

Asynchronous Subtyping by Trace Relaxation

Anonymized

Anonymized Institute

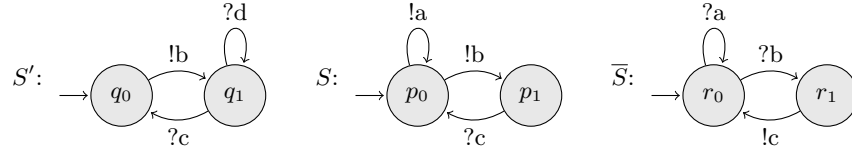
Abstract. Session subtyping answers the question of whether a program in a communicating system can be safely substituted for another, when their communication behaviours are described by session types. Asynchronous session subtyping is undecidable, hence the interest in devising sound, although incomplete, subtyping algorithms. State-of-the-art algorithms are formulated in terms of a data-structure called input trees. We show how input trees can be replaced by sets of traces, which opens up opportunities for applying techniques abstract interpretation techniques to the problem of asynchronous session subtyping. Sets of traces can be relaxed (enlarged) whilst still allowing subtyping to be observed, and one can choose relaxations that can be finitely represented, even when the input trees are arbitrarily large. We instantiate this strategy using regular expressions and show that it allows subtyping to be mechanically proven for communication patterns that were previously out of reach.

Keywords: asynchrony, session subtyping, automata, abstract interpretation

1 Introduction

Protocols, which are used to communicate and orchestrate activity in distributed systems, are notoriously difficult to write and understand. Session types [23, 34] have thus been proposed for specifying protocol interaction and automatically checking whether an implementation conforms to its specification. Session types extend data types to describe communication behaviour, and express the behaviour of units of design (sessions) in terms of which types of messages can be sent or received, and in what order. They have been integrated into mainstream languages and proved to be a powerful tool for static [25, 26, 28, 31, 32] and dynamic [1, 2] verification as well as API generation [24, 30].

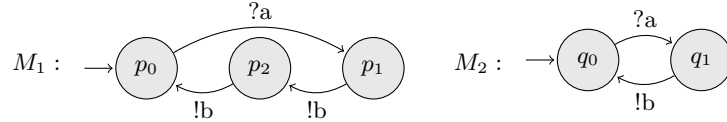
A fundamental problem in the application of session types is checking whether the implementation of one component in a distributed system can be substituted for another, without violating an overarching protocol. This problem can be formulated as *session subtyping* [11, 18, 20, 21], which is a preorder relation on session types: S' is a sub-type of S , written $S' \leq S$, if a program with type S can be safely substituted by a program with type S' . Consider S and S' below:



S and S' are expressed in automata notation where $!a$ (resp. $?a$) denotes a send (resp. receive) action on channel a . S models a process, which in state p_0 , can repeatedly request a service a , or request b and then receive a confirmation c .

The overarching protocol is defined, in a binary (client-server) session, as the parallel composition of S with its dual \bar{S} , written $S \mid \bar{S}$. The dual \bar{S} is obtained by swapping each send action with a corresponding receive action and vice versa. Due to syntactic constraints posed by session types [23], $S \mid \bar{S}$ enjoys a number of key properties (e.g., deadlock freedom, communication safety). A process behaving as S can be safely substituted with another behaving as S' that has less sends (e.g. the absent $!a$) and more receives (e.g. the additional $?d$). This notion of substitutability is co-variant on send actions and contra-variant on receive actions, and preserves the key properties in protocol $S' \mid \bar{S}$.

