Abstract Subtyping for Asynchronous Multiparty Sessions

Laura Bocchi
University of Kent, Canterbury, UK

Andy King
University of Kent, Canterbury, UK

Maurizio Murgia
Gran Sasso Science Institute, L'Aquila, Italy

Simon Thompson
University of Kent, Canterbury, UK

Abstract

Session subtyping answers the question of whether a program in a communicating system can be safely substituted for another, when their communication behaviour is described by session types. Asynchronous session subtyping is undecidable, even for two participants, hence the interest in sound, but incomplete, subtyping algorithms. Asynchronous multiparty subtyping can be formulated by decomposing session types into single input and output types which preclude, respectively, external and internal choice. This paper shows how abstract interpretation can sit atop this approach and how it leads to an algorithm that can prove subtyping for intricate communication patterns.

2012 ACM Subject Classification Theory of computation → Program reasoning; Theory of computation → Process calculi

Keywords and phrases asynchrony, session subtyping, automata, abstract interpretation

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.10

Funding This work was supported by EPSRC project EP/T014512/1 (STARDUST) and the PRIN 2022 PNRR project DeLiCE (F53D23009130001).

1 Introduction

A significant challenge in message-passing concurrency is ensuring that each process in the system adheres to a desired protocol and is free from communication errors. Session types have proved to be powerful in addressing this challenge: they provide a theory for designing protocols with desirable behavioural guarantees, and are directly applicable to programming languages (notably Java [31], Python [22], Rust [33], Go [34] and Erlang [24, 37]) via static type-checking, runtime monitoring, or API generation. Session types were initially proposed for binary sessions involving only two roles (aka participants) [29], and were later extended to multiparty sessions, to allow for many roles [30]. Multiparty session types produce, via automated projection from a global protocol, a safe (by construction) realisation given as a *collection of local types*, one for each role. In this work we use this as a starting point: a collection of local types whose communications are *safe* in the sense that no role will ever: (1) get stuck (until it terminates); and (2) receive an unexpected message.

To illustrate protocol realisation, consider the three local types below: a server S, a client C, and a logger L, where s, c, and l denote the respective roles. The server can initially receive two kinds of messages from the client: $c?\mathrm{next}$ for more feeds or $c?\mathrm{stop}$ for termination. Feed requests are served with a feed ($c!\mathrm{feed}$) and waiting for further requests. The logging process gets regular updates by the server and sends a final summary to the client:

$$S = (c?\text{next}.\, l!\text{next}.\, c!\text{feed}.\, S \;+\; c?\text{stop}.\, l!\text{stop}) \qquad L = (s?\text{next}.\, L + s?\text{stop}.\, c!\text{sum})$$
$$C = (s!\text{next}.\, s?\text{feed}.\, C + s!\text{stop}.\, l?\text{sum})$$

We omit formalities on the semantics for the purposes of this introduction, and instead use color coding to hint at the correspondence between pairs of send/receive actions. This correspondence ensures that, in a system (S, C, L) behaving as the parallel composition of the three local types, no role ever gets stuck or receives an unexpected message.[1]

A fundamental problem in the application of session types is checking whether the implementation of one component in a distributed system can be substituted for another, without compromising safety of the overarching protocol. This problem can be addressed at the level of types and formulated as *session subtyping* [13, 21, 25, 26, 27]. Session subtyping establishes a preorder relation on local types: C' is a subtype of C, written $C' \leq C$, if a program with type C can be *safely* substituted by a program with type C'. Specifically, given a safe protocol (S, C, L), $C' \leq C$ ensures that (S, C', L) is also safe. For example, consider the variant $C' = s!\text{next}.\, (s?\text{feed}.\, s!\text{stop}.\, l?\text{sum} \;+\; s?\text{else})$ of C given earlier. With respect to C, C' has: (1) *less* send options (initially it can only select next, and after receiving one feed it can only select stop), (2) *more* receive options (the additional option $s?\text{else}$). This notion of substitutability is *covariant on send actions and contravariant on receive actions* [25]. In fact, one can verify $C' \leq C$, which ensures that if (S, C, L) is safe then so is (S, C', L).

Asynchronous session types are particularly interesting due to the use of FIFO channels in distributed systems and languages such as Go and Rust. When considering asynchronous session types, channels are unidirectional and unbounded (hence two are used for each pair of roles, one in each direction). For example, the asynchronous protocol (S, C', L) behaves as the parallel composition of 3 roles communicating over 6 unidirectional channels, where a send action enqueues a message to a channel and a receive action dequeues it.

Establishing whether subtyping holds is non-trivial, especially when communication is asynchronous. Recent work [27] has provided, for the first time, a *sound and complete* definition of Multiparty Asynchronous session Subtyping (MAS). This definition of session subtyping is sound and complete in the sense that it precisely capture only and all the safety-preserving modification one can perform on a local type. Besides co- and contra-variance, MAS enables some actions to be *swapped*, reflecting the asynchronous nature of interaction. MAS allows two kinds of swapping:

SW1 a role can anticipate an output action with respect to an input action;

SW2 a role can swap actions of the same directions from/to different roles.

Below, C'' is a variant of C' where the client *anticipates* the end-of-session request $s!\text{stop}$ before the processing of the feed $c?\text{next}$. This swap may be desirable to allow the client to process the feed from its own local queue at a later stage, and is allowed by SW1. S' is a variant of S where the send actions $c!\text{feed}$ and $l!\text{next}$ have been swapped, which is allowed by SW2. Indeed $C'' \leq C$ and $S' \leq S$.

$$C'' = (s!\text{next}.\, s!\text{stop}.\, s?\text{feed}.\, l?\text{sum}) \qquad S' = (c?\text{next}.\, c!\text{feed}.\, l!\text{next}.\, S' \;+\; c?\text{stop}.\, l!\text{stop})$$

This flexibility comes at a price: for interactions over unbounded channels MAS is *undecidable* [7, 35]. The subtyping problem can be viewed as a simulation game between the candidate subtype and the supertype, in which supertype is required to mirror any action

[1] Session types also accompany labels with payloads of type String, Int, higher-order, etc. We omit these details in our presentation as they are not relevant to subtyping.

performed by the candidate subtype. Consider checking whether C''' is a subtype of C where:

$$C''' = s!\text{next}.\, s!\text{next}.\, s?\text{feed}.\, C'''$$

The client requests feeds at double the pace of its own feed processing time. It is actually the case that $C''' \leq C$ according to asynchronous subtyping [7, 27]. However to verify $C''' \leq C$, one needs to show that C can follow the actions of C''', albeit with send actions being delayed. Due to unbounded channel size, an unbounded number of input actions (the feed processing) can be deferred, yielding an infinite space of possible interleavings.

Sound algorithms have emerged which aim to establish subtyping in practical scenarios. Most of these algorithms are limited to *binary* sessions (only two roles) and follow the definition of binary subtyping in [13]. The definition in [13] is *operational* in nature, based on simulation trees, which makes it relatively straightforward to translate it into a practical subtyping algorithm [3, 6]. Unlike the binary case [13], the multiparty definition of MAS in [27] is declarative. While MAS concisely expresses an intrinsically complex problem, it is also highly non-intuitive. Therefore, it is unrealistic to expect a developer to be able to check the correctness of their code substitutions using the definition directly (beside the fact that the problem is undecidable). MAS can only be practically used if a test for subtyping can be *automated*, and since the problem is undecidable, the solution will necessarily entail approximation. This paper has exactly this purpose: automation of MAS. Automation still raises the question "how can one trust the algorithm?" Our paper also answers this: our black-box (automatically) generates a *certificate* which can be, if needed, further checked off with a proof assistant [23].

In this introduction, to clarify the purpose of our work, we have provided an overview of the problem space, and hinted at the underlying asynchronous communication model and the notion of safety that subtyping preserves. However, these details are neither needed nor used in the technical development which follows, since our work builds entirely on the declarative definition of MAS [27] that elegantly abstracts over the communication model.

1.1 Challenges and contributions

The undecidability of MAS is unfortunate, as safety-preserving substitutions are most valuable in the asynchronous setting to check the validity of performance optimisations, namely sending messages earlier or postponing receives [38]. A MAS subtyping checker would encourage developers to experiment with communication structures, providing a litmus test for checking the correctness of an optimisation. While polynomial checking algorithms do exist [36], they do not support optimisation even for the simpler binary case. A potential solution lies in abstract interpretation, previously applied to *binary* subtyping, where sets of *words* model asynchronicity [3]. This addresses undecidability with systematic (lattice-theoretic) approximation. To apply abstract interpretation to multiparty interaction, we reformulate MAS as subtyping on sets of *session trees*, and make the following contributions:

- We introduce an *equivalent* but more tractable formulation of MAS [27], based on a new *lite refinement* preorder. MAS [27] is defined using nested quantification over infinite structures. One of the challenges of checking MAS is that it requires a check for *inclusion* of the actions being postponed over infinite structures. Lite refinement provides a way to automation by reducing the inclusion check to a simple, local check.
- Abstraction has been applied to binary subtyping by adapting subtyping to sets of words and then representing these sets with regular expressions [3]. MAS is formulated in terms of session trees, and likewise we generalise subtyping to sets of session trees. To derive continuations after performing an action, we adapt the Brzozowski derivative [10] from

regular expressions to session trees, and further extend it to sets of session trees. This provides a semantic foundation for subtyping, similar to a collecting semantics [17], ready for abstract interpretation and the deployment of an abstract domain.
- We provide a new abstract domain for representing sets of session trees based on type graphs [43] and provide new domain operations for send and receive. Since the abstract domain is not finite, we show how widening can be applied to derive a *terminating MAS algorithm* founded on *lite refinement*.

Overall, we show how the theoretical formulation of MAS [27] can be distilled into an algorithm for checking subtyping. Moreover, our algorithm yields a *certificate* (enabling subtyping to be double-checked by a third-party without consideration of low-level algorithmic details). We complement our theoretical contribution by evaluating our algorithm on some challenging problems and show it can prove subtyping more often than existing methods, whilst being an order of magnitude faster than the state-of-the-art binary algorithm [3].

