Denotational reasoning for asynchronous multiparty session types

Dylan McDermott[0000−0002−6705−1449] and Nobuko Yoshida[0000−0002−3925−8557]

University of Oxford
`dylan@dylanm.org, nobuko.yoshida@cs.ox.ac.uk`

Abstract. We provide the first denotational semantics for asynchronous multiparty session types with precise asynchronous subtyping. Our semantics enables us to reason about asynchronous message-passing, in which message-sending is non-blocking. It enables us to prove the correctness of communication optimisations, in particular, those involving reordering of messages. Our development crucially relies on modelling message-passing as a computational effect. We apply grading, a paradigm for tracking computational effects, to asynchronous message-passing, demonstrating that multiparty session typing can be viewed as an instance of grading. We demonstrate the utility of our model by showing that it forms an adequate denotational semantics for a call-by-value asynchronous message-passing calculus, that ensures communication safety, deadlock-freedom and liveness in the presence of communication optimisations.

Keywords: multiparty session type · asynchronous subtyping · denotational semantics · graded monad

1 Introduction

Multiparty session types (MPST) provide a typing discipline for ensuring that multiple participants communicating via message-passing conform to a *multiparty protocol*, satisfying desired safety and liveness properties such as (1) **communication safety** (aka *type-safety*) (no participant will receive a message with a value of an unexpected type or unexpected label); (2) **deadlock-freedom** (the participants will never get stuck because they are waiting for each other to send a message); (3) **liveness** (aka *progress* [8,11,10]) (if one participant wishes to communicate with another, then that communication will *eventually* happen following the protocol). Deadlock-freedom implies that two participants p and q cannot both be blocked waiting for messages from each other (wait-free), while liveness requires that, if either of p and q wants to communicate with the other, then that communication will eventually happen. The theory of MPST can guarantee these properties for multiple participants either by taking the *top-down approach* [21,44,17], which uses a *global type* to specify a global protocol, or by taking the *bottom-up approach* [38,18], which model-checks a set of local types to ensure a desired property. A vast number of session type theories have been proposed, but one simple question has been left open: while several semantic studies have been made for binary (2-party) session types based on, e.g., logical relations [1] and game semantics [9], no extensional denotational semantics for an expressive MPST calculus exists (see Section 8). Our challenge is to find a simple but effective denotational semantics for MPST, one that is useful for proving properties of message-passing programs.

We address this challenge by constructing a denotational semantics for MPST that is *extensional* in the sense that it hides non-observable behaviours of programs (e.g. internal reductions), but still captures observable behaviours (e.g. sending a message to another participant). This enables us to reason about systems, without non-observable details getting in the way. For programming languages without message-passing, extensional denotational semantics have been successfully used for program reasoning, for instance, proofs of equivalences between syntactically distinct programs (e.g. [23]). We do the same, but for MPST; we can use our semantics to prove the validity of simple optimizations of MPST systems (e.g. Example 13 below).

Specifically, we introduce a simple notion of **computation tree**, and show that it provides a model of asynchronous message-passing. We introduce `SafeMP`, an idealised call-by-value programming language with a type system based on session types. `SafeMP` features *asynchronous* message-passing, in which messages are buffered into queues so that sending a message does not block. We show that our computation trees form a denotational semantics for `SafeMP`, and this semantics is **adequate** with respect to a typed notion of *bisimilarity* for `SafeMP`. This notion of bisimilarity accounts for asynchrony, and thus our model can reason about program equivalences that rely on asynchrony.

We design `SafeMP`, and its interpretation using computation trees, by following the key insight that sending and receiving of messages are *computational effects*, analogous to mutating state or raising an exception. Recent work [24,32] has established **grading** as the key technique of tracking computational effects compositionally, and thus we design `SafeMP` as a graded type system (aka an *effect system*). A graded type system assigns both a type and a *grade* to each computation; here the grades are multiparty session types, which provide enough information to enforce our desired safety and liveness properties. Our type system is not based on linear logic, unlike previous session type systems (see Section 8). Integrating session types into a call-by-value λ-calculus elucidates the connection between effects and session types, and enables us to apply techniques from the computational effects literature to session types. This insight is crucial to the design of our semantics: the standard way of modelling a graded type systems is to use a *graded monad* [4,39,31,24], so we show that computation trees form a graded monad that can be used to model `SafeMP`.

Asynchronous message-passing is more widely adopted in distributed systems than *synchronous* message-passing, where a computation that sends a message has to wait until that message is received before making any further progress. We give an operational semantics for `SafeMP`, accounting for asynchrony in the usual way, namely by buffering messages into a notion of *queue*. For our denotational semantics, we can use a simpler setup that does not involve queues, but is still asynchronous. We also account for asynchrony in our types, using the sound and complete (**precise**) **asynchronous multiparty session subtyping** of Ghilezan et al. [18]. Asynchrony enables a more permissive subtyping, and hence a more permissive type system, than synchronous message-passing, because some deadlocks that occur with synchronous semantics do not occur with an asynchronous semantics. The precise asynchronous subtyping enables practically useful *communication optimisations*. Several sound algorithms have been developed [12,13,3] and implemented in programming languages such as Rust, MPI and C. Asynchronous subtyping has also been mechanised in Rocq [16].

In this paper, instead of taking the definition of subtyping from [18], we reformulate the definition from the ground up. Our definition of subtyping improves on [18] in that it only involves session types (not their infinite unfoldings as session *trees*), and also enables subtyping for session types with free type variables (which are used in `SafeMP`). Nevertheless, we show for closed session types that our definition is equivalent to [18].

Contributions. This paper provides an effective denotational semantics for MPST, based on the observation that message-passing is a computational effect. **Section 2** introduces our calculus `SafeMP`, the subsequent sections provide the main contributions of the paper. **Section 3** introduces our new formulation of asynchronous session subtyping. This is the first sound and complete definition of asynchronous session subtyping that permits subtyping in the presence of type variables, and the first that does not rely on session trees. **Section 4** introduces our session type system for `SafeMP`. This is the first session type system for a call-by-value language based on grading instead of linearity, and demonstrates the connection between session types and computational effects. **Section 5** introduces *computation trees* as a simple representation of asynchronous message-passing computation. It demonstrates how to account for asynchrony without using queues, and defines a graded monad for asynchronous message-passing. **Section 6** shows that computation trees provide a model of asynchronous message-passing, in the form of an **adequate** denotational semantics for `SafeMP`. **Section 7** demonstrates the utility of our model, by using the model to prove safety and liveness properties of `SafeMP`.

2 A call-by-value message-passing calculus

We begin by introducing the syntax and operational semantics of our message-passing calculus `SafeMP`, and giving a few examples. The purpose of `SafeMP` is to demonstrate our denotational semantics, and as such we only include enough features to have non-trivial message-passing programs. As such we do not focus on the *practicality* of `SafeMP`, neither for writing programs in `SafeMP`, nor for implementing `SafeMP` itself.[1] Since we want to emphasize the connection with computational effects, we base `SafeMP` on an existing effectful calculus, namely the *fine-grain call-by-value* calculus [29]. We take a simplified[2] fine-grain call-by-value, and add asynchronous message-passing as an effect, along with first-order guarded recursive functions.

First, we start with some basic terminology. A *message* $m = (\ell, v)$ is a pair of a *label* ℓ and a *payload* v. We assume throughout the paper that there is some fixed set $\mathcal{L}$ of labels; ℓ ranges over elements of $\mathcal{L}$. A payload value is a constant value, for concreteness, we consider three *ground types* b, namely **unit**, **bool**, **int**, and require a message payload to be a constant v of one of those types, i.e. either $\star$ (of type **unit**), an integer n (of type **int**) or one of the booleans **true** and **false** (of type **bool**). We write $v : \mathsf{b}$ for the typing relation between constants and ground types. We also assume there is a fixed set of *participants* $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$. The idea is that the participants perform tasks in parallel and communicate by exchanging messages between each other. They could, for instance, be implemented as separate programs running on separate servers, with messages being sent across a network.

Terms of `SafeMP` are stratified into *values* v, w and *computations* t, u. They are generated inductively by the following grammar, subject to the constraints discussed below.

$$\begin{aligned} v, w \;&::=\; x \mid \star \mid n \mid \textbf{true} \mid \textbf{false} \\ t, u \;&::=\; \textbf{return}\, v \mid \textbf{let}\ x = t\ \textbf{in}\ u \mid v + w \mid v < w \mid \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \\ &\mid \textbf{send}\,(\ell, v)\,\textbf{to}\,\mathsf{p}\,\textbf{then}\,t \mid \textbf{recv}\ \mathsf{p}\ \{\ell_i\langle x_i : \mathsf{b}_i\rangle.\, t_i\}_{i\in I} \\ &\mid \textbf{let rec}\, f(x_1, \ldots, x_n) = t\ \textbf{in}\ u \mid f(v_1, \ldots, v_n) \end{aligned}$$

A value v is either a variable x, or one of the constants $\star$, n, **true** or **false**.

The first part of the grammar of computations is the core of fine-grain call-by-value: a computation can *return* a value v, or it can be a sequence **let** $x = t$ **in** u of computations. The latter first evaluates t and then, if t returns a result, evaluates u with x bound to the result of t. Computations also include integer addition $v + w$, integer comparison $v < w$, and conditionals **if** v **then** t_1 **else** t_2. These

[1] In particular, we do not consider decidability. Indeed, asynchronous session subtyping is known to be undecidable [5], so an implementation would be of a sound and decidable approximation of subtyping, as discussed in [3]. An implementation would also have *grade polymorphism*, as implemented for instance in Granule [32].

[2] We designed our calculus so that deadlock-freedom and liveness are non-trivial, while the calculus is otherwise as simple as possible. In particular, compared to fine-grain call-by-value, we do not have higher-order functions. This is not because they would be difficult to handle, indeed an advantage of the graded approach is that it easily takes care of higher-order functions, at the cost of a little extra complexity. Compared to some works on session types, we also do not have session *delegation*. For our session types we targeted the same level of expressiveness as Ghilezan et al. [18], who also do not have delegation (precise asynchronous subtyping with session delegation is an open problem).

operate on values; thus to e.g. compare two integer-returning computations one would first use **let** to first evaluate the operands.

The computation **send** (ℓ, v) **to** p **then** t sends the message (ℓ, v) to participant p, and then continues as the computation t. The value v could for instance be the result of a computation evaluated using **let**. Since we use an *asynchronous* semantics, sending a message does not block; the computation does not have to wait for the message to be received before continuing as t. The computation **recv** $\mathsf{p}\,\{\ell_1\langle x_1 : \mathsf{b}_1\rangle.\, t_1, \ldots, \ell_n\langle x_n : \mathsf{b}_n\rangle.\, t_n\}$ receives one message from the participant p; if that message has label ℓ_i, it continues as t_i, with x_i bound to the message payload. If the message label is not one of the ℓ_i, or the payload does not have the corresponding type b_i, then this is a *communication error*; in our operational semantics below, reduction gets stuck. The message labels ℓ_i are drawn from our fixed set $\mathcal{L}$. We require the labels ℓ_i to be distinct from each other, and we require $n > 0$. We typically write **recv** $\mathsf{p}\,\{\ell_i\langle x_i : \mathsf{b}_i\rangle.\, t_i\}_{i\in I}$ where I is a finite non-empty set, but we do not consider I to be part of the syntax. This computation blocks until the corresponding message is received.

Computations also include recursive function definitions, which are written as **let rec** $f(x_1, \ldots, x_n) = t$ **in** u, and applications $f(v_1, \ldots, v_n)$ of these functions to arguments. Function names f are distinct from variables x, and they are not values. We require every recursive definition to be guarded, in the sense that every occurrence of f in t appears inside a **send** or a **recv**. This means that to do a recursive call, one first has to send or receive a message; this ensures that a computation cannot simply diverge, it eventually has to *do* something.

A value or computation is *closed* when it has no free variables x and no free function names f (though we permit it to have free participants).

We define an operational semantics[3] for `SafeMP`, in the form of a transition system labelled by *local actions* α.

$$\begin{array}{rll} \alpha ::= & \tau & \text{(internal action)} \\ \mid & \mathsf{p}!m & \text{(send message } m \text{ to participant } \mathsf{p}) \\ \mid & \mathsf{p}?m & \text{(receive message } m \text{ from participant } \mathsf{p}) \end{array}$$

We define our operational semantics in two steps. First, we define a synchronous notion of reduction $t \overset{\alpha}{\rightsquigarrow} u$ for closed computations, one that does not incorporate any form of reordering of messages from different participants. This is generated by the rules in the first part of Fig. 1. We use a notion of reduction context $\mathcal{R}[\square]$ for congruence rules; such a context is a computation with a single hole $\square$, indicating the position of the computation to reduce. We write $\mathcal{R}[t]$ for the replacement of the hole with a computation t.

The second step is to use queues to model *asynchronous* message-passing. A *queue* σ is a function that assigns, to every participant p, a finite ordered list of messages, such that this list is empty for all but finitely many p (so each queue only contains a finite number of messages). We write $\emptyset$ for the empty queue (which maps every participant to the empty list), write $(\mathsf{p} \lhd m) :: \sigma$ for adding

[3] We extend this to an operational semantics for a notion of *session* in Section 7 below.

$\boxed{\text{Computation reduction } t \overset{\alpha}{\rightsquigarrow} u}$

$$\text{Reduction contexts } \mathcal{R}[\Box] ::= \Box \mid \textbf{let } x = \mathcal{R}[\Box] \textbf{ in } u \mid \textbf{let rec } f(x_1, \ldots, x_n) = t \textbf{ in } \mathcal{R}[\Box]$$

$$[\text{Cong}]\ \frac{t \overset{\alpha}{\rightsquigarrow} u}{\mathcal{R}[t] \overset{\alpha}{\rightsquigarrow} \mathcal{R}[u]} \qquad [\text{LetR}]\ \frac{}{\textbf{let } x = \textbf{return } v \textbf{ in } u \overset{\tau}{\rightsquigarrow} u[x \mapsto v]}$$

$$[\text{IfT}]\ \frac{}{\textbf{if true then } t_1 \textbf{ else } t_2 \overset{\tau}{\rightsquigarrow} t_1} \qquad [\text{IfF}]\ \frac{}{\textbf{if false then } t_1 \textbf{ else } t_2 \overset{\tau}{\rightsquigarrow} t_2}$$

$$[\text{Sub}]\ \frac{m = n_1 - n_2}{(n_1 - n_2) \overset{\tau}{\rightsquigarrow} \textbf{return } m} \qquad [\text{LeT}]\ \frac{n_1 < n_2}{(n_1 < n_2) \overset{\tau}{\rightsquigarrow} \textbf{return true}} \qquad [\text{LeF}]\ \frac{n_1 \geq n_2}{(n_1 < n_2) \overset{\tau}{\rightsquigarrow} \textbf{return false}}$$

$$[\text{Send}]\ \frac{}{\textbf{send } (\ell, v) \textbf{ to } \mathsf{p} \textbf{ then } t \overset{\mathsf{p}!(\ell,v)}{\rightsquigarrow} t} \qquad [\text{Recv}]\ \frac{v : \mathsf{b}_j}{\textbf{recv } \mathsf{p}\ \{\ell_i\langle x_i : \mathsf{b}_i\rangle.\, t_i\}_{i \in I} \overset{\mathsf{p}?(\ell_j,v)}{\rightsquigarrow} t_j[x_j \mapsto v]}$$

$$[\text{Apply}]\ \frac{}{\begin{array}{l}\textbf{let rec } f(x_1, \ldots, x_n) = t \textbf{ in } \mathcal{R}[f(v_1, \ldots, v_n)] \\ \qquad \overset{\tau}{\rightsquigarrow} \textbf{let rec } f(x_1, \ldots, x_n) = t \textbf{ in } \mathcal{R}[t[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]]\end{array}}$$

$\boxed{\text{(Asynchronous) configuration reduction } (\rho, t, \sigma) \overset{\alpha}{\rightsquigarrow} (\rho, u, \sigma)}$

$$[\text{CInt}]\ \frac{t \overset{\tau}{\rightsquigarrow} u}{(\rho, t, \sigma) \overset{\tau}{\rightsquigarrow} (\rho, u, \sigma)} \qquad [\text{CProd}]\ \frac{t \overset{\mathsf{p}!m}{\rightsquigarrow} u}{(\rho, t, \sigma) \overset{\tau}{\rightsquigarrow} (\rho, u, (\mathsf{p} \lhd m) :: \sigma)} \qquad [\text{CCons}]\ \frac{t \overset{\mathsf{p}?m}{\rightsquigarrow} u}{(\rho :: (\mathsf{p} \lhd m), t, \sigma) \overset{\tau}{\rightsquigarrow} (\rho, u, \sigma)}$$

$$[\text{CSend}]\ \frac{}{(\rho, t, \sigma :: (\mathsf{p} \lhd m)) \overset{\mathsf{p}!m}{\rightsquigarrow} (\rho, t, \sigma)} \qquad [\text{CRecv}]\ \frac{}{(\rho, t, \sigma) \overset{\mathsf{p}?m}{\rightsquigarrow} ((\mathsf{p} \lhd m) :: \rho, t, \sigma)}$$

Fig. 1. Operational semantics of `SafeMP`.

a message m to the front of p's list, and write $\sigma :: (\mathsf{p} \lhd m)$ for adding m to the back of p's list. Thus, if m and m' have distinct labels or payloads, the queues $(\mathsf{p} \lhd m) :: (\mathsf{q} \lhd m') :: \sigma$ and $(\mathsf{q} \lhd m') :: (\mathsf{p} \lhd m) :: \sigma$ are equal exactly when $\mathsf{p} \neq \mathsf{q}$; the ordering of messages between different participants does not matter. A `SafeMP` *configuration* $\mathcal{C} = (\rho, t, \sigma)$ consists of a (receive) queue ρ, closed computation t, and a (send) queue σ. The queue ρ contains messages yet to be consumed by t, while the queue σ contains messages that have been produced by t.