We focus on *asynchronous session subtyping* (async subtyping for short) as asynchronous communications (over FIFO channels) are key in distributed systems and languages such as Go and Rust. Async subtyping, however, is undecidable [6, 27] so the search is on for sound algorithms which are sufficiently robust to prove subtyping in the majority of cases. Given a candidate subtype and a supertype, the subtyping problem can be viewed as a simulation game in which the supertype is required to mirror any input and output action performed by the subtype. Since communication is asynchronous, the subtype can send early in the sense that the supertype can only realise the same output after some inputs. Consider M_2 below, which models a server producing a news feed ($!b$) on request from a client ($?a$), where M_1 is a candidate subtype for M_2 :



After receiving on a , M_2 can immediately mimic the first send on b of M_1 , but it can only perform the second send on b after receiving another request. The input a is said to guard the output b . One needs to reason about these dependencies to verify that M_2 can follow the actions of M_1 , albeit with (a possibly unbounded number of) send actions being delayed. This is the challenge of asynchronous subtyping. Apart from substitutability, asynchronous subtyping enables protocol optimisation in which receives are postponed, so as to minimise busy waiting for messages [29]. In M_2 , if feed production was more efficient than request processing then it would be better if the server bundled feeds, as in M_1 .

The state-of-the-art approach to async subtyping [4, 5] represents a simulation game between the (candidate) subtype and supertype, in its entirety, with a simulation tree. The state of the supertype is modelled using an input tree [4, 5, 11, 10], which records and accumulates input actions which guard outputs. Figure 1 gives the simulation tree for M_1 and M_2 . Simulation commences at $p_0 \leq q_0$ where M_1 and M_2 are in their initial states p_0 and q_0 . The edges in the tree follow the actions of M_1 , with M_2 following along using its input tree. Step $p_0 \leq T_2$ models the scenario where M_1 is in state p_0 but, in M_2 , a second send on

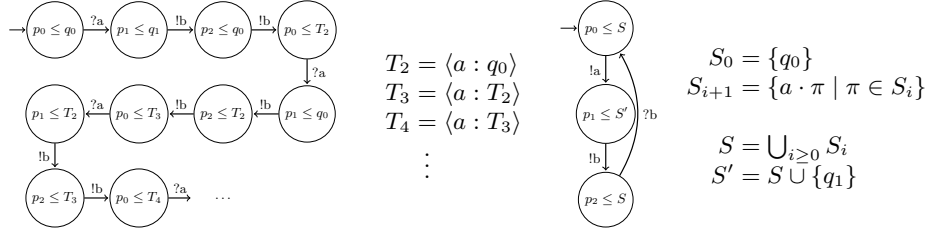


Fig. 1. A simulation tree (left) and collecting simulation graph (right) for M_1 and M_2

b is guarded by a receive on a . Input tree $T_2 = \langle a : q_0 \rangle$ expresses this dependency by recording that M_2 can continue at q_0 , after performing the pending receive on a . As the simulation of M_1 unfolds, however, the input trees for M_2 grow without bound, yielding an infinite simulation tree.

Previous work [4, 5] proposed a multi-step algorithm that computes a simulation tree until violation of a syntactic condition [5, Theorem 3.8] that is formulated in terms of the depth of input trees. The simulation tree is then divided into sub-trees, which are checked against a safety property [5, Definition 3.16]. The sub-trees are then used to generate systems of equations which are solved and checked against a compatibility condition [5, Definition 3.12]. The construction is ingenious, but the length of the proofs [5, p. 14, p. 19-20, p. 22-26] begs the question of whether subtyping can be solved more simply. Furthermore, can a strategy be found that is amenable to independent algorithmic checking? This would explain why subtyping holds, further instilling confidence.

Contribution Our development starts with the observation that an input tree can be represented, without loss of information, as a set of traces: one trace for each branch through the input tree. The rationale behind this encoding is that sets of traces can: (1) be relaxed (enlarged) and (2) be described as regular expressions. As to (1), a trace-based representation allows the subtyping algorithm to relax a set of traces to a strictly larger (possibly infinite) set, whilst still allowing subtyping to be observed. By covering all the sets of traces that arise in a simulation tree with a finite number of trace sets we can fold a simulation tree onto a graph to obtain a tractable (finite) representation. Regarding (2), (possibly infinite) sets of traces can themselves be finitely represented as regular expressions. For example, Figure 1 (right) shows a collecting simulation graph where the states of M_2 are relaxed to the set of traces S and S' , which can be represented, say, as a^*q_0 and $a^*q_0 + q_1$ respectively. The result is a subtyping algorithm equipped with relaxation and termination machinery which can prove subtyping on more (and more complex) problems than existing methods.