2 Preliminaries

2.1 Syntax

Let $\mathbb{P}$ denote a finite set of roles and Σ denote a finite alphabet of communication labels. Let $\mathbb{A}^! = \{p!a \mid p \in \mathbb{P}, a \in \Sigma\}$, $\mathbb{A}^? = \{p?a \mid p \in \mathbb{P}, a \in \Sigma\}$ and $\mathbb{A} = \mathbb{A}^! \cup \mathbb{A}^?$. The syntactic category of session types, $\mathbb{S}$, is inductively defined by:

$$\mathbb{S} ::= \oplus_{i\in I}\, p!a_i.\mathbb{S}_i \mid \&_{i\in I}\, p?a_i.\mathbb{S}_i \mid \mathsf{end} \mid \mu \mathtt{t}.\mathbb{S} \mid t$$

where I is drawn from the category of (finite) index sets $\mathbb{I}$. Session type $\oplus_{i\in I}\, p!a_i.\mathbb{S}_i$ is for selection/send: the role selects one label in $\{a_i\}_{i\in I}$, sends it to role p, and continues as the corresponding $\mathbb{S}_i$. Session type $\&_{i\in I}\, p?a_i.\mathbb{S}_i$ is a branching/receive action, where the role receives a label from p. end is for termination, $\mu\mathtt{t}.\mathbb{S}$ for recursion, and $\mathtt{t}$ is a type variable used for recursive call.

Asynchronous multiparty subtyping [27] is based on a co-inductive representation of session types called *session trees*. The category $\mathbb{T}$ for session trees is defined co-inductively below. Subtyping is defined using three sub-classes of session trees called single-output (SO) trees $\mathbb{U}$, single-input (SI) trees $\mathbb{V}$, and single-input-single-output (SISO) trees $\mathbb{W}$, also defined co-inductively below:

$$\mathbb{T} ::= \oplus_{i\in I}\, p!a_i.\mathbb{T}_i \mid \&_{i\in I}\, p?a_i.\mathbb{T}_i \mid \mathsf{end} \qquad \mathbb{U} ::= p!a.\mathbb{U} \mid \&_{i\in I}\, p?a_i.\mathbb{U}_i \mid \mathsf{end}$$
$$\mathbb{V} ::= \oplus_{i\in I}\, p!a_i.\mathbb{V}_i \mid p?a.\mathbb{V} \mid \mathsf{end} \qquad \mathbb{W} ::= p!a.\mathbb{W} \mid p?a.\mathbb{W} \mid \mathsf{end}$$

We let S, T, U, V and W denote typical members $\mathbb{S}$, $\mathbb{T}$, $\mathbb{U}$, $\mathbb{V}$ and $\mathbb{W}$, respectively.

If a sequence W contains the action $p \diamond a$ where $\diamond \in \{!, ?\}$, we write $p \diamond a \in W$. Given the sets $\mathbb{Q}^! = \{p! \mid p \in \mathbb{P}\}$, $\mathbb{Q}^? = \{p? \mid p \in \mathbb{P}\}$ and $\mathbb{Q} = \mathbb{Q}^! \cup \mathbb{Q}^?$ we define the mapping $act : \mathbb{A} \to \mathbb{Q}$ where $act(p \diamond a) = p\diamond$. Given a set of actions $A \subseteq \mathbb{A}$ and a sequence of actions $W \in \mathbb{W}$, we define $act(A) = \{act(p \diamond a) \mid p \diamond a \in A\}$ and $act(W) = \{act(p \diamond a) \mid p \diamond a \in W\}$.

2.2 MAS: a declarative definition

MAS [27] is formulated in terms of two classes of sequence of actions, $\mathcal{A}^p$ and $\mathcal{B}^p$, which are parameterised by a participant p. Classes $\mathcal{A}^p$ and $\mathcal{B}^p$ embed the principles of *swapability* encoded into MAS. These *finite* sequences are inductively defined as follows:

▶ **Definition 1** (Finite sequences). *$\mathcal{A}^p ::= A^p \mid A^p.\mathcal{A}^p$ and $\mathcal{B}^p ::= B^p \mid B^p.\mathcal{B}^p$ where $A^p = \{q?a \in \mathbb{A}^? \mid p \neq q\}$ and $B^p = \{q!a \in \mathbb{A}^! \mid p \neq q\} \cup \mathbb{A}^?$.*

$\mathcal{A}^p$ defines the sequences of actions that can be anticipated before a *receive action* from p (namely any receive action that is not from p), whereas $\mathcal{B}^p$ defines the sequences of actions that can be anticipated before a *send action* to p (namely any receive action – outputs can always be done before inputs – and any send action that is not to p). Note in particular that $p?a \notin \mathcal{A}^p$ and $p!a \notin \mathcal{B}^p$. Using $\mathcal{A}^p$ and $\mathcal{B}^p$, we can now define refinement for SISO trees:

▶ **Definition 2** (SISO refinement). *The refinement relation $\precsim \subseteq \mathbb{W} \times \mathbb{W}$ for SISO-trees is co-inductively defined as below, where $A \in \mathcal{A}^p$ and $B \in \mathcal{B}^p$:*

$$\frac{}{\mathsf{end} \precsim \mathsf{end}}\ \mathsf{RefEnd} \qquad \frac{W \precsim W'}{p?a.W \precsim p?a.W'}\ \mathsf{RefIn} \qquad \frac{W \precsim W'}{p!a.W \precsim p!a.W'}\ \mathsf{RefOut}$$

$$\frac{W \precsim A.W'\ act(W) = act(A.W')}{p?a.W \precsim A.p?a.W'}\ \mathsf{RefA} \qquad \frac{W \precsim B.W'\ act(W) = act(B.W')}{p!a.W \precsim B.p!a.W'}\ \mathsf{RefB}$$

▶ **Example 3.** Since $\precsim$ is the largest relation which is closed backwards under these rules it follows $(p!a)^\omega \precsim (p!a)^\omega$ holds as well as $p!a.\mathsf{end} \precsim p!a.\mathsf{end}$. To see $(p!a)^\omega \precsim (p!a)^\omega$ observe $(p!a)^\omega = p!a.(p!a)^\omega$ therefore if $p!a.(p!a)^\omega \precsim p!a.(p!a)^\omega$ then $(p!a)^\omega \precsim (p!a)^\omega$ holds too. Thus RefOut is applicable. Rule RefB is not violated since $p!a \notin \mathcal{B}^p$ hence no rule is contradicted.

▶ **Example 4.** Let $(q?b)^\omega$ denote the infinite sequence of actions $q?b.q?b.\cdots$ and consider the following SISO trees: $W = p?a.(q?b)^\omega$ and $W' = (q?b)^\omega$. Observe that $W \precsim W'$ *cannot* be derived by RefIn since W' does not start with $p?a$ or RefA since W' does not contain $p?a$.

SISO-tree refinement provides an elegant vehicle for expressing swapability over the sequences of actions. However, a session type is not a sequence of actions, but rather a tree with selection and branching points. Definition 5 decomposes session trees in sets of SI- and SO- trees so as to express the co- and contra-variance of selection and branching.

▶ **Definition 5** (SO- and SI-tree decomposition for session trees). *The tree decomposition maps $[\![.]\!]_{\mathsf{SO}} : \mathbb{T} \to \wp(\mathbb{U})$ and $[\![.]\!]_{\mathsf{SI}} : \mathbb{T} \to \wp(\mathbb{V})$ are defined:*

$$\begin{array}{ll}
[\![\mathsf{end}]\!]_{\mathsf{SO}} = \{\mathsf{end}\} & [\![\mathsf{end}]\!]_{\mathsf{SI}} = \{\mathsf{end}\} \\
[\![\bigoplus_{i\in I} p!a_i.T_i]\!]_{\mathsf{SO}} = \{p!a_i.U_i \mid i \in I, U_i \in [\![T_i]\!]_{\mathsf{SO}}\} & [\![\bigoplus_{i\in I} p!a_i.T_i]\!]_{\mathsf{SI}} = \{\bigoplus_{i\in I} p!a_i.V_i \mid i \in I, V_i \in [\![T_i]\!]_{\mathsf{SI}}\} \\
[\![\&_{i\in I} p?a_i.T_i]\!]_{\mathsf{SO}} = \{\&_{i\in I} p?a_i.U_i \mid i \in I, U_i \in [\![T_i]\!]_{\mathsf{SO}}\} & [\![\&_{i\in I} p?a_i.T_i]\!]_{\mathsf{SI}} = \{p?a_i.V_i \mid i \in I, V_i \in [\![T_i]\!]_{\mathsf{SI}}\}
\end{array}$$

The intuition is that SO-tree decomposition removes the selection points in a session tree and expresses them using *sets* of single-output sequences, and dually for SI-trees. The maps $[\![\cdot]\!]_{\mathsf{SO}}$ and $[\![\cdot]\!]_{\mathsf{SI}}$ lift to a set of session trees $\mathcal{T} \in \wp(\mathbb{T})$ by, respectively, $[\![\mathcal{T}]\!]_{\mathsf{SO}} = \cup_{T\in\mathcal{T}} [\![T]\!]_{\mathsf{SO}}$ and $[\![\mathcal{T}]\!]_{\mathsf{SI}} = \cup_{T\in\mathcal{T}} [\![T]\!]_{\mathsf{SI}}$.

▶ **Example 6.** $[\![U]\!]_{\mathsf{SI}} = \{(p?a)^\omega\} \cup \{(p?a)^k.p?b.q?b.\mathsf{end} \mid k \geq 0\}$ where $U = p?a.U \,\&\, p?b.q?b.\mathsf{end}$.

