Verification of Probabilistic Distributed Protocols with Multiparty Session Types

Leo Seojun Lee

A project report submitted in partial fulfilment of the requirements for

Part B of the *Honour School of Computer Science*

Trinity Term 2025

Oxford, United Kingdom

4997 words

Abstract

Multiparty session types (MPST) ensure that distributed systems correctly implement error-free communication protocols. However, many real-world protocols exhibit probabilistic behaviour, which MPST cannot capture. To address this, we propose the first probabilistic extension to bottom-up MPST and implement a verification procedure using the PRISM model checker. We define a translation from types into PRISM and prove its soundness and completeness with respect to their operational semantics. Moreover, we evaluate the practicality of our implementation through case studies and measuring its performance across a diverse set of examples.

Acknowledgements

Firstly, I would like to thank my supervisors, Nobuko Yoshida and Joe Paulus, for their invaluable guidance and support throughout this project. I'm truly grateful for the patience and clarity with which they introduced me to this field. Their dedication to research has been a constant source of inspiration, and I feel fortunate to have learned under their mentorship.

I would also like to thank my wonderful friends, whose company has made the past three years so immensely enjoyable. I'll cherish our dinners, walks, and study sessions.

Last, but most definitely not least, I would like to thank my parents for their unwavering love and encouragement. I am endlessly grateful for their everlasting support.

Contents

Chapter 1

Introduction

Distributed systems are everywhere. They underpin technologies we rely on daily, ranging from online communication to financial services. As these systems grow in complexity and scale, ensuring reliable and correct communication becomes increasingly critical.

A prominent approach to verifying such systems is through *session types* [Honda et al., 1998], a typing discipline for specifying communication protocols between two message-passing concurrent processes. Multiparty session types (MPST) [Honda et al., 2008] extend this to protocols involving multiple participants. The original *top-down* MPST framework guarantees deadlock freedom, while the newer *bottom-up* framework [Scalas and Yoshida, 2019] generalises it to support broader properties. MPST is now a prominent method for formalising and verifying distributed protocols, with implementations in over 16 languages including Rust, Go and Java [Yoshida, 2024].

However, many real-world distributed protocols are *probabilistic* in nature. They may utilise *randomised algorithms* to improve their efficiency [Aspnes and Herlihy, 1990], or take *stochastic failures* into account [Fehnker and Gao, 2006]. The standard MPST framework is insufficient to capture such probabilistic behaviours.

Several probabilistic extensions to session types have been proposed, but these either target binary sessions [Inverso et al., 2020, Das et al., 2023] or build off top-down MPST [Aman and Ciobanu, 2019]. The latter is often too restrictive, as it only types deadlock-free processes – a condition that may not hold in real-world probabilistic systems. Instead, we aim to explore richer

properties such as the *probability* of deadlock freedom.

To address this gap, we present the first probabilistic extension to bottom-up MPST. Our main contributions are as follows.

- We introduce our type system and formally define their properties (Chapter 2).
- We define a translation from types into the PRISM language and prove its correctness with respect to their operational semantics (Chapter 3). Notably, the recursive nature of session types precludes many natural inductive proofs; instead, we define new relations and functions that enable *coinductive* proofs.
- We present Prose[1], a tool implemented in OCaml which verifies probabilistic protocols using our theoretical contributions (Chapter 4).
- We demonstrate that our approach is both practical and performant through a wide range of case studies and performance benchmarks (Chapter 5).

[1] https://github.com/smjleo/prose

Chapter 2

A type system

In this chapter, we introduce a type system for probabilistic concurrent message-passing programs. To do this, we briefly touch on a formal notion of such programs by presenting a process calculus. Next, we define a type system, capturing a certain abstract communication protocol that these programs must follow. Finally, we argue that the type system, as presented thus far, is too weak.

We model concurrent processes using a session calculus closely modelling the multiparty session π-calculus presented in [Scalas and Yoshida, 2019], but extended with probabilities. A formal definition of the π-calculus is beyond the scope of this report; instead, we proceed with a small example.

2.1 A simple session

Example 2.1.1. Suppose alice and bob are two participants in a distributed system. Their processes are given by

$$P_{\text{alice}} = \text{bob}\boxplus\begin{cases} 0.4 \;:\; \texttt{inc}\langle 41\rangle.\text{bob}\Sigma\texttt{result}(x).\mathbf{0} \\ 0.6 \;:\; \texttt{not}\langle \texttt{true}\rangle.\text{bob}\Sigma\texttt{result}(x).\mathbf{0} \end{cases}$$

$$P_{\text{bob}} = \text{alice}\Sigma\begin{cases} \texttt{inc}(x).\text{alice}\boxplus\texttt{result}\langle x+1\rangle.\mathbf{0} \\ \texttt{not}(x).\text{alice}\boxplus\texttt{result}\langle \neg x\rangle.\mathbf{0} \end{cases}$$

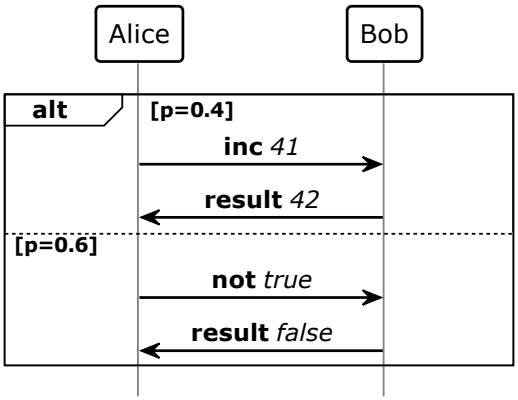

Figure 2.1: Illustration of Example 2.1.1.

With probability 0.4, alice sends (⊞) `inc` with payload 41, receives (Σ) a `result` from bob, binds it to x, and terminates (**0**). With probability 0.6, she sends `not` with payload `true`, receives a `result`, and terminates.

Meanwhile, bob awaits either message. If he receives `inc`, he increments the payload, sends the result, and terminates. If he receives `not`, he negates the payload, replies, and terminates.

We can construct a *session* by *assigning* these processes to alice and bob and *composing* them:

$$\mathscr{M}_{\text{alice,bob}} = \text{alice} \triangleleft P_{\text{alice}} \mid \text{bob} \triangleleft P_{\text{bob}}$$

Running $\mathscr{M}_{\text{alice,bob}}$, alice obtains $x = 42$ with probability 0.4, and $x = \text{false}$ with probability 0.6.

2.2 Introducing types

The π-calculus describes how sessions behave; the *session type* describes how sessions *should* behave. At a high level, session types encode communication protocols. Accordingly, they omit computational details, solely focusing on the messages passed between participants.

Definition 2.2.1. The syntax of *probabilistic multiparty session types* is given by:

$$\begin{aligned}
\mathsf{T} ::= {} & \mathsf{p}\&_{i\in I}\ell_i(\mathsf{B}_i).\mathsf{T}_i && \text{(external choice, with } |I| > 0) \\
\mid {} & \mathsf{p}\oplus_{i\in I} p_i : \ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}'_i && \text{(internal choice, with } |I| > 0,\ \textstyle\sum_{i\in I} p_i = 1 \text{ and } p_i \in [0,1]) \\
\mid {} & \mu\mathbf{t}.\mathsf{T} \mid \mathbf{t} && \text{(recursion and recursion variable)}
\end{aligned}$$

$$\mid \mathsf{end} \qquad \text{(termination)}$$

$$\mathsf{B} ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \qquad \text{(basic types)}$$

We assume types are closed (i.e. cannot have any unbound variables), unless otherwise stated. Types with open variables are called *partial types*, since they may appear as part of a closed type.

$\mathsf{p}\&_{i\in I}\ell_i(\mathsf{B}_i).\mathsf{T}_i$ denotes *external choice*: the process waits to receive from participant p one of the labels ℓ_i with payload type B_i, continuing as T_i. The choice of $i \in I$ is made by p, so the process must be ready for all $|I|$ options.

$\mathsf{p}\oplus_{i\in I} p_i : \ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}'_i$ represents *internal choice*: the process selects $i \in I$ with probability p_i, sends ℓ_i with payload B_i to p, then continues as T'_i.

$\mu\mathbf{t}.\mathsf{T}$ models recursion, binding T to $\mathbf{t}$. For example, $\mu\mathbf{t}.\mathsf{p}\oplus\{1 : \ell.\mathbf{t}\}$ sends ℓ to p forever. Recursion must be *guarded*: variables must appear under a choice. Thus, types like $\mu\mathbf{t}.\mathbf{t}$ are invalid.

Finally, end represents a terminated process.

Notation 2.2.1. We often omit the unit type and probability 1: $\mathsf{p}\&\ell.\mathsf{T}$ means $\mathsf{p}\&\{\ell(\mathsf{unit}).\mathsf{T}\}$, and $\mathsf{p}\oplus\ell.\mathsf{T}$ means $\mathsf{p}\oplus\{1 : \ell\langle\mathsf{unit}\rangle.\mathsf{T}\}$.

Example 2.2.1. The process P_{alice} from Example 2.1.1 inhabits

$$\mathsf{T}_{\mathsf{alice}} = \mathsf{bob}\oplus\begin{cases} 0.4 : \mathtt{inc}\langle\mathsf{int}\rangle.\mathsf{bob}\&\mathtt{result}(\mathsf{int}).\mathsf{end} \\ 0.6 : \mathtt{not}\langle\mathsf{bool}\rangle.\mathsf{bob}\&\mathtt{result}(\mathsf{bool}).\mathsf{end}. \end{cases}$$

Similarly, P_{bob} inhabits

$$\mathsf{T}_{\mathsf{bob}} = \mathsf{alice}\&\begin{cases} \mathtt{inc}(\mathsf{int}).\mathsf{alice}\oplus\mathtt{result}\langle\mathsf{int}\rangle.\mathsf{end} \\ \mathtt{not}(\mathsf{bool}).\mathsf{alice}\oplus\mathtt{result}\langle\mathsf{bool}\rangle.\mathsf{end}. \end{cases}$$

2.3 Typing contexts

We extend the notion of types into *typing contexts* – mappings from participants to types. This allows us to reason about the *interaction* between types.

$$\text{[CT-OUT]}\ \frac{k \in I \qquad p_k > 0}{\mathsf{p} : \mathsf{q} \oplus_{i \in I} p_i : \ell_i\langle \mathsf{B}_i \rangle.\mathsf{T}_i \xrightarrow{\mathsf{p}:\mathsf{q}!\ell_k\langle \mathsf{B}_k\rangle}_{p_k} \mathsf{p} : \mathsf{T}_k} \qquad \text{[CT-IN]}\ \frac{k \in I}{\mathsf{p} : \mathsf{q} \&_{i \in I} \ell_i(\mathsf{B}_i).\mathsf{T}_i \xrightarrow{\mathsf{p}:\mathsf{q}?\ell_k(\mathsf{B}_k)}_{1} \mathsf{p} : \mathsf{T}_k}$$

$$\text{[CT-}\tau\text{]}\ \frac{\Delta_1 \xrightarrow{\mathsf{p}:\mathsf{q}!\ell\langle \mathsf{B}\rangle}_{p} \Delta_1' \qquad \Delta_2 \xrightarrow{\mathsf{q}:\mathsf{p}?\ell(\mathsf{B})}_{1} \Delta_2'}{\Delta_1, \Delta_2 \xrightarrow{(\mathsf{p},\mathsf{q})\ell}_{p} \Delta_1', \Delta_2'} \qquad \text{[CT-REC]}\ \frac{\mathsf{p} : \mathsf{T}\{\mu \mathbf{t}.\mathsf{T}/\mathbf{t}\} \xrightarrow{\alpha}_{p} \mathsf{p} : \mathsf{T}'}{\mathsf{p} : \mu \mathbf{t}.\mathsf{T} \xrightarrow{\alpha}_{p} \mathsf{p} : \mathsf{T}'} \qquad \text{[CT-SC]}\ \frac{\Delta \xrightarrow{\alpha}_{p} \Delta'}{\mathsf{p} : \mathsf{T}, \Delta \xrightarrow{\alpha}_{p} \mathsf{p} : \mathsf{T}, \Delta'}$$

Figure 2.2: Reduction rules for typing contexts.

Definition 2.3.1. The syntax of *typing contexts* is given by

$$\Delta ::= \emptyset \mid \mathsf{p} : \mathsf{T},\ \Delta$$

Example 2.3.1. We put Example 2.2.1 into a context:

$$\Delta_{\mathsf{alice},\mathsf{bob}} = \mathsf{alice} : \mathsf{T}_{\mathsf{alice}},\ \mathsf{bob} : \mathsf{T}_{\mathsf{bob}}$$

Thus we have a *typing judgement* $\vdash \mathscr{M}_{\mathsf{alice},\mathsf{bob}} : \Delta_{\mathsf{alice},\mathsf{bob}}$.

Typing contexts *reduce* according to the rules in Figure 2.2. A reduction $\Delta \xrightarrow{(\mathsf{p},\mathsf{q})\ell}_{p} \Delta'$ means that p sends a message labelled ℓ to q with probability p, evolving context Δ to Δ'.

Intermediate reductions $\xrightarrow{\mathsf{p}:\mathsf{q}!\ell\langle \mathsf{B}\rangle}_{p}$ and $\xrightarrow{\mathsf{p}:\mathsf{q}?\ell(\mathsf{B})}_{1}$ (introduced by [CT-OUT] and [CT-IN], respectively) indicate readiness to send or receive. When both sides are ready, [CT-τ] performs the synchronisation.

[CT-REC] handles recursion by *unrolling* once, replacing occurrences of $\mathbf{t}$ with $\mu \mathbf{t}.\mathsf{T}$ using a standard capture-avoiding substitution. [CT-SC] preserves the types of uninvolved participants.

Example 2.3.2. Figure 2.3 shows the possible reductions of $\Delta_{\mathsf{alice},\mathsf{bob}}$ (Example 2.3.1).

Notation 2.3.1. For notational brevity, we use the following shorthands:

- $\Delta \to \Delta'$ if $\exists \mathsf{p}, \mathsf{q}, \ell, p \cdot \Delta \xrightarrow{(\mathsf{p},\mathsf{q})\ell}_{p} \Delta'$;
- $\Delta \xrightarrow{\alpha}_{p}$ if $\exists \Delta' \cdot \Delta \xrightarrow{\alpha}_{p} \Delta'$;

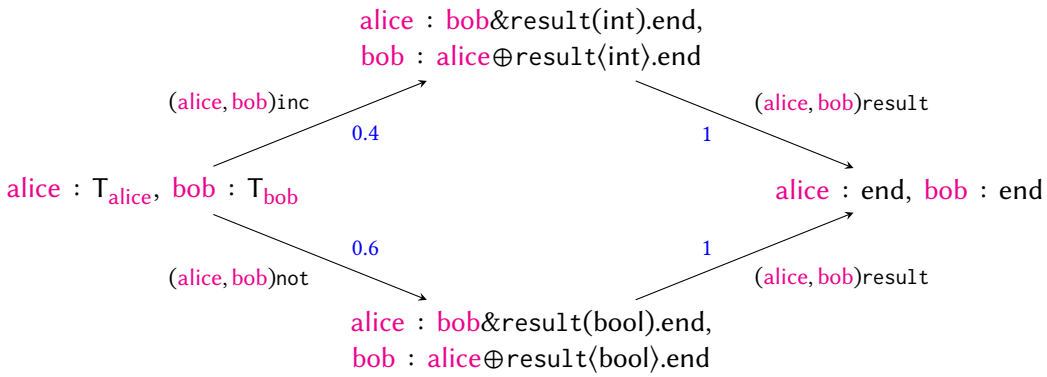

Figure 2.3: Reductions of Example 2.3.1.

- $\Delta \to_p$ if $\exists p, q, \ell \cdot \Delta \xrightarrow{(p,q)\ell}_p$;
- $\Delta \not\to$ if $\nexists p \cdot \Delta \to_p$.

2.4 Properties of typing contexts

Consider a typing context

$$\Delta_{\text{bad}} = \text{alice} : \text{bob} \oplus \ell_1.\text{end},\ \text{bob} : \text{alice} \& \ell_2.\text{end},$$

with $\ell_1 \neq \ell_2$. This seems bad: alice expects to send to bob, and bob expects to receive from alice, but they disagree on a label. Δ_{bad} is an *unsafe* context.

Definition 2.4.1. A typing context Δ is *safe*, written safe(Δ), if:

- $\Delta \xrightarrow{p : q!\ell\langle B\rangle}_p$ and $\Delta \xrightarrow{q : p?\ell'(B')}_1$ implies $\Delta \xrightarrow{(p,q)\ell}_p$; and
- $\Delta \to \Delta'$ implies safe(Δ').

Now consider

$$\Delta'_{\text{bad}} = \text{alice} : \text{bob} \& \ell.\text{end},\ \text{bob} : \text{alice} \& \ell.\text{end}.$$

This time, alice and bob are waiting on each other to send a message: Δ'_{bad} is *deadlocked*.

Definition 2.4.2. A context Δ is *deadlocked*, written deadlock(Δ), if $\Delta \not\to$ and $\exists p \cdot \Delta(p) \neq \text{end}$.

However, we can also ask a more interesting question: what is the *probability* that the typing

context will *stay* deadlock free? For example, consider

$$\Delta_{\text{pbad}} = \text{alice} : \text{bob}\oplus\begin{cases}0.8 : \text{good.end}\\ 0.2 : \text{bad.bob}\&\ell.\text{end},\end{cases} \quad \text{bob} : \text{alice}\&\begin{cases}\text{good.end}\\ \text{bad.alice}\&\ell.\text{end}.\end{cases}$$

Then we have

$$\Delta_{\text{pbad}} \xrightarrow{(\text{alice},\text{bob})\text{good}}_{0.8} \text{alice} : \text{end}, \text{bob} : \text{end}$$
$$\Delta_{\text{pbad}} \xrightarrow{(\text{alice},\text{bob})\text{bad}}_{0.2} \Delta'_{\text{bad}}.$$

Hence Δ_{pbad} is *deadlock free with probability* 0.8. We make this notion more precise with the subsequent definitions.

Definition 2.4.3. A *path* ξ is a (potentially infinite) sequence of contexts $\Delta_1; \Delta_2; \cdots$ such that $\forall i \in [1, |\xi|) \cdot \Delta_i \to \Delta_{i+1}$.

Definition 2.4.4. A *path* $\xi = \Delta_1; \cdots$ is *complete* if it is either infinite or $\Delta_{|\xi|} \not\to$. We write $\Xi(\Delta)$ for the set of all complete paths starting from Δ.

Definition 2.4.5. Given a context Δ and a set $S \subseteq \Xi(\Delta)$, we write $\mathbb{P}_\Delta(S)$ for the probability of obtaining a path in S starting from Δ.

Definition 2.4.6. Given a context Δ, its *probability* of *deadlock freedom* $\mathbb{P}_{\mathsf{DF}}(\Delta)$ is given by

$$\mathbb{P}_{\mathsf{DF}}(\Delta) = \mathbb{P}_\Delta(\{\xi = \Delta_1; \cdots \in \Xi(\Delta) \mid |\xi| = \infty \vee \neg\text{deadlock}(\Delta_{|\xi|})\}).$$

We say that Δ is *deadlock free with probability* $\mathbb{P}_{\mathsf{DF}}(\Delta)$.

Lastly, we might want a system to *terminate*.

Definition 2.4.7. A context Δ has *terminated*, written $\text{term}(\Delta)$, if $\forall \text{p} \in \text{dom}(\Delta) \cdot \Delta(\text{p}) = \text{end}$.