The use of sets of traces separates the proof for correctness of the core algorithm, from the problem of how to finitely represent sets of traces. This separation simplifies the theoretical development. If higher fidelity was required, regular expressions could be replaced with context-free grammars [13]; alterna-

tively the relaxations employed with regular expressions (string widening [12]) can be tuned without revisiting the correctness of the core algorithm.

Synopsis Section 2 introduces (session types as) communicating machines; Section 3 defines asynch subtyping with the formulation in [5] to facilitate comparison and Section 4 gives a sound formulation based on collecting simulation graphs. Section 5 gives an algorithm based on regular expressions and widening over collecting simulation graph, and introduces and evaluates our tool. Conclusion and related work are in Section 6. Proofs are in a separate appendix.

2 Preliminaries on Communicating Machines

Let A denote a finite alphabet, and $\mathbb{A} = \{!, ?\} \times A$ denote a finite set of send and receive actions. A communicating machine $M = (Q, q_0, \delta)$ (machine for short) is defined by a finite set of states Q , an initial state $q_0 \in Q$, and a transition relation $\delta \subseteq Q \times \mathbb{A} \times Q$. For a fixed machine $M = (Q, q_0, \delta)$, we write: $q \xrightarrow{w} q'$ iff $(q, w, q') \in \delta$; $q \xrightarrow{w}$ iff there exists q' such that $q \xrightarrow{w} q'$; $q_0 \xrightarrow{w_1, \dots, w_n} q_n$ iff there exist $q_1, \dots, q_{n-1} \in Q$ such that $q_i \xrightarrow{w_{i+1}} q_{i+1}$ for $0 \leq i \leq n-1$.

Given a sequence of labels $\vec{a} = a_1, \dots, a_k$ and a direction $\star \in \{!, ?\}$, we write $\star \vec{a}$ for the sequence of actions $\star a_1, \dots, \star a_k$. The maps $\text{in}_M : Q \rightarrow \wp(A)$ and $\text{out}_M : Q \rightarrow \wp(A)$ are defined: $\text{in}_M(q) = \{a \in A \mid q \xrightarrow{?a}\}$ and $\text{out}_M(q) = \{a \in A \mid q \xrightarrow{!a}\}$. The predicate $\text{send}_M(q)$ holds iff $\text{out}_M(q) \neq \emptyset$ and $\text{recv}_M(q)$ holds iff $\text{in}_M(q) \neq \emptyset$. The predicate $\text{final}_M(q)$ holds iff $\neg \text{send}_M(q)$ and $\neg \text{recv}_M(q)$.

Definition 1 (Session types correspondence). *For a given $M = (Q, q_0, \delta)$, M is deterministic iff $(q, w, q_1), (q, w, q_2) \in \delta$ implies $q_1 = q_2$; M has no mixed states iff $\neg \text{send}_M(q)$ or $\neg \text{recv}_M(q)$ for all $q \in Q$. A session type corresponds [19] to a deterministic machine without mixed states.*

Henceforth we focus on systems of *two* deterministic machines without mixed states, which correspond to *binary* session types. Binary session types describe two-party protocols (e.g., client-server as POP2, SMTP). State-of-the-art asynchronous subtyping algorithms [5] are formulated on binary sessions (each session involving two rather than many participants). We focus on demonstrating how abstraction can be applied to these algorithms and thus, likewise, adopt the binary setting.