Multiparty Asynchronous Subtyping (MAS) is formally defined as a binary relation on session trees in a forall-exists construction. Covariance and contravariance are encoded as a forall requirement on the SO-trees (resp. SI-trees) of the subtype (resp. supertype). The problem is then reduced to checking SISO-tree refinement:

▶ **Definition 7** (MAS). *The binary relation $\leq$ on $\mathbb{T} \times \mathbb{T}$ is defined by $T \leq T'$ holds iff $\forall U \in [\![T]\!]_{\mathsf{SO}} : \forall V' \in [\![T']\!]_{\mathsf{SI}} : \exists W \in [\![U]\!]_{\mathsf{SI}} : \exists W' \in [\![V']\!]_{\mathsf{SO}} : W \precsim W'$.*

▶ **Example 8.** To illustrate MAS, consider showing $T \leq T'$ where $T = p?c.q!d.\mathsf{end} \,\&\, p?e.q!d.\mathsf{end}$ and $T' = q!d.(p?c.\mathsf{end} \,\&\, p?e.\mathsf{end})$. Since T is an SO-tree, $[\![T]\!]_{\mathsf{SO}} = \{T\}$. T' is not an SI-tree hence $[\![T']\!]_{\mathsf{SI}} = \{q!d.p?c.\mathsf{end}, q!d.p?e.\mathsf{end}\}$. Universal quantification over $[\![T']\!]_{\mathsf{SI}}$ requires that T is checked against *both* $q!d.p?c.\mathsf{end}$ and $q!d.p?e.\mathsf{end}$. To do so, note $p?c.q!d.\mathsf{end} \in [\![T]\!]_{\mathsf{SI}}$ and $p?c.q!d.\mathsf{end} \precsim q!d.p?c.\mathsf{end}$ likewise $p?e.q!d.\mathsf{end} \in [\![T]\!]_{\mathsf{SI}}$ and $p?e.q!d.\mathsf{end} \precsim q!d.p?e.\mathsf{end}$ since

$$\dfrac{\dfrac{\dfrac{}{\mathsf{end} \precsim \mathsf{end}}\,\mathsf{RefEnd}}{q!d.\mathsf{end} \precsim q!d.\mathsf{end}}\,\mathsf{RefOut}}{p?c.q!d.\mathsf{end} \precsim q!d.p?c.\mathsf{end}}\,\mathsf{RefA} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{end} \precsim \mathsf{end}}\,\mathsf{RefEnd}}{q!d.\mathsf{end} \precsim q!d.\mathsf{end}}\,\mathsf{RefOut}}{p?e.q!d.\mathsf{end} \precsim q!d.p?e.\mathsf{end}}\,\mathsf{RefA}$$

3 Lite SISO-tree refinement

To develop a subtyping algorithm we introduce a novel form of SISO-refinement on $\mathbb{W} \times \mathbb{W}$, called lite SISO-tree refinement, that is **equivalent to classical SISO-refinement** but is more attractive computationally. Lite refinement builds on the observation that the equality checks $act(W) = act(A.W')$ and $act(W) = act(B.W')$ in the premises RefA and RefB of classical SISO-refinement can be relaxed to inclusion checks $act(W) \subseteq act(A.W')$ and $act(W) \subseteq act(B.W')$ without enlarging the refinement relation. Furthermore, the inclusions can be further relaxed to lightweight checks $act(hd(A)) \in act(W)$ and $act(hd(B)) \in act(W)$ involving only the single actions $hd(A)$ and $hd(B)$ where hd denotes head:

▶ **Definition 9.** *The lite refinement relation $\precsim_l \subseteq \mathbb{W} \times \mathbb{W}$ on SISO-trees is co-inductively defined as:*

$$\dfrac{W \precsim_l A.W' \; act(hd(A)) \in act(W)}{p?a.W \precsim_l A.p?a.W'}\,\mathsf{RefA} \qquad \dfrac{W \precsim_l B.W' \; act(hd(B)) \in act(W)}{p!a.W \precsim_l B.p!a.W'}\,\mathsf{RefB}$$

where $A \in \mathcal{A}^p$ and $B \in \mathcal{B}^p$ with RefEnd, RefIn *and* RefOut *as in Definition 2.*

▶ **Corollary 10.** *$W \precsim W'$ iff $W \precsim_l W'$.*

In the sequel, we develop a subtyping algorithm, based on $\precsim_l$, the correctness of which follows from the equivalence asserted by Corollary 10.

4 Subtyping Sets of Session Trees

Reasoning on *sets* of session trees paves the way for abstract interpretation [17] to be applied, where an algorithm is obtained by representing sets of session trees using an *abstract domain* constructed from type graphs [32]. Let $[\![\mathcal{T}']\!]_{\mathsf{SI}} = \cup\{[\![T']\!]_{\mathsf{SI}} \mid T' \in \mathcal{T}'\}$ to faciliate the following generalisaton of classical subtyping [27] to sets of session trees:

▶ **Definition 11.** *The binary relation $\leq \subseteq \mathbb{T} \times \wp(\mathbb{T})$ is defined $T \leq \mathcal{T}'$ holds iff $\mathcal{T}' \neq \emptyset$ and $\forall\, U \in [\![T]\!]_{\mathsf{SO}}$: $\forall\, V' \in [\![\mathcal{T}']\!]_{\mathsf{SI}}$: $\exists\, W \in [\![U]\!]_{\mathsf{SI}}$: $\exists W' \in [\![V']\!]_{\mathsf{SO}}$: $W \precsim W'$.*

The section builds towards a subtyping relation on sets of subtrees, given in Figure 1, that provides a basis for abstract interpretation. Subtyping is defined in terms two notions of derivative: one for receive actions (Definition 16) and one for send actions (Definition 17). These operations scan a session tree from its root for a specific action, and return the session trees obtained by removing that action and any sub-trees that do not include it. These operations are inspired by the Brzozowski derivative [10] of a regular expression

e wrt to a symbol u: if e represents a set of words W, the derivative $\partial_u(e)$ represents $\{w \in \Sigma^* \mid u \cdot w \in W\}$. Our derivatives, presented in Sections 4.1 and 4.2, generalise the Brzozowski derivative to operate on (sets of) session trees rather than (sets of) words. Section 4.3 formulates subtyping using the derivatives. The subtyping relation is defined co-inductively, each deduction reading the action at the head of the subtype to apply either the send or receive derivative to the supertype. Our approach is justified by lite refinement.

4.1 Receive derivative of a session tree

Derivatives differ depending on whether they look for receive or send actions. A receive derivative is calculated using a depth bound derived using the $\mathsf{always}_{p?}$ predicate:

▶ **Definition 12.** *The predicate* $\mathsf{always}_{p?} \subseteq \mathbb{T} \times \mathbb{N}_0$ *is inductively defined as follows:*

$$\frac{}{\mathsf{always}_{p?}(\&_{i\in I} p?a_i.T'_i, 0)} \qquad \frac{p \neq q \quad \forall i \in I : \mathsf{always}_{p?}(T'_i, k_i) \quad k = 1 + \max_{i\in I} k_i}{\mathsf{always}_{p?}(\&_{i\in I} q?a_i.T'_i, k)} \qquad \frac{\forall i \in I : \mathsf{always}_{p?}(T'_i, k_i) \quad k = 1 + \max_{i\in I} k_i}{\mathsf{always}_{p?}(\bigoplus_{i\in I} q!a_i.T'_i, k)}$$

Furthermore $\mathsf{always}_{p?}(T)$ *holds iff there exists* $k \in \mathbb{N}_0$ *such that* $\mathsf{always}_{p?}(T, k)$ *holds.*

▶ **Example 13.** Let $T'_1 = q?a.p?a.\mathsf{end} + q?b.p?b.T'_1$ and $T'_2 = q?a.p?a.\mathsf{end} + q?b.T'_2$. Since $\mathsf{always}_{p?}(p?a.\mathsf{end}, 0)$ and $\mathsf{always}_{p?}(p?b.T'_1, 0)$ then $\mathsf{always}_{p?}(T'_1, 1)$. However, $\mathsf{always}_{p?}(T'_2, k)$ does not hold $(\forall\, k \in \mathbb{N}_0)$ since $\mathsf{always}_{p?}$ is defined inductively.

Lemma 14 asserts that the absence of a bound k implies the absence of subtyping. Lemma 15 asserts that there exists at most one k for which $\mathsf{always}_{p?}(T', k)$ holds. The receive derivative, which follows, is then defined in terms of the bound.

▶ **Lemma 14.** *If* $T, T' \in \mathbb{T}$ *and* $p?a.T \leq T'$ *then* $\mathsf{always}_{p?}(T')$.

▶ **Lemma 15.** *If* $T' \in \mathbb{T}$, $\mathsf{always}_{p?}(T', k)$ *and* $\mathsf{always}_{p?}(T', k')$ *then* $k = k'$.

▶ **Definition 16** (Receive derivative)**.** *The receive derivative map* $\delta_{p?a} : \mathbb{T} \to \wp(\mathbb{T})$ *is defined as* $\delta_{p?a}(T) =$ *if* $\mathsf{always}_{p?}(T, k)$ *then* $\{D \mid \delta^k_{p?a}(T, D)\}$ *else* $\emptyset$*, where the derivative relation* $\delta^k_{p?a} : \mathbb{T} \times \mathbb{T}$ *for some* $k \geq 0$ *is inductively defined:*

$$\frac{j \in I \quad a = a_j}{\delta^0_{p?a}(\&_{i\in I} p?a_i.T'_i, T'_j)} \qquad \frac{p \neq q \quad \forall i \in I : \delta^{k_i}_{p?a}(T'_i, D_i) \quad k = 1 + \max\{k_i\}_{i\in I} \quad j \in I}{\delta^k_{p?a}(\&_{i\in I} q?a_i.T'_i, q?a_j.D_j)}$$

Following MAS, a receive action from p can only be found immediately (first rule) or after other receive actions from other roles (second rule). The second rule can be applied non-deterministically for any $j \in I$ and derives at least $|I|$ distinct derivatives (which $\delta_{p?a}(T)$ returns as a set). We write $q?a_j.D_j$ for the concatenation of $q?a$, the action preceding $p?a$, with a subtree D_j that was the continuation of $p?a$ in that path. The receive derivative is well-defined because there exists at most one k such that $\mathsf{always}_{p?}(T', k)$ holds.

4.2 Send derivative of a session tree

Unfortunately, Lemma 14 does not help us define a send derivative on supertypes as we no longer rely on a known bound k. Hence, the definition of $\Delta^k_{p!a}$ is parametric in k.