Definition 2.4.8. Given a context Δ, its *termination probability* $\mathbb{P}_{\mathsf{Term}}(\Delta)$ is given by

$$\mathbb{P}_{\mathsf{Term}}(\Delta) = \mathbb{P}_\Delta(\{\xi = \Delta_1; \cdots \in \Xi(\Delta) \mid |\xi| < \infty \wedge \text{term}(\Delta_{|\xi|})\}).$$

2.5 Typable ≠ well-behaved (for now)

Well-typed programs cannot "go wrong".

—Robin Milner, *A theory of type polymorphism in programming*

This well-known slogan from [Milner, 1978] succinctly captures a (arguably, *the*) desirable property of type systems. Does it hold for ours?

Admittedly, we have so far avoided the discussion of typing rules. However, the naïve rules we might expect prove to be problematic.

Example 2.5.1. We define two sessions with undesirable properties:

1. $\mathscr{M}_{\text{bad}} = \text{alice} \triangleleft \text{bob} \boxplus \ell_1.\mathbf{0} \mid \text{bob} \triangleleft \text{alice} \Sigma \ell_2.\mathbf{0}$
2. $\mathscr{M}'_{\text{bad}} = \text{alice} \triangleleft \text{bob} \Sigma \ell.\mathbf{0} \mid \text{bob} \triangleleft \text{alice} \Sigma \ell.\mathbf{0}$

Then we might expect that $\vdash \mathscr{M}_{\text{bad}} : \Delta_{\text{bad}}$ and $\vdash \mathscr{M}'_{\text{bad}} : \Delta'_{\text{bad}}$ (from Section 2.4), but in fact $\neg\text{safe}(\Delta_{\text{bad}})$ and $\mathbb{P}_{\text{DF}}(\Delta'_{\text{bad}}) = 0$.

This suggests that we have an incomplete picture. We'd prefer typed programs to not deadlock, and they definitely shouldn't be unsafe! Following the approach in [Scalas and Yoshida, 2019], we partially resolve this dilemma through the following typing rule:

$$\frac{\text{safe}(\{\text{p}_i : \text{T}_i\}_{i \in I}) \qquad \forall i \in I \cdot \Gamma \vdash P_i : \text{T}_i}{\Gamma \vdash \prod_{i \in I} \text{p}_i \triangleleft P_i : \{\text{p}_i : \text{T}_i\}_{i \in I}}$$

This rule says contexts assigned in a typing judgment must be safe, so at least $\mathscr{M}_{\text{bad}}$ is untypable. Still $\mathscr{M}'_{\text{bad}}$ *is* typable, but perhaps that's fine: different applications have different thresholds for "going wrong". Certain systems may demand termination ($\mathbb{P}_{\text{Term}}(\Delta) = 1$), while others may tolerate a weaker guarantee (say, $\mathbb{P}_{\text{DF}}(\Delta) \geq 0.95$). This motivates a procedure for verifying probabilistic properties of typing contexts.

Chapter 3

Probabilistic model checking

In the previous chapter, we argued for the need to verify typing contexts. This chapter explores one way to do so via *probabilistic model checking*.

Model checking is a technique for automated formal verification [Baier and Katoen, 2008]. We first construct a *model* – an abstract representation of the system under consideration – and express the desired properties in a formal system of logic. We then use an algorithm to verify whether the properties hold in the model.

Prior work [Scalas and Yoshida, 2019] employed the *modal μ-calculus* and the mCRL2 model-checker [Bunte et al., 2019] to verify typing contexts in a non-probabilistic setting. We instead use PRISM [Kwiatkowska et al., 2011], a *probabilistic* model checker developed here at Oxford.

3.1 PRISM semantics

To enable a precise discussion, we first formally define the relevant subset of the PRISM language. We closely follow the PRISM documentation[1] and [Carbone and Veschetti, 2024], with some modifications to syntax and a more precise discussion of states.

Definition 3.1.1. The syntax of PRISM *models* is given by

$$M, N ::= \{C_i\}_{i \in I} \qquad \text{(module)}$$

$$\mid M \parallel N \qquad \text{(composition)}$$

[1] https://www.prismmodelchecker.org/doc/

$$
\begin{aligned}
C &::= [\alpha]\, g \to \textstyle\sum_{i\in I} p_i \,:\, u_i && \text{(command, with } \textstyle\sum_{i\in I} p_i = 1 \text{ and } p_i \in [0,1]\text{)}\\
g, g' &::= (x = v) \mid \neg g \mid g \wedge g' \mid g \vee g' && \text{(guard)}\\
u, u' &::= (x' = v) \mid u \wedge u' && \text{(update)}\\
v, v' &::= v \odot v' && \text{(arithmetic operations, with } \odot \in \{+, -, \times, \div\}\text{)}\\
&\quad \mid x \mid n \in \mathbb{Z} && \text{(variable or constant integer)}
\end{aligned}
$$

PRISM models are a *composition* of *modules*, each containing *commands* C_i. Each command has an *action* α, *guard* g, and updates u_i, each with probability p_i. Guards and updates operate on *states* throughout the execution of the model.

Definition 3.1.2. A *state* S is a mapping $\{x_i \mapsto n_i\}_{i\in I}$ from variables x_i to constants n_i. *Updates* override mappings: $S[(x' = n)] = (S \setminus \{x \mapsto S(x)\}) \cup \{x \mapsto n\}$.

Remark. PRISM distinguishes equality from updates by denoting them as $(x = v)$ and $(x' = v)$, respectively. In the latter, we update x, not x'!

For the sake of presentational hygiene, we treat zero-valued variables differently:

Notation 3.1.1. For all $x \notin \mathrm{dom}(S)$, we treat $S(x) = 0$.

Notation 3.1.2. We treat $S \cup \{x \mapsto 0\} \equiv S$.

Hence we succinctly describe the state $\{x \mapsto 42, y \mapsto 0\}$ as $\{x \mapsto 42\}$. Since PRISM initialises every integer variable to 0, the *initial state* of any model is $\emptyset$.

Definition 3.1.3. A state S is *a state for* model M if every $x \in \mathrm{dom}(S)$ appears in some guard or update in M. We call (M, S) a *model-state pair*.

Notation 3.1.3. We write M as a shorthand for $(M, \emptyset)$.

We now define the semantics of PRISM. Following [Carbone and Veschetti, 2024], we first introduce a relation $M \rightsquigarrow C$ (pronounced *"M has command C"*) to capture the effects of composition.

$$\text{[M-MODULE]}\quad \frac{[\alpha]\, g \to \sum_{i\in I} p_i \,:\, u_i \in \{C_i\}_{i\in I}}{\{C_i\}_{i\in I} \rightsquigarrow [\alpha]\, g \to \sum_{i\in I} p_i \,:\, u_i}$$

$$\text{[M-MOVE]}\quad \frac{M \rightsquigarrow [\alpha]\, g \to \sum_{i\in I} \lambda_i \,:\, u_i \qquad \forall J, p'_j, u'_j, g \cdot N \not\rightsquigarrow [\alpha]\, g' \to \sum_{j\in J} p'_j \,:\, u'_j}{M \,\|\, N \rightsquigarrow [\alpha]\, g \to \sum_{i\in I} p_i \,:\, u_i}$$

$$\text{[M-SYNC]}\quad \frac{M \rightsquigarrow [\alpha]\, g \to \sum_{i\in I} p_i \,:\, u_i \qquad N \rightsquigarrow [\alpha]\, g' \to \sum_{j\in J} p'_j \,:\, u'_j}{M \,\|\, N \rightsquigarrow [\alpha]\, g \wedge g' \to \sum_{i,j} p_i * p'_j \,:\, u_i \wedge u'_j}$$

Figure 3.1: Rules for $\rightsquigarrow$.

$$\text{[S-STEP]}\quad \frac{M \rightsquigarrow [\alpha]\, g \to \sum_{i\in I} p_i \,:\, u_i \qquad S \vdash g}{(M, S) \xrightarrow[p_i]{\alpha} (M, S[u_i])}$$

Figure 3.2: Transition rule for PRISM model-state pairs.

The rules for $\rightsquigarrow$ are shown in Figure 3.1. [M-MODULE] says a model with one module inherits its commands. If M has action α and N does not, [M-MOVE] keeps M's command in their composition $M \,\|\, N$. If both contain α, [M-SYNC] synchronises them, executing both commands together.

Based on this, we define the transition relation $\xrightarrow[p]{\alpha}$ between model–state pairs (Figure 3.2). Its sole rule [S-STEP] does what we expect: if M has a command guarded by g, and state S satisfies g ($S \vdash g$), then with probability p_i, we apply the update u_i.

Example 3.1.1. Consider the PRISM model

$$\begin{aligned} M_{\text{ex}} = \{\ & [\alpha]\,(x = 0) \to 0.7 \,:\, (x' = 1) + 0.3 \,:\, (x' = 2), \\ & [\beta]\,(x = 1) \to 1 \,:\, (x' = 0)\ \} \\ \|\ \{\ & [\alpha]\,(y = 0) \to 0.2 \,:\, (y' = 1) + 0.8 \,:\, (y' = 2), \\ & [\alpha]\,(y = 1) \to 1 \,:\, (y' = 0), \\ & [\gamma]\,(y = 2) \to 1 \,:\, (y' = 0)\ \}. \end{aligned}$$

Then by the rules in Figure 3.1,

$$M_{\text{ex}} \rightsquigarrow [\alpha]\,(x=0)\wedge(y=0) \to 0.14 : (x'=1)\wedge(y'=1) + 0.56 : (x'=1)\wedge(y'=2)$$
$$+\, 0.06 : (x'=2)\wedge(y'=1) + 0.24 : (x'=2)\wedge(y'=2)$$
$$M_{\text{ex}} \rightsquigarrow [\alpha]\,(x=0)\wedge(y=1) \to 0.7 : (x'=1)\wedge(y'=0) + 0.3 : (x'=2)\wedge(y'=0)$$
$$M_{\text{ex}} \rightsquigarrow [\beta]\,(x=1) \to 1 : (x'=0)$$
$$M_{\text{ex}} \rightsquigarrow [\gamma]\,(y=2) \to 1 : (y'=0)$$

Hence by [S-MOVE], some possible transitions are

$$M_{\text{ex}} \xrightarrow{\alpha}_{0.56} (M_{\text{ex}}, \{x \mapsto 1, y \mapsto 2\}) \xrightarrow{\gamma}_{1} (M_{\text{ex}}, \{x \mapsto 1\}) \xrightarrow{\beta}_{1} M_{\text{ex}} \text{ and}$$
$$M_{\text{ex}} \xrightarrow{\alpha}_{0.14} (M_{\text{ex}}, \{x \mapsto 1, y \mapsto 1\}) \xrightarrow{\beta}_{1} (M_{\text{ex}}, \{y \mapsto 1\}) \xrightarrow{\alpha}_{0.3} (M_{\text{ex}}, \{x \mapsto 2\}).$$

3.2 A first look at translation

To verify typing contexts using PRISM, we need a systematic procedure for translating them into PRISM models. Before we examine the details of the encoding, though, let us grow our intuition with a simple example.

Example 3.2.1. In this section, we consider the following context:

$$\Delta_{\text{tran}} = \mathsf{p} : \mathsf{q}\oplus\begin{cases} 0.2 : \ell_1.\mu\mathbf{t}.\mathsf{q}\oplus\ell_1.\mathbf{t} \\ 0.3 : \ell_2.\mathsf{q}\oplus\ell_2.\mathsf{end} \\ 0.5 : \ell_3.\mathsf{end}, \end{cases} \quad \mathsf{q} : \mathsf{p}\&\begin{cases} \ell_1.\mu\mathbf{t}.\mathsf{p}\&\ell_1.\mathbf{t} \\ \ell_2.\mathsf{p}\&\ell_2.\mathsf{end} \\ \ell_3.\mathsf{end} \end{cases}$$

With probability 0.2, p sends ℓ_1 to q indefinitely; w.p. 0.3, ℓ_2 twice; and w.p. 0.5, ℓ_3 once.

3.2.1 Dealing with states

MPST is a term rewriting system, whereas PRISM is state-based. Thus, we must first determine how to map terms into states. Our plan of attack: translate each participant in the typing context into its own module, with variable S_{p} tracking p's current position in its type. We then compose these modules into a single model.

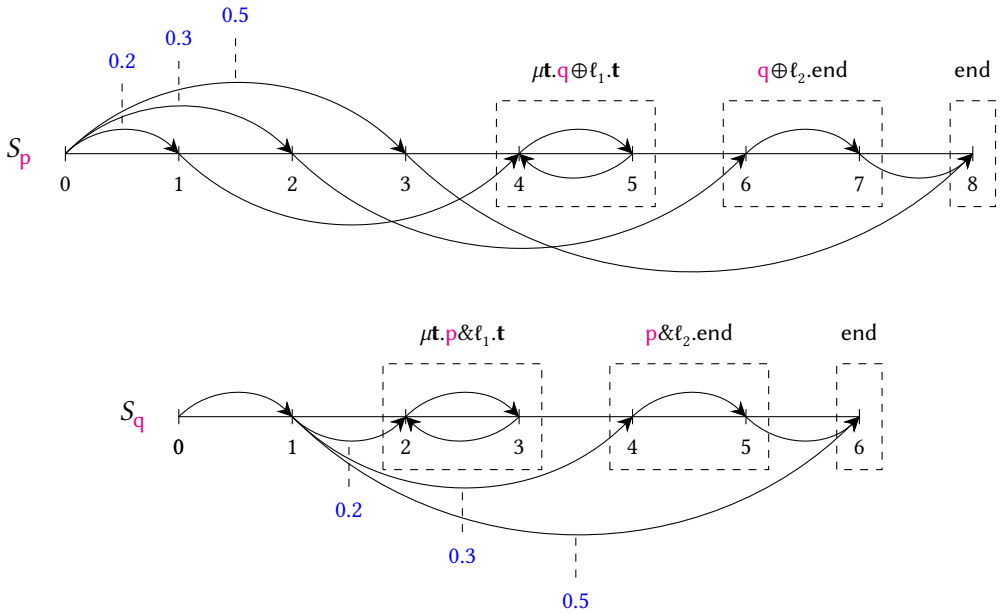

Figure 3.3: Illustration of states in the encoding of Δ_{tran} (Example 3.2.1). Points on the number line represent values of S_p and S_q, and arrows represent transitions. Unlabelled transitions have probability 1.

Figure 3.3 shows the state encoding for Δ_{tran}. Each typing context reduction corresponds to *two* PRISM transitions. For internal choice $p \oplus_{i \in I} p_i : \ell_i.T_i$, we first make a probabilistic transition to one of $|I|$ intermediary states, then move to the state for T_i. For external choice $p \&_{i \in I} \ell_i.T_i$, we first move to a shared intermediary state, then branch to each T_i.

We deal with recursion in the obvious way: by looping back to the state of the corresponding μ-binding. A special state handles end, to which all terminated participants are sent.

Definition 3.2.1. The *state space* of a type T, written $SS(T)$, is defined inductively:

$$SS(\text{end}) = 0$$

$$SS(\mu \mathbf{t}.T') = SS(T')$$

$$SS(\mathbf{t}) = 0$$

$$SS(p \&_{i \in I} \ell_i(B_i).T_i) = 2 + \sum_{i \in I} SS(T_i)$$

$$SS(p \oplus_{i \in I} p_i : \ell_i\langle B_i \rangle.T_i) = 1 + |I| + \sum_{i \in I} SS(T_i)$$

Each variable S_p in the encoding of $\Delta = p : T, \Delta'$ has a value between 0 and $SS(T)$, inclusive.

Example 3.2.2. For the context Δ_{tran} given in Example 3.2.1,

$$\begin{aligned}
SS(\Delta_{\text{tran}}(\mathsf{p})) &= 1 + 3 + SS(\mu\mathbf{t}.\mathsf{q}\oplus\ell_1.\mathbf{t}) + SS(\mathsf{q}\oplus\ell_2.\mathsf{end}) + SS(\mathsf{end}) \\
&= 4 + SS(\mathsf{q}\oplus\ell_1.\mathbf{t}) + (1 + 1 + SS(\mathsf{end})) + 0 \\
&= 4 + (1 + 1 + SS(\mathbf{t})) + (2 + 0) + 0 \\
&= 4 + 2 + 2 = 8 \\
SS(\Delta_{\text{tran}}(\mathsf{q})) &= 2 + SS(\mu\mathbf{t}.\mathsf{p}\&\ell_1.\mathbf{t}) + SS(\mathsf{p}\&\ell_2.\mathsf{end}) + SS(\mathsf{end}) \\
&= 2 + SS(\mathsf{p}\&\ell_1.\mathbf{t}) + (2 + SS(\mathsf{end})) + 0 \\
&= 2 + (2 + SS(\mathbf{t})) + (2 + 0) \\
&= 2 + 2 + 2 = 6,
\end{aligned}$$

and indeed $S_{\mathsf{p}} \in [0, 8]$ and $S_{\mathsf{q}} \in [0, 6]$ in Figure 3.3.

3.2.2 Synchronising modules

With an understanding of the structure of state transitions, we turn to the translation of Δ_{tran} (Example 3.2.1). From this, we will study how communication is encoded.

