \Rightarrow \mathsf{flip}\ 0.166667\ (\mathsf{H} \Rightarrow \ldots \mid \mathsf{T} \Rightarrow \ldots) \mid \mathbf{false} \Rightarrow \ldots) :: (c : \mathrm{ubool})}$$

We show a part of the derivation tree of the debias process (**false** branch not shown). Although we know the initial type c : ubool (conclusion at the bottom), we need to guess its intermediate type in the **true** and **false** branches. Suppose we guess type $\mathbf{A}$ for c in the **true** branch and similarly type $\mathbf{B}$ in the **false** branch (not shown). We would obtain the constraint ubool $= 0.6 \cdot \mathbf{A} +^{\mathsf{R}} 0.4 \cdot \mathbf{B}$ taking the probability of each branch into account. We again need to typecheck a flip expression. So, we again guess types $\mathbf{A_1}$ and $\mathbf{A_2}$ in the H and T branches respectively. Then, we obtain the constraint $\mathbf{A} = 0.166667 \cdot \mathbf{A_1} +^{\mathsf{R}} 0.833333 \cdot \mathbf{A_2}$ as shown in the derivation. The outcome of the type checking of this process would depend on the satisfaction of these constraints.

Our type checker solves this issue using a 2-phase reconstruction algorithm. Before initiating the type checking algorithm, we perform a simple pass on the process code and replace all $*$ annotations with variables. These variables later become a part of the linear constraints that are generated by the type checker.

The first phase of type checking is *top-down* where all probability values are ignored and the remaining basic session typing rules are checked. NomosPro's type system is based on syntax-directed typing rules. In other words, given a process typed as $\Delta \vdash P :: (x : A)$, the rule that needs to be applied depends on the structure of P. For instance, if $P = x.k\ ;\ Q$, i.e., send a label k on channel x and continue with Q, then the $\oplus R$ rule is applied. If A is a type name, its definition is looked up in the signature and expanded. Since A must expand to an internal choice type (if it does not, a type error is reported), suppose its definition is $\oplus\{\ell : A_\ell\}_{\ell \in L}$. Then, the continuation type for x after sending label k is A_k. Therefore, the type checker proceeds to typing the expression $\Delta \vdash Q :: (x : A_k)$. The other non-probabilistic rules follow a similar pattern.

The second phase of type checking is *bottom-up* where we apply the probabilistic typing rules (Figure 4). We explain this phase on the debias process by recalling the implementation

```
decl debias : (b : pbool) ⊢ (c : ubool)
proc c ← debias b = pcase b ( true ⇒ flip 0.166667 ( H ⇒ c..false; wait b; close c
                                                   | T ⇒ c..true; wait b; close c )
                            | false ⇒ c..false; wait b; close c )
```

As described earlier, suppose the type of c is $\mathbf{A}$ in the **true** branch and $\mathbf{B}$ in the **false** branch. We know that ubool $= 0.6 \cdot \mathbf{A} +^{\mathsf{R}} 0.4 \cdot \mathbf{B}$, which implies that $\mathbf{A}$ and $\mathbf{B}$ have the same *type structure* as ubool, but only differ in the probability annotations. We then introduce probability variables to stand in for the annotations (which will later be inferred). Suppose we define

$$\mathbf{A} \triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{p_A} : \mathbf{1}, \mathbf{false}^{p'_A} : \mathbf{1}\} \qquad \mathbf{B} \triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{p_B} : \mathbf{1}, \mathbf{false}^{p'_B} : \mathbf{1}\}$$

The validity of these types straight away derive the constraint $p_A + p'_A = 1$ and $p_B + p'_B = 1$. And from the constraint ubool $= 0.6 \cdot \mathbf{A} +^{\mathsf{R}} 0.4 \cdot \mathbf{B}$, we obtain the constraints $0.6p_A + 0.4p_B = 0.5$ and $0.6p'_A + 0.4p'_B = 0.5$. Moving to type checking the flip expression, we assign types $\mathbf{A_1}$ and $\mathbf{A_2}$ to channel c in the H and T branches respectively. Again, we define

$$\mathbf{A_1} \triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{p_1} : \mathbf{1}, \mathbf{false}^{p'_1} : \mathbf{1}\} \qquad \mathbf{A_2} \triangleq \oplus_{\mathsf{P}}\{\mathbf{true}^{p_2} : \mathbf{1}, \mathbf{false}^{p'_2} : \mathbf{1}\}$$