▶ **Definition 17** (Send derivative). *The k-th order send derivative map $\Delta^k_{p!a} : \mathbb{T} \to \wp(\mathbb{T})$ is defined as*

$$\Delta^k_{p!a}(T') = \begin{cases} \mathcal{D}^k & \text{if } \mathcal{D}^k \neq \emptyset \ \vee \ k = 0 \qquad \text{where } \mathcal{D}^k = \{D \mid \Delta^k_{p!a}(T', D)\} \\ \Delta^{k-1}_{p!a}(T') & \text{otherwise} \end{cases}$$

and the derivative relation $\Delta^k_{p!a} \subseteq \mathbb{T} \times \mathbb{T}$ is, itself, inductively defined by:

$$\frac{j \in I \quad a = a_j}{\Delta^0_{p!a}(\bigoplus_{i\in I} p!a_i.T'_i, T'_j)} \qquad \frac{\forall i \in I : \Delta^{k_i}_{p!a}(T'_i, D_i) \wedge k_i \text{ maximal} \quad k = 1 + \max\{k_i\}_{i\in I} \quad j \in I}{\Delta^k_{p!a}(\&_{i\in I} q?a_i.T'_i, q?a_j.D_j)}$$

$$\frac{p \neq q \quad J = \{i \in I \mid \Delta^{k_i}_{p!a}(T'_i, D_i) \ \wedge \ k_i \text{ maximal } \} \neq \emptyset \quad k = 1 + \max\{k_i\}_{i\in I}}{\Delta^k_{p!a}(\bigoplus_{i\in I} q!a_i.T'_i, \bigoplus_{j\in J} q!a_j.D_j)}$$

The k-th derivative is defined to be the set of derivatives $\mathcal{D}^\ell$ for which $\mathcal{D}^\ell \neq \emptyset$, the degree ℓ is maximal and $\ell \leq k$. If $\mathcal{D}^\ell = \emptyset$ for all $0 \leq \ell \leq k$ then the k-th derivative is itself null. Following MAS, in a supertype, a send action can be found immediately (first rule), after receive actions (e.g., the subtype may have anticipated the output), or after other send actions from different roles (third rule). Condition k_i maximal means: $\forall_{i\in I} : \Delta^{k^*_i}_{p!a}(T'_i, D^*_i) \Rightarrow k^*_i \leq k_i$. The third rule derives at least one derivative equipped with $|J|$ branches $q!a_j.D_j$, where the D_j are again selected to maximise their degree k_i.

▶ **Example 18** (Derivatives). Let $T = r?a.q?a.p!a.T \& r?b.r?b.q?a.\mathsf{end}$. The receive derivative $\delta_{q?a}(T)$ exists because both branches of T possess a receive action $q?a$. Indeed $\mathsf{always}_{q?}(T, 2)$ holds, hence $\delta_{q?a}(T) = \{r?a.p!a.T,\ r?b.r?b.\mathsf{end}\}$. Now consider $\Delta^k_{p!a}(T)$. For $k < 2$, the send derivatives $\Delta^0_{p!a}(T) = \Delta^1_{p!a}(T) = \emptyset$. Yet for $k \geq 2$, $\Delta^k_{p!a}(T) = \{r?a.q?a.T\}$. In this case, $p!a$ does not need to occur in all branches of T for the derivative to exist.

4.3 Subtyping with receive and send derivatives

To apply one derivative after another in a chain of actions, receive and send derivatives are lifted to operators on sets of supertypes as follows:

▶ **Definition 19.** *The operators $\delta_{p?a} : \wp(\mathbb{T}) \to \wp(\mathbb{T})$ and $\Delta_{p!a} : \wp(\mathbb{T}) \to \wp(\mathbb{T})$ are defined:*

$$\begin{array}{l} \delta_{p?a}(\mathcal{T}') = \cup\{\delta_{p?a}(T') \mid T' \in \mathcal{T}'\} \\ \Delta^k_{p!a}(\mathcal{T}') = \uplus\{\Delta^k_{p!a}(T') \mid T' \in \mathcal{T}'\} \end{array} \quad \text{where} \quad \uplus\,\mathcal{T}' = \begin{cases} \emptyset & \emptyset \in \mathcal{T}' \\ \cup\mathcal{T}' & \text{otherwise} \end{cases}$$

The use of $\uplus$ in $\Delta^k_{p!a}(\mathcal{T}')$ reflects covariance over sends: the $p!a$ action of the subtype must be supported by all session-trees $T' \in \mathcal{T}'$. Conversely, the use of $\cup$ in $\delta_{p?a}(\mathcal{T}')$ models contravariance over receives: the $p?a$ of the subtype does not need to be found in all $T' \in \mathcal{T}'$.

▶ **Definition 20.** *Given a partial map $\Gamma \in \mathbb{T} \rightharpoonup \wp(\mathbb{Q})$, the update and accumulate operations are respectively defined:*

- $\Gamma[T \mapsto Q] = \{T \mapsto Q\} \cup \{T' \mapsto Q' \in \Gamma \mid T \neq T'\}$
- $\Gamma + p\diamond = \{T \mapsto Q \cup \{p\diamond\} \mid T \mapsto Q \in \Gamma\}$ *where* $\diamond \in \{!, ?\}$

▶ **Definition 21** (Subtyping on sets of session trees). *The ternary subtyping relation $\leq \subseteq (\mathbb{T} \rightharpoonup \wp(\mathbb{Q})) \times \mathbb{T} \times \wp(\mathbb{T})$ is co-inductively defined by the rules of Figure 1.*

The ternary subtyping relation is defined using an environment Γ, a partial map $\mathbb{T} \rightharpoonup \wp(\mathbb{Q})$, that records the actions performed by T before T is encountered again as a sub-tree. This device assumes that session trees are regular [40, Section 21.7.2] (aka rational [16]). T is

$$\frac{}{\Gamma \vdash \mathsf{end} \leq \{\mathsf{end}\}}\ \mathsf{RefEnd}$$

$$\frac{\begin{array}{c}\forall T' \in \mathcal{T}' : \exists i \in I : \delta_{p?a_i}(T') \neq \emptyset \qquad J = \{i \in I \mid \delta_{p?a_i}(\mathcal{T}') \neq \emptyset\} \neq \emptyset \\ \forall i \in J : \exists \mathcal{D}_i \supseteq \delta_{p?a_i}(\mathcal{T}) : \Gamma' \vdash T_i \leq \mathcal{D}_i \\ \hdashline T \mapsto Q \in \Gamma \Rightarrow Q \supseteq act(hd(\mathcal{T}')) \quad \Gamma' = \Gamma[T \mapsto \emptyset] + p? \quad \text{where} \quad T = \&_{i\in I} p?a_i.T_i\end{array}}{\Gamma \vdash \&_{i\in I} p?a_i.T_i \leq \mathcal{T}'}\ \mathsf{RefInA}$$

$$\frac{\begin{array}{c}\exists k \in \mathbb{N} : \forall i \in I : \Delta^k_{p!a_i}(\mathcal{T}') \neq \emptyset \qquad \forall i \in I : \exists \mathcal{D}_i \supseteq \Delta^k_{p!a_i}(\mathcal{T}) : \Gamma' \vdash T_i \leq \mathcal{D}_i \\ \hdashline T \mapsto Q \in \Gamma \Rightarrow Q \supseteq act(hd(\mathcal{T}')) \quad \Gamma' = \Gamma[T \mapsto \emptyset] + p! \quad \text{where} \quad T = \oplus_{i\in I} p!a_i.T_i\end{array}}{\Gamma \vdash \oplus_{i\in I} p!a_i.T_i \leq \mathcal{T}'}\ \mathsf{RefOutB}$$

Figure 1 Rules for the ternary subtyping relation $\leq \subseteq (\mathbb{T} \rightharpoonup \wp(\mathbb{Q})) \times \mathbb{T} \times \wp(\mathbb{T})$.

regular iff its set of distinct subtrees, $\mathsf{sub}(T)$, is finite. This is not limiting since session types correspond to regular session trees. The premises of RefInA and RefOutB are partitioned by the dotted lines into: (bottom) requirements based on the syntax of the subtype T that decompose the problem onto sub-problems of the form $\Gamma_i \vdash T_i \leq \mathcal{D}_i$ and ensure a *progress condition*, and (top) subtyping requirements based on derivatives. In both rules, the premise $T \mapsto Q \in \Gamma \Rightarrow Q \supseteq act(hd(\mathcal{T}'))$ stipulates that the actions at the head of $\mathcal{T}'$ must be performed within one cycle of T, that is, before T is revisited while traversing the subtype. This progress condition [15] ensures that a head action of $\mathcal{T}'$ is not postponed indefinitely.[2]

Rule RefInA lifts RefIn and RefA of lite refinement (Definition 9) to sets of session trees. The first premise of RefInA (top) is a coverage condition which ensures that for every $T' \in \mathcal{T}'$ a receive derivative $\delta_{p?a_i}(T')$ is defined for least one $i \in I$. This guarantees that the selection of receive actions offered by T cover those occurring in $\mathcal{T}'$.

Rule RefOutB lifts RefOut and RefB (Definition 9) to session trees by applying the send derivative. Note RefInA permits a relaxation $\mathcal{D}_i$ to be used for $\delta_{p?a_i}(\mathcal{T}')$; likewise a superset $\mathcal{D}_i$ is used for $\Delta^{k_i}_{p!a_i}(\mathcal{T}')$ in RefOutB. Observe k is not fixed upfront: it can vary with $\mathcal{T}'$.

▶ **Theorem 22.** *If $T \in \mathbb{T}$ is regular, $\mathcal{T}' \in \wp(\mathbb{T})$ and $\emptyset \vdash T \leq \mathcal{T}'$ then $T \leq \mathcal{T}'$.*

▶ **Example 23** (Violated progress condition)**.** Consider $U = p?a.(q?a.U + q?b.r?a.\mathsf{end})$ and the subtyping problem $U \leq r?a.W'$ where $W' = p?a.q?a.W'$ and $W' \in \mathbb{W}$. Observe that if $W \precsim_l r?a.W'$ then $r? \in act(W)$ hence W is finite which presents a contradiction therefore $U \not\leq r?a.W'$. Likewise $\emptyset \vdash U \not\leq r?a.W'$ as is demonsrated below:

$$\dfrac{\dfrac{\{U \mapsto \{p?, q?\}, q?a.U \mapsto \{q?\}\} \vdash U \leq \{r?a.W'\}}{\{U \mapsto \{q?\}\} \vdash q?a.U + q?b.r?a.\mathsf{end} \leq \{r?a.q?a.W'\}}\ \mathsf{RefInA}}{\emptyset \vdash U \leq \{r?a.W'\}}\ \mathsf{RefInA}$$

In the first application of RefInA, $\delta_{p?a}(r?a.W') = \delta_{p?a}(r?a.p?a.q?a.W') = \{r?a.q?a.W'\}$ and $\delta_{p?a'}(r?a.W') = \emptyset$ for all $a' \neq a$. In the second, $\delta_{q?a}(r?a.q?a.W') = \{r?a.W'\}$ but $\delta_{q?b}(r?a.q?a.W') = \emptyset$ thus J is a singleton. The development cannot be progressed since $\{p?, q?\} \not\supseteq \{r?\} = act(hd(\{r?a.W'\}))$, in effect, detecting that $r?a$ is an orphan.