Example 3.2.3. Translating Δ_{tran} from Example 3.2.1 yields the model

$$M_{\text{tran}} = M_{\text{tran},\mathsf{p}} \parallel M_{\text{tran},\mathsf{q}},$$

where the modules $M_{\text{tran},\mathsf{p}}$ and $M_{\text{tran},\mathsf{q}}$ are given by

$$\begin{aligned}
M_{\text{tran},\mathsf{p}} = \{\ & [\mathsf{p} :: \mathsf{q}](S_{\mathsf{p}} = 0) \to 0.2 : (S_{\mathsf{p}}' = 1) \\
& \qquad + 0.3 : (S_{\mathsf{p}}' = 2) \\
& \qquad + 0.5 : (S_{\mathsf{p}}' = 3), \\
& [\mathsf{p} :: \mathsf{q} :: \ell_1](S_{\mathsf{p}} = 1) \to 1 : (S_{\mathsf{p}}' = 4), \\
& [\mathsf{p} :: \mathsf{q} :: \ell_2](S_{\mathsf{p}} = 2) \to 1 : (S_{\mathsf{p}}' = 6), \\
& [\mathsf{p} :: \mathsf{q} :: \ell_3](S_{\mathsf{p}} = 3) \to 1 : (S_{\mathsf{p}}' = 8), \\
& [\mathsf{p} :: \mathsf{q}](S_{\mathsf{p}} = 4) \to 1 : (S_{\mathsf{p}}' = 5), \\
& [\mathsf{p} :: \mathsf{q} :: \ell_1](S_{\mathsf{p}} = 5) \to 1 : (S_{\mathsf{p}}' = 4),
\end{aligned}$$

$$[\mathsf{p} :: \mathsf{q}]\,(S_\mathsf{p} = 6) \to 1 : (S_\mathsf{p}' = 7),$$
$$[\mathsf{p} :: \mathsf{q} :: \ell_2]\,(S_\mathsf{p} = 7) \to 1 : (S_\mathsf{p}' = 8)\ \}$$

$$\begin{aligned} M_{\text{tran},\mathsf{q}} = \{\ & [\mathsf{p} :: \mathsf{q}]\,(S_\mathsf{q} = 0) \to 1 : (S_\mathsf{q}' = 1), \\ & [\mathsf{p} :: \mathsf{q} :: \ell_1]\,(S_\mathsf{q} = 1) \to 1 : (S_\mathsf{q}' = 2), \\ & [\mathsf{p} :: \mathsf{q} :: \ell_2]\,(S_\mathsf{q} = 1) \to 1 : (S_\mathsf{q}' = 4), \\ & [\mathsf{p} :: \mathsf{q} :: \ell_3]\,(S_\mathsf{q} = 1) \to 1 : (S_\mathsf{q}' = 6), \\ & [\mathsf{p} :: \mathsf{q}]\,(S_\mathsf{q} = 2) \to 1 : (S_\mathsf{q}' = 3), \\ & [\mathsf{p} :: \mathsf{q} :: \ell_1]\,(S_\mathsf{q} = 3) \to 1 : (S_\mathsf{q}' = 3), \\ & [\mathsf{p} :: \mathsf{q}]\,(S_\mathsf{q} = 4) \to 1 : (S_\mathsf{q}' = 5), \\ & [\mathsf{p} :: \mathsf{q} :: \ell_2]\,(S_\mathsf{q} = 5) \to 1 : (S_\mathsf{q}' = 6)\ \}. \end{aligned}$$

Hence, a possible reduction sequence is

$$\begin{aligned} M_{\text{tran}} &\xrightarrow{\mathsf{p}::\mathsf{q}}_{0.3} (M_{\text{tran}}, \{S_\mathsf{p} \mapsto 2, S_\mathsf{q} \mapsto 1\}) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell_2}_{1} (M_{\text{tran}}, \{S_\mathsf{p} \mapsto 6, S_\mathsf{q} \mapsto 4\}) \\ &\xrightarrow{\mathsf{p}::\mathsf{q}}_{1} (M_{\text{tran}}, \{S_\mathsf{p} \mapsto 7, S_\mathsf{q} \mapsto 5\}) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell_2}_{1} (M_{\text{tran}}, \{S_\mathsf{p} \mapsto 8, S_\mathsf{q} \mapsto 6\}). \end{aligned}$$

This corresponds to the context reductions

$$\Delta_{\text{tran}} \xrightarrow{(\mathsf{p},\mathsf{q})\ell_2}_{0.3} \mathsf{p} : \mathsf{q} \oplus \ell_2.\mathsf{end}, \mathsf{q} : \mathsf{p} \,\&\, \ell_2.\mathsf{end} \xrightarrow{(\mathsf{p},\mathsf{q})\ell_2}_{1} \mathsf{p} : \mathsf{end}, \mathsf{q} : \mathsf{end}.$$

Earlier we mentioned that each context reduction corresponds to two transitions in PRISM. We now make this notion more precise: $\xrightarrow{(\mathsf{p},\mathsf{q})\ell}_{p}$ maps to $\xrightarrow{\mathsf{p}::\mathsf{q}}_{p}$ followed by $\xrightarrow{\mathsf{p}::\mathsf{q}::\ell}_{1}$.

The first transition lets p make a probabilistic choice. The second uses synchronisation to inform q of p's choice, allowing both to move to the correct next state. Crucially, only the action chosen by p can synchronise – no other transitions are possible.

3.3 A second look at translation

We now generalise the earlier example by defining a formal encoding function. We will first define a nearly-complete *inner* encoding $(\!|\cdot|\!)$, and then extend this with the *closure module* to

$$
\begin{aligned}
(\!|\varnothing|\!) &= \varnothing \\
(\!|\mathsf{p} : \mathsf{T}, \Delta|\!) &= \{\!|\mathsf{T}|\!\}_{(\mathsf{p},0,SS(\mathsf{T}),\varnothing)} \parallel (\!|\Delta|\!) \\
\{\!|\mathsf{end}|\!\}_{(\mathsf{p},n,m,f)} &= \varnothing \\
\{\!|\mathbf{t}|\!\}_{(\mathsf{p},n,m,f)} &= \varnothing \\
\{\!|\mu\mathbf{t}.\mathsf{T}|\!\}_{(\mathsf{p},n,m,f)} &= \{\!|\mathsf{T}|\!\}_{(\mathsf{p},n,m,f\cup\{\mathbf{t}\mapsto n\})} \\
\{\!|\mathsf{q}\&_{i\in I}\ell_i(\mathsf{B}_i).\mathsf{T}_i|\!\}_{(\mathsf{p},n,m,f)} &= [\mathsf{q} :: \mathsf{p}]\,(S_{\mathsf{p}} = n) \to 1 : (S'_{\mathsf{p}} = n+1) \\
&\quad \cup \bigcup_{i\in I} [\mathsf{q} :: \mathsf{p} :: \ell_i]\,(S_{\mathsf{p}} = n+1) \to \\
&\qquad 1 : \begin{cases} (S'_{\mathsf{p}} = m) & \text{if } \mathsf{T}_i = \mathsf{end} \\ (S'_{\mathsf{p}} = f(\mathbf{t})) & \text{if } \mathsf{T}_i = \mathbf{t} \\ (S'_{\mathsf{p}} = \mathsf{next}(i)) & \text{otherwise} \end{cases} \\
&\quad \cup \bigcup_{i\in I} \{\!|\mathsf{T}_i|\!\}_{(\mathsf{p},\mathsf{next}(i),m,f)} \\
&\text{where } \mathsf{next}(i) = n + 2 + \textstyle\sum_{\{j\in I \mid j<i\}} SS(\mathsf{T}_j) \\
\{\!|\mathsf{q}\oplus_{i\in I} p_i : \ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}'_i|\!\}_{(\mathsf{p},n,m,f)} &= [\mathsf{p} :: \mathsf{q}]\,(S_{\mathsf{p}} = n) \to \sum_{i\in I}(p_i : (S'_{\mathsf{p}} = n+i)) \\
&\quad \cup \bigcup_{i\in I} [\mathsf{p} :: \mathsf{q} :: \ell_i]\,(S_{\mathsf{p}} = n+i \to \\
&\qquad 1 : \begin{cases} (S'_{\mathsf{p}} = m) & \text{if } \mathsf{T}_i = \mathsf{end} \\ (S'_{\mathsf{p}} = f(\mathbf{t})) & \text{if } \mathsf{T}_i = \mathbf{t} \\ (S'_{\mathsf{p}} = \mathsf{next}(i)) & \text{otherwise} \end{cases} \\
&\quad \cup \bigcup_{i\in I} \{\!|\mathsf{T}_i|\!\}_{(\mathsf{p},\mathsf{next}(i),m,f)} \\
&\text{where } \mathsf{next}(i) = n + 1 + |I| + \textstyle\sum_{\{j\in I \mid j<i\}} SS(\mathsf{T}_j)
\end{aligned}
$$

Figure 3.4: The inner encoding function $(\!|\cdot|\!)$.

give the final encoding $[\![\cdot]\!]$.

3.3.1 The inner encoding

Definition 3.3.1. The inner encoding function $(\!|\cdot|\!)$ is given in Figure 3.4.

The first two lines say that a typing context is translated by composing the modules produced by the type-level encoding $\{\!|\mathsf{T}|\!\}_{(\mathsf{p},n,m,f)}$, which takes a type T being translated, participant p, current state n, end state m and a mapping f from variables to their state value. The remainder concerns the type-level translation.

No new commands are added for end or $\mathbf{t}$. For recursion $\mu\mathbf{t}.\mathsf{T}$, we extend the mapping with $\mathbf{t} \mapsto n$

and continue translating T. For & and ⊕, we follow our discussion in Section 3.2.1 by first moving to an intermediary state and proceeding to the continuation (using f if it's a variable).

3.3.2 Some loose ends

The inner encoding is almost correct, except it has slightly more transitions than we ought to have.

Example 3.3.1. Consider

$$\Delta_1 = \mathsf{p} : \mathsf{q} \oplus \ell.\mathsf{end}.$$

Then $\Delta_1 \nrightarrow$, but its inner translation

$$\begin{aligned}(\!|\Delta_1|\!) = \{\ &[\mathsf{p} :: \mathsf{q}]\,(S_{\mathsf{p}} = 0) \to 1 : (S'_{\mathsf{p}} = 1),\\ &[\mathsf{p} :: \mathsf{q} :: \ell]\,(S_{\mathsf{p}} = 1) \to 1 : (S'_{\mathsf{p}} = 2)\ \}\end{aligned}$$

allows

$$(\!|\Delta_1|\!) \xrightarrow{\mathsf{p}::\mathsf{q}}_1 ((\!|\Delta_1|\!), \{S_{\mathsf{p}} \mapsto 1\}) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}_1 ((\!|\Delta_1|\!), \{S_{\mathsf{p}} \mapsto 2\}).$$

Example 3.3.2. Consider

$$\Delta_2 = \mathsf{p} : \mathsf{q} \oplus \ell_1.\mathsf{end}, \mathsf{q} : \mathsf{p} \,\&\, \ell_2.\mathsf{end}.$$

Then again $\Delta_2 \nrightarrow$, but its inner translation

$$\begin{aligned}(\!|\Delta_2|\!) = \{\ &[\mathsf{p} :: \mathsf{q}]\,(S_{\mathsf{p}} = 0) \to 1 : (S'_{\mathsf{p}} = 1),\\ &[\mathsf{p} :: \mathsf{q} :: \ell_1]\,(S_{\mathsf{p}} = 1) \to 1 : (S'_{\mathsf{p}} = 2)\ \}\\ \|\ \{\ &[\mathsf{p} :: \mathsf{q}]\,(S_{\mathsf{q}} = 0) \to 1 : (S'_{\mathsf{q}} = 1),\\ &[\mathsf{p} :: \mathsf{q} :: \ell_2]\,(S_{\mathsf{q}} = 1) \to 1 : (S'_{\mathsf{q}} = 2)\ \}\end{aligned}$$

allows

$$\begin{aligned}(\!|\Delta_2|\!) &\xrightarrow{\mathsf{p}::\mathsf{q}}_1 ((\!|\Delta_2|\!), \{S_{\mathsf{p}} \mapsto 1, S_{\mathsf{q}} \mapsto 1\}) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell_1}_1 ((\!|\Delta_1|\!), \{S_{\mathsf{p}} \mapsto 2, S_{\mathsf{q}} \mapsto 1\})\\ (\!|\Delta_2|\!) &\xrightarrow{\mathsf{p}::\mathsf{q}}_1 ((\!|\Delta_2|\!), \{S_{\mathsf{p}} \mapsto 1, S_{\mathsf{q}} \mapsto 1\}) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell_2}_1 ((\!|\Delta_1|\!), \{S_{\mathsf{p}} \mapsto 1, S_{\mathsf{q}} \mapsto 2\}).\end{aligned}$$

More precisely, we treat $\xrightarrow{\mathsf{p}::\mathsf{q}}$ and $\xrightarrow{\mathsf{p}::\mathsf{q}::\ell}$ transitions as communications, implicitly assuming both p and q participate. However, in some cases, only one of p or q contains this action, and [M-MOVE] permits such transitions to occur without synchronisation.

Thus, we introduce a *closure module* that blocks these spurious transitions.

Definition 3.3.2. Let $\mathsf{actions}_\Delta(\mathsf{p})$ be the set of actions in $(\!|\mathsf{p} : \Delta(\mathsf{p})|\!)$. If $\mathsf{p} \notin \mathsf{dom}(\Delta)$, then we treat $\mathsf{actions}_\Delta(\mathsf{p}) = \emptyset$. We write $\mathsf{actions}(\Delta) = \bigcup_{\mathsf{p}\in\mathsf{dom}(\Delta)} \mathsf{actions}_\Delta(\mathsf{p})$.

Definition 3.3.3. We define $\mathsf{diff}_\Delta(\mathsf{p},\mathsf{q})$ as the *symmetric difference* of actions:

$$\mathsf{diff}_\Delta(\mathsf{p},\mathsf{q}) = \mathsf{actions}_\Delta(\mathsf{p}) \oplus \mathsf{actions}_\Delta(\mathsf{q}).$$

Definition 3.3.4. The *closure* module for the context Δ is given by

$$\begin{aligned}\mathsf{closure}(\Delta) = {} & \{\mathsf{disallow}(\mathsf{p} :: \mathsf{q}) \mid \mathsf{p} :: \mathsf{q} \in \mathsf{diff}_\Delta(\mathsf{p},\mathsf{q})\} \\ & \cup \{\mathsf{disallow}(\mathsf{p} :: \mathsf{q} :: \ell) \mid \mathsf{p} :: \mathsf{q} :: \ell \in \mathsf{diff}_\Delta(\mathsf{p},\mathsf{q})\},\end{aligned}$$

where $\mathsf{disallow}(\alpha)$ blocks α:

$$\mathsf{disallow}(\alpha) = [\alpha]\,\mathsf{false} \to 1 : ().$$

The () denotes a dummy update – it will never be applied.

Definition 3.3.5. The encoding $[\![\cdot]\!]$ of contexts into PRISM models is defined by

$$[\![\Delta]\!] = (\!|\Delta|\!) \parallel \mathsf{closure}(\Delta).$$

Example 3.3.3. Translating Δ_1 (Example 3.3.1) yields

$$\begin{aligned}[\![\Delta_1]\!] = {} & \{\ [\mathsf{p} :: \mathsf{q}]\,(S_\mathsf{p} = 0) \to 1 : (S'_\mathsf{p} = 1), \\ & \quad [\mathsf{p} :: \mathsf{q} :: \ell]\,(S_\mathsf{p} = 1) \to 1 : (S'_\mathsf{p} = 2)\ \} \\ & \parallel \{\ [\mathsf{p} :: \mathsf{q}]\,\mathsf{false} \to 1 : (), \\ & \quad [\mathsf{p} :: \mathsf{q} :: \ell]\,\mathsf{false} \to 1 : ()\ \},\end{aligned}$$

and thus $[\![\Delta_1]\!] \not\rightarrow$.

3.4 Correctness

We now address correctness of the encoding with respect to operational semantics. First, we show that the encoding is *sound* (Theorem 3.4.9): every typing context reduction corresponds to transitions in the encoded module. Next, we demonstrate that the encoding is *complete* (Theorem 3.4.15): PRISM transitions also correspond to typing context reductions.

3.4.1 Soundness

Taking a compositional approach, we first show that the inner encoding $(\!|\Delta|\!)$ is sound with respect to the internal transitions $\xrightarrow{\mathsf{p}:\mathsf{q}!\ell\langle \mathsf{B}\rangle}$ and $\xrightarrow{\mathsf{p}:\mathsf{q}?\ell(\mathsf{B})}$ (Lemma 3.4.5). We then use this result to demonstrate the soundness of $[\![\Delta]\!]$.

A result we ought to have is that $(\!|\cdot|\!)$ preserves behaviour under recursion unfolding: $(\!|\mathsf{p} : \mu\mathbf{t}.\mathsf{T}|\!)$ should be *equivalent* to $(\!|\mathsf{p} : \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!)$. But what does it mean for PRISM modules to be equivalent? Though their internal states and commands may differ, they should exhibit the same observable transition behaviour. This motivates the use of *bisimulation* to define equivalence.

Definition 3.4.1. Let (M_1, I_1) and (M_2, I_2) be PRISM model-state pairs. A binary relation R is a *bisimulation* if:

1. $(I_1, I_2) \in R$, and
2. For all $(S_1, S_2) \in R$, action α and $p \in [0, 1]$,

   $(M_1, S_1) \xrightarrow{\alpha}_p (M_1, S_1') \implies \exists S_2' \text{ s.t. } (M_2, S_2) \xrightarrow{\alpha}_p (M_2, S_2')$ and $(S_1', S_2') \in R$; and

   $(M_2, S_2) \xrightarrow{\alpha}_p (M_2, S_2') \implies \exists S_1' \text{ s.t. } (M_1, S_1) \xrightarrow{\alpha}_p (M_1, S_1')$ and $(S_1', S_2') \in R$.

We say (M_1, I_1) and (M_2, I_2) are *bisimilar* if such a relation R exists, and write $(M_1, I_1) \sim (M_2, I_2)$.

Bisimulation allows us to write a *coinductive* proof. To show two model-state pairs are bisimilar, we define a relation R and verify it satisfies the bisimulation conditions. To define R, though, we need a precise way of reasoning about the reductions available from a state. We therefore

$$
\begin{aligned}
\mathsf{source}_{\mathsf{p}:\mathsf{T}} &= \mathsf{source}'_{(\mathsf{p},0,\emptyset)}(\mathsf{T}) \\
&\quad \cup \{\{S_{\mathsf{p}} \mapsto SS(\mathsf{T})\} \mapsto (\mathsf{end}, 0, \emptyset)\} \\
\mathsf{source}'_{(\mathsf{p},n,f)}(\mathsf{end}) &= \emptyset \\
\mathsf{source}'_{(\mathsf{p},n,f)}(\mathbf{t}) &= \emptyset \\
\mathsf{source}'_{(\mathsf{p},n,f)}(\mu\mathbf{t}.\mathsf{T}) &= \mathsf{source}'_{(\mathsf{p},n,f\cup\{\mathbf{t}\mapsto\{S_{\mathsf{p}}\mapsto n\}\})}(\mathsf{T}) \\
\mathsf{source}'_{(\mathsf{p},n,f)}(\mathsf{q}\&_{i\in I}\ell_i(\mathsf{B}_i).\mathsf{T}_i) &= \{\{S_{\mathsf{p}} \mapsto n\} \mapsto (\mathsf{q}\&_{i\in I}\ell_i(\mathsf{B}_i).\mathsf{T}_i, 0, f), \\
&\qquad \{S_{\mathsf{p}} \mapsto n+1\} \mapsto (\mathsf{q}\&_{i\in I}\ell_i(\mathsf{B}_i).\mathsf{T}_i, 1, f)\} \\
&\quad \cup \bigcup_{i\in I} \mathsf{source}'_{(\mathsf{p},\mathsf{next}(i),f)}(\mathsf{T}_i) \\
&\text{where } \mathsf{next}(i) = n + 2 + \textstyle\sum_{\{j\in I | j<i\}} SS(\mathsf{T}_i)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{source}'_{(\mathsf{p},n,f)}(\mathsf{q}\oplus_{i\in I}p_i : \ell_i\langle\mathsf{B}_i\rangle.\mathsf{T}_i) &= \{\{S_{\mathsf{p}} \mapsto n\} \mapsto (\mathsf{q}\oplus_{i\in I}p_i : \ell_i\langle\mathsf{B}_i\rangle.\mathsf{T}_i, 0, f)\} \\
&\quad \cup \bigcup_{i\in I} \{\{S_{\mathsf{p}} \mapsto n+i\} \mapsto (\mathsf{q}\oplus_{i\in I}p_i : \ell_i\langle\mathsf{B}_i\rangle.\mathsf{T}_i, i, f)\} \\
&\quad \cup \bigcup_{i\in I} \mathsf{source}'_{(\mathsf{p},\mathsf{next}(i),f)}(\mathsf{T}_i) \\
&\text{where } \mathsf{next}(i) = n + 1 + |I| + \textstyle\sum_{\{j\in I | j<i\}} SS(\mathsf{T}_i)
\end{aligned}
$$

Figure 3.5: Definition of $\mathsf{source}_{\mathsf{p}:\mathsf{T}}(S)$.

introduce a *source function* that identifies the location of a state within its encoding.

Definition 3.4.2. The *source* of a state S within the module $(\!|\mathsf{p} : \mathsf{T}|\!)$, written $\mathsf{source}_{\mathsf{p}:\mathsf{T}}(S)$, is given in Figure 3.5.

The function outputs a triple $\mathsf{source}_{\mathsf{p}:\mathsf{T}}(S) = (\mathsf{T}', x, f)$. The first element T' is an end, $\oplus$, or $\&$ occurring as a *partial type* of T, representing the substructure of T associated with the state. The value x indicates the state's position: $x = 0$ marks the entry point, while higher values denote later stages in the transition. Finally, f maps variables to their corresponding states.