Because M is deterministic, the relation δ can be interpreted as a partial function $Q \times \mathbb{A} \rightarrow Q$ defined by $\delta(q, \ell) = q'$ iff $q \xrightarrow{\ell} q'$. Following [5] we introduce the predicate $\text{cycle}_M(!, q)$ to aid the characterisation of orphan messages:

Definition 2. *The predicate $\text{cycle}_M(!, q)$ holds iff there exist $\vec{a} \in A^*, \vec{b} \in A^+$ and $q' \in Q$ such that $q \xrightarrow{! \vec{a}} q'$ and $q' \xrightarrow{! \vec{b}} q$.*

The predicate $\text{cycle}_M(!, q)$ thus holds iff from q one can reach, using a possibly empty sequence of send actions, a cycle (from q' to q' itself) of send actions. The predicate $\text{cycle}_M(?, q)$ is defined analogously.

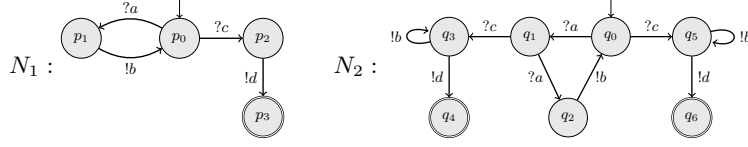


Fig. 2. Communicating machines N_1 (14may2) and N_2 (14may1)

3 Asynchronous Subtyping with Input Trees

We define input trees and asynchronous subtyping, adopting the formulation of [5]. Input trees are defined over the states Q of a supertype. Asynchronous subtyping is then defined in terms of input trees, the trees capturing input accumulation for guarded outputs.

Definition 3. *The set of input trees T_Q over Q is the least set such that: (1) if $q \in Q$ then $q \in T_Q$; (2) if I is an index set, $\forall i \in I. a_i \in A, t_i \in T_Q$ and $\forall i, j \in I. i \neq j \implies a_i \neq a_j$ then $\langle a_i : t_i \mid i \in I \rangle \in T_Q$*

An input tree over Q is either a state in Q or an accumulated input. A term of the form $\langle a_i : t_i \mid i \in I \rangle$ represents an accumulated input that presents an options a_i for each $i \in I$, followed by a tree t_i . Note that any input tree of T_Q is necessarily finite. The following definition shows how to build the input tree $\text{inTree}_M(q)$ for a state q of a given machine M , and defines the associated set of leaves $\text{leaf}(t)$ of the input tree t .

Definition 4 (Input tree). *Define $\text{inTree}_M : Q \rightarrow T_Q$ and $\text{leaf} : T_Q \rightarrow \wp(Q)$*

$$\text{inTree}_M(q) = \begin{cases} \perp & \text{if } \text{cycle}_M(?, q) \\ q & \text{else if } \text{in}_M(q) = \emptyset \\ \langle a_i : \text{inTree}_M(\delta(q, ?a_i)) \mid i \in I \rangle & \text{else if } \text{in}_M(q) = \{a_i \mid i \in I\} \end{cases}$$

$$\text{leaf}(t) = \begin{cases} \{t\} & \text{if } t \in Q \\ \bigcup \{\text{leaf}(t_i) \mid i \in I\} & \text{else if } t = \langle a_i : t_i \mid i \in I \rangle \end{cases}$$

The $\text{cycle}_M(?, q)$ condition (also used in [5]) ensures that $\text{inTree}_M(q)$, if defined, is finite. Note that $a_i : \text{inTree}_M(q_i)$ is well-defined in the above. To see why, suppose $\delta(q, a_i) = q_i$. Observe that if $\neg \text{cycle}_M(?, q)$ then $\neg \text{cycle}_M(?, q_i)$. Repeating this argument it follows $\text{inTree}_M(q_i) \neq \perp$, as required.

Example 1 (Running example: input trees and leaves). The machines 14may2 and 14may1 specified in Figure 2 originate from the GitHub repository which accompanies [5]. Henceforth let $N_1 = 14\text{may}2$ and $N_2 = 14\text{may}1$.

$$\begin{aligned} \text{inTree}_{N_2}(q_0) &= \langle a : \langle a : q_2, c : q_3 \rangle, c : q_5 \rangle & \text{leaf}(\text{inTree}_{N_2}(q_0)) &= \{q_2, q_3, q_5\} \\ \text{inTree}_{N_2}(q_1) &= \langle a : q_2, c : q_3 \rangle & \text{leaf}(\text{inTree}_{N_2}(q_1)) &= \{q_2, q_3\} \\ \text{inTree}_{N_2}(q_i) &= q_i \text{ for all } 2 \leq i \leq 6 & \text{leaf}(q_3) &= \{q_3\} \end{aligned}$$

Next, we introduce a substitution θ that we use, in the definition of asynchronous subtyping, to model the accumulation of inputs as simulation unfolds. Input trees are extended at their leaves by the application of a substitution θ .