Our operational semantics is defined at the bottom of Fig. 1; it consists of a reduction relation $(\rho, t, \sigma) \overset{\alpha}{\rightsquigarrow} (\rho', u, \sigma')$ for configurations. This essentially reduces t synchronously as above, except that all messages are buffered into one of the queues. A configuration of the form $(\emptyset, \mathcal{R}[\textbf{return } v], \emptyset)$, where $\mathcal{R}$ has no non-recursive **let**s, does not reduce; we consider this to be a completed execution, with result v. We define a partial function Result from configurations to values, with $\mathrm{Result}(\emptyset, \mathcal{R}[\textbf{return}\,v], \emptyset) = v$ and Result undefined otherwise.

Example 1 (modified from an example of [18]). The following computation t receives the outcome of a computation from a participant p, then tells q whether to continue with some other computation, or stop. The result of t is a boolean, indicating whether it sent a `stop` message to q.

$$\begin{aligned} t = \textbf{recv}\ \mathsf{p}\ \{&\mathtt{success}\langle x : \textbf{int}\rangle.\ \textbf{let}\ y = 0 < x\ \textbf{in}\\ &\qquad\qquad\textbf{if}\ y\ \textbf{then send}\ (\mathtt{cont}, x)\ \textbf{to}\ \mathsf{q}\ \textbf{then return false}\\ &\qquad\qquad\quad\textbf{else send}\ (\mathtt{stop}, \textbf{true})\ \textbf{to}\ \mathsf{q}\ \textbf{then return true}\\ &\mathtt{error}\langle x : \textbf{bool}\rangle.\ \textbf{send}\ (\mathtt{stop}, \textbf{false})\ \textbf{to}\ \mathsf{q}\ \textbf{then return true}\} \end{aligned}$$

For instance, if p's computation fails, then t will send $(\mathtt{stop}, \textbf{false})$ to q. We have computation reductions, and hence configuration reductions, as follows (omitting the intermediate configurations).

$$t \xrightarrow{\mathsf{p}?(\mathtt{error},\textbf{true})} \textbf{send}\ (\mathtt{stop}, \textbf{false})\ \textbf{to}\ \mathsf{q}\ \textbf{then return true} \xrightarrow{\mathsf{q}!(\mathtt{stop},\textbf{false})} \textbf{return true}$$

$$(\emptyset, t, \emptyset) \xrightarrow{\mathsf{p}?(\mathtt{error},\textbf{true})}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\mathsf{q}!(\mathtt{stop},\textbf{false})} (\emptyset, \textbf{return true}, \emptyset)$$

The above reductions do not require asynchrony. To demonstrate asynchronous message-passing, consider a computation u that decides which message to send to q, without first waiting for the message from p.

$$\begin{aligned} u = \textbf{if}\ y\ &\textbf{then send}\ (\mathtt{cont}, 0)\ \textbf{to}\ \mathsf{q}\ \textbf{then}\ u'(\textbf{false})\\ &\textbf{else send}\ (\mathtt{stop}, \textbf{false})\ \textbf{to}\ \mathsf{q}\ \textbf{then}\ u'(\textbf{true})\\ \text{where } u'(v) = {}&\textbf{recv}\ \mathsf{p}\ \{\ell\langle x : \mathsf{b}\rangle.\ \textbf{return}\ v\}_{\ell\langle\mathsf{b}\rangle\in\{\mathtt{success}\langle\textbf{int}\rangle,\mathtt{error}\langle\textbf{bool}\rangle\}} \end{aligned}$$

Since we buffer messages into queues, we have the following reductions, in which a message is received from p first – just like in the reduction of $(\emptyset, t, \emptyset)$ above. Here $\rho = (\mathsf{p} \lhd (\mathtt{error}, \textbf{true})) :: \emptyset$ and $\sigma = (\mathsf{q} \lhd (\mathtt{stop}, \textbf{false})) :: \emptyset$.

$$\begin{aligned} (\emptyset, \textbf{let}\ y = \textbf{return false in}\ u, \emptyset) &\xrightarrow{\mathsf{p}?(\mathtt{error},\textbf{true})} (\rho, \textbf{let}\ y = \textbf{return false in}\ u, \emptyset)\\ &\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} (\rho, u'(\textbf{true}), \sigma)\\ &\xrightarrow{\tau}\xrightarrow{\mathsf{q}!(\mathtt{stop},\textbf{false})} (\emptyset, \textbf{return true}, \emptyset) \end{aligned}$$

Example 2 (Global state). Our second example shows that we can emulate other computational effects via message-passing, in a manner reminiscent of *effect handlers* [35]. Specifically, we focus on mutable state. For concreteness, we assume the state consists of a single integer.

Consider two participants s (for state) and c (for client). Tracking the state is the role of s, which waits for a message from c: a `get` message indicates it should send the value of the state back to c, as the payload of a `st` message;

a `put` message indicates it should update the current value of the state; and a `done` message indicates that the interaction is finished, the state is no longer needed. A possible implementation of s would be the following computation t_{s}, where x is the current value of the state (initially 0).

$$
\begin{aligned}
t_{\mathsf{s}} = \textbf{let rec}\, f\, x = \textbf{recv}\, \mathsf{c}\, \{ \\
&\quad \mathtt{get}\langle z : \textbf{unit}\rangle.\, \textbf{send}\, (\mathtt{st}, x)\, \textbf{to}\, \mathsf{c}\, \textbf{then}\, f\, x, \\
&\quad \mathtt{put}\langle y : \textbf{int}\rangle.\, f\, y, \\
&\quad \mathtt{done}\langle z : \textbf{unit}\rangle.\, \textbf{return}\, \star \\
&\}\ \textbf{in}\, f\, 0
\end{aligned}
$$

$$
\begin{aligned}
t_{\mathsf{c},1} = {} & \textbf{send}\, (\mathtt{get}, \star)\, \textbf{to}\, \mathsf{s}\, \textbf{then} \\
& \textbf{recv}\, \mathsf{s}\, \{\mathtt{st}\langle x : \textbf{int}\rangle. \\
& \textbf{send}\, (\mathtt{put}, 0)\, \textbf{to}\, \mathsf{s}\, \textbf{then return}\, x\} \\
t_{\mathsf{c},2} = {} & \textbf{send}\, (\mathtt{get}, \star)\, \textbf{to}\, \mathsf{s}\, \textbf{then} \\
& \textbf{send}\, (\mathtt{put}, 0)\, \textbf{to}\, \mathsf{s}\, \textbf{then} \\
& \textbf{recv}\, \mathsf{s}\, \{\mathtt{st}\langle x : \textbf{int}\rangle.\textbf{return}\, x\}
\end{aligned}
$$

On the right above, we also give two partial implementations of the client; each gets the value of the state, and then sets the state to 0. One waits for the `st` message before sending `put`; the other eagerly sends the `put` message, which is valid because of asynchrony. The latter can be thought of an *optimisation* of the former. Neither sends a `done` message; one example of a complete implementation of c is $\textbf{let}\ x = t_{\mathsf{c},1}\ \textbf{in send}\, (\mathtt{done}, \star)\, \textbf{to}\, \mathsf{s}\, \textbf{then return}\, x$.

3 Asynchronous multiparty session types

Our type system for `SafeMP` is based around (multiparty) session types. A session type $\mathbb{T}$ describes a *protocol* that must be followed by a given participant in a distributed system. They are generated as follows, where X ranges over *session type variables*.

$$\mathbb{T}, \mathbb{U} ::= \mathsf{end} \mid \mathsf{p} \oplus_{i\in I} \ell_i\langle \mathsf{b}_i\rangle.\, \mathbb{T}_i \mid \mathsf{p}\, \&_{i\in I}\, \ell_i\langle \mathsf{b}_i\rangle.\, \mathbb{T}_i \mid \mathsf{X} \mid \mu\mathsf{X}.\, \mathbb{T}$$

end denotes the end of the protocol, it requires that there are no further interactions with the other participants. An *internal choice* $\mathsf{p} \oplus_{i\in I} \ell_i\langle \mathsf{b}_i\rangle.\, \mathbb{T}_i$ means send some message (ℓ_i, v), where $v : \mathsf{b}_i$, to participant p, and continue according to $\mathbb{T}_i$. Sending a message p with a label not in $\{\ell_i \mid i \in I\}$, or with the wrong payload type, is not permitted. We require the labels ℓ_i to be distinct from each other, and that I is non-empty. An *external choice* $\mathsf{p}\, \&_{i\in I}\, \ell_i\langle \mathsf{b}_i\rangle.\, \mathbb{T}_i$ means receive a message (ℓ_i, v) from participant p, and then continue according to $\mathbb{T}_i$. The participant p chooses the message, but is required to ensure that the label is one of the labels ℓ_i, and that the payload has the corresponding type b_i. Similar syntactic constraints apply, in particular distinctness of message labels. Finally, $\mu\mathsf{X}.\, \mathbb{T}$ is a recursive protocol, binding the type variable X; the session type $\mu\mathsf{X}.\, \mathbb{T}$ is equivalent to $\mathbb{T}[\mathsf{X} \mapsto \mu\mathsf{X}.\, \mathbb{T}]$. Recursion is required to be *guarded*, in the sense that every occurrence of X in $\mathbb{T}$ is inside an internal or external choice.

The (single-step) *unfolding* $\mathcal{U}(\mathbb{T})$ of a session type $\mathbb{T}$ is defined as follows. The unfolding is an equivalent session type to $\mathbb{T}$, but due to guardedness, $\mathcal{U}(\mathbb{T})$ is always either end, or an internal or external choice, or a type variable.

$$\mathcal{U}(\mu\mathsf{X}.\, \mathbb{T}) = \mathcal{U}(\mathbb{T})[\mathsf{X} \mapsto \mu\mathsf{X}.\, \mathbb{T}] \qquad \mathcal{U}(\mathbb{T}) = \mathbb{T}\ \text{ if } \mathbb{T} \text{ is not a recursive type}$$

If Θ is a set containing all of the free type variables of a session type $\mathbb{T}$, we will refer to $\mathbb{T}$ as a session type *over* Θ. A session type is *closed* when it has no free type variables.

Example 3. Recall the computations t and u from Example 1, where t first receives a message from p, but u first sends a message to q. The computations t and u follow the protocols described by $\mathbb{T}$ and $\mathbb{U}$ respectively, and our type system in Section 4 assigns these session types to the computations.

$$\mathbb{T} = \mathsf{p}\,\&\begin{cases}\mathtt{success}\langle\textbf{int}\rangle.\mathbb{T}'\\ \mathtt{error}\langle\textbf{bool}\rangle.\mathbb{T}'\end{cases} \quad \text{where } \mathbb{T}' = \mathsf{q}\oplus\begin{cases}\mathtt{cont}\langle\textbf{int}\rangle.\,\mathsf{end}\\ \mathtt{stop}\langle\textbf{bool}\rangle.\,\mathsf{end}\end{cases}$$

$$\mathbb{U} = \mathsf{q}\oplus\begin{cases}\mathtt{cont}\langle\textbf{int}\rangle.\mathbb{U}'\\ \mathtt{stop}\langle\textbf{bool}\rangle.\mathbb{U}'\end{cases} \quad \text{where } \mathbb{U}' = \mathsf{p}\,\&\begin{cases}\mathtt{success}\langle\textbf{int}\rangle.\,\mathsf{end}\\ \mathtt{error}\langle\textbf{bool}\rangle.\,\mathsf{end}\end{cases}$$

The computation u is also a valid implementation of $\mathbb{T}$; indeed, under the definition of *subtyping* below (Definition 4), we have $\mathbb{U} <: \mathbb{T}$ (but not $\mathbb{T} <: \mathbb{U}$).

Both of $\mathbb{T}$ and $\mathbb{U}$ permit sending a message to q, without necessarily waiting for a message from p; in the case of $\mathbb{T}$ this relies on asynchrony. Similarly, they both require the implementation to accept a message from p. On the other hand, an implementation of $\mathbb{U}$ is required to send a message to q, without waiting for one from p, while an implementation of $\mathbb{T}$ is permitted to wait.

3.1 Asynchronous multiparty session subtyping

Subtyping is a crucial aspect of session type systems. The intuition is that $\mathbb{T} <: \mathbb{U}$ holds whenever each implementation of $\mathbb{T}$ can be used as an implementation of $\mathbb{U}$, without causing any safety or liveness issues. Ghilezan et al. [18] provide a definition of $\mathbb{T} <: \mathbb{U}$, for closed session types, that is precise for asynchronous message-passing, in the sense that it exactly captures this intuition. By this metric, it is the best possible notion of subtyping, and so we adopt it here. However, their definition is inconvenient for several reasons: the definition relies on session *trees* (which are infinite data structures, unlike session *types*), and they do not provide a definition of subtyping for non-closed session types (which we rely on to type Example 2 above). We therefore provide a more convenient reformulation of asynchronous session typing. For closed session types, our definition of subtyping is equivalent to that of [18].

As Example 3 demonstrates, in the presence of asynchrony, determining whether an implementation is permitted or required to send or receive a message is non-trivial. We define relations and predicates on session types that capture exactly when these are the case; these are key to our definition of subtyping.

For sending, the relation $\mathbb{U} \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathbb{U}'$ means an implementation of $\mathbb{U}$ is permitted to send a message (ℓ, v) with $v : \mathsf{b}$, to p, and then follow $\mathbb{U}'$. (But it does not necessarily have to send such a message.) The predicate $\mathrm{Sends}_{\mathsf{p}}(\mathbb{T})$ means an implementation of $\mathbb{T}$ is required to send a message to p, and it is not permitted to wait for a message before doing so. These are both defined inductively, the rules are at the top of Fig. 2.