Example 3.4.1. Consider the type T defined by

$$\mathsf{T} = \mathsf{q}\oplus\begin{cases} 0.1 : \ell_1.\mathsf{end} \\ 0.9 : \ell_2.\mu\mathbf{t}.\mathsf{q}\oplus\ell_3.\mathbf{t} \end{cases}$$

The possible states of $(\!|\mathsf{p} : \mathsf{T}|\!)$ and their corresponding sources are illustrated in Figure 3.6.

The following proposition says that $\mathsf{source}_{\mathsf{p}:\mathsf{T}}(S)$ really does what we expect.

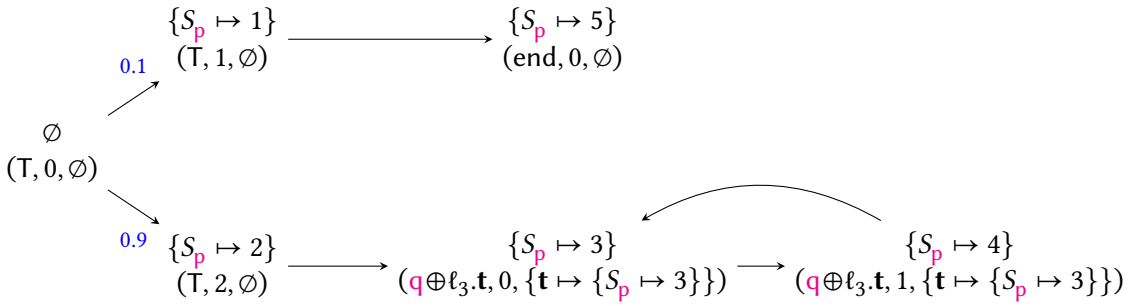

Figure 3.6: Transitions of $(\!|p : \mathsf{T}|\!)$ (Example 3.4.1). Nodes are labelled with their state S and corresponding $\mathsf{source}_{p:\mathsf{T}}(S)$ on separate lines. Unlabelled transitions have probability 1.

Proposition 3.4.1. *Let* T *be a type and* S *a state for* $(\!|p : \mathsf{T}|\!)$. *Then:*

1. *If* $\mathsf{source}_{p:\mathsf{T}}(S) = (\mathsf{end}, 0, \emptyset)$, *then* $((\!|p : \mathsf{T}|\!), S) \not\rightarrow$.
2. *If* $\mathsf{source}_{p:\mathsf{T}}(S) = (q \& _{i\in I} \ell_i(\mathsf{B}_i).\mathsf{T}_i, 0, f)$, *then* $((\!|p : \mathsf{T}|\!), S) \xrightarrow{q::p}_1 ((\!|p : \mathsf{T}|\!), S')$ *where* $\mathsf{source}_{p:\mathsf{T}}(S') = (q \& _{i\in I} \ell_i(\mathsf{B}_i).\mathsf{T}_i, 1, f)$.
3. *If* $\mathsf{source}_{p:\mathsf{T}}(S) = (q \& _{i\in I} \ell_i(\mathsf{B}_i).\mathsf{T}_i, 1, f)$, *then* $\forall i \in I \cdot ((\!|p : \mathsf{T}|\!), S) \xrightarrow{q::p::\ell_i}_1 ((\!|p : \mathsf{T}|\!), S')$ *where* $S' = f(\mathbf{t})$ *if* $\mathsf{T}_i = \mathbf{t}$, *and otherwise*

$$\mathsf{source}_{p:\mathsf{T}}(S') = \begin{cases} (\mathsf{end}, 0, \emptyset) & \text{if } \mathsf{T}_i = \mathsf{end} \\ (\mathsf{T}', 0, f \cup \bigcup_i \{\mathbf{t}_i \mapsto S'\}) & \text{if } \mathsf{T}_i = \mu\mathbf{t}_1 \cdots \mu\mathbf{t}_n.\mathsf{T}' \\ (\mathsf{T}_i, 0, f) & \text{otherwise.} \end{cases}$$

4. *If* $\mathsf{source}_{p:\mathsf{T}}(S) = (q \oplus _{i\in I} p_i : \ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}_i, 0, f)$, *then* $\forall i \in I \cdot ((\!|p : \mathsf{T}|\!), S) \xrightarrow{p::q}_{p_i} ((\!|p : \mathsf{T}|\!), S')$ *with* $\mathsf{source}_{p:\mathsf{T}}(S') = (q \oplus _{i\in I} p_i : \ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}_i, i, f)$.
5. *If* $\mathsf{source}_{p:\mathsf{T}}(S) = (q \oplus _{i\in I} p_i : \ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}_i, i, f)$ *with* $i > 0$, *then* $((\!|p : \mathsf{T}|\!), S) \xrightarrow{p::q::\ell_i}_1 ((\!|p : \mathsf{T}|\!), S')$ *where* $S' = f(\mathbf{t})$ *if* $\mathsf{T}_i = \mathbf{t}$, *and otherwise*

$$\mathsf{source}_{p:\mathsf{T}}(S') = \begin{cases} (\mathsf{end}, 0, \emptyset) & \text{if } \mathsf{T}_i = \mathsf{end} \\ (\mathsf{T}', 0, f \cup \bigcup_i \{\mathbf{t}_i \mapsto S'\}) & \text{if } \mathsf{T}_i = \mu\mathbf{t}_1 \cdots \mu\mathbf{t}_n.\mathsf{T}' \\ (\mathsf{T}_i, 0, f) & \text{otherwise.} \end{cases}$$

Moreover, $(\!|p : \mathsf{T}|\!), S)$ *cannot make any other transitions.*

Proof. By induction on T, walking through the recursive calls of $\mathsf{source}'_{(p,n,f)}(\mathsf{T})$ and $\{\!|\mathsf{T}|\!\}_{(p,n,m,f')}$ in parallel. In particular, we maintain the invariant that $\mathsf{source}'_{(p,n,f)}(\mathsf{T})$ is called iff $\{\!|\mathsf{T}|\!\}_{(p,n,m,f')}$ is called, where $f(\mathbf{t}) = \{S_p \mapsto f'(\mathbf{t})\}$ for all $\mathbf{t}$. □

Figure 3.7: Unfold correspondence for Example 3.4.2. Partial types that are unfold correspondent are annotated with the same circled number.

How shall we define the bisimulation relation for $(\!|\mathsf{p} : \mu\mathbf{t}.\mathsf{T}|\!) \sim (\!|\mathsf{p} : \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!)$? We'd like to relate equivalent partial types of $\mu\mathbf{t}.\mathsf{T}$ and $\mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}$, modulo unrolling of $\mathbf{t}$ (see Figure 3.7). We tackle this by introducing a novel *unfold correspondence* relation.

Definition 3.4.3. The *unfold correspondence* of two partial types *with respect to* T *and* $\mathbf{t}$, written $\sqsubseteq_{\mathsf{T},\mathbf{t}}$, is the least relation satisfying the following rules.

$$\frac{}{\mathsf{T}' \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{T}'}\ [\textsc{uc-refl}] \qquad \frac{}{\mathbf{t} \sqsubseteq_{\mathsf{T},\mathbf{t}} \mu\mathbf{t}.\mathsf{T}}\ [\textsc{uc-t}] \qquad \frac{\mathbf{t}' \neq \mathbf{t} \qquad \mathsf{T}_1 \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{T}_2}{\mu\mathbf{t}'.\mathsf{T}_1 \sqsubseteq_{\mathsf{T},\mathbf{t}} \mu\mathbf{t}'.\mathsf{T}_2}\ [\textsc{uc-rec}]$$

$$\frac{\forall i \in I : \mathsf{T}_i \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{T}'_i}{\mathsf{q}\oplus_{i\in I}\mathsf{p}_i : \ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}_i \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{q}\oplus_{i\in I}\mathsf{p}_i : \ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}'_i}\ [\textsc{uc-out}] \qquad \frac{\forall i \in I : \mathsf{T}_i \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{T}'_i}{\mathsf{q}\&_{i\in I}\ell_i(\mathsf{B}_i).\mathsf{T}_i \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{q}\&_{i\in I}\ell_i(\mathsf{B}_i).\mathsf{T}'_i}\ [\textsc{uc-in}]$$

Example 3.4.2. Let $\mathsf{T} = \mathsf{q} \oplus \{0.5 : \ell_1.\mathbf{t}, 0.5 : \ell_2.\mathsf{end}\}$. The following derivation tree shows $\mathsf{T} \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}$:

$$\frac{\dfrac{}{\mathbf{t} \sqsubseteq_{\mathsf{T},\mathbf{t}} \mu\mathbf{t}.\mathsf{q}\oplus\{0.5 : \ell_1.\mathbf{t}, 0.5 : \ell_2.\mathsf{end}\}}\ [\textsc{uc-t}] \qquad \dfrac{}{\mathsf{end} \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{end}}\ [\textsc{uc-refl}]}{\mathsf{q}\oplus\{0.5 : \ell_1.\mathbf{t}, 0.5 : \ell_2.\mathsf{end}\} \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{q}\oplus\{0.5 : \ell_1.\mu\mathbf{t}.\mathsf{q}\oplus\{0.5 : \ell_1.\mathbf{t}, 0.5 : \ell_2.\mathsf{end}\}, 0.5 : \ell_2.\mathsf{end}\}}\ [\textsc{uc-out}]$$

Figure 3.7 illustrates all unfold correspondences between partial types of $\mu\mathbf{t}.\mathsf{T}$ and $\mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}$.

Lemma 3.4.2. *For all types* T, T' *and variable* $\mathbf{t}$, $\mathsf{T}' \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{T}'\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}$.

Proof. Straightforward induction on the structure of T'. □

Lemma 3.4.3. *For every type* T, $(\!|\mathsf{p} : \mu\mathbf{t}.\mathsf{T}|\!) \sim (\!|\mathsf{p} : \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!)$.

Proof. Suppose S_1 and S_2 are states for $(\!|\mathsf{p} : \mu\mathbf{t}.\mathsf{T}|\!)$ and $(\!|\mathsf{p} : \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!)$ respectively, and let

$(\mathsf{T}_1, x_1, f_1) = \mathsf{source}_{\mathsf{p}:\mu\mathbf{t}.\mathsf{T}}(S_1)$ and $(\mathsf{T}_2, x_2, f_2) = \mathsf{source}_{\mathsf{p}:\mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}}(S_2)$.

Define a binary relation R to be the least relation such that $(S_1, S_2) \in R$ if:

- $x_1 = x_2$, and
- $\mathsf{T}_1 \sqsubseteq_{\mathsf{T},\mathbf{t}} T_2$, and
- $\forall \mathbf{t}' \in \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2) \cdot (f_1(\mathbf{t}'), f_2(\mathbf{t}')) \neq (S_1, S_2) \implies (f_1(\mathbf{t}'), f_2(\mathbf{t}')) \in R$.

We will argue that R is a bisimulation relation.

Take any $(S_1, S_2) \in R$, and again let $(\mathsf{T}_1, x, f_1) = \mathsf{source}_{\mathsf{p}:\mu\mathbf{t}.\mathsf{T}}(S_1)$ and $(\mathsf{T}_2, x, f_2) = \mathsf{source}_{\mathsf{p}:\mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}}(S_2)$. Since $\mathsf{T}_1 \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{T}_2$, both types must be end, $\oplus$, or & (the latter two sharing the same participants, labels, basic types, and probabilities).

In the first case, both $(\!|\mathsf{p} \,:\, \mu\mathbf{t}.\mathsf{T}|\!), S_1) \not\rightarrow$ and $(\!|\mathsf{p} \,:\, \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!), S_2) \not\rightarrow$.

For the second case, suppose $\mathsf{T}_1 = \mathsf{q}\oplus_{i\in I} p_i : \ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}'_i$, and $\mathsf{T}_2 = \mathsf{q}\oplus_{i\in I} p_i : \ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}''_i$. There are two cases to consider, depending on the value of x.

1. If $x = 0$, then by Proposition 3.4.1, the only possible transitions are

$$(\!|\mathsf{p} \,:\, \mu\mathbf{t}.\mathsf{T}|\!), S_1) \xrightarrow{\mathsf{p}::\mathsf{q}}_{p_k} (\!|\mathsf{p} \,:\, \mu\mathbf{t}.\mathsf{T}|\!), S'_1)$$
$$(\!|\mathsf{p} \,:\, \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!), S_2) \xrightarrow{\mathsf{p}::\mathsf{q}}_{p_k} (\!|\mathsf{p} \,:\, \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!), S'_2)$$

   for all $k \in I$, with $\mathsf{source}_{\mathsf{p}:\mu\mathbf{t}.\mathsf{T}}(S'_1) = (\mathsf{T}_1, k, f_1)$ and $\mathsf{source}_{\mathsf{p}:\mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}}(S'_2) = (\mathsf{T}_2, k, f_1)$. Since $(S_1, S_2) \in R$, we already have $\mathsf{T}_1 \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{T}_2$; therefore, $(S'_1, S'_2) \in R$ also.

2. If $x > 0$, then the only possible transitions are

$$(\!|\mathsf{p} \,:\, \mu\mathbf{t}.\mathsf{T}|\!), S_1) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell_x}_{1} (\!|\mathsf{p} \,:\, \mu\mathbf{t}.\mathsf{T}|\!), S'_1)$$
$$(\!|\mathsf{p} \,:\, \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!), S_2) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell_x}_{1} (\!|\mathsf{p} \,:\, \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!), S'_2).$$

   We consider the structurally possible cases of T'_x and T''_x. By Proposition 3.4.1:

   (a) if $\mathsf{T}'_x = \mathsf{T}''_x = \mathsf{end}$, then $\mathsf{source}_{\mathsf{p}:\mu\mathbf{t}.\mathsf{T}}(S'_1) = \mathsf{source}_{\mathsf{p}:\mu\mathbf{t}.\mathsf{T}}(S'_2) = (\mathsf{end}, 0, \emptyset)$;

   (b) if $\mathsf{T}'_x = \mathsf{T}''_x = \mathbf{t}'$ (potentially $\mathbf{t}' = \mathbf{t}$), then $S'_1 = f_1(\mathbf{t})$ and $S'_2 = f_2(\mathbf{t})$;

   (c) if $\mathsf{T}'_x = \mu\mathbf{t}'_1 \cdots \mu\mathbf{t}'_n.\mathsf{T}'$ and $\mathsf{T}''_x = \mu\mathbf{t}'_1 \cdots \mu\mathbf{t}'_n.\mathsf{T}''$, then $\mathsf{source}_{\mathsf{p}:\mu\mathbf{t}.\mathsf{T}}(S'_1) = (\mathsf{T}', 0, f_1 \cup \bigcup_i \{\mathbf{t}'_i \mapsto$

$S_1'\}$) and $\mathsf{source}_{\mathsf{p}:\mu\mathbf{t}.\mathsf{T}}(S_2') = (\mathsf{T}'', 0, f_2 \cup \bigcup_i \{\mathbf{t}_i' \mapsto S_2'\})$, and $\mathsf{T}' \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{T}''$ by the definition of $\sqsubseteq_{\mathsf{T},\mathbf{t}}$;

(d) if $\mathsf{T}_x' = \mathbf{t}$ and $\mathsf{T}_x'' = \mu\mathbf{t}.\mathsf{T}$, then $\mathsf{source}_{\mathsf{p}:\mu\mathbf{t}.\mathsf{T}}(S_1') = (\mathsf{T}, 0, \{\mathbf{t} \mapsto S_1'\})$ (in fact, $S_1' = \emptyset$) and $\mathsf{source}_{\mathsf{p}:\mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}}(S_2') = (\mathsf{T}, 0, f_2 \cup \{\mathbf{t} \mapsto S_2'\})$;

(e) otherwise, $\mathsf{source}_{\mathsf{p}:\mu\mathbf{t}.\mathsf{T}}(S_x') = (\mathsf{T}_x', 0, f_1)$ and $\mathsf{source}_{\mathsf{p}:\mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}}(S_2') = (\mathsf{T}_x'', 0, f_2)$, with $\mathsf{T}_x' \sqsubseteq_{\mathsf{T},\mathbf{t}} \mathsf{T}_x''$ by the definition of $\sqsubseteq_{\mathsf{T},\mathbf{t}}$.

Thus, in all cases, $(S_1', S_2') \in R$.

The third case is analogous. Finally, $\mathsf{source}_{\mathsf{p}:\mu\mathbf{t}.\mathsf{T}}(\emptyset) = (\mathsf{T}, 0, \{\mathbf{t} \mapsto \emptyset\})$ and $\mathsf{source}_{\mathsf{p}:\mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}}(\emptyset) = (\mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}, 0, \emptyset)$, so applying Lemma 3.4.2, $(\emptyset, \emptyset) \in R$. Hence R is a bisimulation relation, and we conclude $(\!|\mathsf{p} : \mu\mathbf{t}.\mathsf{T}|\!) \sim (\!|\mathsf{p} : \mathsf{T}\{\mu\mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!)$. □

We now prove a simple result about modules that enter the state space of a closed type.

> **Lemma 3.4.4.** *Let T, T' be (closed) types, and S a state for $(\!|\mathsf{p} : \mathsf{T}|\!)$ such that $\mathsf{source}_{\mathsf{p}:\mathsf{T}}(S) = (\mathsf{T}', 0, f)$. Then $((\!|\mathsf{p} : \mathsf{T}|\!), S) \sim (\!|\mathsf{p} : \mathsf{T}'|\!)$.*

Proof. Let S_1, S_2 be states for $(\!|\mathsf{p} : \mathsf{T}|\!)$ and $(\!|\mathsf{p} : \mathsf{T}'|\!)$, respectively. Let $(\mathsf{T}_1, x_1, f_1) = \mathsf{source}_{\mathsf{p}:\mathsf{T}}(S_1)$ and $(\mathsf{T}_2, x_2, f_2) = \mathsf{source}_{\mathsf{p}:\mathsf{T}'}(S_2)$. We define a relation R as the least relation such that:

1. $(\mathsf{T}_1, x_1) = (\mathsf{T}_2, x_2)$;
2. $\forall \mathbf{t}' \in \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2) \cdot (f_1(\mathbf{t}'), f_2(\mathbf{t}')) \neq (S_1, S_2) \implies (f_1(\mathbf{t}'), f_2(\mathbf{t}')) \in R$.

We proceed as in Lemma 3.4.3 to argue that R is a bisimulation relation. Notably, $(S, \emptyset) \in R$. □

This enables us to prove the soundness of $(\!|\mathsf{p} : \mathsf{T}|\!)$.

> **Lemma 3.4.5.**
>
> 1. *If* $\mathsf{p} : \mathsf{T} \xrightarrow{\mathsf{p}:\mathsf{q}!\ell\langle\mathsf{B}\rangle}_{p} \mathsf{p} : \mathsf{T}'$, *then*
>
> $$\exists S_1, S_2 \cdot (\!|\mathsf{p} : \mathsf{T}|\!) \xrightarrow{\mathsf{p}::\mathsf{q}}_{p} ((\!|\mathsf{p} : \mathsf{T}|\!), S_1) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}_{1} ((\!|\mathsf{p} : \mathsf{T}|\!), S_2) \sim (\!|\mathsf{p} : \mathsf{T}'|\!).$$
>
> 2. *If* $\mathsf{p} : \mathsf{T} \xrightarrow{\mathsf{p}:\mathsf{q}?\ell(\mathsf{B})}_{1} \mathsf{p} : \mathsf{T}'$, *then*
>
> $$\exists S_1, S_2 \cdot (\!|\mathsf{p} : \mathsf{T}|\!) \xrightarrow{\mathsf{q}::\mathsf{p}}_{1} ((\!|\mathsf{p} : \mathsf{T}|\!), S_1) \xrightarrow{\mathsf{q}::\mathsf{p}::\ell}_{1} ((\!|\mathsf{p} : \mathsf{T}|\!), S_2) \sim (\!|\mathsf{p} : \mathsf{T}'|\!).$$

Proof.