[2] Binary subtyping [3, 5, 6, 13] makes use of a check that tests for cycles of send actions in the candidate subtype. We reframed these checks here as a more intuitive and simple progress condition, which amounts to no more than a set inclusion test that discharges a requirement on swapability.

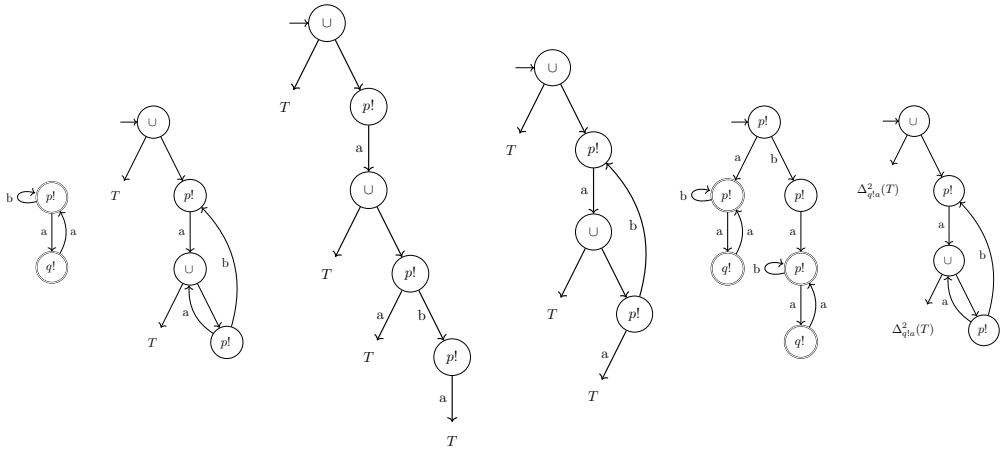

Figure 2 (i) T (ii) G (iii) W_0 (iv) W_1 (v) $\Delta^2_{q!a}(T)$ (vi) $\Pi^2_{q!a}(G)$.

▶ **Example 24** (Satisfied progress conditions)**.** To illustrate where a message is postponed, but is not orphan, consider the question $W \leq W'$ where $W = p!a.p!a.p?b.W$ and $W' = p!a.p?b.W'$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{W \mapsto \{p!\}, p!a.p?b.W \mapsto \{p!,p?\}, p?b.W \mapsto \{p?,p!\}\} \vdash p!a.p?b.W \leq \{p?b.p?b.W'\}}\,\mathsf{RefOutB}}{\{W \mapsto \{p!,p?\}, p!a.p?b.W \mapsto \{p!,p?\}, p?b.W \mapsto \{p?\}\} \vdash W \leq \{p?b.W'\}}\,\mathsf{RefInA}}{\{W \mapsto \{p!\}, p!a.p?b.W \mapsto \{p!\}\} \vdash p?b.W \leq \{p?b.p?b.W'\}}\,\mathsf{RefOutB}}{\{W \mapsto \{p!\}\} \vdash p!a.p?b.W \leq \{p?b.W'\}}\,\mathsf{RefOutB}}{\emptyset \vdash W \leq \{W'\}}$$

To show $\emptyset \vdash W \leq \{W'\}$, it is sufficient to put $k = 0$ since $\Delta^0_{p!a}(W') = \{p?b.W'\}$. To establish the second, it is necessary to use $k = 1$ because $\Delta^0_{p!a}(p?b.W') = \emptyset$ but $\Delta^1_{p!a}(p?b.W') = \{p?b.p?b.W'\}$. To show the fourth, $k = 2$ is needed, illustrating that it is not sufficient to fix k throughout. Notice too that W is first revisited when attempting to establish the the fourth judgement. Its environment includes $W \mapsto \{p!, p?\} \in \Gamma$ and $\{p?\} = act(hd(\{p?b.W'\}))$ hence the environment (orphan) premise is satisfied, and the development continues.

5 Subtyping sets of session trees using type graphs

Type graphs [32, Section 4.1.2] were proposed for representing (possibly infinite) sets of finite trees. We adapt them to represent (possibly infinite) sets of session trees by inductively defining the category $\mathbb{G}$ of type graphs to be: $\mathbb{G} ::= \emptyset \mid \mathbb{T} \mid \cup_{i\in I}\mathbb{G}_i \mid \&_{i\in I}p?a_i.\mathbb{G}_i \mid \oplus_{i\in I}p!a_i.\mathbb{G}_i$. Thus the leaves of a type graph G are session trees drawn from $\mathbb{T}$. A type graph is interpreted by a *concretisation* map γ [17] which describes the set of session trees it represents.

▶ **Definition 25** (Concretisation map)**.** *The concretisation map $\gamma : \mathbb{G} \to \wp(\mathbb{T})$ is defined as the least solution to the following equations:*

$$\begin{array}{rl} \gamma(\emptyset) = \emptyset & \gamma(\&_{i\in I}p?a_i.G_i) = \{\&_{i\in I}p?a_i.T_i \mid \forall i \in I : T_i \in \gamma(G_i)\} \\ \gamma(T) = \{T\} & \gamma(\oplus_{i\in I}p!a_i.G_i) = \{\oplus_{i\in I}p!a_i.T_i \mid \forall i \in I : T_i \in \gamma(G_i)\} \\ \gamma(\cup_{i\in I}G_i) = \cup_{i\in I}\gamma(G_i) & \end{array}$$

▶ **Example 26.** Consider T (co-inductively) defined by $T = p!a.q!a.T \oplus p!b.T$ and the type graph $G = T \cup p!a.G'$ where G' is (inductively) defined by $G' = T \cup (p!a.G' \oplus p!b.p!a.G')$. T and G are illustrated in Figure 2 (i) and (ii) as regular trees where the terminal symbols distinguish T from G. If $\mathcal{T}_0 = \{T\}$ and $\mathcal{T}_{i+1} = \{T', p!a.T', p!a.(p!a.T' \oplus p!b.p!a.T') \mid T' \in \mathcal{T}_i\}$ then $\gamma(G) = \cup_{i=0}^{\infty}\mathcal{T}_i$. Observe that G has a finite representation even though $\gamma(G)$ is infinite.

Concretisation induces a preorder $\preceq$ on $\mathbb{G}$ by $G \preceq G'$ iff $\gamma(G) \subseteq \gamma(G')$. The preorder $\langle \mathbb{G}, \preceq \rangle$ may possess a bottom element $\emptyset$ but admits infinite ascending chains, as shown below:

▶ **Example 27.** Define $G_0 = T$ and $G_{i+1} = G_i \cup p!a.(G_i \cup p!a.G_i \oplus p!b.p!a.G_i)$. Observe $\gamma(G_i) \subseteq \gamma(G_{i+1})$ but $\gamma(G_i) \neq \gamma(G_{i+1})$ for all $i \geq 0$. Nevertheless the chain G_i has an upper bound: G of Example 26 since $\gamma(G_i) \subseteq \gamma(G)$ for all $i \geq 0$.

As we shall see, calculating the upper bound of a sequence of type graphs is key to finitely computing a co-inductive proof. Such an upper bound can classically [17] be found with a widening operator $\mathbb{G} \triangledown \mathbb{G} \to \mathbb{G}$. Widening provides a way to finitely derive an upper bound: given any increasing sequence $G_0 \preceq G_1 \cdots$ the (widened) sequence defined by $G'_0 = G_0$ and $G'_{i+1} = G'_i \triangledown G_{i+1}$ is a sequence $G'_0 \preceq G'_1 \preceq \cdots$ which is ultimately stable with a limit G'_ℓ. Widening is required [17] to be monotonic in both arguments: $G \preceq G \triangledown G'$ and $G' \preceq G \triangledown G'$. Then the limit G'_ℓ is an upper bound of the whole G_i sequence, that is, $G_i \preceq G'_\ell$ for all $i \geq 0$.

▶ **Example 28.** Type graphs can be widened [43] by introducing back-arcs to curb growth and thereby induce stability. To provide some intuition, consider widening the type graph W_0 given in Figure 2(iii). Widening is achieved by repeatedly introducing cycles into the type graph by replacing a sub-tree with a back-arc to an ancestor. To illustrate, W_0 is transformed into W_1, see Figure 2(iv), by replacing the sub-tree $p!a.T$ with a back-arc to an ancestor. Observe $W_0 \preceq W_1$. Moreover, the deepest T of W_1 has a direct ancestor which is disjunctive and has T as a child. This triggers the introduction of another back-arc which finally yields G, see Figure 2(ii). Again $W_1 \preceq G$. The widening algorithm of [43] guarantees [43, Theorem 7.1] that a widened sequence is ultimately stable, while attempting to minimise the introduction of cycles. The widening of [43] will terminate with G, though a more aggressive widening would also replace the deepest T of G with a back-arc to the root.

5.1 An abstract domain for sets of session trees

Operations on type graphs can be designed to mimic those on sets of session trees. In particular, to faithfully mimic the derivatives, it is both necessary to detect the absence of a derivative and preserve the derivation in a relaxation. The following definition formulates these requires and the proposition shows how they can be satisfied by lifting $\Delta^k_{p!a}$ and $\delta_{p?a}$:

▶ **Definition 29.** *An abstract send derivative is any operator $\Pi^k_{p!a} : \mathbb{G} \to \mathbb{G}$ such that:*

- *if $\Delta^k_{p!a}(\gamma(G)) = \emptyset$ then $\Pi^k_{p!a}(G) = \emptyset$*
- *$\Delta^k_{p!a}(\gamma(G)) \subseteq \gamma(\Pi^k_{p!a}(G))$*

for all $G \in \mathbb{G}$. An abstract receive derivative $\pi_{p?a} : \mathbb{G} \to \mathbb{G}$ is specified analogously.