The two base cases [$\overset{!}{\hookrightarrow}$-BASE] and [SENDS-BASE] are self-explanatory. The rule [$\overset{!}{\hookrightarrow}$-$\oplus$] encodes the fact that asynchrony does not impose any ordering between messages sent to different participants; since $\mathsf{p} \neq \mathsf{q}$, the two messages will be placed into different parts of the send queue, and can be removed from the queue in any order. Thus, if we know that the implementation is permitted to send a message to p in *some* branch of the internal choice on q, then it is permitted to send a message to p. The implementation gets to choose which branches of the internal choice on q wishes to keep, because it is an *internal* choice. [SENDS-$\oplus$] similarly enables us to reorder messages in session types. For instance, a participant of type $\mathsf{q} \oplus \ell'\langle \mathsf{b}' \rangle.\, \mathsf{p} \oplus \ell\langle \mathsf{b} \rangle.\, \mathsf{end}$ may send a message to p first (the session types $\mathsf{q} \oplus \ell'\langle \mathsf{b}' \rangle.\, \mathsf{p} \oplus \ell\langle \mathsf{b} \rangle.\, \mathsf{end}$ and $\mathsf{p} \oplus \ell\langle \mathsf{b} \rangle.\, \mathsf{q} \oplus \ell'\langle \mathsf{b}' \rangle.\, \mathsf{end}$ are asynchronous subtypes of each other). For [$\overset{!}{\hookrightarrow}$-&], even though the implementation is required to accept a message from q, that does not prevent it from eagerly sending a message to p, even if $\mathsf{p} = \mathsf{q}$. This is the case because sending a message will not block; after sending the message to p, the implementation will immediately be ready to accept a message from q. However, note that we cannot discard branches of the external choice: the implementation does not get to choose to disallow certain messages from q just because it is sending a message to p. There is no analogous rule for $\mathrm{Sends}_{\mathsf{p}}$, because $\mathrm{Sends}_{\mathsf{p}}$ forbids waiting for a message. Finally, the two recursion rules [$\overset{!}{\hookrightarrow}$-REC] and [SENDS-REC] ensure that every session type $\mathbb{T}$ is treated as equivalent to its unfolding $\mathcal{U}(\mathbb{T})$.

For receiving, the relation $\mathbb{T} \xhookrightarrow{\mathsf{p}?\ell\langle \mathsf{b} \rangle} \mathbb{T}'$ means an implementation of $\mathbb{T}$ is *required* to accept a message (ℓ, v) with $v : \mathsf{b}$, from p, if p chooses to send one; after doing so, the implementation is required to follow $\mathbb{T}'$. The predicate $\mathrm{Recvs}_{\mathsf{p}}(\mathbb{T})$ means an implementation of $\mathbb{T}$ is permitted to wait for a message from p, in particular, it is not required to send any messages before such a message from p arrives. These are again defined inductively, the rules (bottom half of Fig. 2) are exactly the duals of the sending rules (i.e. we obtain them by swapping sending and receiving).

These relations and predicates provide the bulk of our subtyping definition.

Definition 4. *Let Θ be a set of session type variables.* Asynchronous subtyping *is the largest binary relation $<:_\Theta$ between session types over Θ, such that the following hold when $\mathbb{T} <:_\Theta \mathbb{U}$. (When Θ is empty, we write just $\mathbb{T} <: \mathbb{U}$.)*

1. *If $\mathcal{U}(\mathbb{T}) = \mathsf{p} \oplus_{i \in I} \ell_i\langle \mathsf{b}_i \rangle.\, \mathbb{T}_i$, then for every $i \in I$, there is some $\mathbb{U}_i$ such that $\mathbb{U} \xhookrightarrow{\mathsf{p}!\ell_i\langle \mathsf{b}_i \rangle} \mathbb{U}_i$ and $\mathbb{T}_i <:_\Theta \mathbb{U}_i$.*
2. *If $\mathcal{U}(\mathbb{T}) = \mathsf{p} \,\&_{i \in I}\, \ell_i\langle \mathsf{b}_i \rangle.\, \mathbb{T}_i$, then $\mathrm{Recvs}_{\mathsf{p}}(\mathbb{U})$.*
3. *If $\mathcal{U}(\mathbb{U}) = \mathsf{p} \oplus_{i \in I} \ell_i\langle \mathsf{b}_i \rangle.\, \mathbb{U}_i$, then $\mathrm{Sends}_{\mathsf{p}}(\mathbb{T})$.*
4. *If $\mathcal{U}(\mathbb{U}) = \mathsf{p} \,\&_{i \in I}\, \ell_i\langle \mathsf{b}_i \rangle.\, \mathbb{U}_i$, then for every $i \in I$, there is some $\mathbb{T}_i$ such that $\mathbb{T} \xhookrightarrow{\mathsf{p}?\ell_i\langle \mathsf{b}_i \rangle} \mathbb{T}_i$ and $\mathbb{T}_i <:_\Theta \mathbb{U}_i$.*
5. *For every $\mathsf{X} \in \Theta$, we have $\mathcal{U}(\mathbb{T}) = \mathsf{X}$ if and only if $\mathcal{U}(\mathbb{U}) = \mathsf{X}$.*

This is a coinductive definition, and as is usual for coinductive definitions, we can construct $<:_\Theta$ as the union over all relations satisfying the above conditions.

$$\boxed{\mathbb{U} \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathbb{U}'}$$

$$[\overset{!}{\hookrightarrow}\text{-BASE}]\ \frac{}{\mathsf{p} \oplus_{i\in I} \ell_i\langle\mathsf{b}_i\rangle.\,\mathbb{U}_i \xhookrightarrow{\mathsf{p}!\ell_i\langle\mathsf{b}_i\rangle} \mathbb{U}_i}$$

$$[\overset{!}{\hookrightarrow}\text{-}\oplus]\ \frac{\mathsf{p}\neq\mathsf{q} \qquad J\subseteq I \qquad \mathbb{U}_i \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathbb{U}'_i \text{ for all } i\in J}{\mathsf{q} \oplus_{i\in I} \ell'_i\langle\mathsf{b}'_i\rangle.\,\mathbb{U}_i \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathsf{q} \oplus_{i\in J} \ell'_i\langle\mathsf{b}'_i\rangle.\,\mathbb{U}'_i}$$

$$[\overset{!}{\hookrightarrow}\text{-}\&]\ \frac{\mathbb{U}_i \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathbb{U}'_i \text{ for all } i\in I}{\mathsf{q} \,\&_{i\in I} \ell'_i\langle\mathsf{b}'_i\rangle.\,\mathbb{U}_i \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathsf{q} \,\&_{i\in I} \ell'_i\langle\mathsf{b}'_i\rangle.\,\mathbb{U}'_i}$$

$$[\overset{!}{\hookrightarrow}\text{-REC}]\ \frac{\mathbb{U}[\mathsf{X}\mapsto\mu\mathsf{X}.\,\mathbb{U}] \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathbb{U}'}{\mu\mathsf{X}.\,\mathbb{U} \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathbb{U}'}$$

$$\boxed{\mathrm{Sends}_{\mathsf{p}}(\mathbb{T})}$$

$$[\text{SENDS-BASE}]\ \frac{}{\mathrm{Sends}_{\mathsf{p}}(\mathsf{p} \oplus_{i\in I} \ell_i\langle\mathsf{b}_i\rangle.\,\mathbb{T}_i)}$$

$$[\text{SENDS-}\oplus]\ \frac{\mathrm{Sends}_{\mathsf{p}}(\mathbb{T}_j) \quad \text{for all } j\in J}{\mathrm{Sends}_{\mathsf{p}}(\mathsf{q} \oplus_{j\in J} \ell_j\langle\mathsf{b}_i\rangle.\,\mathbb{T}_j)}$$

$$[\text{SENDS-REC}]\ \frac{\mathrm{Sends}_{\mathsf{p}}(\mathbb{T})}{\mathrm{Sends}_{\mathsf{p}}(\mu\mathsf{X}.\,\mathbb{T})}$$

$$\boxed{\mathbb{T} \xhookrightarrow{\mathsf{p}?\ell\langle\mathsf{b}\rangle} \mathbb{T}'}$$

$$[\overset{?}{\hookrightarrow}\text{-BASE}]\ \frac{}{\mathsf{p} \,\&_{i\in I} \ell_i\langle\mathsf{b}_i\rangle.\,\mathbb{T}_i \xhookrightarrow{\mathsf{p}?\ell_i\langle\mathsf{b}_i\rangle} \mathbb{T}_i}$$

$$[\overset{?}{\hookrightarrow}\text{-}\&]\ \frac{\mathsf{p}\neq\mathsf{q} \qquad J\subseteq I \qquad \mathbb{T}_i \xhookrightarrow{\mathsf{p}?\ell\langle\mathsf{b}\rangle} \mathbb{T}'_i \text{ for all } i\in J}{\mathsf{q} \,\&_{i\in I} \ell'_i\langle\mathsf{b}'_i\rangle.\,\mathbb{T}_i \xhookrightarrow{\mathsf{p}?\ell\langle\mathsf{b}\rangle} \mathsf{q} \,\&_{i\in J} \ell'_i\langle\mathsf{b}'_i\rangle.\,\mathbb{T}'_i}$$

$$[\overset{?}{\hookrightarrow}\text{-}\oplus]\ \frac{\mathbb{T}_i \xhookrightarrow{\mathsf{p}?\ell\langle\mathsf{b}\rangle} \mathbb{T}'_i \text{ for all } i\in I}{\mathsf{q} \oplus_{i\in I} \ell'_i\langle\mathsf{b}'_i\rangle.\,\mathbb{T}_i \xhookrightarrow{\mathsf{p}?\ell\langle\mathsf{b}\rangle} \mathsf{q} \oplus_{i\in I} \ell'_i\langle\mathsf{b}'_i\rangle.\,\mathbb{T}'_i}$$

$$[\overset{?}{\hookrightarrow}\text{-REC}]\ \frac{\mathbb{T}[\mathsf{X}\mapsto\mu\mathsf{X}.\,\mathbb{T}] \xhookrightarrow{\mathsf{p}?\ell\langle\mathsf{b}\rangle} \mathbb{T}'}{\mu\mathsf{X}.\,\mathbb{T} \xhookrightarrow{\mathsf{p}?\ell\langle\mathsf{b}\rangle} \mathbb{T}'}$$

$$\boxed{\mathrm{Recvs}_{\mathsf{p}}(\mathbb{U})}$$

$$[\text{RECVS-BASE}]\ \frac{}{\mathrm{Recvs}_{\mathsf{p}}(\mathsf{p} \,\&_{i\in I} \ell_i\langle\mathsf{b}_i\rangle.\,\mathbb{U}_i)}$$

$$[\text{RECVS-}\&]\ \frac{\mathrm{Recvs}_{\mathsf{p}}(\mathbb{U}_j) \quad \text{for all } j\in J}{\mathrm{Recvs}_{\mathsf{p}}(\mathsf{q} \,\&_{j\in J} \ell_j\langle\mathsf{b}_i\rangle.\,\mathbb{U}_j)}$$

$$[\text{RECVS-REC}]\ \frac{\mathrm{Recvs}_{\mathsf{p}}(\mathbb{U})}{\mathrm{Recvs}_{\mathsf{p}}(\mu\mathsf{X}.\,\mathbb{U})}$$

Fig. 2. Four inductively defined relations and predicates on multiparty session types

Example 5. Consider the session types of Example 3. We do not have $\mathbb{T} <: \mathbb{U}$; (2) and (3) do not hold, because $\mathrm{Recvs}_{\mathsf{p}}(\mathbb{U})$ and $\mathrm{Sends}_{\mathsf{q}}(\mathbb{T})$ are both false. However, we do have $\mathbb{U} <: \mathbb{T}$. This is because we have $\mathbb{T}' <: \mathbb{T}'$ and $\mathbb{U}' <: \mathbb{U}'$ (subtyping is reflexive by Lemma 7 below). To satisfy (1), it is therefore enough to note that $\mathbb{T} \xhookrightarrow{\mathsf{q}!\mathtt{cont}\langle\textbf{int}\rangle} \mathbb{U}'$ and $\mathbb{T} \xhookrightarrow{\mathsf{q}!\mathtt{stop}\langle\textbf{bool}\rangle} \mathbb{U}'$. To satisfy (4), it enough to note that $\mathbb{U} \xhookrightarrow{\mathsf{p}?\mathtt{success}\langle\textbf{int}\rangle} \mathbb{T}'$ and $\mathbb{U} \xhookrightarrow{\mathsf{p}?\mathtt{error}\langle\textbf{bool}\rangle} \mathbb{T}'$. (2), (3) and (5) are trivial.

Example 6. Asynchronous subtyping requires us to take great care with infinite protocols; the following example demonstrates one of the subtleties. Consider the following closed session types, where $\mathsf{p}\neq\mathsf{q}$.

$$\mathbb{U} = \mu\mathsf{X}.\,\mathsf{q}\,\&\begin{cases}\ell_1\langle\mathsf{b}_1\rangle.\,\mathsf{p}\,\&\,\ell\langle\mathsf{b}\rangle.\,\mathsf{end}\\ \ell_2\langle\mathsf{b}_2\rangle.\,\mathsf{X}\end{cases} \qquad \mathbb{U}' = \mathsf{p}\,\&\,\ell\langle\mathsf{b}\rangle.\,\mu\mathsf{X}.\,\mathsf{q}\,\&\begin{cases}\ell_1\langle\mathsf{b}_1\rangle.\,\mathsf{end}\\ \ell_2\langle\mathsf{b}_2\rangle.\,\mathsf{X}\end{cases}$$

We have neither $\mathbb{U} <: \mathbb{U}'$ nor $\mathbb{U}' <: \mathbb{U}$; we first explain informally why these instances of subtyping should not hold. Suppose that q only sends messages with label ℓ_2; this is permitted by both $\mathbb{U}$ and $\mathbb{U}'$. In this case, $\mathbb{U}' <: \mathbb{U}$ would lead to a failure of liveness: a participant implementing $\mathbb{U}'$ is permitted to wait for a message from p, but according to $\mathbb{U}$, no such message will ever come. The converse $\mathbb{U} <: \mathbb{U}'$ would lead to a different liveness failure: $\mathbb{U}'$ permits p to send a message, but a participant implementing $\mathbb{U}$ will not consume that message; it will keep waiting for more messages from q instead.

Proving that $\mathbb{U}' <: \mathbb{U}$ does not hold is easy: it would imply $\mathrm{Recvs}_{\mathsf{p}}(\mathbb{U})$, which is false. To see why $\mathbb{U} <: \mathbb{U}'$ does not hold, consider the following closed session types, where k ranges over natural numbers.

$$\mathbb{T}_0 = \mathsf{q} \mathbin{\&} \ell_1\langle \mathsf{b}_1\rangle.\,\mathsf{end} \qquad \mathbb{T}_{k+1} = \mathsf{q} \mathbin{\&} \begin{cases} \ell_1\langle \mathsf{b}_1\rangle.\,\mathsf{end} \\ \ell_2\langle \mathsf{b}_2\rangle.\,\mathbb{T}_k \end{cases} \qquad \mathbb{T}_\infty = \mu \mathsf{X}.\,\mathsf{q} \mathbin{\&} \begin{cases} \ell_1\langle \mathsf{b}_1\rangle.\,\mathsf{end} \\ \ell_2\langle \mathsf{b}_2\rangle.\,\mathsf{X} \end{cases}$$

We have $\mathbb{T}_{k+1} \xhookrightarrow{\mathsf{q}?\ell_2\langle \mathsf{b}_2\rangle} \mathbb{T}_k$ and $\mathbb{T}_\infty \xhookrightarrow{\mathsf{q}?\ell_2\langle \mathsf{b}_2\rangle} \mathbb{T}_\infty$; indeed $\mathbb{T}_\infty <: \mathbb{T}_k$ for all k. However, there is no k such that $\mathbb{T}_k <: \mathbb{T}_\infty$; this would imply, by an inductive argument, that $\mathbb{T}_0 <: \mathbb{T}_\infty$, which is false because there is no $\mathbb{T}'$ such that $\mathbb{T}_0 \xhookrightarrow{\mathsf{q}?\ell_2\langle \mathsf{b}_2\rangle} \mathbb{T}'$. We have $\mathbb{U} \xhookrightarrow{\mathsf{p}?\ell\langle \mathsf{b}\rangle} \mathbb{T}_k$ for every k, and in fact $\mathbb{U} \xhookrightarrow{\mathsf{p}?\ell\langle \mathsf{b}\rangle} \mathbb{T}'$ implies $\mathbb{T}'$ is a supertype of some $\mathbb{T}_k$. In particular, we do not have $\mathbb{U} \xhookrightarrow{\mathsf{p}?\ell\langle \mathsf{b}\rangle} \mathbb{T}_\infty$; informally this is because, while an implementation of $\mathbb{U}$ is required to receive a message from p, it is not then required to implement $\mathbb{T}_\infty$. We therefore do not have $\mathbb{U} <: \mathbb{U}'$: this subtyping would imply $\mathbb{U} \xhookrightarrow{\mathsf{p}?\ell\langle \mathsf{b}\rangle} \mathbb{T}'$ for some $\mathbb{T}' <: \mathbb{T}_\infty$, which cannot exist by the above and transitivity of $<:$ (Lemma 7 below).