1. We proceed by induction on the rules of $\to$.

   Case [CT-OUT] Let $\mathsf{T} = \mathsf{q} \oplus_{i \in I} p_i : \ell_i\langle \mathsf{B}_i \rangle.\mathsf{T}_i$, and suppose $\mathsf{p} : \mathsf{T} \xrightarrow{\mathsf{p}:\mathsf{q}!\ell_k\langle \mathsf{B}_k \rangle}_{p_k} \mathsf{p} : \mathsf{T}_k$ for some $k \in I$. Then by the definition of $(\!| \cdot |\!)$,

   $$\exists S_1, S_2 \cdot (\!|\mathsf{p} : \mathsf{T}|\!) \xrightarrow{\mathsf{p}::\mathsf{q}}_{p_k} ((\!|\mathsf{p} : \mathsf{T}|\!), S_1) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell_k}_{1} ((\!|\mathsf{p} : \mathsf{T}|\!), S_2).$$

   Since T_k is closed, $((\!|\mathsf{p} : \mathsf{T}|\!), S_2) \sim (\!|\mathsf{p} : \mathsf{T}_k|\!)$ by Lemma 3.4.4.

   Case [CT-REC] Suppose $\mathsf{p} : \mu \mathbf{t}.\mathsf{T} \xrightarrow{\mathsf{p}:\mathsf{q}!\ell\langle \mathsf{B} \rangle}_{p} \mathsf{p} : \mathsf{T}'$. By the inductive hypothesis,

   $$\exists S_1, S_2 \cdot (\!|\mathsf{p} : \mathsf{T}\{\mu \mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!) \xrightarrow{\mathsf{p}::\mathsf{q}}_{p} ((\!|\mathsf{p} : \mathsf{T}\{\mu \mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!), S_1) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}_{1} ((\!|\mathsf{p} : \mathsf{T}\{\mu \mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!), S_2) \sim (\!|\mathsf{p} : \mathsf{T}'|\!).$$

   But $(\!|\mathsf{p} : \mu \mathbf{t}.\mathsf{T}|\!) \sim (\!|\mathsf{p} : \mathsf{T}\{\mu \mathbf{t}.\mathsf{T}/\mathbf{t}\}|\!)$ by Lemma 3.4.3, and hence

   $$\exists S_1', S_2' \cdot (\!|\mathsf{p} : \mu \mathbf{t}.\mathsf{T}|\!) \xrightarrow{\mathsf{p}::\mathsf{q}}_{p} ((\!|\mathsf{p} : \mu \mathbf{t}.\mathsf{T}|\!), S_1') \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}_{1} ((\!|\mathsf{p} : \mu \mathbf{t}.\mathsf{T}|\!), S_2') \sim (\!|\mathsf{p} : \mathsf{T}'|\!).$$

2. Analogous.

□

We now explore results that link the inner encoding $(\!| \cdot |\!)$ with the full encoding $[\![\cdot]\!]$. Firstly, we define what it means to *combine* states.

Definition 3.4.4. Let S_1 and S_2 be states for a PRISM module M. Their *union* is given by

$$(S_1 \cup S_2)(x) = \begin{cases} S_1(x) & \text{if } x \in \mathrm{dom}(S_1), x \notin \mathrm{dom}(S_2) \\ S_2(x) & \text{if } x \in \mathrm{dom}(S_2), x \notin \mathrm{dom}(S_1) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Using this definition, we formalise the relationship between the two encodings.

Lemma 3.4.6. *Let* $\Delta = \mathsf{p} : \mathsf{T}, \mathsf{q} : \mathsf{T}', \Delta'$ *be a context. If* $((\!|\mathsf{p} : \mathsf{T}|\!), S_1) \xrightarrow{\alpha}_{p} ((\!|\mathsf{p} : \mathsf{T}|\!), S_1')$ *and*

> $((\!|\mathsf{q} : \mathsf{T}'|\!), S_2) \xrightarrow{\alpha}_1 ((\!|\mathsf{q} : \mathsf{T}'|\!), S_2')$, *then for any state* S *of* $(\!|\Delta'|\!)$,
>
> $$([\![\Delta]\!], S_1 \cup S_2 \cup S) \xrightarrow{\alpha}_p ([\![\Delta]\!], S_1' \cup S_2' \cup S).$$

Proof. By case analysis on $\alpha = \mathsf{p} :: \mathsf{q}$ or $\mathsf{p} :: \mathsf{q} :: \ell$. □

> **Lemma 3.4.7.** *Let* $\Delta = \mathsf{p} : \mathsf{T}, \mathsf{q} : \mathsf{T}', \Delta'$ *be a context. Further let* S_1 *and* S_1' *be states of* $(\!|\mathsf{p} : \mathsf{T}|\!)$, S_2 *and* S_2' *states of* $(\!|\mathsf{q} : \mathsf{T}'|\!)$, *and* S *a state of* $(\!|\Delta'|\!)$. *If*
>
> $$([\![\Delta]\!], S_1 \cup S_2 \cup S) \xrightarrow{\alpha}_p ([\![\Delta]\!], S_1' \cup S_2' \cup S)$$
>
> *with* $\alpha = \mathsf{p} :: \mathsf{q}$ *or* $\mathsf{p} :: \mathsf{q} :: \ell$, *then*
>
> $$((\!|\mathsf{p} : \mathsf{T}|\!), S_1) \xrightarrow{\alpha}_p ((\!|\mathsf{p} : \mathsf{T}|\!), S_1') \text{ and } ((\!|\mathsf{q} : \mathsf{T}'|\!), S_2) \xrightarrow{\alpha}_1 ((\!|\mathsf{q} : \mathsf{T}'|\!), S_2').$$

Proof. We first argue using the definition of closure$(\cdot)$ that $\alpha \in \text{actions}_\Delta(\mathsf{p}) \cap \text{actions}_\Delta(\mathsf{q})$, and proceed by case analysis on α. □

These results reveal a useful property: bisimulation on $(\!|\cdot|\!)$ is closed under composition!

> **Lemma 3.4.8.** *Let* $\Delta = \mathsf{p}_1 : \mathsf{T}_1, \cdots, \mathsf{p}_n : \mathsf{T}_n$ *and* $\Delta' = \mathsf{p}_1 : \mathsf{T}_1', \cdots, \mathsf{p}_n : \mathsf{T}_n'$. *If* $((\!|\mathsf{p}_i : \mathsf{T}_i|\!), S_i) \sim (\!|\mathsf{p}_i : \mathsf{T}_i'|\!)$ *for all* i, *then* $([\![\Delta]\!], S_1 \cup \cdots \cup S_n) \sim [\![\Delta']\!]$.

Proof. For all i, let R_i be a bisimulation relation for $((\!|\mathsf{p}_i : \mathsf{T}_i|\!), S_i) \sim (\!|\mathsf{p}_i : \mathsf{T}_i'|\!)$. Now let

$$R = \{((T_1 \cup T_2 \cup \cdots \cup T_n), (T_1' \cup T_2' \cup \cdots \cup T_n')) \mid \forall i \cdot (T_i, T_i') \in R_i\}.$$

We then check that R is a bisimulation relation by case analysis on α and using Lemmas 3.4.6 and 3.4.7. □

We are now ready to prove soundness.

> **Theorem 3.4.9** (Soundness). *For every context* Δ, Δ' *and probability* p *with* $\Delta \to_p \Delta'$, *there exists* S *such that* $[\![\Delta]\!] \to_p^2 ([\![\Delta]\!], S)$ *and* $[\![\Delta']\!] \sim ([\![\Delta]\!], S)$.

Proof. Let $\Delta = \mathsf{p}_1 : \mathsf{T}_1, \cdots, \mathsf{p}_n : \mathsf{T}_n$. Without loss of generality, we assume $\Delta \xrightarrow{(\mathsf{p}_1, \mathsf{p}_2)\ell}_p \Delta'$, so

$\Delta' = \mathsf{p}_1 : \mathsf{T}_1', \mathsf{p}_2 : \mathsf{T}_2', \mathsf{p}_3 : \mathsf{T}_3, \cdots, \mathsf{p}_n : \mathsf{T}_n$. Then,

$$[\![\Delta]\!] = (\!|\mathsf{p}_1 : \mathsf{T}_1|\!) \parallel \ldots \parallel (\!|\mathsf{p}_n : \mathsf{T}_n|\!) \parallel \text{closure}(\Delta) \text{ and}$$

$$[\![\Delta']\!] = (\!|\mathsf{p}_1 : \mathsf{T}_1'|\!) \parallel (\!|\mathsf{p}_2 : \mathsf{T}_2'|\!) \parallel (\!|\mathsf{p}_3 : \mathsf{T}_3|\!) \parallel \ldots \parallel (\!|\mathsf{p}_n : \mathsf{T}_n|\!) \parallel \text{closure}(\Delta').$$

By [ST-τ], we must have $\mathsf{p}_1 : \mathsf{T}_1 \xrightarrow{\mathsf{p}_1 : \mathsf{p}_2 ! \ell \langle \mathsf{B} \rangle}_p \mathsf{p}_1 : \mathsf{T}_1'$ and $\mathsf{p}_2 : \mathsf{T}_2 \xrightarrow{\mathsf{p}_2 : \mathsf{p}_1 ? \ell (\mathsf{B})}_1 \mathsf{p}_2 : \mathsf{T}_2'$. Applying Lemma 3.4.5, there exist states S_1, S_1', S_2, S_2' such that

$$(\!|\mathsf{p}_1 : \mathsf{T}_1|\!) \xrightarrow{\mathsf{p}_1 :: \mathsf{p}_2}_p ((\!|\mathsf{p}_1 : \mathsf{T}_1|\!), S_1) \xrightarrow{\mathsf{p}_1 :: \mathsf{p}_2 :: \ell}_1 ((\!|\mathsf{p}_1 : \mathsf{T}_1|\!), S_1') \sim (\!|\mathsf{p}_1 : \mathsf{T}_1'|\!),$$
$$(\!|\mathsf{p}_2 : \mathsf{T}_2|\!) \xrightarrow{\mathsf{p}_1 :: \mathsf{p}_2}_1 ((\!|\mathsf{p}_2 : \mathsf{T}_2|\!), S_2) \xrightarrow{\mathsf{p}_1 :: \mathsf{p}_2 :: \ell}_1 ((\!|\mathsf{p}_2 : \mathsf{T}_2|\!), S_2') \sim (\!|\mathsf{p}_2 : \mathsf{T}_2'|\!).$$

We collapse the left three columns using Lemma 3.4.6:

$$[\![\Delta]\!] \xrightarrow{\mathsf{p}_1 :: \mathsf{p}_2}_p ([\![\Delta]\!], S_1 \cup S_2) \xrightarrow{\mathsf{p}_1 :: \mathsf{p}_2 :: \ell}_1 ([\![\Delta]\!], S_1' \cup S_2').$$

Finally, we appeal to Lemma 3.4.8 to conclude $([\![\Delta]\!], S_1' \cup S_2') \sim [\![\Delta']\!]$. □

3.4.2 Completeness

We first prove a weaker result (Lemma 3.4.11), following the approach for soundness by first considering $(\!| \cdot |\!)$ (Lemma 3.4.10) then generalising to $[\![\cdot]\!]$.

Lemma 3.4.10.

1. *If* $(\!|\mathsf{p} : \mathsf{T}|\!) \xrightarrow{\mathsf{p} :: \mathsf{q}}_p ((\!|\mathsf{p} : \mathsf{T}|\!), S) \xrightarrow{\mathsf{p} :: \mathsf{q} :: \ell}_1 ((\!|\mathsf{p} : \mathsf{T}|\!), S')$, *then*

$$\exists \mathsf{T}' \cdot \mathsf{p} : \mathsf{T} \xrightarrow{\mathsf{p} : \mathsf{q} ! \ell \langle \mathsf{B} \rangle}_p \mathsf{p} : \mathsf{T}' \text{ and } ((\!|\mathsf{p} : \mathsf{T}|\!), S') \sim (\!|\mathsf{p} : \mathsf{T}'|\!).$$

2. *If* $(\!|\mathsf{p} : \mathsf{T}|\!) \xrightarrow{\mathsf{q} :: \mathsf{p}}_1 ((\!|\mathsf{p} : \mathsf{T}|\!), S) \xrightarrow{\mathsf{q} :: \mathsf{p} :: \ell}_1 ((\!|\mathsf{p} : \mathsf{T}|\!), S')$, *then*

$$\exists \mathsf{T}' \cdot \mathsf{p} : \mathsf{T} \xrightarrow{\mathsf{p} : \mathsf{q} ? \ell (\mathsf{B})}_1 \mathsf{p} : \mathsf{T}' \text{ and } ((\!|\mathsf{p} : \mathsf{T}|\!), S') \sim (\!|\mathsf{p} : \mathsf{T}'|\!).$$

Proof. Examining the definition of $(\!| \cdot |\!)$, T must have the form

$$\mathsf{T} = \mu \mathbf{t}_1 . \mu \mathbf{t}_2 . \cdots \mu \mathbf{t}_n . \mathsf{q} \oplus_{i \in I} \mathsf{p}_i : \ell_i \langle \mathsf{B}_i \rangle . \mathsf{T}_i'.$$

for (1), and

$$\mathsf{T} = \mu \mathbf{t}_1.\mu \mathbf{t}_2. \cdots \mu \mathbf{t}_n.\mathsf{q}\&_{i\in I}\ell_i(\mathsf{B}_i).\mathsf{T}'_i.$$

for (2). We proceed by induction on n, using Lemma 3.4.4 for the base case and Lemma 3.4.3 for the inductive case. □

Lemma 3.4.11. *For every context Δ and states S, S' such that $[\![\Delta]\!] \xrightarrow{\mathsf{p}::\mathsf{q}}_p ([\![\Delta]\!], S) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}_1 ([\![\Delta]\!], S')$, there exists Δ' with $\Delta \to_p \Delta'$ and $([\![\Delta]\!], S') \sim [\![\Delta']\!]$.*

Proof. Let $\Delta = \mathsf{p}_1 : \mathsf{T}_1, \cdots, \mathsf{p}_n : \mathsf{T}_n$, and without loss of generality, suppose $\mathsf{p} = \mathsf{p}_1$ and $\mathsf{q} = \mathsf{p}_2$. Then,

$$[\![\Delta]\!] = (\!|\mathsf{p}_1 : \mathsf{T}_1|\!) \parallel \ldots \parallel (\!|\mathsf{p}_n : \mathsf{T}_n|\!) \parallel \mathsf{closure}(\Delta).$$

Let us rewrite $S = S_1 \cup S_2$ and $S' = S'_1 \cup S'_2$ where S_1, S'_1 are states for $(\!|\mathsf{p}_1 : \mathsf{T}_1|\!)$ and S_2, S'_2 are states for $(\!|\mathsf{p}_2 : \mathsf{T}_2|\!)$, so that

$$[\![\Delta]\!] \xrightarrow{\mathsf{p}_1::\mathsf{p}_2}_p ([\![\Delta]\!], S_1 \cup S_2) \xrightarrow{\mathsf{p}_1::\mathsf{p}_2::\ell}_1 ([\![\Delta]\!], S'_1 \cup S'_2).$$

Then by two applications of Lemma 3.4.7,

$$(\!|\mathsf{p}_1 : \mathsf{T}_1|\!) \xrightarrow{\mathsf{p}_1::\mathsf{p}_2}_p ((\!|\mathsf{p}_1 : \mathsf{T}_1|\!), S_1) \xrightarrow{\mathsf{p}_1::\mathsf{p}_2::\ell}_1 ((\!|\mathsf{p}_1 : \mathsf{T}_1|\!), S'_1),$$
$$(\!|\mathsf{p}_2 : \mathsf{T}_2|\!) \xrightarrow{\mathsf{p}_1::\mathsf{p}_2}_1 ((\!|\mathsf{p}_2 : \mathsf{T}_2|\!), S_2) \xrightarrow{\mathsf{p}_1::\mathsf{p}_2::\ell}_1 ((\!|\mathsf{p}_2 : \mathsf{T}_2|\!), S'_2).$$

Applying Lemma 3.4.10, there exist $\mathsf{T}'_1, \mathsf{T}'_2$ such that

$$\mathsf{p}_1 : \mathsf{T}_1 \xrightarrow{\mathsf{p}_1:\mathsf{p}_2!\ell\langle\mathsf{B}\rangle}_p \mathsf{p}_1 : \mathsf{T}'_1, \qquad \mathsf{p}_2 : \mathsf{T}_2 \xrightarrow{\mathsf{p}_2:\mathsf{p}_1?\ell(\mathsf{B})}_1 \mathsf{p}_2 : \mathsf{T}'_2,$$
$$((\!|\mathsf{p}_1 : \mathsf{T}_1|\!), S'_1) \sim (\!|\mathsf{p}_1 : \mathsf{T}'_1|\!), \qquad ((\!|\mathsf{p}_2 : \mathsf{T}_2|\!), S'_2) \sim (\!|\mathsf{p}_2 : \mathsf{T}'_2|\!).$$

Let $\Delta' = \mathsf{p}_1 : \mathsf{T}'_1, \mathsf{p}_2 : \mathsf{T}'_2, \mathsf{p}_3 : \mathsf{T}_3, \cdots, \mathsf{p}_n : \mathsf{T}_n$. Then $\Delta \xrightarrow{(\mathsf{p}_1,\mathsf{p}_2)\ell}_p \Delta'$ by [ct-τ] and [ct-sc], and by Lemma 3.4.8, $([\![\Delta]\!], S') = ([\![\Delta]\!], S'_1 \cup S'_2) \sim [\![\Delta']\!]$, as required. □

This is a weaker result as it mandates $\xrightarrow{\mathsf{p}::\mathsf{q}}$ and $\xrightarrow{\mathsf{p}::\mathsf{q}::\ell}$ to occur consecutively, rather than allowing an arbitrary interleaving of transitions.

However, defining the general behaviour is tricky – there's no typing context reduction directly corresponding to a single partial transition $\xrightarrow{\mathsf{p}::\mathsf{q}}$! Intuitively, if $[\![\Delta]\!] \xrightarrow{\mathsf{p}::\mathsf{q}} ([\![\Delta]\!], S)$, then $([\![\Delta]\!], S)$

should still be *related* to $[\![\Delta]\!]$ even if not bisimilar, since the transition isn't complete until the corresponding $\xrightarrow{\mathsf{p} :: \mathsf{q} :: \ell}$ occurs.

We encode this intuition into a notion of *refinement via partial transitions*.

Definition 3.4.5 (Refinement via partial transitions)**.** We say $([\![\Delta]\!], S)$ *refines* $([\![\Delta']\!], S')$ *via partial transitions with probability* p, written $([\![\Delta]\!], S) \subsetneq_p ([\![\Delta']\!], S')$, if they satisfy the following rules:

1. if $(M, S) \sim (M', S')$, then $([\![\Delta]\!], S) \subsetneq_1 ([\![\Delta']\!], S')$;
2. if $\exists S'' \cdot ([\![\Delta']\!], S') \xrightarrow{\mathsf{p} :: \mathsf{q}}_p ([\![\Delta']\!], S'')$ and $([\![\Delta]\!], S) \subsetneq_{p'} ([\![\Delta']\!], S'')$, then $([\![\Delta]\!], S) \subsetneq_{p \cdot p'} ([\![\Delta']\!], S')$.