Definition 5 (Substitution). *If $q_i \in Q$ and $t_i \in T_Q$ for all $i \in I$ then $\theta = \{q_i \mapsto t_i \mid i \in I\}$ denotes an operator $T_Q \rightarrow T_Q$ where $\theta(t)$ is the input tree obtained by simultaneously substituting each occurrence of q_i in t with t_i .*

In Definition 6 we introduce the notion of an async subtyping relation between states of a candidate subtype and input trees of a supertype. We follow [11] and, like [5], adopt the conventional orphan-free version of asynchronous subtyping [7, Definition 2.4] adapted to the setting of communicating machines:

Definition 6. *An async subtyping relation for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ is a binary relation $\mathcal{R} \subseteq P \times T_Q$ such that $(p, t) \in \mathcal{R}$ implies:*

1. *if $\text{final}_{M_1}(p)$ then $t = q$ for some $q \in Q$ and $\text{final}_{M_2}(q)$*
2. *if $\text{recv}_{M_1}(p)$ then*
 - (a) *if $t = q$ for some $q \in Q$ then $\text{recv}_{M_2}(q)$ and if $q \xrightarrow{?a} q'$ there exist $p \xrightarrow{?a} p'$ and $(p', q') \in \mathcal{R}$*
 - (b) *if $t = \langle a_i : t_i \mid i \in I \rangle$ then for all $i \in I$ there exist $p \xrightarrow{?a_i} p'$ and $(p', t_i) \in \mathcal{R}$*
3. *if $\text{send}_{M_1}(p)$ then:*
 - (a) *if $t = q$ for some $q \in Q$ and $\text{send}_{M_2}(q)$ then if $p \xrightarrow{!a} p'$ there exist $q \xrightarrow{!a} q'$ and $(p', q') \in \mathcal{R}$*
 - (b) *otherwise if $\text{leaf}(t) = \{q_i \mid i \in I\}$ then $\neg \text{cycle}_{M_1}(!, p)$ and*
 - i. *$t_i = \text{inTree}_{M_2}(q_i) \neq \perp$ for all $i \in I$*
 - ii. *if $p \xrightarrow{!a} p'$ and $\theta = \{q \mapsto q' \mid q \in Q, q \xrightarrow{!a} q'\}$ then $\text{leaf}(t_i) \subseteq \text{dom}(\theta)$ for all $i \in I$ and $(p', \kappa(t)) \in \mathcal{R}$ where $\kappa = \{q_i \mapsto \theta(t_i) \mid i \in I\}$*

Case (1) is self-explanatory. Case (2) is for input actions in M_1 and realises contra-variance with respect to inputs. Case (2.a) applies when the states p and q are in sync, whereas case (2.b) applies when an accumulated input a_i in M_2 is consumed by a corresponding input action of M_1 . In case (2.a), condition $\text{recv}_{M_2}(q)$ ensures that the guarded clause $q \xrightarrow{?a} q'$ does not hold vacuously. Case (3) is for output actions in M_1 and implements output co-variance. Case (3.a) applies when M_1 and M_2 are in sync, while case (3.b) is for accumulated inputs. The negated cycle_{M_1} predicate mirrors [5] and prevents orphan messages, ensuring that accumulated inputs are eventually considered. Clause (3.b.i) was implicit in [5] but is used in the proofs for structuring, and is thus made explicit. Clause (3.b.ii) ensures that if p in M_1 can send, then every leaf of the corresponding input tree t in M_2 can make a matching send action.