Subtyping is reflexive, transitive, a congruence, and respects substitution:

Lemma 7. *Each of the following rules is admissible (if the premises hold, then so does the conclusion):*

$$\frac{}{\mathbb{T} <:_\Theta \mathbb{T}} \qquad \frac{\mathbb{S} <:_\Theta \mathbb{T} \qquad \mathbb{T} <:_\Theta \mathbb{U}}{\mathbb{S} <:_\Theta \mathbb{U}}$$

$$\frac{J \subseteq I \qquad \mathbb{T}_i <:_\Theta \mathbb{U}_i \textit{ for all } i \in J}{(\mathsf{p} \oplus_{i\in J} \ell_i\langle \mathsf{b}_i\rangle.\,\mathbb{T}_i) <:_\Theta (\mathsf{p} \oplus_{i\in I} \ell_i\langle \mathsf{b}_i\rangle.\,\mathbb{U}_i)} \qquad \frac{J \subseteq I \qquad \mathbb{T}_i <:_\Theta \mathbb{U}_i \textit{ for all } i \in J}{(\mathsf{p} \mathbin{\&}_{i\in I} \ell_i\langle \mathsf{b}_i\rangle.\,\mathbb{T}_i) <:_\Theta (\mathsf{p} \mathbin{\&}_{i\in J} \ell_i\langle \mathsf{b}_i\rangle.\,\mathbb{U}_i)}$$

$$\frac{\mathsf{X} \notin \Theta \qquad \mathbb{T} <:_{\Theta,\mathsf{X}} \mathbb{U}}{\mu \mathsf{X}.\,\mathbb{T} <:_\Theta \mu \mathsf{X}.\,\mathbb{U}} \qquad \frac{\mathbb{T} <:_{\mathsf{X}_1,\ldots,\mathsf{X}_n} \mathbb{U} \qquad \mathbb{T}_1 <:_\Theta \mathbb{U}_1 \quad \cdots \quad \mathbb{T}_n <:_\Theta \mathbb{U}_n}{\mathbb{T}[\mathsf{X}_1 \mapsto \mathbb{T}_1, \ldots, \mathsf{X}_n \mapsto \mathbb{T}_n] <:_\Theta \mathbb{U}[\mathsf{X}_1 \mapsto \mathbb{U}_1, \ldots, \mathsf{X}_n \mapsto \mathbb{U}_n]}$$

3.2 Sequencing

Unlike in previous works on session types, we have a notion of *sequencing* of computations, namely **let** $x = t$ **in** u. To assign a type to a sequence, we use a *multiplication* operation $\mathbb{T} \cdot \mathbb{T}'$ on session types. This essentially replaces end

with $\mathbb{T}'$ in $\mathbb{T}$ (once t is done, we run u). The definition is by recursion on $\mathbb{T}$:

$$\begin{aligned}
\mathsf{end} \cdot \mathbb{T}' &= \mathbb{T}' & (\mathsf{p} \oplus_{i \in I} \ell_i \langle \mathsf{b}_i \rangle . \mathbb{T}_i) \cdot \mathbb{T}' &= \mathsf{p} \oplus_{i \in I} \ell_i \langle \mathsf{b}_i \rangle . (\mathbb{T}_i \cdot \mathbb{T}') \\
\mathsf{X} \cdot \mathbb{T}' &= \mathsf{X} & (\mathsf{p} \,\&_{i \in I} \ell_i \langle \mathsf{b}_i \rangle . \mathbb{T}_i) \cdot \mathbb{T}' &= \mathsf{p} \,\&_{i \in I} \ell_i \langle \mathsf{b}_i \rangle . (\mathbb{T}_i \cdot \mathbb{T}') \\
(\mu \mathsf{X}. \mathbb{T}) \cdot \mathbb{T}' &= \mu \mathsf{X}. (\mathbb{T} \cdot \mathbb{T}') \ \text{ if } \mathsf{X} \text{ not free in } \mathbb{T}'
\end{aligned}$$

The set of closed session types forms a preordered monoid: the preorder is the asynchronous subtyping relation $\mathbb{T} <: \mathbb{U}$; the monoid operation is our multiplication $\mathbb{T} \cdot \mathbb{U}$; and the unit of the monoid is end. Multiplication $(\cdot)$ is associative, and also monotone (if $\mathbb{T} <: \mathbb{U}$ and $\mathbb{T}' <: \mathbb{U}'$ then $\mathbb{T} \cdot \mathbb{T}' <: \mathbb{U} \cdot \mathbb{U}'$).

In general, following Katsumata [24], a preordered monoid of *grades* is the basic structure required to give a graded type system; in particular, the multiplication operation is required to give a graded typing rule for sequencing. The grades in this paper are session types, and thus the fact that we can organize session types into a preordered monoid is crucial. We use the multiplication operation in the typing rule [Let] of the following section.

4 Session-type-graded type system for SafeMP

We now come to our type system for SafeMP. The goal is to assign, to each configuration, a type b and a session type $\mathbb{T}$, but to do so in such a way that we can prove safety and liveness properties. This is a *graded* type system in the terminology of for instance [32]; the *grades* here are the session types. Indeed, the typing rules for sequencing, returning and for subtyping are standard from graded type systems. This section demonstrates that we can view session types as an instance of grading.

We first define a typing judgement for values. This has the form $\Gamma \vdash^{\mathsf{v}} v : \mathsf{b}$, where the typing context Γ assigns types b' to variables x. The rules for value typing are at the top of Fig. 3.

We then define a typing judgement for computations, of the form $\Theta \,;\, \Psi \,;\, \Gamma \vdash t : \mathsf{b} \,;\, \mathbb{T}$. The session type $\mathbb{T}$ describes the behaviour of the computation t, in terms of the protocol it follows. Here Θ is a finite set of type variables; this contains all of the free type variables in the other components of the judgement. The *function typing context* Ψ tracks the recursive function definitions that are in scope. It maps function names f to triples of the form $(\mathsf{b}_1, \dots, \mathsf{b}_n) \xrightarrow{\mathbb{U}} \mathsf{b}'$, where $(\mathsf{b}_1, \dots, \mathsf{b}_n)$ is a possibly empty list of (argument) types, $\mathbb{U}$ is a session type over Θ, and b' is a (result) type. The session type $\mathbb{U}$ describes the protocol followed by an application $f\,(v_1, \dots, v_n)$. Annotating function types with a grade in this way is standard from the grading literature, the annotation is sometimes called the *latent effect* of the function. The symbol $;$ separates the different components of the judgement, it has no meaning by itself.

The computation typing judgement is defined inductively, by the rules in the middle of Fig. 3. We include a session subtyping rule [<:], using the above definition of subtyping; this is useful for typing conditionals, because [If] requires the two branches to have the same type. The [Send] and [Recv] rules record the

Value typing $\boxed{\varGamma \vdash^{\mathsf{v}} v : \mathsf{b}}$

$$\frac{(x : \mathsf{b}) \in \varGamma}{\varGamma \vdash^{\mathsf{v}} x : \mathsf{b}} \qquad \frac{}{\varGamma \vdash^{\mathsf{v}} \star : \textbf{unit}} \qquad \frac{}{\varGamma \vdash^{\mathsf{v}} n : \textbf{int}} \qquad \frac{}{\varGamma \vdash^{\mathsf{v}} \textbf{true} : \textbf{bool}} \qquad \frac{}{\varGamma \vdash^{\mathsf{v}} \textbf{false} : \textbf{bool}}$$

Computation typing $\boxed{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash t : \mathsf{b} \fatsemi \mathbb{T}}$

$$[<:]\,\frac{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash t : \mathsf{b} \fatsemi \mathbb{T} \quad \mathbb{T} <:_{\varTheta} \mathbb{U}}{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash t : \mathsf{b} \fatsemi \mathbb{U}} \qquad [\text{Ret}]\,\frac{\varGamma \vdash^{\mathsf{v}} v : \mathsf{b}}{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash \textbf{return}\, v : \mathsf{b} \fatsemi \mathsf{end}}$$

$$[\text{Let}]\,\frac{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash t : \mathsf{b} \fatsemi \mathbb{T} \quad \varTheta \fatsemi \varPsi \fatsemi \varGamma, x : \mathsf{b} \vdash u : \mathsf{b}' \fatsemi \mathbb{T}'}{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash \textbf{let}\ x = t\ \textbf{in}\ u : \mathsf{b}' \fatsemi \mathbb{T} \cdot \mathbb{T}'} \qquad [+]\,\frac{\varGamma \vdash^{\mathsf{v}} v : \textbf{int} \quad \varGamma \vdash^{\mathsf{v}} w : \textbf{int}}{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash v + w : \textbf{int} \fatsemi \mathsf{end}}$$

$$[<]\,\frac{\varGamma \vdash^{\mathsf{v}} v : \textbf{int} \quad \varGamma \vdash^{\mathsf{v}} w : \textbf{int}}{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash v < w : \textbf{bool} \fatsemi \mathsf{end}} \qquad [\text{If}]\,\frac{\varGamma \vdash^{\mathsf{v}} v : \textbf{bool} \quad \varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash t_i : \mathsf{b} \fatsemi \mathbb{T} \text{ for all } i \in \{1, 2\}}{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : \mathsf{b} \fatsemi \mathbb{T}}$$

$$[\text{Send}]\,\frac{\varGamma \vdash^{\mathsf{v}} v : \mathsf{b} \quad \varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash t : \mathsf{b}' \fatsemi \mathbb{T}}{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash \textbf{send}\,(\ell, v)\,\textbf{to}\,\mathsf{p}\,\textbf{then}\,t : \mathsf{b}' \fatsemi (\mathsf{p} \oplus \ell\langle \mathsf{b} \rangle.\, \mathbb{T})}$$

$$[\text{Recv}]\,\frac{\varTheta \fatsemi \varPsi \fatsemi \varGamma, x_i : \mathsf{b}_i \vdash t_i : \mathsf{b}' \fatsemi \mathbb{T}_i \text{ for all } i \in I}{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash \textbf{recv}\ \mathsf{p}\ \{\ell_i\langle x_i : \mathsf{b}_i\rangle.\, t_i\}_{i \in I} : \mathsf{b}' \fatsemi (\mathsf{p}\ \&_{i \in I}\ \ell_i\langle \mathsf{b}_i \rangle.\, \mathbb{T}_i)}$$

$$[\text{LetRec}]\,\frac{\begin{array}{c}\varTheta, X \fatsemi \varPsi, f : (\mathsf{b}_1, \ldots, \mathsf{b}_n) \xrightarrow{\mathsf{X}} \mathsf{b}' \fatsemi \varGamma, x_1 : \mathsf{b}_1, \ldots, x_n : \mathsf{b}_n \vdash t : \mathsf{b}' \fatsemi \mathbb{T} \\ \varTheta \fatsemi \varPsi, f : (\mathsf{b}_1, \ldots, \mathsf{b}_n) \xrightarrow{\mu\mathsf{X}.\,\mathbb{T}} \mathsf{b}' \fatsemi \varGamma \vdash u : \mathsf{b}'' \fatsemi \mathbb{T}'\end{array}}{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash \textbf{let rec}\ f(x_1, \ldots, x_n) = t\ \textbf{in}\ u : \mathsf{b}'' \fatsemi \mathbb{T}'}$$

$$[\text{App}]\,\frac{(f : (\mathsf{b}_1, \ldots, \mathsf{b}_n) \xrightarrow{\mathbb{T}} \mathsf{b}') \in \varPsi \quad \varGamma \vdash^{\mathsf{v}} v_1 : \mathsf{b}_1 \ \cdots\ \varGamma \vdash^{\mathsf{v}} v_n : \mathsf{b}_n}{\varTheta \fatsemi \varPsi \fatsemi \varGamma \vdash f(v_1, \ldots, v_n) : \mathsf{b}' \fatsemi \mathbb{T}}$$

Configuration typing $\boxed{\vdash (\rho, t, \sigma) : \mathsf{b} \fatsemi \mathbb{T}}$

$$[\text{CBase}]\,\frac{\cdot \fatsemi \cdot \fatsemi \cdot \vdash t : \mathsf{b} \fatsemi \mathbb{T}}{\vdash (\emptyset, t, \emptyset) : \mathsf{b} \fatsemi \mathbb{T}} \qquad [\text{CSend}]\,\frac{\begin{array}{c} v : \mathsf{b}' \quad \mathbb{U} \xhookrightarrow{\mathsf{p}!\ell\langle \mathsf{b}' \rangle} \mathbb{T} \\ \vdash (\rho, t, \sigma) : \mathsf{b} \fatsemi \mathbb{T}\end{array}}{\vdash (\rho, t, \sigma :: (\mathsf{p} \lhd (\ell, v))) : \mathsf{b} \fatsemi \mathbb{U}} \qquad [\text{CRecv}]\,\frac{\begin{array}{c} v : \mathsf{b}' \quad \mathbb{T} \xhookrightarrow{\mathsf{p}?\ell\langle \mathsf{b}' \rangle} \mathbb{U} \\ \vdash (\rho, t, \sigma) : \mathsf{b} \fatsemi \mathbb{T}\end{array}}{\vdash ((\mathsf{p} \lhd (\ell, v)) :: \rho, t, \sigma) : \mathsf{b} \fatsemi \mathbb{U}}$$

Fig. 3. Typing of values, computations, and configurations in `SafeMP`

send/receive in the session type assigned to the computation. The [LetRec] rule requires the body of the recursive function to have type $\mathbb{T}$ under the assumption that a recursive call will have type X; it then assigns the recursive session type $\mu\mathsf{X}.\,\mathbb{T}$ to the recursive function.

Finally, the typing judgement for configurations has the form $\vdash (\rho, t, \sigma) : \mathsf{b} \fatsemi \mathbb{T}$, and is defined inductively by the rules at the bottom of Fig. 3. There are no contexts because t is a closed computation; the rules ensure that $\mathbb{T}$ is a closed session type. The [CBase] rule uses our computation typing judgement with empty contexts (we write $\cdot$ for an empty context). The intuition for [CSend] is that the configuration in the conclusion is sending a message to p; we therefore need to ensure that the type $\mathbb{U}$ permits it to do so, and we do this using our relation $\xhookrightarrow{!}$. For [CRecv], the conclusion is only well-typed if the configuration in the assumption is able to receive a message from p; we ensure this is the case using $\overset{?}{\hookrightarrow}$. We do not explicitly include a subtyping rule for configurations, but such a rule is admissible: if $\vdash (\rho, t, \sigma) : \mathsf{b} \mathbin{;} \mathbb{T}$ and $\mathbb{T} <: \mathbb{U}$, then $\vdash (\rho, t, \sigma) : \mathsf{b} \mathbin{;} \mathbb{U}$.

Example 8. Recall the computations and session types from Example 1 and Example 3. We have $\vdash (\emptyset, t, \emptyset) : \textbf{bool} \mathbin{;} \mathbb{T}$, and $\cdot \mathbin{;} \cdot \mathbin{;} y : \textbf{bool} \vdash u : \textbf{bool} \mathbin{;} \mathbb{U}$. Both of these use subtyping to type the conditionals. Since $\mathbb{U} <: \mathbb{T}$, we therefore also have $\cdot \mathbin{;} \cdot \mathbin{;} y : \textbf{bool} \vdash u : \textbf{bool} \mathbin{;} \mathbb{T}$, so for instance we can assign to $(\emptyset, \textbf{let } y = \textbf{false in } u, \emptyset)$ the same type as $(\emptyset, t, \emptyset)$.