This allows us to relate model-state pairs separated only by partial transitions. It turns out that this happens exactly when partial transitions involve distinct participants:

Notation 3.4.1. We write $([\![\Delta]\!], S) \xrightarrow{(\_ :: \_)*}_p ([\![\Delta]\!], S')$ if $\exists S_1, \ldots, S_n$ such that

$$([\![\Delta]\!], S) \xrightarrow{\mathsf{p}_1 :: \mathsf{q}_1}_{p_1} ([\![\Delta]\!], S_1) \xrightarrow{\mathsf{p}_2 :: \mathsf{q}_2}_{p_2} \cdots \xrightarrow{\mathsf{p}_n :: \mathsf{q}_n}_{p_n} ([\![\Delta]\!], S_n) = ([\![\Delta]\!], S'),$$

with $\mathsf{p}_1, \cdots, \mathsf{p}_n, \mathsf{q}_1, \cdots, \mathsf{q}_n$ all distinct participants, and $p_1 p_2 \cdots p_n = p$.

Lemma 3.4.12. $([\![\Delta]\!], S_1) \subsetneq_p ([\![\Delta]\!], S_2)$ *iff* $\exists S_2' \cdot ([\![\Delta']\!], S_2) \xrightarrow{(\_ :: \_)*}_p ([\![\Delta']\!], S_2') \sim ([\![\Delta]\!], S_1)$.

Proof. The backwards direction is immediate from the definition of $\subsetneq$.

For the forwards direction, we examine the definition of $(\!|\cdot|\!)$. Intuitively, if a participant has already made a partial transition, they can't make another one. □

This is a useful characterisation since transitions with disjoint participants commute.

Lemma 3.4.13. *Let* $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}$ *be distinct participants. Furthermore, let actions* $\alpha_1 = \mathsf{p} :: \mathsf{q}$ *or* $\mathsf{p} :: \mathsf{q} :: \ell_1$*, and* $\alpha_2 = \mathsf{r} :: \mathsf{s}$ *or* $\mathsf{r} :: \mathsf{s} :: \ell_2$*. If* $([\![\Delta]\!], S) \xrightarrow{\alpha_1}_p ([\![\Delta]\!], S_1) \xrightarrow{\alpha_2}_{p'} ([\![\Delta]\!], S')$*, then* $\exists S_2 \cdot ([\![\Delta]\!], S) \xrightarrow{\alpha_2}_{p'} ([\![\Delta]\!], S_2) \xrightarrow{\alpha_1}_p ([\![\Delta]\!], S')$.

Proof. We show that variables used in the guards and updates of each transition are independent.

□

We use this property to generalise Lemma 3.4.11 to the full completeness result.

Lemma 3.4.14. *Suppose* $(\llbracket\Delta\rrbracket, S) \subsetneq_{p_1} \llbracket\Delta'\rrbracket$ *and* $(\llbracket\Delta\rrbracket, S) \xrightarrow{\alpha}_{p_2} (\llbracket\Delta\rrbracket, S')$. *Then* $\exists\Delta'' \cdot \Delta' \to^*_{p_3} \Delta''$ *and* $(\llbracket\Delta\rrbracket, S') \subsetneq_{p_4} \llbracket\Delta''\rrbracket$, *with* $p_1 p_2 = p_3 p_4$.

Proof. By Lemma 3.4.12,

$$\exists T \cdot \llbracket\Delta'\rrbracket \xrightarrow{(\_::\_)*}_{p_1} (\llbracket\Delta'\rrbracket, T) \sim (\llbracket\Delta\rrbracket, S), \tag{3.1}$$

and by bisimilarity,

$$\exists T' \cdot (\llbracket\Delta'\rrbracket, T) \xrightarrow{\alpha}_{p_2} (\llbracket\Delta'\rrbracket, T') \sim (\llbracket\Delta'\rrbracket, S'). \tag{3.2}$$

We consider the two possible cases of α.

Case $\alpha = \mathsf{p} :: \mathsf{q}$ Then

$$\llbracket\Delta'\rrbracket \xrightarrow{(\_::\_)*}_{p_1 p_2} (\llbracket\Delta'\rrbracket, T') \sim (\llbracket\Delta\rrbracket, S),$$

so setting $\Delta'' = \Delta'$ (and therefore $p_3 = 1$), indeed $(\llbracket\Delta\rrbracket, S') \subsetneq_{p_1 p_2} \llbracket\Delta'\rrbracket$.

Case $\alpha = \mathsf{p} :: \mathsf{q} :: \ell$ Then $p_2 = 1$. Since $(\llbracket\Delta'\rrbracket, T) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}$, there must've been a $\xrightarrow{\mathsf{p}::\mathsf{q}}$ within $\llbracket\Delta'\rrbracket \xrightarrow{(\_::\_)*}_{p_1} (\llbracket\Delta'\rrbracket, T)$. Hence, we apply Lemma 3.4.13 to (3.1) and (3.2) so that

$$\exists U, U' \cdot \llbracket\Delta'\rrbracket \xrightarrow{\mathsf{p}::\mathsf{q}}_{p_3} (\llbracket\Delta'\rrbracket, U) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}_{1} (\llbracket\Delta'\rrbracket, U') \xrightarrow{(\_::\_)*}_{p_1/p_3} (\llbracket\Delta'\rrbracket, T'). \tag{3.3}$$

Applying Lemma 3.4.11 to (3.3), $\exists\Delta'' \cdot \Delta' \to_{p_3} \Delta''$ and $(\llbracket\Delta'\rrbracket, U') \sim \llbracket\Delta''\rrbracket$. By bisimilarity,

$$\exists V \cdot \llbracket\Delta''\rrbracket \xrightarrow{(\_::\_)*}_{p_1/p_3} (\llbracket\Delta''\rrbracket, V) \sim (\llbracket\Delta'\rrbracket, T').$$

Recalling $(\llbracket\Delta'\rrbracket, T') \sim (\llbracket\Delta'\rrbracket, S')$, we have

$$(\llbracket\Delta'\rrbracket, S) \subsetneq_{p_1/p_3} \llbracket\Delta''\rrbracket,$$

with $p_1 \cdot 1 = p_3 \cdot \frac{p_1}{p_3}$, as required. □

Theorem 3.4.15 (Completeness). *For every context* Δ, *state* S *and probability* p *with* $\llbracket\Delta\rrbracket \to^*_p (\llbracket\Delta\rrbracket, S)$, *there exists* Δ' *such that* $\Delta \to^*_{p'} \Delta'$ *and* $(\llbracket\Delta\rrbracket, S) \subsetneq_{p''} \llbracket\Delta'\rrbracket$, *with* $p = p' p''$.

Proof. For some $n \geq 0$ and states $S_1, \cdots, S_n$,

$$[\![\Delta]\!] \xrightarrow{\alpha_1}_{p_1} ([\![\Delta]\!], S_1) \xrightarrow{\alpha_2}_{p_2} \cdots \xrightarrow{\alpha_n}_{p_n} ([\![\Delta]\!], S_n) = ([\![\Delta]\!], S),$$

with $p_1 p_2 \cdots p_n = p$. We proceed by induction on n.

Case $n = 0$ Then $([\![\Delta]\!], S) = [\![\Delta]\!]$ and $p = 1$. We pick $\Delta' = \Delta$ (and therefore $p' = 1$), and indeed $[\![\Delta]\!] \subsetneq_1 [\![\Delta]\!]$.

Case $n = k + 1$ By the induction hypothesis, $\exists \Delta' \cdot \Delta \to^*_{p'} \Delta'$ and $([\![\Delta]\!], S_n) \subsetneq_{p''} [\![\Delta']\!]$, with $p' p'' = p_1 \cdots p_k$. Since $([\![\Delta]\!], S_n) \xrightarrow{\alpha_{k+1}}_{p_{k+1}} ([\![\Delta]\!], S)$, by Lemma 3.4.14, $\exists \Delta'' \cdot \Delta' \to^*_q \Delta''$ and $([\![\Delta]\!], S) \subsetneq_{q'} [\![\Delta'']\!]$, with $p'' p_{k+1} = q q'$. Hence $\Delta \to^*_{p'q} \Delta''$, and $(p'q)q' = p' p'' p_{k+1} = p$, as required. □

3.5 Property checking

With the encoding in place, we must now write desirable properties as logical formulae.

3.5.1 A brief introduction to PCTL*

We first present the relevant fragment of PCTL*, an extension of PCTL (*probabilistic computation tree logic*). Due to space constraints, we opt for a brief presentation; a more principled discussion can be found in [Baier, 1998].

Definition 3.5.1. The syntax of our fragment of PCTL* is given by

$$\begin{aligned}
\Phi &::= \mathsf{Pmin}(\varphi) && \text{(probability)} \\
\varphi, \psi &::= a && \text{(atomic proposition)} \\
&\mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi && \text{(negation, conjunction, disjunction)} \\
&\mid \Box \varphi && \text{(always)} \\
&\mid \Diamond \varphi && \text{(eventually)}
\end{aligned}$$

We write $\varphi \implies \psi$ as a shorthand for $\neg\varphi \vee \psi$.

Definition 3.5.2. A *path* π of a PRISM module M is a (potentially infinite) sequence of states $S_1; S_2; \cdots$ such that $\forall i \in [1, |\pi|) \cdot (M, S_i) \to (M, S_{i+1})$.

We call φ a *path formula*. Path formulae act on paths π of a module M, and we write $\pi \vDash \varphi$ if π satisfies φ. Given a model M, $\mathsf{Pmin}(\varphi)$ tells us the probability[2] of obtaining a path $\pi = \emptyset; S_2; \cdots$ starting from $\emptyset$ such that $\pi \vDash \varphi$.

Definition 3.5.3. Let $\pi = S_1; S_2; \cdots$ be a path, and denote $\pi[i..]$ to be the subpath $S_i; S_{i+1}; \cdots$. The semantics of a path formula φ is given inductively:

$$
\begin{aligned}
\pi \vDash a &\iff S_1 \vdash a \\
\pi \vDash \neg\varphi &\iff \pi \nvDash \varphi \\
\pi \vDash \varphi \wedge \psi &\iff \pi \vDash \varphi \wedge \pi \vDash \psi \\
\pi \vDash \varphi \vee \psi &\iff \pi \vDash \varphi \vee \pi \vDash \psi \\
\pi \vDash \Box\, \varphi &\iff \forall i \cdot \pi[i..] \vDash \varphi \\
\pi \vDash \Diamond\, \varphi &\iff \exists i \cdot \pi[i..] \vDash \varphi
\end{aligned}
$$

3.5.2 Defining specifications

We define several PRISM *labels* – boolean expressions over variables – to act as atomic propositions. For conciseness, we only give informal meanings here, though it is straightforward to express them as conjunctions or disjunctions of state variables.

Definition 3.5.4. Given an encoding $[\![\Delta]\!]$, we define the following *labels* (their names denoted with *"quotes"*):

- $S \vdash$ *"end"* $\iff \forall \mathsf{p} \in \mathrm{dom}(\Delta) \cdot S(S_{\mathsf{p}}) = SS(\Delta(\mathsf{p}))$
- $\forall \mathsf{p} :: \mathsf{q} :: \ell \in \mathrm{actions}_\Delta(\mathsf{p}) \cup \mathrm{actions}_\Delta(\mathsf{q}) \cdot$
  $S \vdash$ *"send* $::$ p $::$ q $::$ ℓ*"* $\iff (\!|\Delta(\mathsf{p})|\!), S) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}$
- $\forall \mathsf{p}, \mathsf{q} \in \mathrm{dom}(\Delta) \cdot S \vdash$ *"receive* $::$ p $::$ q*"* $\iff \exists \ell \cdot (\!|\Delta(\mathsf{q})|\!), S) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}$

[2] More precisely, this gives the *minimum* probability across all realisations of nondeterminism. In our case, however, all probabilities are equal thanks to *confluence*.

- $\forall \mathsf{p} :: \mathsf{q} :: \ell \in \mathsf{actions}_\Delta(\mathsf{p}) \cup \mathsf{actions}_\Delta(\mathsf{q}) \cdot$
  $S \vdash \text{“}receive :: \mathsf{p} :: \mathsf{q} :: \ell \text{”} \iff (\!|\Delta(\mathsf{q})|\!), S) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}$

Moreover, PRISM has an inbuilt label *“deadlock”* such that $S \vdash \text{“}deadlock\text{”} \iff ([\![\Delta]\!], S) \nrightarrow$.

Using the labels, we write the following *property specifications*:

$$\mathsf{safe}(\Delta) \iff \mathsf{Pmin}\Big(\Box\Big(\bigwedge_{\mathsf{p}::\mathsf{q}::\ell \in \mathsf{actions}(\Delta)} \Big(\text{“}send :: \mathsf{p} :: \mathsf{q} :: \ell\text{”} \wedge \text{“}receive :: \mathsf{p} :: \mathsf{q}\text{”}\Big) \implies \text{“}receive :: \mathsf{p} :: \mathsf{q} :: \ell\text{”}\Big)\Big) = 1$$

$$\mathbb{P}_{\mathsf{DF}}(\Delta) = \mathsf{Pmin}\left(\Box\left(\text{“}deadlock\text{”} \implies \text{“}end\text{”}\right)\right)$$

$$\mathbb{P}_{\mathsf{Term}}(\Delta) = \mathsf{Pmin}\left(\Diamond\ \text{“}end\text{”}\right)$$

In a safe context, if p can send ℓ to q and q is ready to receive from p, then q must accept ℓ. The probability of deadlock freedom is the probability of obtaining a path where no reductions are possible only when all participants are in the end state. Finally, the probability of termination is the probability of obtaining a path where all participants eventually enter the end state.

Chapter 4

Implementation

With the theoretical foundations in place, we now turn our attention to implementation. In this chapter, we present Prose, a tool for the verification of **pro**babilistic **se**ssion types. Prose is implemented in OCaml 🐫, following the compilation pipeline illustrated in Figure 4.1. We start with an input context, which is converted into an *abstract syntax tree* (AST) by the *lexing* and *parsing* stages (Section 4.1). The *validation* stage (Section 4.2) checks that the AST is well-formed, after which the *translation* stage (Section 4.3) produces abstract representations of the translated PRISM model and properties. Finally, the *pretty-printing* stage outputs `.prism` and `.pctl` files for verification via PRISM.

4.1 Lexing and parsing

The *lexing* stage converts the input context file into a sequence of *tokens*. For example,

```
p : q (+) l . end
```

might be converted into

```
IDENT "p"; COLON; IDENT "q"; OPLUS; IDENT "l"; DOT; END.
```

The full set of tokens and their regular expressions are given in Figure 4.2. We then invoke `ocamllex`[1] on these rules.

The *parsing* stage converts the stream of tokens produced by the lexer into an AST. To do this, we

[1] https://ocaml.org/manual/5.3/lexyacc.html

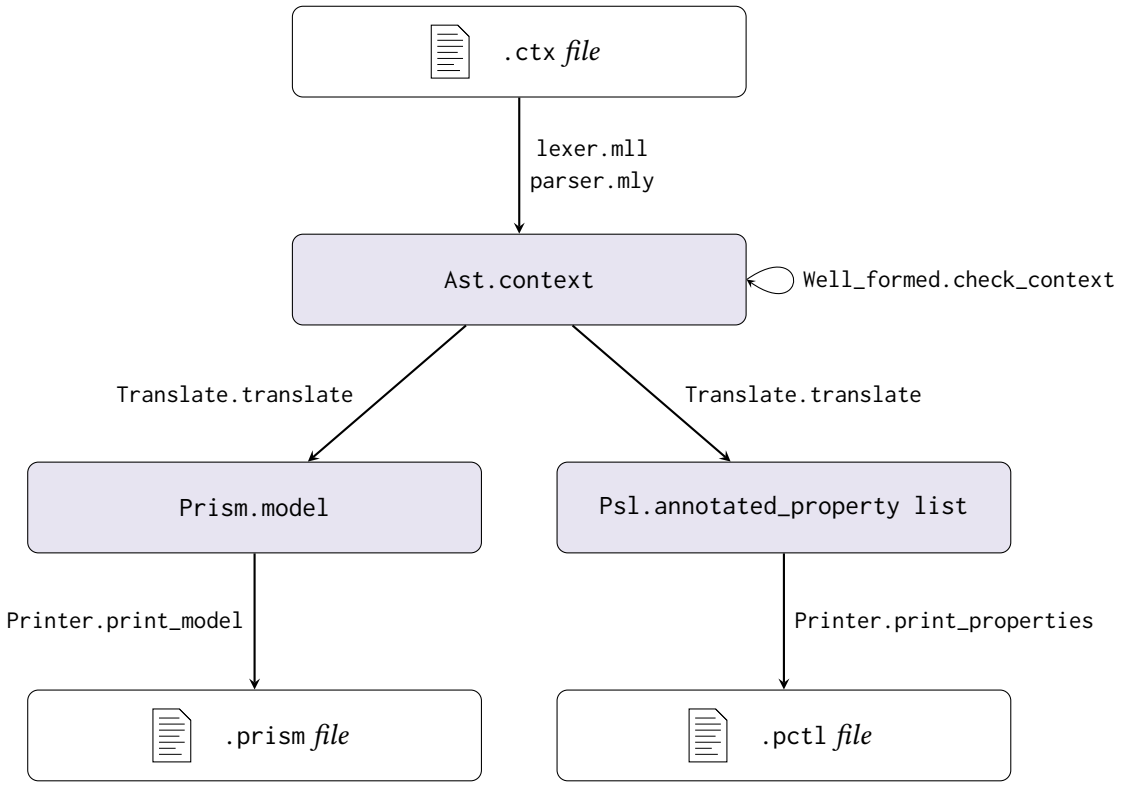

Figure 4.1: An overview of the Prose compilation pipeline.

Token	Regular expression
IDENT	(a-z | A-Z | _) (a-z | A-Z | 0-9 | _)*
PROB	0 | 1 | 1.0 | 0.(0-9)*
COLON	:
DOT	.
COMMA	,
END	end
MU	mu
OPLUS	(+)
AND	&
LBRACE	{
RBRACE	}
LPAREN	(
RPAREN	)
INT	Int
BOOL	Bool
UNIT	Unit

Figure 4.2: Lexer tokens and their corresponding regular expressions.

⟨*context*⟩ ::= ∅ | ⟨*context-item*⟩ ⟨*context*⟩
⟨*context-item*⟩ ::= `IDENT` : ⟨*session-type*⟩
⟨*session-type*⟩ ::= `end` | `IDENT`
| `mu IDENT .` ⟨*session-type*⟩
| `IDENT (+)` ⟨*int-choices*⟩
| `IDENT &` ⟨*ext-choices*⟩
⟨*int-choices*⟩ ::= ⟨*choice*⟩ | { ⟨*prob-choices*⟩ }
⟨*ext-choices*⟩ ::= ⟨*choice*⟩ | { ⟨*choices-list*⟩ }
⟨*choice*⟩ ::= `IDENT .` ⟨*session-type*⟩
| `IDENT (` ⟨*basic-type*⟩ `) .` ⟨*session-type*⟩
⟨*choices*⟩ ::= ⟨*choice*⟩ | ⟨*choice*⟩ , ⟨*choices*⟩
⟨*prob-choice*⟩ ::= `PROB :` ⟨*choice*⟩
⟨*prob-choices*⟩ ::= ⟨*prob-choice*⟩ | ⟨*prob-choice*⟩ , ⟨*prob-choices*⟩
⟨*basic-type*⟩ ::= `Int` | `Bool` | `Unit`

Figure 4.3: Context-free grammar for machine-readable typing contexts. For clarity, we present all tokens except `IDENT` and `PROB` in their expanded form.

first write down the syntax of session type contexts as a *context-free grammar* (CFG). The rules are given in Figure 4.3 – note this largely mirrors the mathematical syntax of session types.