Definition 7 (Async Subtyping). *$M_1 = (P, p_0, \delta_1)$ is an (async) subtype of $M_2 = (Q, q_0, \delta_2)$, written $M_1 \leq M_2$, iff there exists an async subtyping relation $\mathcal{R} \subseteq P \times T_Q$ for M_1 and M_2 such that $(p_0, q_0) \in \mathcal{R}$.*

4 Asynchronous Subtyping with Input Traces

Simulation trees [5] provide a foundation for checking subtyping, but because their branches can grow arbitrarily long, they are not tractable in themselves. To obtain a model which is amenable to abstraction, we substitute an input tree with a set of input traces. Sets of input traces can be easily relaxed by adding more input traces, which is key to deriving a finite alternative representation.

Definition 8 (Input Traces). *Given a fixed alphabet A and a set of states Q , input traces (traces for short) are words formed from the alphabet A (which are ranged over by π) followed by a state in Q : $Tr_Q = \{\pi \cdot q \mid \pi \in A^*, q \in Q\}$. The empty word is denoted ϵ .*

The development begins by lifting a simulation tree to sets of traces, a construction which itself requires some set-level auxiliary operations:

Definition 9 (Traces of an input tree). *The set of traces of an input tree is given by the map $tr : T_Q \cup \{\perp\} \rightarrow \wp(Tr_Q)$ defined by:*

$$tr(t) = \begin{cases} \emptyset & \text{if } t = \perp \\ \{t\} & \text{if } t \in Q \\ \{a_i \cdot \pi \mid \pi \in tr(t_i), i \in I\} & \text{if } t = \langle a_i : t_i \mid i \in I \rangle \end{cases}$$

Example 2 (Running example: traces). Continuing with N_1 and N_2 of Example 1 (Figure 2), $tr(\text{inTree}_{N_2}(q_0)) = \{aaq_2, acq_3, cq_5\}$ and $tr(\text{inTree}_{N_2}(q_1)) = \{aq_2, cq_3\}$.

4.1 Collecting simulation

A (collecting) simulation tree is formulated in terms of a (collecting) simulation relation, defined below. The term collecting has been chosen to resonate with abstract interpretation [15] where a semantics is lifted to operate on sets of data points (to give a so-called collecting semantics) which provides a semantic substrate for synthesising an algorithm.

Definition 10 (Collecting simulation). *The collecting simulation relation of two machines $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ is the least 5-place relation $\hookrightarrow \in P \times \wp(Tr_Q) \times A \times P \times \wp(Tr_Q)$, satisfying the rules in Figure 3, where $p \leq S \xrightarrow{\ell} p' \leq S'$ abbreviates $(p, S, \ell, p', S') \in \hookrightarrow$.*

In Figure 3, rules **Recv** and **RecvTr** collectively realise the second case of Definition 6: rule **Recv** realises case (2.a) for interactions in sync, and **RecvTr** realises case (2.b) that consumes an accumulated input. The contra-variance of receive manifests as $\text{in}_{M_2}(q) \subseteq \text{in}_{M_1}(p)$ in **Recv** and $a \in \text{in}_{M_1}(p)$ in **RecvTr**. Rules **Send** and **SendTr** realise case (3.a) and case (3.b) of Definition 6, respectively. In these rules, the co-variance of send appears as premise $\text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q)$ in **Send** and $\forall i \in I : \text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q_i)$ in **SendTr**. In rule **SendTr**, the $\text{leaf}(t_j) \subseteq \text{dom}(\theta)$ condition in case (3.b.) follows from the premise $\text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q_i)$ for