▶ **Proposition 30.** *The operator $\mathbb{G} \to \mathbb{G}$ is an abstract send derivative where:*

$$\frac{}{\Gamma \vdash \Pi^k_{p!a}(\emptyset) = \emptyset} \qquad \frac{G \in \mathbb{T} \quad \Delta^k_{p!a}(G) = \emptyset}{\Gamma \vdash \Pi^k_{p!a}(G) = \emptyset} \qquad \frac{G \in \mathbb{T} \quad \Delta^k_{p!a}(G) = \{T_i\}_{i\in I}}{\Gamma \vdash \Pi^k_{p!a}(G) = \cup_{i\in I} T_i} \qquad \frac{(\cup_{i\in I} G_i) \mapsto G \in \Gamma}{\Gamma \vdash \Pi^k_{p!a}(\cup_{i\in I} G_i) = G}$$

$$\frac{\cup_{i\in I} G_i \notin \mathsf{dom}(\Gamma) \quad \Gamma' = \Gamma \cup \{\cup_{i\in I} G_i \mapsto \uplus_{i\in I} G'_i\} \quad \forall i \in I : \Gamma' \vdash \Pi^k_{p!a}(G_i) = G'_i}{\Gamma \vdash \Pi^k_{p!a}(\cup_{i\in I} G_i) = \uplus_{i\in I} G'_i}$$

with additional rules for $\Pi^k_{p!a}(\&_{i\in I} p?a_i.G_i)$ and $\Pi^k_{p!a}(\oplus_{i\in I} p!a_i.G_i)$ that follow Definition 17.

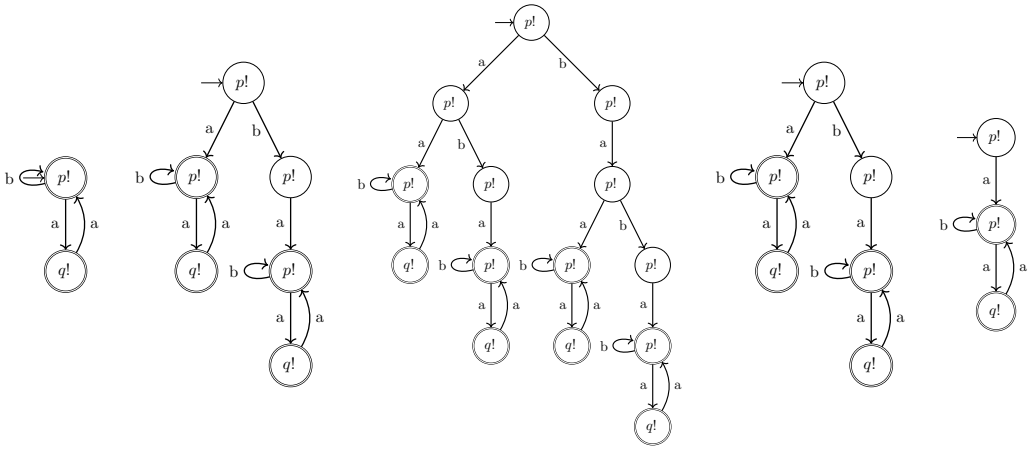

Figure 3 Intermediate type graphs: (i) $G'_{0,0}$ (ii) $G_{0,1}$ (iii) $G_{0,2}$ (iv) $G_{0,3}$ and (v) $G_{1,0}$.

An abstract receive derivative $\pi_{p?a} : \mathbb{G} \to \mathbb{G}$ can also be constructed using an environment Γ.

▶ **Example 31.** Consider calculating $\Pi^2_{q!a}(G)$ where G is as defined in Example 26. Figure 2 (i) and (ii) illustrate T and G. Following Proposition 30, $\Pi^2_{q!a}(G)$ is constructed by replacing each leaf T of G with the derivative $\Delta^2_{q!a}(T)$, given in (v), to produce the type graph of (vi).

An algorithm for checking subtyping follows from Figure 1 by substituting $\Delta^k_{p!a}$ with $\Pi^k_{p!a}$ and $\delta_{p?a}$ with $\pi_{p?a}$; thus operations on sets of session trees are simulated with operations on type graphs. We provide insight by first showing how widening can be applied to type graphs to create a co-inductive proof of subtyping, for a subtype that is linearly recursive.

▶ **Example 32.** To illustrate our strategy, consider $T \leq Q$ where $T = q!a.q!a.p!a.p!b.T$ and $Q = p!a.q!a.Q \oplus p!b.Q$. For brevity, define $T_0 = q!a.T_1$, $T_1 = q!a.T_2$, $T_2 = p!a.T_3$ and $T_3 = p!b.T_0$ thus $T = T_0$. Suppose $G'_{0,0} = Q$ and $G'_{i+1,0} = G'_{i,0} \cup G_{i+1,0}$ for all $i \geq 0$ where

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma'_0 \vdash T_0 \leq G_{1,0}}{\Gamma'_3 \vdash T_3 \leq G_{0,3}}}{\Gamma'_2 \vdash T_2 \leq G_{0,2}}}{\Gamma'_1 \vdash T_1 \leq G_{0,1}}}{\Gamma'_0 \vdash T_0 \leq G'_{0,0}} \quad \frac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma'_0 \vdash T_0 \leq G_{i+1,0}}{\Gamma'_3 \vdash T_3 \leq G_{i,3}}}{\Gamma'_2 \vdash T_2 \leq G_{i,2}}}{\Gamma'_1 \vdash T_1 \leq G_{i,1}}}{\Gamma'_0 \vdash T_0 \leq G'_{i,0}} \quad \frac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma'_0 \vdash T_0 \leq G_{\ell+1,0}}{\Gamma'_3 \vdash T_3 \leq G_{\ell,3}}}{\Gamma'_2 \vdash T_2 \leq G_{\ell,2}}}{\Gamma'_1 \vdash T_1 \leq G_{\ell,1}}}{\Gamma'_0 \vdash T_0 \leq G'_{\ell,0}}$$

Observe $G'_{0,0} \preceq G'_{1,0} \preceq \cdots$ constitutes an increasing sequence, albeit one that is possibility infinite. Repeating the construction with $G'_{i+1,0} = G'_{i,0} \triangledown G_{i+1,0}$, however, yields a sequence which is ultimately stable, that contains its limit $G'_{\ell,0}$. Because $G'_{\ell,0} \triangledown G_{\ell+1,0} = G'_{\ell+1,0} = G'_{\ell,0}$ it follows $G_{\ell+1,0} \preceq G'_{\ell,0}$. An infinite co-inductive derivation can then be assembled by repeating the derivation on the right (since RefInA and RefOutB support relaxation). Figure 3 illustrates $G'_{0,0}$, $G_{0,1}$, $G_{0,2}$, $G_{0,3}$ and $G_{1,0}$ whereas Figure 4 gives the widened sequence $G'_{0,0} \preceq G'_{1,0} \preceq G'_{2,0} \preceq G'_{3,0} \cdots$ which has $G'_{3,0}$ as its limit. Altogether this shows $T \leq Q$. Note that $G'_{3,0}$ constitutes a certificate. Finally observe that existence of $G'_{3,0}$, hence an infinite co-inductive derivation, follows by widening, which was originally conceived [17] as a device for enforcing termination.

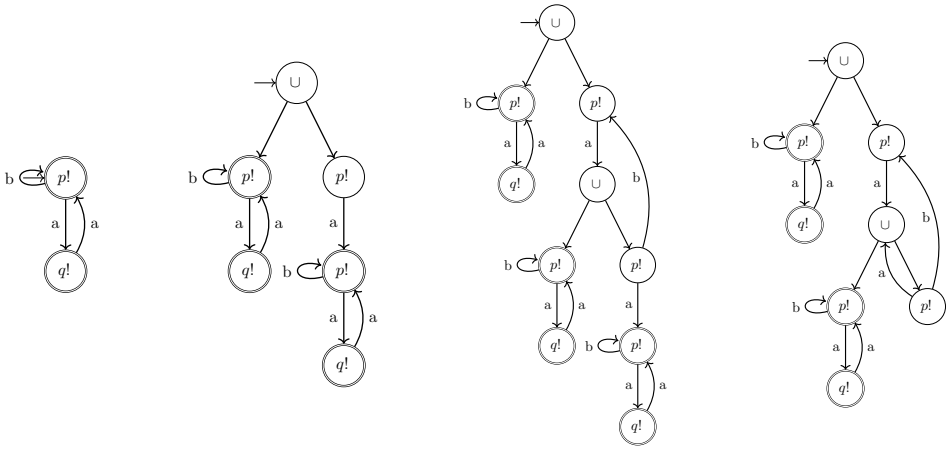

Figure 4 Widened type graphs: (i) $G'_{0,0}$ (ii) $G'_{1,0}$ (iii) $G'_{2,0}$ (iv) $G'_{3,0}$.

5.2 Subtyping algorithm for type graphs

Example 32 illustrates how widening can be applied to establish the existence of an infinite co-inductive proof for $T \leq Q$. Algorithm 1 outlines an algorithm, which applies the same strategy for proving $T \leq T'$, but is applicable when T is not linearly recursive. The key caveat for our algorithm is that T is regular. This ensures $\mathsf{sub}(T)$ is finite. It follows there exists a finite set $wp \subseteq \mathsf{sub}(T)$ such that every cycle of T crosses at least one sub-tree of wp. (wp is inspired by the idea of widening points which are program locations [4] where widening is applied.)

▶ **Example 33.** Continuing Example 32, observe $\mathsf{sub}(T) = \{T, T_1, T_2, T_3\}$ and the only cycle of T crosses each sub-tree of $\mathsf{sub}(T)$. Thus put $wp = \{T_2\}$, though other choices are possible.

In addition to wp, which is fixed throughout, our subtyping algorithm has a worklist L and a partial map $\mathcal{G} : \mathsf{sub}(T) \rightharpoonup \mathbb{G}$ as parameters. $\mathcal{G}$ mirrors the strategy used Example 32 which associates each sub-tree of T with its own type graph. The algorithm is primed with $L = [\langle T, \Gamma \rangle]$ where $\Gamma = \emptyset$ and $\mathcal{G} = \{T \mapsto T'\}$ (recall $\mathbb{T} \subseteq \mathbb{G}$). The algorithm exits successfully at line (2) returning the certificate $\mathcal{G}$, and otherwise throws an exception at (8), (9), (14) or (15) indicating an inconclusive verdict. The body of SUBTYPING removes the first pair $\langle T, \Gamma \rangle$ from the worklist (if not empty), and checks whether the action at the root of T is a receive, a send, or an end. In the latter case, if $\mathbb{G}(T) = \mathsf{end}$ then rule RefEnd of Figure 1 holds and processing continues, otherwise an exception is thrown.