Example 4.1.1. The context

$$\mathsf{p} : \mu \mathbf{t}.\mathsf{q} \oplus \begin{cases} 0.3 : \ell_1.\mathsf{end} \\ 0.7 : \ell_2.\mathbf{t} \end{cases} \qquad \mathsf{q} : \mu \mathbf{t}.\mathsf{p} \& \begin{cases} \ell_1.\mathsf{end} \\ \ell_2.\mathbf{t} \end{cases}$$

can be written in our machine-readable format as

```
p : mu t . q (+) {
              0.3 : l1 . end,
              0.7 : l2 . t
            }

q : mu t . p & {
              l1 . end,
              l2 . t
            }
```

We augment each production rule in the CFG with rules on how to construct the *abstract syntax*

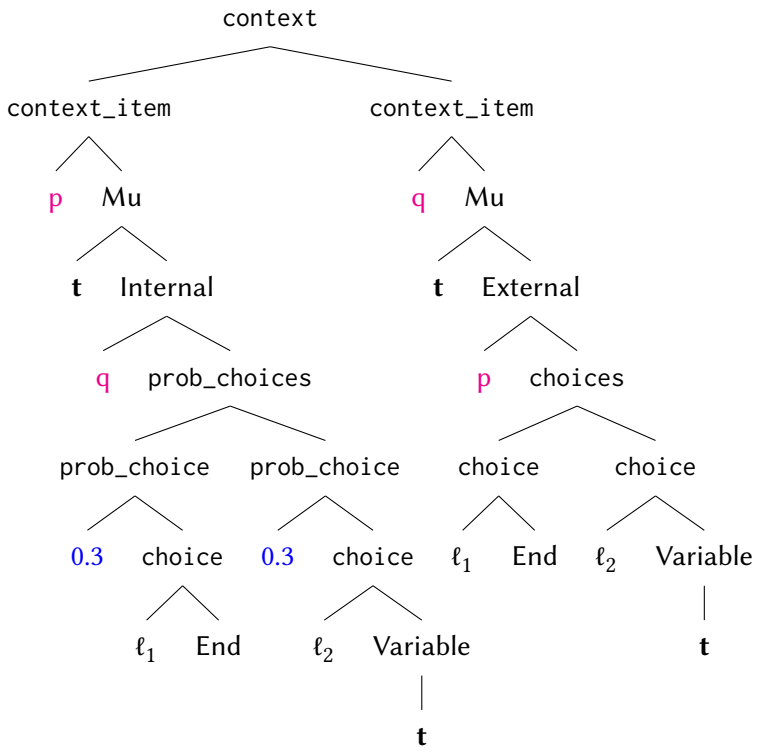

Figure 4.4: Abstract syntax tree for Example 4.1.1.

tree (AST), then invoke the Menhir parser generator.[2]

Example 4.1.2. The parsing rule for ⟨*session-type*⟩ is

```
session_type:
| END
    { End }
| var = IDENT
    { Variable var }
| MU var = IDENT DOT cont = session_type
    { Mu (var, cont) }
| int_part = IDENT OPLUS int_choices = int_choices
    { Internal { int_part; int_choices } }
| ext_part = IDENT AND ext_choices = ext_choices
    { External { ext_part; ext_choices } }
```

For example, the second rule states that the sequence `MU; IDENT; DOT; <session-type>` produces a `Mu` node.

[2] https://gallium.inria.fr/~fpottier/menhir/

```
type model =
  { globals : var_type list
  ; modules : pmodule list
  ; labels : label list
  }
[@@deriving sexp_of]

and pmodule =
  { locals : var_type list
  ; participant : string
  ; commands : command list
  }

and command =
  { action : Action.t
  ; guard : bool expr
  ; updates :
      (float * update list) list
  }

and label =
  { name : label_name
  ; expr : bool expr
  }

and _ expr =
  | IntConst : int -> int expr
  | BoolConst : bool -> bool expr
  | Var : 'a variable -> 'a expr
  | Eq : 'a expr * 'a expr -> bool expr
  | And : bool expr * bool expr -> bool expr
  | Or : bool expr * bool expr -> bool expr

and _ variable =
  | StringVar : string -> 'a variable

and update =
  | IntUpdate of int variable * int expr
  | BoolUpdate of bool variable * bool expr
```

Figure 4.5: An excerpt from `prism.ml`.

Example 4.1.3. The AST for the context in Example 4.1.1 is given in Figure 4.4.

4.2 Validation

The *validation* stage checks that certain invariants hold in the typing context. Namely:

- all variables are bound;
- each probability $p_i \in [0, 1]$;
- internal choices have $\sum_{i \in I} p_i = 1$.

If any of these are violated, Prose displays an error.

4.3 Translation

The *translation* stage implements the procedure from Chapter 3. We adopt several optimisations: for instance, we use sets and maps based on binary search trees (from Base[3]) to track variables and actions efficiently. This stage produces ASTs for the PRISM model and property specifications, with its structure defined by a mutually recursive type shown in Figure 4.5.

[3] https://opensource.janestreet.com/base/

Table 4.1: Available flags for `prose verify`

Flag	Description
`-print-ast`	Print internal AST representation for debugging
`-raw-prism`	Print raw PRISM CLI output for debugging
`-translation-time`	Print time taken for translation of context into PRISM

4.4 Using Prose

To verify typing contexts, users can pass a context file to the `verify` mode.

```
$ prose verify examples/prob-deadlock.ctx
Type safety
Result: true

Probabilistic deadlock freedom
Result: 0.30000000000000004 (exact floating point)

Probabilistic termination
Result: 1.0 (exact floating point)
```

A number of debugging flags are available, as listed in Table 4.1.

To display the PRISM model and property files, we use the `output` mode.

```
$ prose output examples/prob-deadlock.ctx
global fail : bool init false;

module closure
  closure : bool init false;

endmodule

module commander
  commander : [0..4] init 0;
  commander_a_label : [0..2] init 0;

...

// Probabilistic deadlock freedom
Pmin=? [ (G ("deadlock" => "end")) ]

// Probabilistic termination
Pmin=? [ (F ("end") ]
```

Table 4.2: Available flags for `prose output`

Flag	Description
`-o string`	Write PRISM model output to filename (default: print to `stdout`)
`-p string`	Write PRISM property output to filename (default: print to `stdout`)
`-print-ast`	Print internal AST representation for debugging
`-translation-time`	Print time taken for translation of context into PRISM

We can use the flags in Table 4.2 to instead save the output as files.

Chapter 5

Evaluation

In this chapter, we evaluate the effectiveness of Prose in various scenarios. We first verify properties of interesting typing contexts. Then, we run a comprehensive suite of performance benchmarks to measure Prose's efficiency.

5.1 Case studies

5.1.1 Recursive map-reduce

A recursive map-reduce protocol (extended from [Scalas and Yoshida, 2019]) is described by the following context:

$$\begin{aligned}\mathsf{mapper} \;:\; & \mu \mathbf{t}.\mathsf{worker}_1 \oplus \mathtt{datum}\langle \mathsf{int}\rangle. \\ & \mathsf{worker}_2 \oplus \mathtt{datum}\langle \mathsf{int}\rangle. \\ & \mathsf{worker}_3 \oplus \mathtt{datum}\langle \mathsf{int}\rangle. \\ & \mathsf{reducer}\,\&\begin{cases} \mathtt{continue}(\mathsf{int}).\mathbf{t} \\ \mathtt{stop}.\mathsf{worker}_1 \oplus \mathtt{stop}. \\ \qquad \mathsf{worker}_2 \oplus \mathtt{stop}. \\ \qquad \mathsf{worker}_3 \oplus \mathtt{stop}.\mathtt{end} \end{cases}\end{aligned}$$

$$\begin{aligned}\mathsf{reducer} \;:\; & \mu \mathbf{t}.\mathsf{worker}_1 \,\&\, \mathtt{result}(\mathsf{int}). \\ & \mathsf{worker}_2 \,\&\, \mathtt{result}(\mathsf{int}). \\ & \mathsf{worker}_3 \,\&\, \mathtt{result}(\mathsf{int}). \\ & \mathsf{mapper} \oplus \begin{cases} 0.4 \;:\; \mathtt{continue}\langle \mathsf{int}\rangle.\mathbf{t} \\ 0.6 \;:\; \mathtt{stop}.\mathtt{end} \end{cases}\end{aligned}$$

$$\begin{aligned}\forall i \in \{1,2,3\} \cdot \mathsf{worker}_i \;:\; & \mathsf{mapper}\,\&\,\mathtt{datum}(\mathsf{int}). \\ & \mu \mathbf{t}.\mathsf{reducer} \oplus \mathtt{result}\langle \mathsf{int}\rangle. \\ & \mathsf{mapper}\,\&\begin{cases} \mathtt{datum}(\mathsf{int}).\mathbf{t} \\ \mathtt{stop}.\mathtt{end} \end{cases}\end{aligned}$$

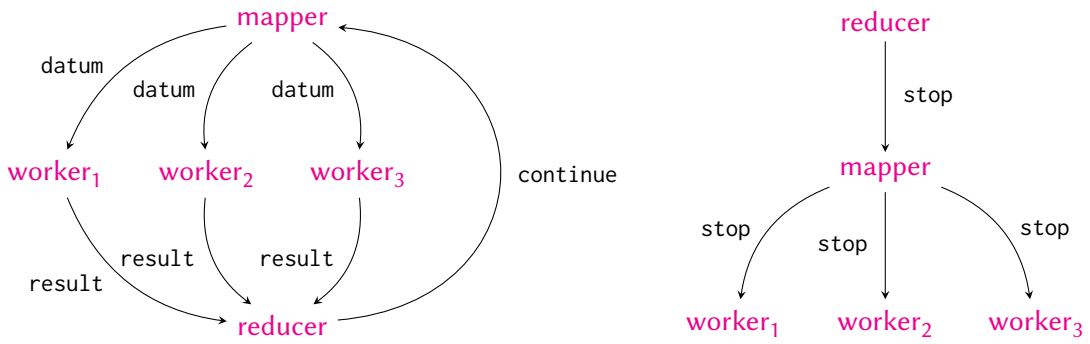

Figure 5.1: Illustration of the recursive map-reduce protocol, while the system is continuing (left) and stopping (right).

An illustration is given in Figure 5.1. The mapper sends `datum` to three workers, which each return a `result` to the reducer. The reducer then either `continues` (w.p. 0.4) or `stops` (w.p. 0.6). If continuing, the mapper distributes more tasks; otherwise, it signals the workers to stop.

Using Prose, we can determine that this protocol is safe, deadlock-free and terminating (the latter two *almost surely*).

```
$ prose verify examples/rec-map-reduce.ctx
Type safety
Result: true

Probabilistic deadlock freedom
Result: 1.0 (exact floating point)

Probabilistic termination
Result: 1.0 (exact floating point)
```

5.1.2 Knuth-Yao dice

Taking inspiration from a PRISM case study[1], we consider a dice program due to [Knuth and Yao, 1976], illustrated in Figure 5.2. It describes a Markov chain modelling a six-sided die using only coin flips.

The corresponding context is given in Figure 5.3. We model each node x with two participants p_x and q_x to receive from and send to different participants in a single $\oplus$ or &. The i-th face of the die is represented by the participant d_i. If outcome 1 is chosen, d_1 sends `repeat` to dummy

[1] https://www.prismmodelchecker.org/casestudies/dice.php

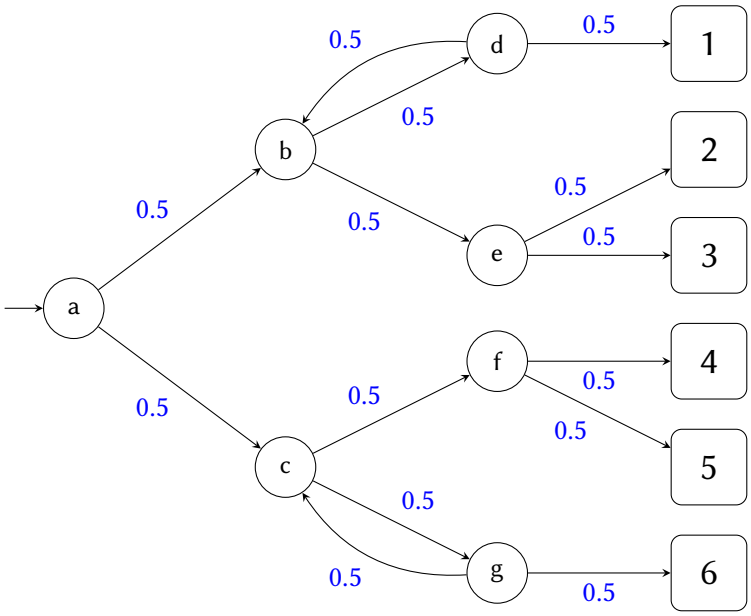

Figure 5.2: Illustration of the Knuth-Yao dice program.

$$p_a : q_a \oplus \begin{cases} 0.5 : \mathsf{a.end} \\ 0.5 : \mathsf{b.end} \end{cases} \qquad q_a : p_a \,\&\, \begin{cases} \mathsf{a}.\mu \mathbf{t}.p_b \oplus \mathsf{go}.q_d \,\&\, \mathsf{redo}.\mathbf{t} \\ \mathsf{b}.\mu \mathbf{t}.p_c \oplus \mathsf{go}.q_g \,\&\, \mathsf{redo}.\mathbf{t} \end{cases}$$

$$p_b : \mu \mathbf{t}.q_a \,\&\, \mathsf{go}.q_b \oplus \begin{cases} 0.5 : \mathsf{d}.\mathbf{t} \\ 0.5 : \mathsf{e}.\mathbf{t} \end{cases} \qquad q_b : \mu \mathbf{t}.p_b \,\&\, \begin{cases} \mathsf{d}.p_d \oplus \mathsf{go}.\mathbf{t} \\ \mathsf{e}.p_e \oplus \mathsf{go}.\mathbf{t} \end{cases}$$

$$p_c : \mu \mathbf{t}.q_a \,\&\, \mathsf{go}.q_c \oplus \begin{cases} 0.5 : \mathsf{f}.\mathbf{t} \\ 0.5 : \mathsf{g}.\mathbf{t} \end{cases} \qquad q_c : \mu \mathbf{t}.p_c \,\&\, \begin{cases} \mathsf{f}.p_f \oplus \mathsf{go}.\mathbf{t} \\ \mathsf{g}.p_g \oplus \mathsf{go}.\mathbf{t} \end{cases}$$

$$p_d : \mu \mathbf{t}.q_b \,\&\, \mathsf{go}.q_d \oplus \begin{cases} 0.5 : \mathsf{b}.\mathbf{t} \\ 0.5 : \mathsf{one.end} \end{cases} \qquad q_d : \mu \mathbf{t}.p_d \,\&\, \begin{cases} \mathsf{b}.q_a \oplus \mathsf{redo}.\mathbf{t} \\ \mathsf{one}.d_1 \oplus \mathsf{done.end} \end{cases}$$

$$p_e : q_b \,\&\, \mathsf{go}.q_e \oplus \begin{cases} 0.5 : \mathsf{two.end} \\ 0.5 : \mathsf{three.end} \end{cases} \qquad q_e : p_e \,\&\, \begin{cases} \mathsf{two}.d_2 \oplus \mathsf{done.end} \\ \mathsf{three}.d_3 \oplus \mathsf{done.end} \end{cases}$$

$$p_f : q_c \,\&\, \mathsf{go}.q_f \oplus \begin{cases} 0.5 : \mathsf{four.end} \\ 0.5 : \mathsf{five.end} \end{cases} \qquad q_f : p_f \,\&\, \begin{cases} \mathsf{four}.d_4 \oplus \mathsf{done.end} \\ \mathsf{five}.d_5 \oplus \mathsf{done.end} \end{cases}$$

$$p_g : \mu \mathbf{t}.q_c \,\&\, \mathsf{go}.q_g \oplus \begin{cases} 0.5 : \mathsf{c}.\mathbf{t} \\ 0.5 : \mathsf{six.end} \end{cases} \qquad q_g : \mu \mathbf{t}.p_g \,\&\, \begin{cases} \mathsf{c}.q_a \oplus \mathsf{redo}.\mathbf{t} \\ \mathsf{six}.d_6 \oplus \mathsf{done.end} \end{cases}$$

$$d_1 : q_d \,\&\, \mathsf{done}.\mu \mathbf{t}.\mathsf{dummy} \oplus \mathsf{repeat}.\mathbf{t} \qquad d_2 : q_e \,\&\, \mathsf{done.end} \qquad d_3 : q_e \,\&\, \mathsf{done.end}$$

$$d_4 : q_f \,\&\, \mathsf{done.end} \qquad d_5 : q_f \,\&\, \mathsf{done.end} \qquad d_6 : q_g \,\&\, \mathsf{done.end}$$

$$\mathsf{dummy} : \mu \mathbf{t}.d_1 \,\&\, \mathsf{repeat}.\mathbf{t}$$

Figure 5.3: Typing context for Knuth-Yao dice.

forever; otherwise, dummy waits for d_1 indefinitely, so the context is deadlocked.

The probability of deadlock freedom therefore equals the probability of obtaining outcome 1.

```
$ prose verify examples/dice.ctx
...
Probabilistic deadlock freedom
Result: 0.16666698455810547 (+/- 1.1920963061161968E-6 estimated;
rel err 7.1525641942636435E-6)
...
```

As expected, outcome 1 has probability $\frac{1}{6}$. We can rearrange d_i to verify that other outcomes have probability $\frac{1}{6}$ too.

5.1.3 Monty Hall problem

The Monty Hall problem [Dickey et al., 1975] is a well-known probabilistic paradox. We consider two contexts

$$\Delta_{\text{stay}} = \text{car} : \mathsf{T}_{\text{car}},\ \text{host} : \mathsf{T}_{\text{host}},\ \text{player} : \mathsf{T}_{\text{stay}}$$

$$\Delta_{\text{change}} = \text{car} : \mathsf{T}_{\text{car}},\ \text{host} : \mathsf{T}_{\text{host}},\ \text{player} : \mathsf{T}_{\text{leave}},$$

where

$$\mathsf{T}_{\text{car}} = \text{host}\oplus\begin{cases}\frac{1}{3} : \ell_1.\text{end}\\ \frac{1}{3} : \ell_2.\text{end}\\ \frac{1}{3} : \ell_3.\text{end},\end{cases} \qquad \mathsf{T}_{\text{host}} = \text{car}\&\begin{cases}\ell_1.\text{player}\oplus\begin{cases}0.5 : \ell_2.\text{player}\&\ell_1.\text{end}\\ 0.5 : \ell_3.\text{player}\&\ell_1.\text{end}\end{cases}\\ \ell_2.\text{player}\oplus\ell_3.\text{player}\&\ell_2.\text{end}\\ \ell_3.\text{player}\oplus\ell_2.\text{player}\&\ell_3.\text{end},\end{cases}$$

$$\mathsf{T}_{\text{stay}} = \text{host}\&\begin{cases}\ell_2.\text{host}\oplus\ell_1.\text{end}\\ \ell_3.\text{host}\oplus\ell_1.\text{end},\end{cases} \qquad \mathsf{T}_{\text{change}} = \text{host}\&\begin{cases}\ell_2.\text{host}\oplus\ell_3.\text{end}\\ \ell_3.\text{host}\oplus\ell_2.\text{end}.\end{cases}$$

The player is on a game show with a choice of three doors, with one hiding a car. They initially pick door 1. The host then opens another door without the car and informs the player, who must then decide whether to stick with door 1 or switch to the remaining unopened door. Is switching advantageous? Intuitively, it might seem not.