Example 9. Recall our global state example, from Example 2. Consider the following session types, where X is a type variable.

$$\mathbb{T}_{\mathsf{s}} = \mathsf{c} \mathbin{\&} \begin{cases} \mathtt{get}\langle \textbf{unit} \rangle.\, \mathsf{c} \oplus \mathtt{st}\langle \textbf{int} \rangle.\, \mathsf{X} \\ \mathtt{put}\langle \textbf{int} \rangle.\, \mathsf{X} \\ \mathtt{done}\langle \textbf{unit} \rangle.\, \mathsf{end} \end{cases} \qquad \mathbb{T}_{\mathsf{c}} = \mathsf{s} \oplus \begin{cases} \mathtt{get}\langle \textbf{unit} \rangle.\, \mathsf{s} \mathbin{\&} \mathtt{st}\langle \textbf{int} \rangle.\, \mathsf{X} \\ \mathtt{put}\langle \textbf{int} \rangle.\, \mathsf{X} \\ \mathtt{done}\langle \textbf{unit} \rangle.\, \mathsf{end} \end{cases}$$

We have $\vdash \mathcal{C}_{\mathsf{s}} : \textbf{unit} \mathbin{;} \mu\mathsf{X}.\, \mathbb{T}_{\mathsf{s}}$, and $\vdash \mathcal{C}_{\mathsf{c},1} : \textbf{int} \mathbin{;} \mu\mathsf{X}.\, \mathbb{T}_{\mathsf{c}}$ and $\vdash \mathcal{C}_{\mathsf{c},2} : \textbf{int} \mathbin{;} \mu\mathsf{X}.\, \mathbb{T}_{\mathsf{c}}$, where

$$\mathcal{C}_{\mathsf{s}} = (\emptyset, t_{\mathsf{s}}, \emptyset)$$
$$\mathcal{C}_{\mathsf{c},i} = (\emptyset, \textbf{let } x = t_{\mathsf{c},i} \textbf{ in send}\, (\mathtt{done}, \star)\, \textbf{to}\, \mathsf{s}\, \textbf{then return}\, x, \emptyset) \quad (i \in \{1, 2\})$$

The derivation for $\mathcal{C}_{\mathsf{s}}$ involves a derivation of

$$\mathsf{X} \mathbin{;} f : \textbf{int} \xrightarrow{\mathsf{X}} \textbf{unit} \mathbin{;} x : \textbf{int} \vdash \textbf{recv}\, \mathsf{c}\, \{\dots\} : \textbf{unit} \mathbin{;} \mathbb{T}_{\mathsf{s}}$$

where $\textbf{recv}\, \mathsf{c}\, \{\dots\}$ is the body of the recursive function in t_{s}.

The subject reduction theorem uses the inductive relations of Fig. 2.

Theorem 10 (Subject reduction). *Assume that* $\vdash (\rho, t, \sigma) : \mathsf{b} \mathbin{;} \mathbb{T}$*, and that* $(\rho, t, \sigma) \overset{\alpha}{\rightsquigarrow} (\rho', t', \sigma')$*. If* $\alpha = \tau$*, then* $\vdash (\rho', t', \sigma') : \mathsf{b} \mathbin{;} \mathbb{T}$*. If* $\alpha = \mathsf{p}!(\ell, v)$ *with* $v : \mathsf{b}'$*, then* $\vdash (\rho', t', \sigma') : \mathsf{b} \mathbin{;} \mathbb{U}$ *for some* $\mathbb{U}$ *such that* $\mathbb{T} \xhookrightarrow{\mathsf{p}!\ell\langle \mathsf{b}' \rangle} \mathbb{U}$*. If* $\alpha = \mathsf{p}?(\ell, v)$ *with* $v : \mathsf{b}'$*, then* $\vdash (\rho', t', \sigma') : \mathsf{b} \mathbin{;} \mathbb{U}$ *for every* $\mathbb{U}$ *such that* $\mathbb{T} \xhookrightarrow{\mathsf{p}?\ell\langle \mathsf{b}' \rangle} \mathbb{U}$*.*

4.1 Typed bisimulation

To compare a configuration to its interpretation in our denotational semantics below, we use a notion of equivalence between states of transition systems. Our notion of equivalence is based on *bisimulation*. However, in the context of session types, the appropriate notion of equivalence is one that is informed by the type. To see why, consider Example 8. The configurations $(\emptyset, t, \emptyset)$ and $(\emptyset, \textbf{let } y = \textbf{false in } u, \emptyset)$ do not have the same behaviour; for instance, if p sends $(\mathtt{success}, 1)$, then they send different messages to q. However, if we know that p cannot send $\mathtt{success}$, then they are equivalent. In particular, since $\mathbb{T} <: (\mathsf{p} \mathbin{\&} \mathtt{error}\langle \textbf{bool} \rangle.\, \mathbb{T}')$, we can assign the session type $\mathsf{p} \mathbin{\&} \mathtt{error}\langle \textbf{bool} \rangle.\, \mathbb{T}'$ to both configurations; by doing so, we are requiring p to send an `error` message and not `success`. The aforementioned configurations have the same behaviour when considered as configurations of type $\mathsf{p}\,\&\,\texttt{error}\langle\textbf{bool}\rangle.\,\mathbb{T}'$. We define a general notion of *typed bisimulation* for *typed transition systems*, to account for the session types.

Definition 11. *A* typed transition system $(S, \rightsquigarrow, \mathrm{Result}, \blacktriangleleft)$ *with result set* X*, consists of a set* S *of* states*, a binary relation* $\overset{\alpha}{\rightsquigarrow}$ *on* S *for each local action* α*, a partial function* Result *from* S *to* X*, and a relation* $\blacktriangleleft$ *between states and closed session types. We require that* $s \blacktriangleleft \mathbb{T}$ *implies* $s \blacktriangleleft \mathbb{U}$ *whenever* $\mathbb{T} <: \mathbb{U}$*.*

We write $s \overset{\alpha}{\rightsquigarrow}{}^{*}\, s'$ when there is a finite sequence of reductions ending in action α, where all the reductions before the last have action τ. When $\alpha = \tau$ we permit the sequence of reductions to be empty ($s = s'$), while for every other α we require there to be at least one reduction.

For each b, we obtain a typed transition system with result set $\{v \mid v : \mathsf{b}\}$: states are configurations, $\rightsquigarrow$ and Result are as defined in Section 2, and the typing relation $(\rho, t, \sigma) \blacktriangleleft \mathbb{T}$ holds when $\vdash (\rho, t, \sigma) : \mathsf{b} \,⨟\, \mathbb{T}$. Our notion of typed bisimulation is as follows.

Definition 12. *Let* $(S, \rightsquigarrow, \mathrm{Result}, \blacktriangleleft)$ *and* $(S', \rightsquigarrow, \mathrm{Result}, \blacktriangleleft)$ *be typed transition systems. A family of relations* $R_{\mathbb{S}} \subseteq S \times S'$*, indexed by closed session types* $\mathbb{S}$*, is a* typed bisimulation *when* $sR_{\mathbb{S}}s'$ *implies* $s \blacktriangleleft \mathbb{S}$*,* $s' \blacktriangleleft \mathbb{S}$*, and also the following.*

1. *(a)* $s \overset{\tau}{\rightsquigarrow} t$ *implies there exists* t' *such that* $s' \overset{\tau}{\rightsquigarrow}{}^{*}\, t'$ *and* $tR_{\mathbb{S}}t'$*; and (b)* $s' \overset{\tau}{\rightsquigarrow} t'$ *implies there exists* t *such that* $s \overset{\tau}{\rightsquigarrow}{}^{*}\, t$ *and* $tR_{\mathbb{S}}t'$*.*
2. *If* $\mathcal{U}(\mathbb{S}) = \mathsf{end}$*, then (a)* $\mathrm{Result}(s) = x$ *implies there exists* t' *such that* $s' \overset{\tau}{\rightsquigarrow}{}^{*}\, t'$ *and* $\mathrm{Result}(t') = x$*; and (b)* $\mathrm{Result}(s') = x$ *implies there exists* t *such that* $s \overset{\tau}{\rightsquigarrow}{}^{*}\, t$ *and* $\mathrm{Result}(t) = x$*.*
3. *If* $\mathrm{Sends}_{\mathsf{p}}(\mathbb{S})$ *and* $v : \mathsf{b}$*, then (a)* $s \overset{\mathsf{p}!(\ell,v)}{\rightsquigarrow} t$ *implies there exist* $\mathbb{T}, t'$ *such that* $\mathbb{S} \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathbb{T}$*,* $s' \overset{\mathsf{p}!(\ell,v)}{\rightsquigarrow}{}^{*}\, t'$ *and* $tR_{\mathbb{T}}t'$*; and (b)* $s' \overset{\mathsf{p}!(\ell,v)}{\rightsquigarrow} t'$ *implies there exist* $\mathbb{T}, t$ *such that* $\mathbb{S} \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathbb{T}$*,* $s \overset{\mathsf{p}!(\ell,v)}{\rightsquigarrow}{}^{*}\, t$ *and* $tR_{\mathbb{T}}t'$*.*
4. *If* $\mathbb{S} \xhookrightarrow{\mathsf{p}?\ell\langle\mathsf{b}\rangle} \mathbb{T}$ *and* $v : \mathsf{b}$*, then (a)* $s \overset{\mathsf{p}?(\ell,v)}{\rightsquigarrow} t \blacktriangleleft \mathbb{T}$ *implies there exists* $\mathbb{T}, t'$ *such that* $s' \overset{\mathsf{p}?(\ell,v)}{\rightsquigarrow}{}^{*}\, t'$ *and* $tR_{\mathbb{T}}t'$*; and (b)* $s' \overset{\mathsf{p}?(\ell,v)}{\rightsquigarrow} t' \blacktriangleleft \mathbb{T}$ *implies there exists* t *such that* $s \overset{\mathsf{p}?(\ell,v)}{\rightsquigarrow}{}^{*}\, t$ *and* $tR_{\mathbb{T}}t'$*.*

We write $\sim$ *for the largest typed bisimulation.*

The crucial points to note are that in (3), we only impose requirements on messages sent to p when $\mathrm{Sends}_{\mathsf{p}}$ holds, because p can only consume a message when that is the case; and that in (4) we only impose requirements on a message received from p when the session type requires that message to be accepted. As is usual for a coinductive definition, $\sim$ is equal to the union of all typed bisimulations.

Example 13. Returning to Example 8, we have $(\emptyset, t, \emptyset) \sim_{\mathbb{S}} (\emptyset, \textbf{let } y = \textbf{false in } u, \emptyset)$ for $\mathbb{S} = \mathsf{p} \,\&\, \mathtt{error}\langle\textbf{bool}\rangle.\,\mathbb{T}'$, but not for $\mathbb{S} = \mathbb{T}$. The difference is that, while $\mathbb{T} \xhookrightarrow{\mathsf{p?success}\langle\textbf{int}\rangle} \mathbb{T}'$ holds, no $\mathbb{S}'$ satisfies $(\mathsf{p} \,\&\, \mathtt{error}\langle\textbf{bool}\rangle.\,\mathbb{T}') \xhookrightarrow{\mathsf{p?success}\langle\textbf{int}\rangle} \mathbb{S}'$.

In the setting of Example 9, we have $(\emptyset, t_{\mathsf{c},1}, \emptyset) \sim_{\mathbb{S}} (\emptyset, t_{\mathsf{c},2}, \emptyset)$ where $\mathbb{S} = \mathsf{s} \oplus \mathtt{get}\langle\textbf{unit}\rangle.\,\mathsf{s} \,\&\, \mathtt{st}\langle\textbf{int}\rangle.\,\mathsf{s} \oplus \mathtt{put}\langle\textbf{int}\rangle.\,\mathsf{end}$, even though $t_{\mathsf{c},1}$ cannot send a `put` message until it receives a message from s. As a consequence, we can view $t_{\mathsf{c},2}$ as a sound optimisation of $t_{\mathsf{c},1}$. Our denotational semantics for `SafeMP` provides a sound and complete technique for proving this bisimilarity; see Example 21 below. This bisimilarity does not rely in any way on the implementation of s.

Lemma 14. *For every $\mathbb{S}$, the relation $\sim_{\mathbb{S}}$ is transitive and symmetric. Moreover, if $s \sim_{\mathbb{S}} s'$, and $\mathbb{S} <: \mathbb{T}$, then $s \sim_{\mathbb{T}} s'$.*

5 Computation trees

A session type describes the protocol that a given participant p must follow, in terms of the interactions it is meant to have with other participants in a system. The purpose of this section is to give some semantic meaning to this. We do this by introducing *computation trees* as a notion of asynchronous message-passing computation, and discussing how they relate to session trees.

Definition 15. Computation trees*, over a set X of* results*, are generated coinductively by the following grammar, where x ranges over elements of X, m ranges over messages, and M ranges over sets of messages.*

$$t ::= \mathtt{return}(x) \mid \mathtt{send}_{\mathsf{p},m}(t) \mid \mathtt{recv}_{\mathsf{p}}(t_m)_{m \in M}$$

Our first task is to show that these support asynchronous message-passing, without the use of queues. To do this, we make them into a transition system labelled by local actions α. Reduction $t \overset{\alpha}{\rightsquigarrow} u$ is defined inductively by the following rules (there are no τ transitions). The base cases [Send] and [Recv] are obvious, while the congruence rules make this notion of reduction an asynchronous one. In particular, to receive a message from p, we do not need the root of the tree to be a $\mathtt{recv}_{\mathsf{p}}$. The congruence rules enable us to reduce occurences of $\mathtt{recv}_{\mathsf{p}}$ appearing inside deeper in the tree, potentially discarding branches of other $\mathtt{recv}$s (which might not contain $\mathtt{recv}_{\mathsf{p}}$).

$$[\textsc{Send}]\ \frac{}{\mathtt{send}_{\mathsf{p},m}(t) \overset{\mathsf{p}!m}{\rightsquigarrow} t} \qquad [\textsc{Recv}]\ \frac{m \in M}{\mathtt{recv}_{\mathsf{p}}(t_{m'})_{m' \in M} \overset{\mathsf{p}?m}{\rightsquigarrow} t_m}$$

$$[\textsc{SendSend}]\ \frac{\mathsf{p} \neq \mathsf{q} \qquad t \overset{\mathsf{p}!m}{\rightsquigarrow} u}{\mathtt{send}_{\mathsf{q},m'}(t) \overset{\mathsf{p}!m}{\rightsquigarrow} \mathtt{send}_{\mathsf{q},m'}(u)} \qquad [\textsc{RecvSend}]\ \frac{t \overset{\mathsf{p}?m}{\rightsquigarrow} u}{\mathtt{send}_{\mathsf{q},m'}(t) \overset{\mathsf{p}?m}{\rightsquigarrow} \mathtt{send}_{\mathsf{q},m'}(u)}$$

$$[\textsc{RecvRecv}]\ \frac{\mathsf{p} \neq \mathsf{q} \qquad M' \subseteq M \qquad t_{m'} \overset{\mathsf{p}?m}{\rightsquigarrow} u_{m'} \text{ for all } m' \in M'}{\mathtt{recv}_{\mathsf{q}}(t_{m'})_{m' \in M} \overset{\mathsf{p}?m}{\rightsquigarrow} \mathtt{recv}_{\mathsf{q}}(u_{m'})_{m' \in M'}}$$

To make this into a typed transition system, we define $\mathrm{Result}(\texttt{return}(x)) = x$, with $\mathrm{Result}(t)$ undefined otherwise. The typing relation for computation trees is defined coinductively, in a similar manner to our definition of subtyping.

Definition 16. *We define a typing relation $t \blacktriangleleft \mathbb{T}$ between computation trees t and closed session types $\mathbb{T}$ coinductively, as the largest relation such that the following hold when $t \blacktriangleleft \mathbb{T}$.*

1. *If $t = \texttt{send}_{\mathsf{p},(\ell,v)}(u)$, with $v : \mathsf{b}$, then there is some $\mathbb{U}$ such that $\mathbb{T} \xhookrightarrow{\mathsf{p}!\ell\langle\mathsf{b}\rangle} \mathbb{U}$ and $u \blacktriangleleft \mathbb{U}$.*
2. *If $t = \texttt{recv}_{\mathsf{p}}(t_m)_{m\in M}$, then $\mathrm{Recvs}_{\mathsf{p}}(\mathbb{T})$.*
3. *If $\mathcal{U}(\mathbb{T}) = \mathsf{p} \oplus_{i\in I} \ell_i\langle\mathsf{b}_i\rangle.\, \mathbb{T}_i$, then there exist m, u such that $t \overset{\mathsf{p}!m}{\rightsquigarrow} u$.*
4. *If $\mathcal{U}(\mathbb{T}) = \mathsf{p} \&_{i\in I} \ell_i\langle\mathsf{b}_i\rangle.\, \mathbb{T}_i$, then there is some natural number h such that, for every $i \in I$ and $v : \mathsf{b}_i$, there is some $u \blacktriangleleft \mathbb{T}_i$ such that $t \overset{\mathsf{p}?(\ell_i,v)}{\rightsquigarrow} u$, where the derivation of the latter has height at most h.*[4]

(1) and (2) permit t to be a `send` or a `recv` only when this is permitted by the session type $\mathbb{T}$. (3) and (4) require t to send or receive a message, when the session type $\mathbb{T}$ says it must do so. We can construct $\blacktriangleleft$ concretely as a union of relations.