Again, we obtain $p_1 + p'_1 = 1$ and $p_2 + p'_2 = 1$. Also, simplifying the constraint $\mathbf{A} = 0.166667 \cdot \mathbf{A_1} +^{\mathrm{R}} 0.833333 \cdot \mathbf{A_2}$, we derive $p_A = 0.166667p_1 + 0.833333p_2$ and $p'_A = 0.166667p'_1 + 0.833333p'_2$. But also, in the H branch, when we type check sending the label **false** on c, we derive $p_1 = 0$ and $p'_1 = 1$. Similarly, from the T branch, we get $p_2 = 1$ and $p'_2 = 0$. With a similar argument, from the **false** branch, we get $p_B = 0$ and $p'_B = 1$. Solving all these constraints, we obtain

$$p_A = 0.833333 \quad p'_A = 0.166667 \quad p_B = 0 \quad p'_B = 1 \quad p_1 = 0 \quad p'_1 = 1 \quad p_2 = 1 \quad p'_2 = 0$$

The second phase generates linear constraints (as described above) on the probability annotations which are shipped to Coin-Or [Clp team 2022], an off-the-shelf LP solver. Coin-Or either returns a satisfying assignment which is substituted into the typing derivation, or reports that the constraints are unsatisfiable which is interpreted as a type checking failure.

Potential and Probability Inference. A note-worthy point here is that Coin-Or also acts as our inference engine to compute both potential and probability annotations automatically. Unknown (probability or potential) annotations are indicated using $*$ in the source program which are first substituted with variables (just like intermediate channel types). Application of bidirectional typing rules generates linear constraints for both the probability annotations and potential. The type-checker separately collects the potential constraints and adds an additional optimization which minimizes the total sum of potential annotations to achieve tight bounds. The constraints w.r.t. probabilities are solved first and partially substituted into the constraints w.r.t. potential. This ensures that we can at least verify the program's probabilistic behavior even if we are unable to infer a cost bound (which is undecidable in general). Since the exact potential annotations depend on the cost assigned to each operation and are difficult to predict statically, we found inference to be extremely useful to make NomosPro practically applicable.

Evaluation. We implemented all examples presented so far and several other benchmarks in the NomosPro prototype. Table 1 compiles the results of our experiments run on an Intel Core i5 2.7 GHz processor with 16 GB 1867 MHz DDR3 memory. For each benchmark, we present the expected execution cost determined by NomosPro's inference engine. Additionally, for the Markov chain benchmarks, we also verify their limiting distribution and/or correctness.

The program **leader** is the Itai and Rodeh [1990] synchronous leader election where the number of processes is 3 and maximum id is 3. The **bdd ret.** examples implement the bounded retransmission protocol [Helmink et al. 1994] that drops messages with 20% probability with $N = 4$ and 5 respectively. Correspondingly, **bdd sym.** implements the symbolic variant of the same protocol representing the file chunks as a list. The Srikanth and Toueg [1987] and Bracha [1987] broadcast protocols are implemented in **auth. broad.** and **bracha** respectively. All these protocols demonstrate that NomosPro *scales to large programs* due to its linear-time type checker and efficient LP solver. The **din. crypto.** benchmark implements the Chaum [1988] dining cryptographers protocol used for exchanging secure messages with sender and receiver untraceability. Briefly, the protocol involves 3 cryptographers who flip an *unbiased* coin and communicate their coin's outcome with one of their neighbors. Crucial to the correctness of this protocol is the use of an unbiased coin which can be statically guaranteed using NomosPro's type system.

The **queue** benchmark implements the amortized queue from Section 6. The programs **3 die** and **6 die** implement the 3-faced and 6-faced die from Section 2 using a coin [Knuth and Yao 1976]. The **fair coin** process recursively flips a biased coin to produce a fair coin. The program **exp. trials** recursively flips a fair coin until it outputs H. The bound shows the expected number of flips needed in either case. The programs **rnd walk**, **repair**, and **weather** describe Markov chains representing a random walk along a 1D line, a faulty machine, and probabilistic weather patterns, respectively. NomosPro infers that if the random walker moves forward with probability 0.6, they will cover n steps in $4n$ time units on average. NomosPro also infers that the faulty machine spends $0.11n$

Table 1. Evaluation of NomosPro. LOC = lines of code; Defs = #type and process definitions; T (ms) = type checking time in milliseconds; Vars = #unknown $*$ annotations replaced by variables; Cons = #constraints generated during type checking; I (ms) = inference time in milliseconds; Bound = expected execution cost counting total number of message exchanges (unless explicitly stated in text).