If T is a receive, the progress condition of RefInA is checked at (8). The coverage condition at (9) can be decided without recourse to γ by an auxiliary predicate cover defined thus: $\mathsf{cover}(\emptyset) = \mathit{true}$, $\mathsf{cover}(\cup_{j \in J} G_j) = \wedge_{j \in J} \mathsf{cover}(G_j)$ and $\mathsf{cover}(G) = \exists i \in I : \delta_{p?a_i}(G) \neq \emptyset$ if $G \in \mathbb{T}$. Then $\mathsf{cover}(\mathcal{G}(T))$ holds iff $\forall T' \in \gamma(\mathcal{G}(T)) : \exists i \in I : \delta_{p?a_i}(T') \neq \emptyset$. Line (9) also checks $J \neq \emptyset$. If these checks are passed, $\mathcal{G}$ is relaxed to $\mathcal{G}'$ by updating $\mathcal{G}$ on the keys $\{T_j \mid j \in J\}$. If the sub-tree $T_j \in wp$ then widening is triggered and $\mathcal{G}'(T_j) = \mathcal{G}'(T_j) \triangledown \pi_{p?a_j}(\mathcal{G}(T))$. It follows $\mathcal{G}(T_j) \preceq \mathcal{G}'(T_j)$ and $\pi_{p?a_j}(\mathcal{G}(T)) \preceq \mathcal{G}'(T_j)$, both assuring soundness while ensuring that $\mathcal{G}'(T_j)$ cannot be relaxed ad infinitum. If $T_j \notin wp$ then $\mathcal{G}'(T_j) = \mathcal{G}'(T_j) \cup \pi_{p?a_j}(\mathcal{G}(T))$, again yielding a sound relaxation. Widening at wp is enough to ensure $\mathcal{G}$ is ultimately stable across all keys [4]: thus $\mathcal{G}' = \mathcal{G}$ after a finite number of updates.

To demonstrate RefInA holds it remains to show $\Gamma[T \mapsto \emptyset] + p? \vdash T_j \leq \gamma(\mathcal{G}'(T_j))$ for all $j \in J$. This is achieved by the list comprehension at (11) which extends the worklist

Algorithm 1 Subtyping algorithm where a type graph represents a set of sessions trees.

(1) **function** SUBTYPE(L, wp, $\mathcal{G}$)
(2) **if** $L = \emptyset$ **then** $\mathcal{G}$
(3) **else**
(4) $\langle T, \Gamma \rangle := hd(L)$
(5) **switch** T
(6) **case** $\&_{i \in I}\, p?a_i.T_i$:
(7) $J := \{j \in I \mid \pi_{p?a_j}(\mathcal{G}(T)) \neq \emptyset\}$
(8) **if** $T \mapsto Q \in \Gamma \wedge Q \not\trianglerighteq act(hd(\mathcal{G}(T)))$ **then throw maybe**
(9) **if** $J = \emptyset \vee \exists T' \in \gamma(\mathcal{G}(T)) : \forall i \in I : \delta_{p?a_i}(T') = \emptyset$ **then throw maybe**
(10) $\mathcal{G}' := \mathcal{G}[T_j \mapsto \mathcal{G}(T_j)$ (**if** $T_j \in wp$ **then** $\triangledown$ **else** $\cup$) $\pi_{p?a_j}(\mathcal{G}(T)) \mid j \in J]$
(11) $L' := [\langle T_j, \Gamma[T \mapsto \emptyset] + p? \rangle \mid j \in J, \Gamma(T_j) = \bot \vee \neg(\mathcal{G}'(T_j) \preceq \mathcal{G}(T_j))] :: tl(L)$
(12) SUBTYPE(L', wp, $\mathcal{G}'$)
(13) **case** $\oplus_{i \in I}\, p!a_i.T_i$:
(14) **if** $T \mapsto Q \in \Gamma \wedge Q \not\trianglerighteq act(hd(\mathcal{G}(T)))$ **then throw maybe**
(15) **if** $\exists i \in I : \Pi^k_{p!a_i}(\mathcal{G}(T)) = \emptyset$ **then throw maybe**
(16) $\mathcal{G}' := \mathcal{G}[T_i \mapsto \mathcal{G}(T_j)$ (**if** $T_j \in wp$ **then** $\triangledown$ **else** $\cup$) $\Pi^k_{p!a_i}(\mathcal{G}(T)) \mid i \in I]$
(17) $L' := [\langle T_i, \Gamma[T \mapsto \emptyset] + p! \rangle \mid i \in I, \Gamma(T_i) = \bot \vee \neg(\mathcal{G}'(T_i) \preceq \mathcal{G}(T_i))] :: tl(L)$
(18) SUBTYPE(L', wp, $\mathcal{G}'$)
(19) **case** end :
(20) **if** $\mathcal{G}(T) \neq$ end **then throw maybe**
(21) SUBTYPE($tl(L)$, wp, $\mathcal{G}$)

with the pair $\langle T_j, \Gamma[T \mapsto \emptyset] + p? \rangle$ for each $j \in J$ providing the pair actually needs to be considered, that is, if $\Gamma(T_j) = \bot$ or $\neg(\mathcal{G}'(T_j) \preceq \mathcal{G}(T_j))$. Subtyping then reduces to invoking SUBTYPE(L', wp, $\mathcal{G}'$). Observe $\Gamma(T_j) \neq \bot$ on any subsequent visit to T_j, and eventually $\mathcal{G}'(T_j) \preceq \mathcal{G}(T_j)$ by virtue of widening. Termination thus follows.

The case for send on lines (14)–(18) is arguably simpler since J does not need to be computed and coverage does not need to be checked.

6 Implementation

To assess practicality, we have implemented the Subtype algorithm in Scala 3.1.0 in 5800 LOC. Our implementation makes substantial use of immutable data-structure and parser combinator libraries [28] to reconstruct the type graph operations of [32] and the widening of [43]. As well as the derivatives, our codebase provides a visualiser for type graphs which outputs tikz.

We tested our algorithm on 12 binary and 12 multiparty problems, as detailed in Table 1. The problem trickySub < trickySup is taken from Example 24 whereas all other problems are drawn from (or inspired by) existing literature. The binary benchmarks include 8 which possess "complex accumulation patterns" [6] which is a catch-all term for communication patterns which defy heuristic methods [6]. The remaining 4 problems were chosen by size.

Our algorithm was able to automatically prove subtyping for *all* problems, strictly improving on [3, 6] for the binary problems and [18] for the multiparty problems. Our algorithm was also able to establishing subtyping for *all* the remaining 74 binary problems in the benchmark suite of [6]. The strict improvement over the binary algorithm of [3] occurs because the string widening of [14] deployed in [3], imposes a restriction on the syntactic form of consecutive regular sub-expressions, whereas the particular widening developed for

Table 1 Minimum k of $\Pi^k_{p!a}$ required for successful subtyping, iterations of the algorithm (up), time in ms (expressed as parse time+analysis time) with comparison against related algorithms.

benchmark (binary)	k	up	time	[6]	[3]	[18]	benchmark (multiparty)	k	up	time	[6]	[3]	[18]
ctxta1 < ctxta2 [6]	1	14	1+14	✗	✓	✗	trickySub < trickySup	2	16	1+15	✗	✗	✗
ctxtb1 < ctxtb2 [6]	1	14	1+4	✗	✓	✗	FFT1 < FFT2 [11]	1	15	2+1	✗	✗	✓
14may2 < 14may1 [6]	1	8	1+2	✗	✓	✗	altbit1 < altbit2 [18]	1	4	1+0	✗	✗	✓
badseq1 < badseq2 [6]	1	24	1+16	✗	✓	✗	dbuff1 < dbuff2 [18]	1	5	1+1	✗	✗	✓
march3testa1 < march3testa2 [6]	1	17	1+24	✗	✓	✗	ring1 < ring2 [18]	1	2	1+0	✗	✗	✓
aaaaaab1 < aaaaaab2 [6]	1	9	1+2	✗	✓	✗	rchoice1 < rchoice2 [18]	1	3	1+0	✗	✗	✓
ex1-ok-loop < ex2-ok-loop [6]	1	15	1+34	✗	✓	✗	ex181 < ex182 [27]	1	3	2+0	✗	✗	✓
march3testa1 < march3testb2 [6]	1	28	1+15	✗	✗	✗	ex192 < ex191 [27]	1	9	2+2	✗	✓	✗
twinstar < ex2 [6]	1	12	1+13	✓	✓	✗	ex172 < ex171 [27]	1	7	1+0	✗	✗	✓
decidex1 < decidex2 [6]	1	22	1+33	✓	✓	✗	LSWAq2 < LSWAq1 [36]	1	8	2+0	✗	✗	✓
sub-runningex < sup-runningex [6]	1	12	1+40	✓	✓	✗	LSWAr2 < LSWAr1 [36]	1	5	2+0	✗	✗	✓
september1 < september2 [6]	1	20	1+75	✓	✓	✗	EYTb2 < EYTb1 [23]	1	3	1+0	✗	✗	✓

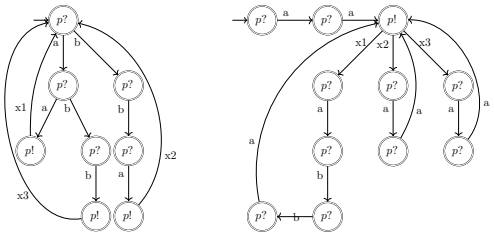

Figure 5 doubleb1 < doubleb2 (adapted from march3testa1 < march3testa2 of [6]).

type graphs [42] used in the present work has no such limitation. However, Figure 5 depicts a binary problem, doubleb1 < doubleb2, delibrately crafted in response to the reviewers, to subvert the type graph widening of [42]. This example shows that the algorithm is incomplete.

Surprisingly, calculating the send derivatives with $k = 1$ was sufficient for all but one problem, trickySub < trickySup, which was crafted to be difficult. The *up* column records how many times the map $\mathcal{G}$ is updated, which governs complexity. Timings are presented as parse+analysis time and were taken on a 32 GB laptop equipped with a 2.8GHz i7 processor. The zero times are runs beneath the granularity of the clock. The slowest runtime was 1+75=76ms, whereas the subtyping tool of [3], takes 1,757ms worse-case for the binary benchmarks (on the same machine), which suggests that performance-wise, multiparty subtyping is no worse than binary subtyping if carefully designed. All certificates are available at `https://www.cs.kent.ac.uk/people/staff/amk/certificates.zip`.

7 Related Work

The undecidability of asynchronous subtyping [7, 35] has given rise to a rich line of research aimed at finding sound algorithms. The first work [8] to explore the boundaries of undecidability, showed that synchronous subtyping can be generalised to k-bounded asynchronous subtyping [8], where input accumulation depth is bounded by k ($k = 0$ is synchronous subtyping). With k fixed, k-bounded asynchronous subtyping is sound and decidable (though the existence of such a k is undecidable [8]).