The definition of $\blacktriangleleft$ respects subtyping, in the sense that $t \blacktriangleleft \mathbb{T}$ implies $t \blacktriangleleft \mathbb{U}$ when $\mathbb{T} <: \mathbb{U}$. Moreover, $t \sim_{\mathbb{T}} t$ holds whenever $t \blacktriangleleft \mathbb{T}$, where $\sim_{\mathbb{T}}$ is typed bisimilarity between computation trees. Typed bisimilarity for computation trees is non-trivial, in that $t \sim_{\mathbb{T}} t'$ does not generally imply $t = t'$. One might therefore expect us to need to consider equivalence classes of computation trees instead of just computation trees. However, we can do better than this: up to typed bisimilarity, every typed computation tree has a normal form. It is these normal forms we will use in our model; this avoids the need for any quotient. The normal forms of type $\mathbb{T}$, with result set X, are the elements of the set $\mathcal{N}(X)_{\mathbb{T}}$ defined coinductively as follows. (Concretely, we can construct $\mathcal{N}(X)$ as a union.)

Definition 17. *We write $\mathcal{N}(X)$ for the largest family of sets, indexed by closed session types $\mathbb{T}$, such that $t \in \mathcal{N}(X)_{\mathbb{T}}$ implies the following.*

1. *If $\mathcal{U}(\mathbb{T}) = \mathsf{end}$, then $t = \texttt{return}(x)$ for some $x \in X$.*
2. *If $\mathcal{U}(\mathbb{T}) = \mathsf{p} \oplus_{i\in I} \ell_i\langle\mathsf{b}_i\rangle.\, \mathbb{T}_i$, then $t = \texttt{send}_{\mathsf{p},(\ell_i,v)}(t')$ for some $i \in I$, some $v : \mathsf{b}_i$, and some $t' \in \mathcal{N}(X)_{\mathbb{T}_i}$.*
3. *If $\mathcal{U}(\mathbb{T}) = \mathsf{p}\&_{i\in I}\ell_i\langle\mathsf{b}_i\rangle.\, \mathbb{T}_i$, then $t = \texttt{recv}_{\mathsf{p}}(t_m)_{m\in M}$ for some family $(t_m)_{m\in M}$, where $M = \{(\ell_i, v) \mid i \in I \wedge v : \mathsf{b}_i\}$, and $t_{(\ell_i,v)} \in \mathcal{N}(X)_{\mathbb{T}_i}$ for all $(\ell_i, v) \in M$.*

Lemma 18 (Normalization of typed computation trees). *Let $\mathbb{T}$ be a closed session type. We have $u \blacktriangleleft \mathbb{T}$ for every $u \in \mathcal{N}(X)_{\mathbb{T}}$. If t is a computation tree such that $t \blacktriangleleft \mathbb{T}$, then there is a unique $u \in \mathcal{N}(X)_{\mathbb{T}}$ such that $t \sim_{\mathbb{T}} u$.*

[4] We need such an h to exist for the same reason that we do not want $\mathbb{U} <: \mathbb{U}'$ in Example 6: if there were no such h, then that would mean there is an infinite reduction sequence beginning with t, that never involves receiving a message from p; this would lead to a failure of liveness.

5.1 Returning, sequencing, and subtyping

As noticed by Katsumata [24], the appropriate structure for interpreting graded computational effects is a *graded monad* [4,39,31,24]. Specifically, when we give our denotational semantics in Section 6 below, we specify an interpretation of each of our typing rules; a graded monad provides the structure needed to interpret the typing rules [<:], [Ret], and [Let].

To interpret `SafeMP`, we therefore make normal forms of computation trees into a graded monad $\mathcal{N}$. This is the appropriate graded monad for interpreting asynchronous message-passing viewed as a computational effect. To do this, we provide three classes of functions involving computation trees.

- [<:]: To interpret *subtyping*, we need a function $\mathcal{N}(X)_{\mathbb{T}<:\mathbb{U}} : \mathcal{N}(X)_{\mathbb{T}} \to \mathcal{N}(X)_{\mathbb{U}}$ for each set X and pair of closed session types such that $\mathbb{T} <: \mathbb{U}$. We define these functions by $\mathcal{N}(X)_{\mathbb{T}<:\mathbb{U}}(t) = u$, where u is the unique $u \in \mathcal{N}(X)_{\mathbb{U}}$ such that $t \sim_{\mathbb{U}} u$. (This exists by Lemma 18 and the fact that $\blacktriangleleft$ respects subtyping.)
- [Ret]: To interpret *returning a value*, we need a *unit* function $\mathrm{return}_X : X \to \mathcal{N}(X)_{\mathsf{end}}$ for each X. We define $\mathrm{return}_X(x) = \mathtt{return}(x)$.
- [Let]: To interpret *sequencing*, we need a *bind* function $(\overset{\mathbb{T},\mathbb{T}'}{\gg\!=}) : \mathcal{N}(X)_{\mathbb{T}} \times (X \to \mathcal{N}(Y)_{\mathbb{T}'}) \to \mathcal{N}(Y)_{\mathbb{T}\cdot\mathbb{T}'}$ for each $X, Y, \mathbb{T}, \mathbb{T}'$. We define $t \overset{\mathbb{T},\mathbb{T}'}{\gg\!=} f$ coinductively by inspecting t, using the following clauses.

$$(\mathtt{return}(x)) \gg\!= f = f\,x \qquad \begin{aligned}(\mathtt{send}_{\mathsf{p},m}(t)) \gg\!= f &= \mathtt{send}_{\mathsf{p},m}(t \gg\!= f)\\ (\mathtt{recv}_{\mathsf{p}}(t_m)_{m\in M}) \gg\!= f &= \mathtt{recv}_{\mathsf{p}}(t_m \gg\!= f)_{m\in M}\end{aligned}$$

6 Denotational semantics of `SafeMP`

We now demonstrate that computation trees, and specifically the graded monad $\mathcal{N}$ defined in the previous section, form the basis of a model of `SafeMP`. The aim is to interpret each `SafeMP` configuration as an equivalent computation tree, where the notion of equivalence is our typed bisimulation. This equivalence provides our *adequacy* result (Corollary 20 below).

We first explain how to interpret `SafeMP` values. We define the interpretation of a ground type b to be the set $[\![\mathsf{b}]\!] = \{v \mid v : \mathsf{b}\}$ of constants of that type. If Γ is a typing context, then a *variable environment* γ is a function that assigns, to every $(x : \mathsf{b}) \in \Gamma$, a constant $\gamma(x) \in [\![\mathsf{b}]\!]$; we write $[\![\Gamma]\!]$ for the set of such functions. We interpret each typed value $\Gamma \vdash^{\mathsf{v}} v : \mathsf{b}$ as a function $[\![v]\!] : [\![\Gamma]\!] \to [\![\mathsf{b}]\!]$, as in Fig. 4.

For computations, we interpret the judgement $\Theta \,⨾\, \Phi \,⨾\, \Gamma \vdash t : \mathsf{b} \,⨾\, \mathbb{T}$ as follows. A *session type environment* θ, for the set Θ of type variables, is a function from Θ to closed session types. Given such a θ, if $\mathbb{T}$ is a session type over Θ, then we write $\mathbb{T}[\theta]$ for the session type that results from substituting the free type variables according to θ. For a function context Φ over Θ, a *function environment* ϕ is a function that assigns, to every $(f : (\mathsf{b}_1, \ldots, \mathsf{b}_n) \xrightarrow{\mathbb{T}} \mathsf{b}') \in \Phi$,

$$\boxed{\llbracket v \rrbracket \colon \llbracket \Gamma \rrbracket \to \llbracket \mathsf{b} \rrbracket \text{ where } \Gamma \vdash^{\mathsf{v}} v : \mathsf{b}} \quad \llbracket x \rrbracket(\gamma) = \gamma(x) \quad \llbracket v \rrbracket(\gamma) = v \text{ if } v \text{ is a constant}$$

$$\boxed{\llbracket t \rrbracket_\theta \colon \llbracket \Phi \rrbracket_\theta \times \llbracket \Gamma \rrbracket \to \mathcal{N}(\llbracket \mathsf{b} \rrbracket)_{\mathbb{T}[\theta]} \text{ where } \Theta \fatsemi \Phi \fatsemi \Gamma \vdash t : \mathsf{b} \fatsemi \mathbb{T}}$$

$$[{<:}]\ \frac{t' = \llbracket t \rrbracket_\theta(\phi,\gamma) \quad \mathbb{T} <:_\Theta \mathbb{U}}{\llbracket t \rrbracket_\theta(\phi,\gamma) = \mathcal{N}(\llbracket \mathsf{b} \rrbracket)_{\mathbb{T}[\theta] <: \mathbb{U}[\theta]}(t')} \qquad [\textsc{Ret}]\ \frac{v' = \llbracket v \rrbracket(\gamma)}{\llbracket \textsf{return}\ v \rrbracket_\theta(\phi,\gamma) = \mathtt{return}(v')}$$

$$[\textsc{Let}]\ \frac{t' = \llbracket t \rrbracket_\theta(\phi,\gamma) \quad f = \lambda x'.\, \llbracket u \rrbracket_\theta(\phi,(\gamma, x \mapsto x'))}{\llbracket \textsf{let}\ x = t\ \textsf{in}\ u \rrbracket_\theta(\phi,\gamma) = (t' \gg\!= f)} \quad [+]\ \frac{v' = \llbracket v \rrbracket(\gamma) \quad w' = \llbracket w \rrbracket(\gamma)}{\llbracket v + w \rrbracket_\theta(\phi,\gamma) = \mathtt{return}(v' + w')}$$

$$[<]\ \frac{v' = \llbracket v \rrbracket(\gamma) \quad w' = \llbracket w \rrbracket(\gamma)}{\llbracket v < w \rrbracket_\theta(\phi,\gamma) = \mathtt{return}(\text{if } v' < w' \text{ then true else false})}$$

$$[\textsc{If}]\ \frac{v' = \llbracket v \rrbracket(\gamma) \quad t_1' = \llbracket t_1 \rrbracket_\theta(\phi,\gamma) \quad t_2' = \llbracket t_2 \rrbracket_\theta(\phi,\gamma)}{\llbracket \textsf{if}\ v\ \textsf{then}\ t_1\ \textsf{else}\ t_2 \rrbracket_\theta(\phi,\gamma) = (\text{if } v' \text{ then } t_1' \text{ else } t_2')}$$

$$[\textsc{Send}]\ \frac{v' = \llbracket v \rrbracket(\gamma) \quad t' = \llbracket t \rrbracket_\theta(\phi,\gamma)}{\llbracket \textsf{send}\ (\ell, v)\ \textsf{to}\ \mathsf{p}\ \textsf{then}\ t \rrbracket_\theta(\phi,\gamma) = \mathtt{send}_{\mathsf{p},(\ell,v')}(t')}$$

$$[\textsc{Recv}]\ \frac{f_i = \lambda x_i'.\, \llbracket t_i \rrbracket_\theta(\phi,(\gamma, x_i \mapsto x_i')) \text{ for all } i \in I}{\llbracket \textsf{recv}\ \mathsf{p}\ \{\ell_i\langle x_i\rangle.\, t_i\}_{i \in I} \rrbracket_\theta(\phi,\gamma) = \mathtt{recv}_{\mathsf{p}}(f_i(v))_{(\ell_i,v) \in \{(\ell_i,v) \mid i \in I \wedge v : \mathsf{b}_i\}}}$$

$$[\textsc{LetRec}]\ \frac{\begin{array}{l} f' = \lambda(x_1',\ldots,x_n').\, \llbracket t \rrbracket_{\theta, \mathsf{X} \mapsto (\mu \mathsf{X}.\, \mathbb{T})}((\phi, f \mapsto f'),(\gamma, x_1 \mapsto x_1', \ldots, x_n \mapsto x_n')) \\ g' = \lambda f''.\, \llbracket u \rrbracket_\theta((\phi, f \mapsto f''),\gamma) \end{array}}{\llbracket \textsf{let rec}\ f(x_1,\ldots,x_n) = t\ \textsf{in}\ u \rrbracket_\theta(\phi,\gamma) = g'(f')}$$

$$[\textsc{App}]\ \frac{v_1' = \llbracket v_1 \rrbracket(\gamma) \ \cdots\ v_n' = \llbracket v_n \rrbracket(\gamma)}{\llbracket f(v_1,\ldots,v_n) \rrbracket_\theta(\phi,\gamma) = \phi(f)(v_1',\ldots,v_n')}$$

$$\boxed{\llbracket (\rho,t,\sigma) \rrbracket \in \mathcal{N}(\mathsf{b})_{\mathbb{T}} \text{ where } \vdash (\rho,t,\sigma) : \mathsf{b} \fatsemi \mathbb{T}}$$

$$[\textsc{CBase}]\ \frac{t' = \llbracket t \rrbracket_{.}(\cdot,\cdot)}{\llbracket (\emptyset,t,\emptyset) \rrbracket = t'} \qquad [\textsc{CSend}]\ \frac{\begin{array}{c} \mathbb{U} \xhookrightarrow{\mathsf{p}!\ell\langle \mathsf{b}' \rangle} \mathbb{T} \\ \mathcal{N}(\mathsf{b})_{\mathbb{U}} \ni u \overset{\mathsf{p}!(\ell,v)}{\rightsquigarrow} \llbracket (\rho,t,\sigma) \rrbracket \end{array}}{\llbracket (\rho,t,\sigma :: (\mathsf{p} \lhd (\ell,v))) \rrbracket = u} \qquad [\textsc{CRecv}]\ \frac{\begin{array}{c} \mathbb{T} \xhookrightarrow{\mathsf{p}?\ell\langle \mathsf{b}' \rangle} \mathbb{U} \\ \llbracket (\rho,t,\sigma) \rrbracket \overset{\mathsf{p}?(\ell,v)}{\rightsquigarrow} u \in \mathcal{N}(\mathsf{b})_{\mathbb{U}} \end{array}}{\llbracket ((\mathsf{p} \lhd (\ell,v)) :: \rho,t,\sigma) \rrbracket = u}$$

Fig. 4. Interpretation of `SafeMP` values, computations and configurations

a function $\phi(f) \colon \llbracket \mathsf{b}_1 \rrbracket \times \cdots \times \llbracket \mathsf{b}_n \rrbracket \to \mathcal{N}(\llbracket \mathsf{b}' \rrbracket)_{\mathbb{T}[\theta]}$; we write $\llbracket \Phi \rrbracket_\theta$ for the set of such function environments. For computations, we interpret each derivation of $\Theta \fatsemi \Phi \fatsemi \Gamma \vdash t : \mathsf{b} \fatsemi \mathbb{T}$ as a function $\llbracket t \rrbracket_\theta \colon \llbracket \Phi \rrbracket_\theta \times \llbracket \Gamma \rrbracket \to \mathcal{N}(\llbracket \mathsf{b} \rrbracket)_{\mathbb{T}[\theta]}$. This is defined in Fig. 4. The definition is by induction on the *derivation*, not on the computatation t, and hence a priori the computation t will have several interpretations – one for each derivation. It turns out this is not the case; we show below that any two derivations of $\Theta \fatsemi \Phi \fatsemi \Gamma \vdash t : \mathsf{b} \fatsemi \mathbb{T}$ necessarily have the same interpretation. The interpretations of subtyping, **return** and sequencing use the graded monad structure. The interpretations of **send** and **recv** are the obvious computation trees.