Program	LOC	Defs	T (ms)	Vars	Cons	I (ms)	Bound	
leader	161	18	0.3	144	360	3.1	1.8	$(N = 3, K = 3)$
bdd ret.	646	122	906.7	403	1323	353.6	5	$(N = 4)$
bdd ret.	806	152	11676.8	503	1652	4403.6	6.25	$(N = 5)$
bdd sym.	79	14	0.1	86	276	3.6	$1.25N$	
auth	397	32	0.9	454	1436	—	—	
bracha	736	58	1.7	559	1731	—	—	
din. crypto.	99	5	0.7	291	870	4.2	2	
queue	118	14	0.4	158	484	3.7	$7n$	
3 die	37	6	0.3	24	72	2.9	2.667	
6 die	109	14	0.3	54	160	4.9	3.667	
fair coin	17	2	0.3	20	54	2.9	4.167	(bias prob. is 0.4)
exp. trials	10	2	0.2	8	24	2.8	2	
rnd walk	27	3	0.1	32	96	3.4	$4n$	(move prob. is 0.6)
repair	22	3	0.3	39	118	2.9	$0.11n$	
weather	23	2	0.3	35	84	2.9	2	
chessboard	811	16	28.5	3371	8354	70.2	5.847	
nat add	25	5	0.3	33	90	3.2	n	
nat double	28	6	0.3	31	95	3.6	$0.4 + 2.4n$	
lottery	16	2	0.3	19	52	2.8	10	(win prob. is 0.1)
slots	17	2	0.4	21	58	3.3	5	(win prob. is 0.2)

days out of n in repair. For the **repair** and **weather** benchmarks, NomosPro also verifies their limiting distribution estimating the fraction of time the faulty machine spends in repair, and the expected fraction of rainy and sunny days, respectively. The chessboard Markov chain described in Section 6.3 is implemented in **chessboard**. We also implemented probabilistic unary natural numbers that send *successor* and *zero* with a fixed probability. Programs **nat add** and **nat double** study the expected behavior of adding and doubling them respectively. For **nat add**, the bound is a function of the number of bits in the first addend. The **lottery** contract guarantees a certain winning probability (10%) in its session type. Finally, **slots** implements slot machines from Section 6. In both examples, the bound describes the amount received by the winner.

8 RELATED WORK

Probabilistic Session Types. Aman and Ciobanu [2019] propose a typing system extending multiparty session types [Honda et al. 2008] with probabilistic internal choice and non-deterministic external choice. Their session typing discipline contains probabilistic intervals as opposed to NomosPro, where probabilities are exact. Inverso et al. [2020] developed a system with probabilistic binary session types. Their system does not support non-probabilistic internal/external choices and channel passing, whereas NomosPro is a conservative extension of intuitionistic session types [Caires and Pfenning 2010]. Another distinguishing feature of NomosPro from all the aforementioned work is that we automatically infer expected cost and probabilities.

Probabilistic Process Algebras. Probabilities were introduced in process algebras [Bergstra and Klop 1984] by a probabilistic internal choice operator and extended with parallel composition [Andova 1999]. Herescu and Palamidessi [2000] later proposed an extension of the asynchronous π-calculus with a notion of random choice distinguishing between probabilistic choice internal to a process and non-deterministic external choice made by an adversarial scheduler. They further use these techniques to prove probabilistic correctness of the leader election protocol under any possible scheduler. Other extensions to model time [Hansson and Fredlund 1994] and performance [Hillston 1996] have also been proposed working on the same principle of a probabilistic choice operator. NomosPro differs from these work by describing probabilistic behavior in a compositional type system.

Horne [2019] uses a processes-as-formulae paradigm and introduces *sub-additives* $\oplus_P$ and $\&_P$ as term formers to construct processes. For instance, their $\oplus_P$ term former is similar to the flip expression in our work. Despite the surface similarity, NomosPro is fundamentally different, as our $\oplus_P$ and $\&_P$ are type formers and NomosPro follows a Curry-Howard interpretation of intuitionistic linear logic [Caires and Pfenning 2010], where processes are proofs and types are formulae.