In Δ_{stay}, the player keeps door 1; in Δ_{change}, they switch. Both contexts terminate iff the correct door is chosen. Running Prose on the both gives:

```
$ prose verify examples/monty-hall-stay.ctx
```

Table 5.1: Benchmark results for Prose.

Typing context	Translation (ms)	Safety (s)	PDF (s)	PTerm (s)	End-to-end (s)
auth	0.237 ± 0.002	0.338 ± 0.004	0.331 ± 0.001	0.328 ± 0.001	0.339 ± 0.002
dice	0.293 ± 0.008	0.386 ± 0.001	0.414 ± 0.001	0.387 ± 0.002	0.448 ± 0.001
different-sort	0.104 ± 0.008	0.330 ± 0.001	0.328 ± 0.001	0.327 ± 0.001	0.330 ± 0.002
monty-hall-change	0.163 ± 0.008	0.327 ± 0.001	0.327 ± 0.001	0.333 ± 0.001	0.337 ± 0.001
monty-hall-stay	0.163 ± 0.008	0.330 ± 0.001	0.334 ± 0.002	0.329 ± 0.001	0.341 ± 0.002
more-choices	0.107 ± 0.008	0.328 ± 0.001	0.328 ± 0.001	0.328 ± 0.001	0.332 ± 0.002
multiparty-workers	0.219 ± 0.008	0.394 ± 0.006	0.434 ± 0.002	0.375 ± 0.001	0.515 ± 0.003
non-terminating	0.118 ± 0.009	0.331 ± 0.002	0.332 ± 0.001	0.329 ± 0.001	0.333 ± 0.001
open	0.121 ± 0.008	0.329 ± 0.001	0.328 ± 0.001	0.338 ± 0.002	0.335 ± 0.001
prob-deadlock	0.123 ± 0.009	0.334 ± 0.001	0.341 ± 0.003	0.329 ± 0.001	0.338 ± 0.001
rec-map-reduce	0.172 ± 0.008	0.335 ± 0.001	0.331 ± 0.001	0.334 ± 0.001	0.345 ± 0.001
rec-two-buyers	0.146 ± 0.008	0.327 ± 0.001	0.341 ± 0.003	0.339 ± 0.002	0.336 ± 0.001
same-labels	0.140 ± 0.008	0.331 ± 0.001	0.337 ± 0.001	0.344 ± 0.001	0.337 ± 0.002
simple	0.108 ± 0.008	0.331 ± 0.001	0.333 ± 0.001	0.332 ± 0.001	0.333 ± 0.001
sync-alone	0.132 ± 0.008	0.337 ± 0.001	0.339 ± 0.001	0.339 ± 0.002	0.334 ± 0.001
translation-example	0.126 ± 0.008	0.332 ± 0.001	0.333 ± 0.001	0.332 ± 0.002	0.336 ± 0.001
unsafe	0.130 ± 0.008	0.333 ± 0.001	0.328 ± 0.002	0.333 ± 0.002	0.337 ± 0.001
unsafe-2	0.123 ± 0.008	0.332 ± 0.002	0.334 ± 0.001	0.332 ± 0.001	0.340 ± 0.001

```
...
Probabilistic termination
Result: 0.333333 (exact floating point)

$ prose verify examples/monty-hall-change.ctx
...
Probabilistic termination
Result: 0.666667 (exact floating point)
```

So in fact switching doubles the player's chance of winning the car!

5.2 Performance

We now evaluate the efficiency of Prose through benchmarks. The experiments were run on a laptop with an Apple M1 Pro processor and 32GB of RAM, with OCaml 5.1.1 and PRISM v4.8.1. We run each measurement 30 times and report their mean and standard error. For each context, we record the translation time, time for invoking PRISM on each property separately, and the overall end-to-end runtime.

The results are presented in Table 5.1, and all context files are provided in Appendix A. We find that all contexts are verified in well under a second, demonstrating that Prose is suitable for practical use. Notably, translation cost is negligible (<0.3ms), making PRISM model checking the

dominating cost.

Interestingly, end-to-end runtimes are similar to those for checking individual properties, as PRISM internal model construction introduces a significant cost. Since the model is built once per PRISM invocation, this cost is reflected in all single-property runtimes but only once in the end-to-end runtime.

Chapter 6

Conclusion

In this project, we have introduced a new method for verifying probabilistic distributed protocols. We extended bottom-up multiparty session types with probabilities (Chapter 2), developed an encoding into PRISM (Chapter 3), and proved its correctness. We then implemented the verification procedure in Prose (Chapter 4) and demonstrated its practicality through case studies and performance benchmarks (Chapter 5).

This project lays the groundwork for many extensions. We are currently extending our type system to support *sub-probabilities*, where internal choices have probabilities summing to less than one. These types represent underspecified protocols, enabling the verification of processes with incomplete behaviour. Experimental support for this extension is already implemented in Prose. We also plan to extend Prose to verify additional properties such as *probabilistic liveness*, and use Prose to verify larger-scale protocols such as those used in the distributed training of machine learning models.

References

[Aman and Ciobanu, 2019] Aman, B. and Ciobanu, G. (2019). Probabilities in session types. *Electronic Proceedings in Theoretical Computer Science*, 303:92–106.

[Aspnes and Herlihy, 1990] Aspnes, J. and Herlihy, M. (1990). Fast randomized consensus using shared memory. *Journal of Algorithms*, 15(1):441–460.

[Baier, 1998] Baier, C. (1998). On algorithmic verification methods for probabilistic systems. Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim.

[Baier and Katoen, 2008] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.

[Bunte et al., 2019] Bunte, O., Groote, J. F., Keiren, J. J. A., Laveaux, M., Neele, T., de Vink, E. P., Wesselink, W., Wijs, A., and Willemse, T. A. C. (2019). The mCRL2 toolset for analysing concurrent systems. In Vojnar, T. and Zhang, L., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39, Cham. Springer International Publishing.

[Carbone and Veschetti, 2024] Carbone, M. and Veschetti, A. (2024). A probabilistic choreography language for PRISM. In Castellani, I. and Tiezzi, F., editors, *Coordination Models and Languages*, pages 20–37, Cham. Springer Nature Switzerland.

[Das et al., 2023] Das, A., Wang, D., and Hoffmann, J. (2023). Probabilistic resource-aware session types. *Proc. ACM Program. Lang.*, 7(POPL).

[Dickey et al., 1975] Dickey, J., Gridgeman, N. T., Kingsley, M. C. S., Good, I. J., Carlson, J. E., Gianola, D., Kutner, M. H., and Selvin, S. (1975). Letters to the editor. *The American Statistician*, 29(3):131–134.

[Fehnker and Gao, 2006] Fehnker, A. and Gao, P. (2006). Formal verification and simulation for performance analysis for probabilistic broadcast protocols. In Kunz, T. and Ravi, S. S., editors, *Ad-Hoc, Mobile, and Wireless Networks*, pages 128–141, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Honda et al., 1998] Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In Hankin, C., editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Honda et al., 2008] Honda, K., Yoshida, N., and Carbone, M. (2008). Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284.

[Inverso et al., 2020] Inverso, O., Melgratti, H., Padovani, L., Trubiani, C., and Tuosto, E. (2020). Probabilistic Analysis of Binary Sessions. In Konnov, I. and Kovács, L., editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:21, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Knuth and Yao, 1976] Knuth, D. and Yao, A. (1976). *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press.

[Kwiatkowska et al., 2011] Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G. and Qadeer, S., editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer.

[Milner, 1978] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375.

[Scalas and Yoshida, 2019] Scalas, A. and Yoshida, N. (2019). Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL).

[Yoshida, 2024] Yoshida, N. (2024). Programming Language Implementations with Multiparty Session Types. In *Active Object Languages: Current Research Trends*, pages 147–165. Springer

Nature Switzerland.

Appendix A

Typing context examples

auth.ctx

```
(* Note: this context uses sub-probabilities as discussed briefly in the conclusion *)

s : b & {
      connect . c (+) {
                  0.1 : login . a & authorise . end,
                  0.3 : cancel . e (+) terminate . end
                },
      networkerror . mu t . b & retry . t
    }

c : s & {
      login . a (+) pass . end,
      cancel . a (+) quit . end
    }

a : c & {
      pass . a (+) authorise . end,
      quit . end
    }

b : s (+) {
      0.6 : connect . end,
      0.4 : networkerror . mu t . s (+) retry . t
    }
```

dice.ctx

```
(* Knuth & Yao's Dice Program. Refer to
   https://www.prismmodelchecker.org/casestudies/dice.php

   We represent each vertex i with two processes (pi, qi), which allows us to
   simulate internal choice sending to different participants.
*)

p0 : q0 (+) {
```

```
        0.5 : l1 . end,
        0.5 : l2 . end
    }

q0 : p0 & {
        l1 . mu t .
             p1 (+) go . q3 & redo . t,
        l2 . mu t .
             p2 (+) go . q6 & redo . t
    }

p1 : mu t .
    q0 & go .
    q1 (+) {
        0.5 : l3 . t,
        0.5 : l4 . t
    }

q1 : mu t.
    p1 & {
        l3 . p3 (+) go . t,
        l4 . p4 (+) go . t
    }

p2 : mu t.
    q0 & go .
    q2 (+) {
        0.5 : l5 . t,
        0.5 : l6 . t
    }

q2 : mu t .
    p2 & {
        l5 . p5 (+) go . t,
        l6 . p6 (+) go . t
    }

p3 : mu t .
    q1 & go .
    q3 (+) {
        0.5 : l1 . t,
        0.5 : d1 . end
    }

q3 : mu t .
    p3 & {
        l1 . q0 (+) redo . t,
        d1 . dice1 (+) done . end
    }

p4 : q1 & go .
    q4 (+) {
        0.5 : d2 . end,
        0.5 : d3 . end
    }

q4 : p4 & {
        d2 . dice2 (+) done . end,
        d3 . dice3 (+) done . end
    }
```

```
p5 : q2 & go .
     q5 (+) {
        0.5 : d4 . end,
        0.5 : d5 . end
     }

q5 : p5 & {
        d4 . dice4 (+) done . end,
        d5 . dice5 (+) done . end
     }

p6 : mu t .
     q2 & go .
     q6 (+) {
        0.5 : d6 . end,
        0.5 : l2 . end
     }

q6 : mu t .
     p6 & {
        d6 . dice6 (+) done . end,
        l2 . q0 (+) redo . t
     }

(* Each of these should be of 1/6 probability *)

dice1 : q3 & done . mu t . dummy (+) repeat . t
dice2 : q4 & done . end
dice3 : q4 & done . end
dice4 : q5 & done . end
dice5 : q5 & done . end
dice6 : q6 & done . end

dummy : mu t . dice1 & repeat . t
```

different-sort.ctx

```
(* What happens if two participants try to communicate on the same label but
  different sorts (basic types)? *)

p : q (+) l(Int) . end

q : p & l(Bool) . end
```

monty-hall-change.ctx

```
(* Monty Hall problem. In this variant, the contestant always switches doors
   to either 2 or 3, depending on whichever door the host opens.

   The probability of deadlock freedom corresponds with the probability of
   picking the door with the car.

   Compare with [monty-hall-stay.ctx]. *)

car : host (+) {
        0.333333 : l1 . end,
        0.333333 : l2 . end,
        0.333334 : l3 . end
```

```
      }

host : car & {
         l1 . player (+) {
           0.5 : l2 . player & l1 . end,
           0.5 : l3 . player & l1 . end
         },
         l2 . player (+) l3 . player & l2 . end,
         l3 . player (+) l2 . player & l3 . end
      }

player : host & {
           l2 . host (+) l3 . end,
           l3 . host (+) l2 . end
         }
```

monty-hall-stay.ctx

```
(* Monty Hall problem. In this variant, the contestant always picks Door 1.
   The probability of deadlock freedom corresponds with the probability of
   picking the door with the car.

   Compare with [monty-hall-change.ctx]. *)

car : host (+) {
        0.333333 : l1 . end,
        0.333333 : l2 . end,
        0.333334 : l3 . end
      }

host : car & {
         l1 . player (+) {
           0.5 : l2 . player & l1 . end,
           0.5 : l3 . player & l1 . end
         },
         l2 . player (+) l3 . player & l2 . end,
         l3 . player (+) l2 . player & l3 . end
      }

player : host & {
           l2 . host (+) l1 . end,
           l3 . host (+) l1 . end
         }
```

more-choices.ctx

```
p : q (+) l1 . end

q : mu t . p & {
             l1 . end,
             l2 . t
           }
```

multiparty-workers.ctx

```
starter : workerA1 (+) datum(Int) .
```

```
          workerA2 (+) datum(Int) .
          workerA3 (+) datum(Int) .
          end

workerA1 : starter & datum(Int) .
           mu t .
             workerB1 (+) {
               0.5 : datum(Int) . workerC1 & result(Int) . t,
               0.5 : stop . end
             }

workerB1 : mu t .
             workerA1 & {
               datum(Int) . workerC1 (+) datum(Int) . t,
               stop . workerC1 (+) stop . end
             }

workerC1 : mu t .
             workerB1 & {
               datum(Int) . workerA1 (+) result . t,
               stop . end
             }


workerA2 : starter & datum(Int) .
           mu t .
             workerB2 (+) {
               0.5 : datum(Int) . workerC2 & result(Int) . t,
               0.5 : stop . end
             }

workerB2 : mu t .
             workerA2 & {
               datum(Int) . workerC2 (+) datum(Int) . t,
               stop . workerC2 (+) stop . end
             }

workerC2 : mu t .
             workerB2 & {
               datum(Int) . workerA2 (+) result . t,
               stop . end
             }


workerA3 : starter & datum(Int) .
           mu t .
             workerB3 (+) {
               0.5 : datum(Int) . workerC3 & result(Int) . t,
               0.5 : stop . end
             }

workerB3 : mu t .
             workerA3 & {
               datum(Int) . workerC3 (+) datum(Int) . t,
               stop . workerC3 (+) stop . end
             }

workerC3 : mu t .
             workerB3 & {
               datum(Int) . workerA3 (+) result . t,
               stop . end
```

```
            }
```

non-terminating.ctx

```
a : b (+) {
      0.5 : l1 . end,
      0.5 : l2 . mu t . b (+) l2 . t
    }

b : mu t .
    a & {
      l1 . end,
      l2 . t
    }
```

open.ctx

```
alice : bob (+) { 0.33 : a.end, 0.33 : b . carol(+) c . end, 0.34 : c . end }

bob : alice & { a.end, b.end, c.end }
```

prob-deadlock.ctx

```
commander : a (+) {
              0.7 : deadlock . end,
              0.3 : nodeadlock . end
            }

a : commander & {
      deadlock . b & msg . end,
      nodeadlock . b (+) msg . end
    }

b : a & msg . end
```

rec-map-reduce.ctx

```
mapper : mu t .
           worker1 (+) datum(Int) .
           worker2 (+) datum(Int) .
           worker3 (+) datum(Int) .
           reducer & {
             continue(Int) . t,
             stop .
               worker1 (+) stop .
               worker2 (+) stop .
               worker3 (+) stop .
               end
           }

worker1 : mapper & datum(Int) .
          mu t .
            reducer (+) result(Int) .
            mapper & {
              datum(Int) . t,
```

```
              stop . end
            }

worker2 : mapper & datum(Int) .
          mu t .
            reducer (+) result(Int) .
            mapper & {
              datum(Int) . t,
              stop . end
            }

worker3 : mapper & datum(Int) .
          mu t .
            reducer (+) result(Int) .
            mapper & {
              datum(Int) . t,
              stop . end
            }

reducer : mu t .
            worker1 & result(Int) .
            worker2 & result(Int) .
            worker3 & result(Int) .
            mapper (+) {
              0.4 : continue(Int) . t,
              0.6 : stop.end
            }
```

rec-two-buyers.ctx

```
alice: shop(+)query(Str) .
       shop&price(Int) .
       mu t .
          bob (+) {
              0.5 : split(Int) . bob & {yes . shop (+) buy . end, no . t},
              0.5 : cancel . shop (+) no . end
          }

shop: alice&query(Str) . alice(+)price(Int) . alice&{buy.end, no.end}

bob: mu t .
       alice & {
         split(Int) . alice (+) {0.5 : yes.end, 0.5 : no.t},
         cancel . end
       }
```

same-labels.ctx

```
(* Previous iterations of the translation used ID(-) to work out the next state.
   This causes a problem in the following case.

   Suppose p::q::l1 is assigned ID 2 and p::q::l2 is assigned ID 1, and the state
   after q (+) l2 to be n. Then, the second q (+) l1 will first do an initial
   translation to n + 1, then skip by two to n + 3. This will exceed the state
   space of p.

   This test checks for this case.
*)
```

```
p : q (+) {
      0.5 : l1 . end,
      0.5 : l2 . q (+) l1 . end
    }

q : p & {
      l1 . end,
      l2 . p & l1 . end
    }


(* Try the symmetric case for if the ID ordering changes *)

p1 : q1 (+) {
      0.5 : l1 . q1 (+) l2 . end,
      0.5 : l2 . end
    }

q1 : p1 & {
      l1 . p1 & l2 . end,
      l2 . end
    }


(* Shuffle the ordering of the two branches *)

q2 : p2 & {
      l1 . end,
      l2 . p2 & l1 . end
    }

p2 : q2 (+) {
      0.5 : l2 . q2 (+) l1 . end,
      0.5 : l1 . end
    }
```

simple.ctx

```
alice : bob (+) { 0.33 : a.end, 0.67 : b(Int).end }

bob : alice & { a.end, b(Int).end }
```

sync-alone.ctx

```
(* What happens if we send to a recipient who does not ever expect to receive? *)

alice : bob (+) {
                0.4 : l1 . end,
                0.6 : l2 . end
        }

bob : charlie & {
               l1 . end,
               l2 . end
      }

charlie : bob (+) {
```

```
                  0.5 : l1 . end,
                  0.5 : l2 . end
          }

(* What about the other way? *)

a : b & {
      l1 . end,
      l2 . end
    }

b : c (+) {
      0.7 : l1 . end,
      0.3 : l2 . end
    }

c : b & {
      l1 . end,
      l2 . end
    }
```

translation-example.ctx

```
(* Translation example from the thesis *)

p : q (+) {
      0.2 : l1 . mu t . q (+) l1 . t,
      0.3 : l2 . q (+) l2 . end,
      0.5 : l3 . end
}

q : p & {
      l1 . mu t. p & l1 . t,
      l2 . p & l2 . end,
      l3 . end
}
```

unsafe.ctx

```
alice : bob (+) {
          0.6 : l1 . end,
          0.3 : l2 .
                bob (+) {
                  0.9 : l3 . end,
                  0.1 : l4 . end
                },
          0.1 : l5 . end
        }

bob : alice & {
        l1 . end,
        l2 . alice & l3 . end
      }
```

unsafe-2.ctx

```
(* Two pairs being unsafe in parallel *)

a : b (+) {
      0.4 : l1 . end,
      0.6 : l2 . end
    }

b : a & {
      l2 . end,
      l3 . end
    }

c : d (+) {
      0.3 : l1 . end,
      0.7 : l2 . end
    }

d : c & {
      l2 . end,
      l3 . end
    }
```