Finally, for configurations, we interpret every derivation of $\vdash (\rho, t, \sigma) : \mathtt{b} \mathbin{⨾} \mathbb{T}$ as a computation tree $[\![(\rho, t, \sigma)]\!] \in \mathcal{N}([\![\mathtt{b}]\!])_{\mathbb{T}}$. Rule [CBase] uses the interpretation of computations. Rule [CSend] interprets a configuration with a message to send as a computation tree u that reduces by sending such a message; since we consider normal forms of computation trees, there is a unique such u. Rule [CRecv] interprets a configuration that has received a message by reducing $[\![(\rho, t, \sigma)]\!]$; again, since we are dealing with normal forms, there is a unique such reduct.

Our main result about our denotational semantics is the following theorem, which establishes that computation trees correctly interpret configurations. It is this result that we appeal to when using our semantics to reason about `SafeMP`.

Theorem 19 (Correctness of the denotational semantics). *We have $(\rho, t, \sigma) \sim_{\mathbb{T}} [\![(\rho, t, \sigma)]\!]$ for every derivation of $\vdash (\rho, t, \sigma) : \mathtt{b} \mathbin{⨾} \mathbb{T}$.*

A corollary is that the denotational semantics is sound and complete with respect to typed bisimilarity; this is *adequacy* of the denotational semantics.

Corollary 20 (Adequacy). *Let $\vdash (\rho, t, \sigma) : \mathtt{b} \mathbin{⨾} \mathbb{T}$ and $\vdash (\rho', t', \sigma') : \mathtt{b} \mathbin{⨾} \mathbb{T}$ be two configurations of the same type. We have $(\rho, t, \sigma) \sim_{\mathbb{T}} (\rho', t', \sigma')$ if and only if $[\![(\rho, t, \sigma)]\!] = [\![(\rho', t', \sigma')]\!]$.*

Proof. By Theorem 19 and transitivity of $\sim_{\mathbb{T}}$, we have $(\rho, t, \sigma) \sim_{\mathbb{T}} (\rho', t', \sigma')$ iff $[\![(\rho, t, \sigma)]\!] \sim_{\mathbb{T}} [\![(\rho', t', \sigma')]\!]$. The latter bisimulation is an equality by Lemma 18.

Adequacy also establishes that the interpretation of a configuration does not depend on the choice of type derivation.

Example 21. We return to our global state example (Example 2), and specifically to the bisimilarity $(\emptyset, t_{\mathsf{c},1}, \emptyset) \sim_{\mathbb{S}} (\emptyset, t_{\mathsf{c},2}, \emptyset)$ claimed in Example 13. Due to adequacy, this bisimilarity is *equivalent* to the two configurations having the same interpretation in our denotational semantics. Indeed they do have the same interpretation; their interpretation is the following element of $\mathcal{N}([\![\mathbf{int}]\!])_{\mathbb{S}}$.

$$\mathtt{send}_{\mathsf{s},(\mathtt{get},\star)}(\mathtt{recv}_{\mathsf{s}}(\mathtt{send}_{\mathsf{s},(\mathtt{put},0)}(\mathtt{return}(n)))_{(\mathtt{st},n)})$$

7 Deadlock-freedom and liveness

A *session* is a collection of participants running in parallel. Deadlock-freedom and liveness are properties of sessions. In this section, we define a notion of `SafeMP` session, and then prove our desired safety and liveness properties. We do this as an application of our denotational semantics; we use the computation-tree interpretation of `SafeMP` to prove these properties.

Definition 22. *A* session $\mathcal{M}$ *is a finite list* $(\mathsf{r}_1 \triangleleft \mathcal{C}_1, \mathsf{r}_2 \triangleleft \mathcal{C}_2, \ldots, \mathsf{r}_n \triangleleft \mathcal{C}_n)$*, where* $\mathsf{r}_1, \ldots, \mathsf{r}_n$ *are distinct participant names, and each* $\mathcal{C}_i$ *is a configuration involving (at most) the participants in* $\{\mathsf{r}_1, \ldots, \mathsf{r}_n\} \setminus \{\mathsf{r}_i\}$.

We use our asynchronous operational semantics of `SafeMP` to define a notion of reduction $\mathcal{M} \overset{\beta}{\rightsquigarrow} \mathcal{M}'$ for sessions. This notion of reduction in particular includes a rule for one participant sending a message to another. A *global action* β is either τ_{p} (denoting an internal action made by participant p), or a triple $(\mathsf{p} \to \mathsf{q} : m)$ with $\mathsf{p} \neq \mathsf{q}$ (denoting that p sends message m to q). Reduction $\mathcal{M} \overset{\beta}{\rightsquigarrow} \mathcal{M}'$ of sessions is defined, using reduction of configurations, by two rules:

$$\frac{\mathcal{C}_i = \mathcal{D}_i \text{ for all } i \neq j \qquad \mathcal{C}_j \overset{\tau}{\rightsquigarrow} \mathcal{D}_j}{(\mathsf{r}_1 \lhd \mathcal{C}_1, \mathsf{r}_2 \lhd \mathcal{C}_2, \ldots, \mathsf{r}_n \lhd \mathcal{C}_n) \overset{\tau_{\mathsf{r}_j}}{\rightsquigarrow} (\mathsf{r}_1 \lhd \mathcal{D}_1, \mathsf{r}_2 \lhd \mathcal{D}_2, \ldots, \mathsf{r}_n \lhd \mathcal{D}_n)}$$

$$\frac{\mathcal{C}_i = \mathcal{D}_i \text{ for all } i \notin \{j,k\} \qquad \mathcal{C}_j \overset{\mathsf{r_k}!m}{\rightsquigarrow} \mathcal{D}_j \qquad \mathcal{C}_k \overset{\mathsf{r_j}?m}{\rightsquigarrow} \mathcal{D}_k}{(\mathsf{r}_1 \lhd \mathcal{C}_1, \mathsf{r}_2 \lhd \mathcal{C}_2, \ldots, \mathsf{r}_n \lhd \mathcal{C}_n) \overset{\mathsf{r_j} \to \mathsf{r_k} : m}{\rightsquigarrow} (\mathsf{r}_1 \lhd \mathcal{D}_1, \mathsf{r}_2 \lhd \mathcal{D}_2, \ldots, \mathsf{r}_n \lhd \mathcal{D}_n)}$$

Our type system for `SafeMP` assigns session types $\mathbb{T}_i$ to the individual configurations $\mathcal{C}_i$ of a session. By itself, this does not ensure that the $\mathbb{T}_i$ are in any way compatible with each other. We ensure the latter by following the *top-down* approach for MPST [21,44,17], in which there is a *global protocol*, and each participant p is required to follow the local *projection* of that protocol onto p. Global protocols are described by *global types*, which are generated inductively by the following grammar.

$$\mathbb{G} \ ::= \ \mathsf{end} \mid \mathsf{p} \to \mathsf{q} \colon \{\ell_i\langle \mathsf{b}_i \rangle.\, \mathbb{G}_i\}_{i \in I} \mid \mathsf{X} \mid \mu\mathsf{X}.\, \mathbb{G}$$

The global type end denotes that no further communication between participants will happen. $\mathsf{p} \to \mathsf{q} \colon \{\ell_i\langle \mathsf{b}_i \rangle.\, \mathbb{G}_i\}_{i \in I}$ denotes that p sends a single message to q; that message will have the form (ℓ_i, v) with $v : \mathsf{b}_i$ for some $i \in I$, and then the protocol will continue as $\mathbb{G}_i$. As for internal and external choices, we require I to be non-empty and finite, and we require the labels ℓ_i to be distinct from each other. We also require $\mathsf{p} \neq \mathsf{q}$. A *recursive protocol* $\mu\mathsf{X}.\,\mathbb{G}$ binds the type variable X; we require that every occurence of X is under some communication $\mathsf{p} \to \mathsf{q}$. Just as for local types, we define a notion of single-step unfolding for global types:

$$\mathcal{U}(\mu\mathsf{X}.\,\mathbb{G}) = \mathcal{U}(\mathbb{G})[\mathsf{X} \mapsto \mu\mathsf{X}.\,\mathbb{G}] \qquad \mathcal{U}(\mathbb{G}) = \mathbb{G} \text{ if } \mathbb{G} \text{ is not a recursive type}$$

A global type $\mathbb{G}$ is *closed* when it has no free type variables.

In the top-down approach, we determine each participant's (local) session type by *projecting* it from the global type $\mathbb{G}$. Projection is a partial function that maps a global type $\mathbb{G}$ and participant r to a session type $G \restriction \mathsf{r}$. The definition is by recursion on $\mathbb{G}$, and is given in Fig. 5. In the case of a communication not involving r, we *merge* the projections of the branches. Merging is a partial binary operation $\mathbb{T} \sqcap \mathbb{T}'$ on session types. We use *full merging*, as defined in [38]. The case split in the projection from a recursive global type ensures guardedness.

$$\mathsf{end} \upharpoonright \mathsf{r} = \mathsf{end} \qquad (\mathsf{p} \to \mathsf{q} \colon \{\ell_i\langle \mathsf{b}_i\rangle .\, \mathbb{G}_i\}_{i\in I}) \upharpoonright \mathsf{r} = \begin{cases} \mathsf{q} \oplus_{i\in I} \ell_i\langle \mathsf{b}_i\rangle .\, (\mathbb{G}_i \upharpoonright \mathsf{r}) & \text{if } \mathsf{p} = \mathsf{r} \\ \mathsf{p} \,\&_{i\in I}\, \ell_i\langle \mathsf{b}_i\rangle .\, (\mathbb{G}_i \upharpoonright \mathsf{r}) & \text{if } \mathsf{q} = \mathsf{r} \\ \prod_{i\in I}(\mathbb{G}_i \upharpoonright \mathsf{r}) & \text{if } \mathsf{r} \notin \{\mathsf{p},\mathsf{q}\} \end{cases}$$

$$\mathsf{X} \upharpoonright \mathsf{r} = \mathsf{X} \qquad (\mu\mathsf{X}.\, \mathbb{G}) \upharpoonright \mathsf{r} = \begin{cases} \mathsf{end} & \text{if } \mathbb{G} \upharpoonright \mathsf{r} = \mathsf{X} \\ \mathsf{X}' & \text{if } \mathbb{G} \upharpoonright \mathsf{r} = \mathsf{X}' \neq \mathsf{X} \\ \mu\mathsf{X}.\, (\mathbb{G} \upharpoonright \mathsf{r}) & \text{otherwise} \end{cases}$$

$$\mathsf{end} \sqcap \mathsf{end} = \mathsf{end} \quad (\mathsf{p} \oplus_{i\in I} \ell_i\langle \mathsf{b}_i\rangle .\, \mathbb{T}_i) \sqcap (\mathsf{p} \oplus_{i\in I} \ell_i\langle \mathsf{b}_i\rangle .\, \mathbb{T}'_i) = (\mathsf{p} \oplus_{i\in I} \ell_i\langle \mathsf{b}_i\rangle .\, \mathbb{T}_i \sqcap \mathbb{T}'_i)$$

$$\begin{array}{l}(\mathsf{p} \,\&_{i\in I}\, \ell_i\langle \mathsf{b}_i\rangle .\, \mathbb{T}_i) \\ \quad \sqcap\, (\mathsf{p} \,\&_{i\in J}\, \ell_i\langle \mathsf{b}_i\rangle .\, \mathbb{T}'_i)\end{array} = \begin{array}{l}(\mathsf{p} \,\&_{i\in I\cap J}\, \ell_i\langle \mathsf{b}_i\rangle .\, \mathbb{T}_i \sqcap \mathbb{T}'_i) \\ \&\ (\mathsf{p} \,\&_{i\in I\setminus J}\, \ell_i\langle \mathsf{b}_i\rangle .\, \mathbb{T}_i)\ \&\ (\mathsf{p} \,\&_{i\in J\setminus I}\, \ell_i\langle \mathsf{b}_i\rangle .\, \mathbb{T}'_i)\end{array}$$

$$\mathsf{X} \sqcap \mathsf{X} = \mathsf{X} \quad (\mu\mathsf{X}.\, \mathbb{T}) \sqcap (\mu\mathsf{X}.\, \mathbb{T}') = \mu\mathsf{X}.\, (\mathbb{T} \sqcap \mathbb{T}')$$

Fig. 5. Definitions of *projection* and of *full merging* of multiparty session types

Example 23. In the context of Example 1, the computation t implements a participant r as part of a global protocol described by the following global type $\mathbb{G}$; we can associate to t the session type $\mathbb{G} \upharpoonright \mathsf{r}$.

$$\mathbb{G} = \mathsf{p} \to \mathsf{r} \colon \begin{cases} \mathtt{success}\langle \mathbf{int}\rangle .\mathsf{r} \to \mathsf{q} \colon \begin{cases} \mathtt{cont}\langle \mathbf{int}\rangle .\, \mathsf{end} \\ \mathtt{stop}\langle \mathbf{bool}\rangle .\, \mathsf{end} \end{cases} \\ \mathtt{error}\langle \mathbf{bool}\rangle .\, \mathsf{r} \to \mathsf{q} \colon \mathtt{stop}\langle \mathbf{bool}\rangle .\, \mathsf{end} \end{cases} \quad \mathbb{G} \upharpoonright \mathsf{p} = \mathsf{r} \oplus \begin{cases} \mathtt{success}\langle \mathbf{int}\rangle .\mathsf{end} \\ \mathtt{error}\langle \mathbf{bool}\rangle .\, \mathsf{end} \end{cases}$$

$$\mathbb{G} \upharpoonright \mathsf{q} = \mathsf{r} \,\&\, \begin{cases} \mathtt{cont}\langle \mathbf{int}\rangle .\, \mathsf{end} \\ \mathtt{stop}\langle \mathbf{bool}\rangle .\, \mathsf{end} \end{cases} \quad \mathbb{G} \upharpoonright \mathsf{r} = \mathsf{p} \,\&\, \begin{cases} \mathtt{success}\langle \mathbf{int}\rangle .\mathsf{q} \oplus \begin{cases} \mathtt{cont}\langle \mathbf{int}\rangle .\, \mathsf{end} \\ \mathtt{stop}\langle \mathbf{bool}\rangle .\, \mathsf{end} \end{cases} \\ \mathtt{error}\langle \mathbf{bool}\rangle .\mathsf{q} \oplus \mathtt{stop}\langle \mathbf{bool}\rangle .\, \mathsf{end} \end{cases}$$

Definition 24. *A session* $(\mathsf{r}_1 \lhd \mathcal{C}_1, \mathsf{r}_2 \lhd \mathcal{C}_2, \ldots, \mathsf{r}_n \lhd \mathcal{C}_n)$ has type $\mathbb{G}$ *if (1)* $\mathbb{G}$ *is closed and contains only the participants in* $\{\mathsf{r}_1, \ldots, \mathsf{r}_n\}$*, (2) the projections* $\mathbb{G} \upharpoonright \mathsf{r}_i$ *are all defined, and (3)* $\vdash \mathcal{C}_i : \mathsf{b}_i \fatsemi \mathbb{G} \upharpoonright \mathsf{r}_i$ *for each* i*. When this is the case, we say that the session is* well-typed*.*

Example 25. We define a global type $\mathbb{G}$ for our global state example. The projections are the session types from Example 9.

$$\mathbb{G} = \mu\mathsf{X}.\, \mathsf{c} \to \mathsf{s} \colon \begin{cases} \mathtt{get}\langle \mathbf{unit}\rangle .\, \mathsf{s} \to \mathsf{c} \colon \mathtt{st}\langle \mathbf{int}\rangle .\, \mathsf{X} \\ \mathtt{put}\langle \mathbf{int}\rangle .\, \mathsf{X} \\ \mathtt{done}\langle \mathbf{unit}\rangle .\, \mathsf{end} \end{cases} \qquad \begin{array}{l} \mathbb{G} \upharpoonright \mathsf{s} = \mu\mathsf{X}.\, \mathbb{T}_{\mathsf{s}} \\ \mathbb{G} \upharpoonright \mathsf{c} = \mu\mathsf{X}.\, \mathbb{T}_{\mathsf{c}} \end{array}$$

The configurations of Example 9 form a session $\mathcal{M} = (\mathsf{s} \lhd \mathcal{C}_{\mathsf{s}}, \mathsf{c} \lhd \mathcal{C}_{\mathsf{c},2})$ of type $\mathbb{G}$.