Probabilistic Model Checking. Probabilistic model checkers (such as PRISM [Kwiatkowska et al. 2011] and Storm [Dehnert et al. 2017]) support analysis of both discrete- and continuous-time Markov chains [Kwiatkowska et al. 2007a], Markov decision processes [Forejt et al. 2011], probabilistic timed automata [Kwiatkowska et al. 2007b], π-calculus [Norman et al. 2007], and randomized distributed algorithms [Norman 2004], including dining cryptographers protocol by Chaum [1988]. 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, the symbolic bounds produced by NomosPro are not simply functions of numbers or sizes but of *interactions* between communicating processes. Moreover, probabilistic session types serve as compositional specifications that enable the analysis of different processes in isolation. NomosPro also supports amortization (e.g. functional queue) to infer constant-time bounds even where certain operations can be linear-time, all while using efficient LP solvers. Typing derivations in NomosPro also provide a succinct proof of the probabilistic behavior of concurrent programs along with their expected cost which can be verified efficiently in linear-time. Bertrand et al. [2019] extend threshold automata to model randomized algorithms parameterized by the number of processes and failures under round-rigid schedules. In contrast, the linear fragment of NomosPro guarantees *deadlock freedom*.

Semantics of Probabilistic Languages. Several denotational models combining probabilistic and non-deterministic choices have been developed [Jones 1989; Mislove et al. 2004; Tix et al. 2009; Varacca 2002; Varacca and Winskel 2006; Wang et al. 2019b], and some of them are focused on probabilistic concurrency [Mislove 2000; Varacca 2003; Varacca and Yoshida 2007]. In NomosPro, we use an operational semantic model. Trace-based operational semantics have been used to analyze probabilistic concurrent programs [Hart et al. 1983; Tassarotti and Harper 2019]. There, the semantics maps a concurrent program to a distribution of execution traces, but each trace is reasoned about separately and traces are connected in the final phase of the analysis. In contrast, our nested-multiverse semantics accounts for connections among traces in different universes via nested distributions.

Reasoning About Probabilistic Programs. There exist some studies on automatic expected cost analysis of sequential (imperative) probabilistic programs [Chatterjee et al. 2016; Kura et al. 2019; Ngo et al. 2018; Wang et al. 2019a]. They can be seen as an automation of Kozen's weakest pre-expectation calculus [Kaminski et al. 2016; Kozen 1981]. An automated type-based variant of this idea has been introduced [Wang et al. 2020] in the context of amortized resource analysis [Hoffmann et al.

2017; Hofmann and Jost 2003]. The novelty of our work, is an automatic analysis for a probabilistic language with concurrency. It builds on prior work on work analysis via (deterministic) resource-aware session types [Das et al. 2018].

We are not aware of any other automated rule-based system for deriving resource bounds for concurrent probabilistic programs. However, there are multiple systems that can be used for manually deriving expected cost bounds [Hansson and Jonsson 1994; Tassarotti and Harper 2018, 2019]. Recent work on analyzing probabilistic networks [Foster et al. 2016; Gehr et al. 2018; Smolka et al. 2019] can be viewed as reasoning systems for probabilistic message passing systems. However, the work on networks focuses on finite state systems and global properties. In contrast, the contribution of this article is the integration of (local) probability distributions in session types and the automatic expected resource analysis for message-passing processes.

9 CONCLUSION

This article presented NomosPro, a probabilistic concurrent language relying on novel *probabilistic resource-aware session types*. We introduced a novel nested-multiverse semantics to prove type safety, correctness of expected bounds, and probability consistency. We employed NomosPro to implement and verify correctness of Markov chains, infer expected cost of randomized distributed algorithms, and analyze expected behavior of digital contracts and amortized data structures. We currently only support constant probabilities. In the future, we would like to explore symbolic probability annotations which are common in distributed algorithms. We also plan to enhance the inference engine with non-linear solvers to produce higher-degree polynomial expected cost bounds.

ACKNOWLEDGMENTS

This article is based on research supported by DARPA under AA Contract FA8750-18-C-0092 and by the National Science Foundation under awards 1801369, 1845514, and 2007784. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

REFERENCES

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

Suzana Andova. 1999. Process Algebra with Probabilistic Choice. In *Formal Methods for Real-Time and Probabilistic Systems*, Joost-Pieter Katoen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–129.

Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. 2019. Type-Based Complexity Analysis of Probabilistic Functional Programs. In *Logic in Computer Science (LICS'19)*. https://doi.org/10.1109/LICS.2019.8785725

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 37 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110281

J.A. Bergstra and J.W. Klop. 1984. Process algebra for synchronous communication. *Information and Control* 60, 1 (1984), 109 – 137. https://doi.org/10.1016/S0019-9958(84)80025-X

Nathalie Bertrand, Igor Konnov, Marijana Lazic, and Josef Widder. 2019. Verification of Randomized Consensus Algorithms Under Round-Rigid Adversaries. In *30th International Conference on Concurrency Theory (CONCUR 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 140)*, Wan Fokkink and Rob van Glabbeek (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 33:1–33:15. https://doi.org/10.4230/LIPIcs.CONCUR.2019.33

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A Lambda-Calculus Foundation for Universal Probabilistic Programming. In *Int. Conf. on Functional Programming (ICFP'16)*. https://doi.org/10.1145/2951913.2951942

Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143. https://doi.org/10.1016/0890-5401(87)90054-X

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, P.Gastin and F.Laroussinie (Eds.). Springer LNCS 6269,

Paris, France, 222–236.
Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verif. (CAV'16)*. https://doi.org/10.1007/978-3-319-41528-4_1
David Chaum. 1988. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *J. Cryptology* 1 (1988), 65–75. https://doi.org/10.1007/BF00206326
Clp team. 2022. COIN-OR Linear Programming Solver. Available on https://projects.coin-or.org/Clp.
A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar. 2021a. Resource-Aware Session Types for Digital Contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 111–126.
Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. 2021b. Nested Session Types. In *30th European Symposium on Programming*, N. Yoshida (Ed.). Springer LNCS, Luxembourg, Luxembourg, 178–206. http://www.cs.cmu.edu/~fp/papers/esop21.pdf Extended version available as arXiv:2010.06482.
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)*. ACM, New York, NY, USA, 305–314. https://doi.org/10.1145/3209108.3209146
Ankush Das and Frank Pfenning. 2020a. 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
Ankush Das and Frank Pfenning. 2020b. Session Types with Arithmetic Refinements. 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, 13:1–13:18. https://doi.org/10.4230/LIPIcs.CONCUR.2020.13
Ankush Das and Frank Pfenning. 2020c. Verified Linear Session-Typed Concurrent Programming. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming* (Bologna, Italy) *(PPDP '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 15 pages. https://doi.org/10.1145/3414080.3414087
Ankush Das, Di Wang, and Jan Hoffmann. 2021c. Probabilistic Resource-Aware Session Types. https://arxiv.org/abs/2011.09037
Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A Storm is Coming: A Modern Probabilistic Model Checker. In *Computer Aided Verif. (CAV'17)*. https://doi.org/10.1007/978-3-319-63390-9_31
Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Princ. of Prog. Lang. (POPL'15)*. https://doi.org/10.1145/2676726.2677001
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
Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic netkat. In *European Symposium on Programming*. Springer, 282–309.
Simon Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2 (01 Nov 2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z
Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: probabilistic inference for networks. *ACM SIGPLAN Notices* 53, 4 (2018), 586–602.
J. Y. Girard and Y. Lafont. 1987. Linear logic and lazy computation. In *TAPSOFT '87*, Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–66.
Hans Hansson and Bengt Jonsson. 1994. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing* 6 (1994), 102–111.
Hans A. Hansson and Lars-Ake Fredlund. 1994. *Time and Probability in Formal Design of Distributed Systems*. Elsevier Science Inc., USA.
Sergiu Hart, Micha Sharir, and Amir Pnueli. 1983. Termination of Probabilistic Concurrent Program. *Trans. on Prog. Lang. and Syst.* 5 (July 1983). Issue 3. https://doi.org/10.1145/2166.357214
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.
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.
Jane Hillston. 1996. *A Compositional Approach to Performance Modelling*. Cambridge University Press, USA.
Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 359–373. https://doi.org/10.1145/3009837.3009842

Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th Symposium on Principles of Programming Languages (POPL'03)*. 185–197.

Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523.