$$\begin{array}{c}
\frac{\text{in}_{M_2}(q) \subseteq \text{in}_{M_1}(p) \quad q \xrightarrow{?a} q'}{p \leq q \xrightarrow{?a} \delta_{M_1}(p, ?a) \leq q'} [\text{Recv}] \quad \frac{\text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q) \quad p \xrightarrow{!a} p'}{p \leq q \xrightarrow{!a} p' \leq \delta_{M_2}(q, !a)} [\text{Send}] \\
\\
\frac{a \in \text{in}_{M_1}(p)}{p \leq a \cdot \pi \xrightarrow{?a} \delta_{M_1}(p, ?a) \leq \pi} [\text{RecvTr}] \quad \frac{\neg \text{cycle}_{M_1}(!, p) \quad \text{tr}(\text{inTree}_{M_2}(q)) = \{\phi_i \cdot q_i \mid i \in I\} \quad k \in I \quad \forall i \in I : \text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q_i) \quad p \xrightarrow{!a} p'}{p \leq \phi \cdot q \xrightarrow{!a} p' \leq \phi \cdot \phi_k \cdot \delta_{M_2}(q_k, !a)} [\text{SendTr}] \\
\\
\frac{\forall \pi \in S : \exists b \in A : p \leq \pi \xrightarrow{?b} \quad S_a = \{\pi' \mid \pi \in S, p \leq \pi \xrightarrow{?a} p' \leq \pi'\} \neq \emptyset}{p \leq S \xrightarrow{?a} p' \leq S_a} [\text{RecvSet}] \\
\\
\frac{\forall \pi \in S : p \leq \pi \xrightarrow{!a} \quad S_a = \{\pi' \mid \pi \in S, p \leq \pi \xrightarrow{!a} p' \leq \pi'\} \neq \emptyset}{p \leq S \xrightarrow{!a} p' \leq S_a} [\text{SendSet}]
\end{array}$$

Fig. 3. Rules for trace-based asynchronous subtyping

all $i \in I$. To see this, let $q \in \text{leaf}(t_j)$ for some $j \in J$. Since $p \xrightarrow{!a} p'$, $a \in \text{out}_{M_1}(p)$ thus $a \in \text{out}_{M_2}(q)$ therefore $q \in \text{dom}(\theta)$.

The absence of mixed states (Definition 1) ensures that if both **Send** and **SendTr** are applicable then the traces which result coincide. The force of this is that clause ‘*otherwise if ...*’ of Definition 6(3.b) can be simplified to ‘*if ...*’ (so there is no need to prioritise the application of **Send** over **SendTr**). The current formulation of Definition 6(3.b) was chosen to align with that used in [5].

Rules **RecvSet** and **SendSet** lift subtyping from traces to sets of traces. In **RecvSet**, the first premise specifies a covering requirement: that a receive is possible for each trace of S . The second premise prescribes a grouping requirement: for a given receive action $?a$, the second precondition accumulates all those traces which can be derived by receiving on a . The requirement $S_a \neq \emptyset$ ensures that a non-empty subset of S contributes to S_a . The $S_a \neq \emptyset$ requirement, which likewise shows up in **SendSet**, also inhibits meaningless transitions of the form $p \leq \emptyset \xrightarrow{?a} p' \leq \emptyset$ and $p \leq \emptyset \xrightarrow{!a} p' \leq \emptyset$, which would otherwise hold vacuously.

For any given $p \leq S$, relaxing S to T , can result in either $p \leq T$ becoming stuck, or a move that preserves the inclusion of traces. To formulate this property, let $p \leq T \not\xrightarrow{\ell}$ denote the absence of a transition of the form $p \leq T \xrightarrow{\ell} p' \leq T'$.

Proposition 1 (Monotonicity). *Let $T \subseteq S \subseteq \text{Tr}_Q$ and $\ell \in \mathbb{A}$. Then if $p \leq T \xrightarrow{\ell} p' \leq T'$ either: $p \leq S \not\xrightarrow{\ell}$ or $p \leq S \xrightarrow{\ell} p' \leq S'$ where $T' \subseteq S'$.*

4.2 Collecting simulation trees and graphs

First, we provide an infinite model for collecting simulation using collecting simulation trees, that is an alternative presentation of simulation trees [5] where we represent the state of a supertype as a set of traces rather than an input tree.