Asynchronous (binary) approaches include the seminal algorithm from [5, 6] for binary sessions and [3], which uses word approximation [14] to enlarge the class of problems for which asynchronous subtyping can be algorithmically established. The extension from the

binary setting to multiparty is not obvious. For example, swapping two send actions or two receive actions (to encode SW2) is not safe in the binary setting: it violates the FIFO ordering of messages, potentially causing communication mismatches (e.g., $s!\mathrm{a}.\, s!\mathrm{b} \not\leq s!\mathrm{b}.\, s!\mathrm{a}$) or deadlocks (e.g., $c?\mathrm{a}.\, c?\mathrm{b} \not\leq c?\mathrm{b}.\, c?\mathrm{a}$). The multiparty setting adds more interleaving options to handle and requires more machinery to keep track of the actions-role association.

As to multiparty asynchronous subtyping, the algorithm in [18], while applicable to several realistic protocols, does not address "complex accumulation patterns" [6] that can be, instead, handled by the tools of [3, 6]. MAS [27] has been mechanized [23] providing a sound and complete tool for verifying subtyping *given a hand-crafted certificate as input*, but provides no solution to the fundamental problem of how to compute a certificate.

Interest in subtyping seems to be increasing rather than abating and another approach to asynchronous multiparty subtyping [36], based on projecting global types (a global model of a collection of local types), was proposed in parallel to [3] at a sister conference. The global types in [36] provide richer select and branching primitives with respect to classic global types: in a given state, a sender can select to send a message to different receivers, and likewise a receives can wait on different senders. A central requirement of multiparty subtyping in [36] is sub-protocol fidelity [36, Definition 5.1(i)] which ensures that any refinement (subtype) is consistent with a given global type. Sub-protocol fidelity yields a very different notion of subtyping to the one adopted here (and [5, 6, 12, 13] and formalised as MAS [27]). Sub-protocol fidelity allows some actions to be swapped, but it does not allow the anticipation of outputs before inputs, which is a requirement for performance optimisation [38]. This means that a session type may be a subtype of another according to our algorithm (and MAS), but not that of [36]. Thus it is no paradox when the authors of [36] claim decidability while pointing out that asynchronous subtyping is undecidable [7, 35]. Classically, the subtyping question is answered pairwise: by comparing a type against a candidate subtype, without reference to a global type. This formulation is favourable when the global type is unknown, for instance, if types are mined from source code.

Session subtyping is related to compliance between session contracts [1], which is inspired by testing-preorders [19]. Must-testing, in the binary and synchronous setting, can be seen as a sort of inductive compliance relation [39]. Multiparty synchronous must-testing has been studied in [20], binary asynchronous in [2]. Another interesting preorder is fair subtyping [9], which takes its cues from should testing preorder [41], and preserves the possibility of correct termination. Difference nuances of session subtyping, such as those mentioned above, suggest the need of generic subtyping algorithms, parameterised by the underlying preorder.

8 Conclusion

The application of abstract interpretation to asynchronous session subtyping is challenging in the multiparty setting. To provide a semantic basis for abstraction, an alternative notion of SISO-tree refinement is given, and as well as new notions of derivative which, together, enables the MAS subtyping relation to be re-expressed over sets of session-trees. This provides a foundation for abstract interpretation, akin to a classical collecting semantics, whose use is illustrated with a new abstract domain for session trees, constructed from type graphs, that were originally proposed for type recovery. Overall, we provide the first algorithmic formulation of MAS together with experimental results which are truly encouraging.

References

1 Franco Barbanera and Ugo de'Liguoro. Sub-behaviour relations for session-based client/server systems. *Mathematical Structures in Computer Science*, 25(6):1339–1381, 2015. doi:10.1017/S096012951400005X.

2 Giovanni Bernardi, Ilaria Castellani, Paul Laforgue, and Léo Stefanesco. Constructive Characterisations of the MUST-preorder for Asynchrony. In *European Symposium on Programming*, volume 15694 of *Lecture Notes in Computer Science*, pages 88–116, 2025. doi:10.1007/978-3-031-91118-7_4.

3 Laura Bocchi, Andy King, and Maurizio Murgia. Asynchronous Subtyping by Trace Relaxation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 14570 of *Lecture Notes in Computer Science*, pages 207–226, Berlin, 2024. Springer-Verlag. doi:10.1007/978-3-031-57246-3_12.

4 François Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141, New York, 1993. Springer-Verlag. doi:10.1007/BFb0039704.

5 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A Sound Algorithm for Asychronous Session Subtyping. In *International Conference on Concurrency Theory*, volume 140 of *LIPIcs*, pages 38:1–38:16, Dagstuhl, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2019.38.

6 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A Sound Algorithm for Asychronous Session Subtyping and its Implementation. *Logical Methods in Computer Science*, 17(1):1–35, 2021. doi:10.23638/LMCS-17(1:20)2021.

7 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of Asynchronous Session Subtyping. *Information and Computation*, 256:300–320, 2017. doi:10.1016/j.ic.2017.07.010.

8 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. doi:10.1016/j.tcs.2018.02.010.

9 Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. Fair Asynchronous Session Subtyping. *Logical Methods in Computer Science*, 20:5:1–5:47, 2024. doi:10.48550/arXiv.2101.08181.

10 Janusz Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.

11 David Castro-Perez and Nobuko Yoshida. CAMP: cost-aware multiparty session protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):155:1–155:30, 2020. doi:10.1145/3428223.

12 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science*, 13(2):1–61, 2017. doi:10.23638/LMCS-13(2:12)2017.

13 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types. In *Principles and Practice of Declarative Programming*, pages 135–146, New York, 2014. ACM Press. doi:10.1145/2643135.2643138.

14 Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A Practical String Analyzer by the Widening Approach. In *Asian Symposium on Programming and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 374–388, Berlin, 2006. Springer-Verlag. doi:10.1007/11924661_23.

15 Liron Cohen. Non-well-founded Deduction for Induction and Coinduction. In *Conference on Automated Deduction*, volume 12699 of *Lecture Notes in Computer Science*, pages 3–24. Springer-Verlag, 2021. doi:10.1007/978-3-030-79876-5_1.

16 Bruno Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.

17 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of*

Programming Languages, pages 238–252, New York, 1977. ACM Press. doi:10.1145/512950.512973.

18 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *Symposium on Principles and Practice of Parallel Programming*, pages 246–261, New York, 2022. ACM Press. doi:10.1145/3503221.3508404.

19 Rocco De Nicola and Matthew Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984. doi:10.1007/BFb0036936.

20 Rocco De Nicola and Hernán Melgratti. Multiparty Testing Preorders. *Logical Methods in Computer Science*, 19:1:1–1:31, 2016. doi:10.46298/lmcs-19(1:1)2023.

21 Romain Demangeon and Kohei Honda. Full Abstraction in a Subtyped pi-Calculus with Linear Types. In *International Conference on Concurrency Theory*, volume 6901 of *Lecture Notes in Computer Science*, pages 280–296, Berlin, 2011. Springer-Verlag. doi:10.1007/978-3-642-23217-6_19.

22 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods in Systems Design*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.

23 Burak Ekici and Nobuko Yoshida. Completeness of asynchronous session tree subtyping in coq. In Yves Bertot, Temur Kutsia, and Michael Norrish, editors, *International Conference on Interactive Theorem Proving*, volume 309 of *LIPIcs*, pages 13:1–13:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ITP.2024.13.

24 Simon Fowler. An Erlang Implementation of Multiparty Session Actors. In *Interaction and Concurrency Experience*, volume 223 of *EPTCS*, pages 36–50, 2016. doi:10.4204/EPTCS.223.3.

25 Simon Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90, Berlin, 1999. Springer-Verlag. doi:10.1007/3-540-49099-X_6.

26 Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi Calculus. *Acta Informatica*, 42:191–225, 2005. doi:10.1007/s00236-005-0177-z.

27 Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434297.

28 Steve Hill. Combinators for Parsing Expressions. *Journal of Functional Programming*, 6(3):445–463, 1996. doi:10.1017/S0956796800001799.

29 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 22–138, Berlin, 1998. Springer-Verlag. doi:10.1007/BFb0053567.

30 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Principles of Programming Languages*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

31 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541, Berlin, 2008. Springer-Verlag. doi:10.1007/978-3-540-70592-5_22.

32 Gerda Janssens and Maurice Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming*, 13(2&3):205–258, 1992. doi:10.1016/0743-1066(92)90032-X.

33 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In *European Conference on Object-Oriented*

Programming, volume 222, pages 4:1–4:29, Dagstuhl, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2022.4.

34 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: liveness and safety for channel-based programming. In *Principles of Programming Languages*, pages 748–761, 2017. doi:10.1145/3009837.3009847.

35 Julien Lange and Nobuko Yoshida. On the Undecidability of Asynchronous Session Subtyping. In *Foundations of Software Science and Computation Structures*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, Berlin, 2017. Springer-Verlag. doi:10.1007/978-3-662-54458-7_26.

36 Elaine Li, Felix Stutz, and Thomas Wies. Deciding Subtyping for Asynchronous Multiparty Sessions. In *European Symposium on Programming*, volume 14576 of *Lecture Notes in Computer Science*, pages 176–205, Berlin, 2024. Springer-Verlag. doi:10.1007/978-3-031-57262-3_8.

37 Dimitris Mostrous and Vasco Vasconcelos. Session Typing for a Featherweight Erlang. In *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 95–109, 2011. doi:10.1007/978-3-642-21464-6_7.

38 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332, Berlin, 2009. Springer-Verlag. doi:10.1007/978-3-642-00590-9_23.

39 Maurizio Murgia. A fixed-points based framework for compliance of behavioural contracts. *J. Log. Algebraic Methods Program.*, 120:100641, 2021. doi:10.1016/j.jlamp.2021.100641.

40 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

41 Arend Rensink and Walter Vogler. Fair testing. *Information and Computation*, 205(2):125–198, 2007. doi:10.1016/j.ic.2006.06.002.

42 Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier. Type Analysis of Prolog Using Type Graphs. In *Programming Language Design and Implementation*, pages 337–348. ACM Press, 1994. doi:10.1145/773473.178479.

43 Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier. Type Analysis of Prolog Using Type Graphs. *Journal of Logic Programming*, 22(3):179–209, 1995. doi:10.1016/0743-1066(94)00021-W.