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

Ross Horne. 2019. The Sub-Additives: A Proof Theory for Probabilistic Choice extending Linear Logic. In *Formal Struct. for Comput. and Deduction (FSCD'19)*. https://doi.org/10.4230/LIPIcs.FSCD.2019.23

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

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

Claire Jones. 1989. *Probabilistic Nondeterminism.* Ph. D. Dissertation. University of Edinburgh.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs. In *European Symp. on Programming (ESOP'16)*. https://doi.org/10.1007/978-3-662-49498-1_15

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-Guided Program Synthesis. In *Prog. Lang. Design and Impl. (PLDI'19)*. https://doi.org/10.1145/3314221.3314602

Donald E. Knuth and Andrew Chi-Chih Yao. 1976. The complexity of nonuniform random number generation.

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22 (June 1981). Issue 3. https://doi.org/10.1016/0022-0000(81)90036-2

Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. 2019. Tail Probability for Randomized Program Runtimes via Martingales for Higher Moments. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'19)*. https://doi.org/10.1007/978-3-030-17465-1_8

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

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.

Marta Kwiatkowska, Gethin Norman, Jeremy Sproston, and Fuzhi Wang. 2007b. Symbolic model checking for probabilistic timed automata. *Information and Computation* 205, 7 (2007), 1027 – 1077. https://doi.org/10.1016/j.ic.2007.01.004

Michael W. Mislove. 2000. Nondeterminism and Probabilistic Choice: Obeying the Laws. In *Int. Conf. on Concurrency Theory (CONCUR'00)*. https://doi.org/10.1007/3-540-44618-4_26

Michael W. Mislove, Joël Ouaknine, and James Worrell. 2004. Axioms for Probability and Nondeterminism. *Electr. Notes Theor. Comp. Sci.* 96 (June 2004). https://doi.org/10.1016/j.entcs.2004.04.019 Proceedings of the 10th International Workshop on Expressiveness in Concurrency.

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Prog. Lang. Design and Impl. (PLDI'18)*. https://doi.org/10.1145/3192366.3192394

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

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.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/ Previous number = SIDL-WP-1999-0120.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100

Steffen Smolka, Praveen Kumar, David M Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable Verification of Probabilistic Networks. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 190–203.

T. K. Srikanth and Sam Toueg. 1987. Simulating Authenticated Broadcasts to Derive Simple Fault-Tolerant Algorithms. *Distributed Comput.* 2, 2 (1987), 80–94. https://doi.org/10.1007/BF01667080

RE Tarjan. 1985. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods* 6, 2 (1985), 306–318.

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

Joseph Tassarotti and Robert Harper. 2019. A Separation Logic for Concurrent Randomized Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 64 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290377

Regina Tix, Klaus Keimel, and Gordon D. Plotkin. 2009. Semantic Domains for Combining Probability and Non-Determinism. *Electr. Notes Theor. Comp. Sci.* 222 (February 2009). https://doi.org/10.1016/j.entcs.2009.01.002

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *European Symp. on Programming (ESOP'13)*. https://doi.org/10.1007/978-3-642-37036-6_20

Daniele Varacca. 2002. The Powerdomain of Indexed Valuations. In *Logic in Computer Science (LICS'02)*. https://doi.org/10.1109/LICS.2002.1029838

Daniele Varacca. 2003. *Probability, Nondeterminism and Concurrency: Two Denotational Models for Probabilistic Computation.* Ph. D. Dissertation. University of Aarhus.

Daniele Varacca and Glynn Winskel. 2006. Distributing Probability over Nondeterminism. *Math. Struct. Comp. Sci.* 16 (February 2006). Issue 1. https://doi.org/10.1017/S0960129505005074

Daniele Varacca and Nobuko Yoshida. 2007. Probabilistic π-Calculus and Event Structures. *Electronic Notes in Theoretical Computer Science* 190, 3 (2007), 147 – 166. https://doi.org/10.1016/j.entcs.2007.07.009 Proceedings of the Fifth Workshop on Quantitative Aspects of Programming Languages (QAPL 2007).

Di Wang, Jan Hoffmann, and Thomas Reps. 2019b. A Denotational Semantics for Low-Level Probabilistic Programs with Nondeterminism. *Electr. Notes Theor. Comp. Sci.* 347 (November 2019). https://doi.org/10.1016/j.entcs.2019.09.016 Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics.

Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising Expectations: Automating Expected Cost Analysis with Types.

Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. 2019a. Cost Analysis of Nondeterministic Probabilistic Programs. In *Prog. Lang. Design and Impl. (PLDI'19)*. https://doi.org/10.1145/3314221.3314581

Received 2022-07-07; accepted 2022-11-07