Subject reduction for configurations provides a similar theorem for sessions.

Theorem 26 (Subject reduction for sessions). *If* $\mathcal{M}$ *is well-typed and* $\mathcal{M} \overset{\beta}{\rightsquigarrow} \mathcal{M}'$*, then* $\mathcal{M}'$ *is also well-typed.*

We now come to our desired safety and liveness properties, which hold for all well-typed sessions. The proofs of these use our denotational semantics. *Deadlock-freedom* means that reduction cannot get stuck; either the session terminates (with every configuration returning a result), or some communication will happen. As stated this theorem only applies to the initial session $\mathcal{M}_0$, but it follows from Theorem 26 that deadlock-freedom also applies to any reduct of $\mathcal{M}_0$.

Theorem 27 (Deadlock-freedom). *If $\mathcal{M}_0$ is well-typed, then there is a reduction sequence $\mathcal{M}_0 \overset{\beta_1}{\rightsquigarrow} \cdots \overset{\beta_c}{\rightsquigarrow} \mathcal{M}_c$, with $c \geq 0$, such that either (1) $c \geq 1$ and β_c has the form $\mathsf{p} \to \mathsf{q} : m$, or (2) $\mathcal{M}_c$ has the form*

$$(\mathsf{r}_1 \lhd (\emptyset, \mathcal{R}_1[\textbf{\textit{return}}\, v_1], \emptyset), \ldots, \mathsf{r}_n \lhd (\emptyset, \mathcal{R}_n[\textbf{\textit{return}}\, v_n], \emptyset))$$

Proof. Let $\mathbb{G}$ be the type of $\mathcal{M}_0 = (\mathsf{r}_1 \lhd \mathcal{C}_1, \ldots, \mathsf{r}_n \lhd \mathcal{C}_n)$. If $\mathcal{U}(\mathbb{G}) = \mathsf{end}$, then for every i we have $\mathcal{U}(\mathbb{G} \upharpoonright \mathsf{r}_i) = \mathsf{end}$, so the computation tree $[\![\mathcal{C}_i]\!]$ has the form $\mathtt{return}(x_i)$, and by Theorem 19 there is a reduction $\mathcal{C}_i \overset{\tau}{\rightsquigarrow}^* (\emptyset, \mathcal{R}_i[\textbf{return}\, v_i], \emptyset)$. Otherwise, $\mathcal{U}(\mathbb{G})$ has the form $\mathsf{r_j} \to \mathsf{r_k} \colon \{\ell_i\langle \mathsf{b}_i\rangle.\, \mathbb{G}_i\}_{i \in I}$. In this case, $\mathcal{U}(\mathbb{G} \upharpoonright \mathsf{r}_j)$ has the form $\mathsf{r_k} \oplus_{i\in I} \ell_i\langle \mathsf{b}_i\rangle.\, \mathbb{T}_i$ and $\mathcal{U}(\mathbb{G} \upharpoonright \mathsf{r}_k)$ has the form $\mathsf{r_j} \&_{i\in I} \ell_i\langle \mathsf{b}_i\rangle.\, \mathbb{U}_i$. It follows from Theorem 19 that there are reductions $\mathcal{C}_j \overset{\mathsf{r_k}!m}{\rightsquigarrow}^* \mathcal{D}_j$ and $\mathcal{C}_k \overset{\mathsf{r_j}?m}{\rightsquigarrow}^* \mathcal{D}_k$, so that we can take $\beta_c = \mathsf{r_j} \to \mathsf{r_k} : m$.

Finally, liveness means that (1) if a configuration is waiting to receive a message, then it will eventually do so; and (2) if a configuration sends a message, then that message will eventually be consumed. This assumes *fair* scheduling (the scheduler does not starve any participant). Our formulation of liveness is analogous to that of [18], and is stronger than the liveness considered in [38] (cf. [18, Footnote 4]).

Theorem 28 (Liveness). *Let $\mathcal{M}_0 \overset{\beta_1}{\rightsquigarrow} \mathcal{M}_1 \overset{\beta_2}{\rightsquigarrow} \cdots$ be a (possibly infinite) reduction sequence, with $\mathcal{M}_i = (\mathsf{r}_1 \lhd \mathcal{C}_{i1}, \ldots, \mathsf{r}_n \lhd \mathcal{C}_{in})$, and $\mathcal{C}_{ij} = (\rho_{ij}, t_{ij}, \sigma_{ij})$. Assume that the reduction sequence is* fair*, meaning for every i such that there exists a reduction $\mathcal{M}_i \overset{\beta'}{\rightsquigarrow} \mathcal{M}'$, there is some $i' > i$ such that $\beta_{i'} = \beta'$. If $\mathcal{M}_0$ is well-typed, then we have the following.*

1. *If t_{ij} has the form $\mathcal{R}[\textbf{\textit{recv}}\, \mathsf{p}\, \{\ell_k\langle x_k\rangle.\, u_k\}_{k\in K}]$, and ρ_{ij} has no messages from p, then there is some $i' > i$ and $m \in M$ such that $\beta_{i'} = (\mathsf{p} \to \mathsf{r_j} : m)$.*
2. *If $\beta_i = (\mathsf{p} \to \mathsf{r_j} : m)$, then there is some $i' > i$ such that $t_{i'j}$ has the form $\mathcal{R}[\textbf{\textit{recv}}\, \mathsf{p}\, \{\ell_k\langle x_k\rangle.\, u_k\}_{k\in K}]$.*

The idea behind the proof of liveness is that, if the global type $\mathbb{G}$ contains a communication $\mathsf{p} \to \mathsf{q}$ between two participants of the session, then at some point during the execution a $\mathsf{p} \to \mathsf{q}$ transition becomes available. We prove the latter by appealing to Theorem 19, using the fact that such a transition becomes available in the model. Fairness implies that the transition will be taken at some point. For both cases of liveness, the required communication appears at some finite depth in $\mathbb{G}$, and thus happens somewhere along the reduction sequence.

8 Related work

Semantics of session types Originating from [19,20], strong foundations of session types have been developed based on linear logic, exploiting a Curry-Howard correspondence between linear logic and typed session π-calculi [40,8,42]. The most effective and advanced semantics of these use *logical relations* to tackle various programming language features including parametricity [7] and higher-order functions [40]. Among them, [41], which is built on [1], enables tracking of cyclic dependencies of channels via classical logic session types, and applies this to information flow analysis. The work in [43] develops logical relations based on instuitionistic linear logic enriched with temporal predicates. The logical relations works are limited to *binary* session types; we do multiparty. Jacobs et al. [22] introduce MPGV, a functional MPST language with multiparty session types, based on the linear logic perspective. They use separation logic to define configuration invariants to maintain the acyclic nature of the communication topology and to establish subject reduction. They support *session delegation*, which we leave to future work, but they do not develop a (denotational) semantics for their calculus. Castellani et al. [10,11] interpret asynchronous and synchronous multiparty sessions as *flow event structures*. They do not have asynchronous subtyping; our work is the first denotational semantics for MPST with asynchronous subtyping. They also do not consider standard programming constructs such as sequencing, unlike us. Moreover, their work focuses on interpreting *sessions* and global protocols. As such, they do not interpret (local) computations in the manner that we do, so their semantics cannot be used to reason about individual participants in isolation. In a separate line of work, Castellan and Yoshida [9] use a connection between linear logics and game semantics to describe a fully abstract game semantics for binary session typed processes. They leave the extension to asynchrony and multiparty open. The aforementioned works do not study MPST from the perspective of computational effects.

Effects and session types Orchard and Yoshida [33] show that one can encode a binary session-typed π-calculus in a graded variant of PCF, session types being encoded as grades, and vice-versa. This work is orthogonal to ours. Their graded PCF does not have message-passing, and they do not consider message-passing as a computational effect. Thus they do not show how to track message-passing in an effectful calculus, nor do they provide a denotational semantics for session types. Instead their motivation is about encodability results, and they do not study safety, deadlock-freedom and liveness properties. Their work also considers only *binary* session types (not multiparty), and does not consider asynchrony. There are few other works that approach message-passing from the perspective of computational effects. Sanada [37] uses message-passing to exemplify *category-graded effect handlers*, but does not give a denotational semantics. Marshall and Orchard [30] take the linear logic perspective on synchronous binary session types, and use grades to track linearity. They observe that communication primitives are effects, but do not use grades to track them directly, and leave deadlock-freedom open (cf. [30, Section 11]).

Asynchronous subtyping Recent work on asynchronous subtyping includes the undecidablity result of [5], works focused on taming this undecidablity [6,3], and mechanization [16]. Our formulation of asynchronous subtyping, by characterising when the protocol requires or permits a message to be sent or received, is entirely different to the formulations appearing in these works, and yet is equivalent to the sound and complete subtyping of [18]. Compared to [18], our formulation avoids any use of session *trees*. Session trees are infinite structures, and thus cause some difficulties when it comes to implementing subtyping [16]; our reformulation shows that we do not need to involve session trees. Moreover, session-tree unfolding is defined only for closed session types, and hence [18] define subtyping only for closed session types. By avoiding session trees, we are able to provide the first extension of this subtyping to non-closed session types.

Typed bisimulations Kouzapas et al. study a bisimulation method which characterises a typed contextual equality in a higher-order binary session π-calculus [27], and applied it to measure the expressiveness of higher-order session processes [26]. Kouzapas and Yoshida [28] propose a typed bisimulation, controlled by declared global types, for a synchronous multiparty session π-calculus. We use typed bisimulations to justify the correctness of our denotational semantics.

Models of computational effects Kavvos [25], improving on some earlier work by Plotkin and Power [34], gives a general adequacy result for computational effects. Our adequacy result (Corollary 20) may follow from a graded variant of Kavvos's result, but there is none in the literature yet. There are various monadic models of shared-state concurrency [2,14,36,15], as opposed to message-passing; these bear little resemblance to our model.

9 Conclusions

This work is the first to provide a formal mathematical model for reasoning about asynchronous message-passing computation. We show that every multiparty session type $\mathbb{T}$ can be interpreted a set of *computation trees*, and thus that computation trees provide a basis for reasoning about asynchrony, even without message queues. Computation trees provide an adequate denotational semantics for a simple call-by-value programing language with message-passing as a computational effect, namely `SafeMP`. Since it is based on well-studied tools for studying computational effects, we expect that we can add more programming features to `SafeMP` without much difficulty. We also hope our computational-effects perspective on session types will enable the application of more of the vast computational effects literature to session types. Asynchronous session subtyping is a particular focus of our work. It is known to be difficult to reason about, but we have found our reformulation to be helpful with such reasoning.

Acknowledgments. We thank Jessica Richards, and the anonymous reviewers, for helpful comments. This work was supported by EPSRC EP/T006544/2, EP/T014709/2, EP/Z533749/1, ARIA and Horizon EU TaRDIS 101093006 (UKRI number 10066667).

References

1. Stephanie Balzer, Farzaneh Derakhshan, Robert Harper, and Yue Yao. Logical relations for session-typed concurrency, 2023.
2. Nick Benton, Martin Hofmann, and Vivek Nigam. Effect-dependent transformations for concurrent programs. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, pages 188–201, 2016.
3. Laura Bocchi, Andy King, Maurizio Murgia, and Simon Thompson. Abstract subtyping for asynchronous multiparty sessions. In Patricia Bouyer and Jaco van de Pol, editors, *36th International Conference on Concurrency Theory, CONCUR 2025, August 26-29, 2025, Aarhus, Denmark*, volume 348 of *LIPIcs*, pages 10:1–10:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.
4. Francis Borceux, George Janelidze, and G Max Kelly. Internal object actions. *Comment. Math. Univ. Carolin.*, 46(2):235–255, 2005.
5. Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Information and Computation*, 256:300–320, 2017.
6. Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theoretical Computer Science*, 722:19–51, 2018.
7. Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *ESOP*, volume 7792 of *LNCS*, pages 330–349. Springer, 2013.
8. Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of CONCUR 2010*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.
9. Simon Castellan and Nobuko Yoshida. Two sides of the same coin: session types and game semantics: a synchronous side and an asynchronous side. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.
10. Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Event structure semantics for multiparty sessions. *J. Log. Algebraic Methods Program.*, 131:100844, 2023.
11. Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Global types and event structure semantics for asynchronous multiparty sessions. *Fundam. Informaticae*, 192(1):1–75, 2024.
12. David Castro-Perez and Nobuko Yoshida. CAMP: cost-aware multiparty session protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):155:1–155:30, 2020.
13. Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in Rust with multiparty session types. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 246–261. ACM, 2022.
14. Yotam Dvir, Ohad Kammar, and Ori Lahav. A denotational approach to release/acquire concurrency. In *European Symposium on Programming*, pages 121–149. Springer, 2024.
15. Yotam Dvir, Ohad Kammar, Ori Lahav, and Gordon Plotkin. Two-sorted algebraic decompositions of brookes's shared-state denotational semantics. In *Foundations of Software Science and Computation Structures: 28th International Conference, FoSSaCS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3–8, 2025, Proceedings*, volume 15691, page 377. Springer, 2025.

16. Burak Ekici and Nobuko Yoshida. Completeness of Asynchronous Session Tree Subtyping in Coq. In *15th International Conference on Interactive Theorem Proving, 2024, Tbilisi, Georgia*, volume 309 of *LIPIcs*, pages 6:1–6:20. Schloss Dagstuhl - Leibniz-Zentrum f"ur Informatik, 2024.
17. Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 104:127–173, 2019.
18. Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *ACM Trans. Comput. Logic*, 24(2), nov 2023.
19. Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer.
20. Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.
21. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, 2016.
22. Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.*, 6(ICFP):466–495, 2022.
23. Ohad Kammar and Gordon D Plotkin. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 349–360, 2012.
24. Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proc. of 41st Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 633–645. ACM Press, New York, 2014.
25. GA Kavvos. Adequacy for algebraic effects revisited. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):927–955, 2025.
26. Dimitrios Kouzapas, Jorge A Pérez, and Nobuko Yoshida. On the relative expressiveness of higher-order session processes. *Information and Computation*, 268:104433, 2019.
27. Dimitrios Kouzapas, Jorge A. Pérez, and Nobuko Yoshida. Characteristic Bisimulation for Higher-Order Session Processes. *Acta Informatica*, 54:271–341, 2017.
28. Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. *Logical Methods in Computer Science*, 10, 2014.
29. Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.
30. Danielle Marshall and Dominic Orchard. Non-linear communication via graded modal session types. *Information and Computation*, 301:105234, 2024.
31. Paul-André Melliès. Parametric monads and enriched adjunctions. Manuscript, 2012.
32. Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

33. Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *POPL 2016*. ACM, 2016.
34. Gordon Plotkin and John Power. Adequacy for algebraic effects. In *International Conference on Foundations of Software Science and Computation Structures*, pages 1–24. Springer, 2001.
35. Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.
36. Exequiel Rivas and Tarmo Uustalu. Concurrent monads for shared state. In *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming*, pages 1–13, 2024.
37. Takahiro Sanada. Category-graded algebraic theories and effect handlers. *Electronic Notes in Theoretical Informatics and Computer Science*, 1, 2023.
38. Alceste Scalas and Nobuko Yoshida. Less is more: Multiparty session types revisited. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, January 2019.
39. A.L. Smirnov. Graded monads and rings of polynomials. *J. Math. Sci.*, 151(3):3032–3051, 2008.
40. Bernardo Toninho, Luis Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 350–369, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
41. Bas van den Heuvel, Farzaneh Derakhshan, and Stephanie Balzer. Information Flow Control in Cyclic Process Networks. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 40:1–40:30, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
42. Philip Wadler. Propositions as sessions. In *Proceedings of ICFP 2012*, pages 273–286. ACM, 2012.
43. Yue Yao, Grant Iraci, Cheng-En Chuang, Stephanie Balzer, and Lukasz Ziarek. Semantic logical relations for timed message-passing protocols. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.
44. Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to multiparty session types. In Dang Van Hung and Meenakshi D´Souza, editors, *Distributed Computing and Internet Technology*, pages 73–93, Cham, 2020. Springer International Publishing.