Definition 11 (Collecting simulation (sim) tree). A collecting sim tree for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ is a labelled tree $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ where $\xrightarrow{\ell}_t \subseteq N \times N$ is a tree rooted at n_0 and $\mathcal{L} : N \rightarrow P \times \wp(\text{Tr}_Q)$ such that:

1. $\mathcal{L}(n_0) = (p_0, \{q_0\})$
2. if $p \leq S \xrightarrow{\ell} p' \leq S'$ and $\mathcal{L}(n) = (p, S)$ then $n \xrightarrow{\ell}_t n'$ for some $n' \in N$ such that $\mathcal{L}(n') = (p', S')$
3. if $n \xrightarrow{\ell}_t n'$ and $\mathcal{L}(n) = (p, S)$ then $\mathcal{L}(n') = (p', S')$ such that $p \leq S \xrightarrow{\ell} p' \leq S'$

Case (2) above ensures that a collecting sim tree enumerates *all* the transitions of $\xrightarrow{\ell}$ whereas case (3) ensures that the tree *only* enumerates $\xrightarrow{\ell}$ transitions. Note that a collecting sim tree is unique up to tree isomorphism.

Theorem 1 shows that subtyping can be expressed in terms of successful branches (Definition 12) of collecting sim trees.

Definition 12 (branches). A branch of a collecting sim tree $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ is a (possibly infinite) sequence $n_0, n_1, \dots \subseteq N$ such that $n_i \xrightarrow{\ell}_t n_{i+1}$ for all consecutive n_i, n_{i+1} . A complete branch of the collecting sim tree is a branch which is not a strict prefix of another branch of the collecting sim tree. A successful branch is a complete branch which is either infinite or whose last node n is labelled $\mathcal{L}(n) = (p, F)$ with $F \subseteq Q$, $\text{final}_{M_1}(p)$, and $\text{final}_{M_2}(q)$ for all $q \in F$.

The concept of successful branch allows for F to include multiple final states. This degree of generality supports supertypes with two or more final states (such as q_4 and q_6 of the machine N_2 of Example 1) when, later, successful branches are deployed in the context of collecting simulation graphs (see Figure 4).

Theorem 1 (Equivalence). Let $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ be a collecting sim tree for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$. $M_1 \leq M_2$ iff every complete branch in $(N, \xrightarrow{\ell}_t)$ is successful.

Simulation trees and collecting simulation trees can grow without bound. However, growth can be curtailed by the judicious application of relaxation:

Definition 13 (Collecting simulation (sim) graph). A collecting sim graph for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ is a labelled graph $(N, n_0, \xrightarrow{\ell}_g, \mathcal{L})$ where $\xrightarrow{\ell}_g \subseteq N \times N$ is a graph rooted at n_0 and $\mathcal{L} : N \rightarrow P \times \wp(\text{Tr}_Q)$ such that:

1. $\mathcal{L}(n_0) = (p_0, \{q_0\})$
2. if $p \leq S \xrightarrow{\ell} p' \leq T$ and $\mathcal{L}(n) = (p, S)$ then there exists $n' \in N$ such that $n \xrightarrow{\ell}_g n'$, $\mathcal{L}(n') = (p', S')$ for some $S' \supseteq T$
3. if $n \xrightarrow{\ell}_g n'$ and $\mathcal{L}(n) = (p, S)$ then $\mathcal{L}(n') = (p', S')$ such that $S' \supseteq T$ and $p \leq S \xrightarrow{\ell} p' \leq T$

Relaxation manifests in case (2) of Definition 13 in that $S' \supseteq T$: S' is thus a relaxation of T . Note too that n' is not necessarily on the branch from n_0 to n . Case (3) ensures that each transition in a collecting sim graph has a counterpart in the collecting sim tree.

The concepts of (complete and successful) branch can be defined analogously for a collecting sim graph. With these concepts in place, the following result, which is consequence of Proposition 1, explains how a collecting sim graph simulates a collecting sim tree: each branch in the tree is described by a branch in the graph with possibly enlarged trace sets. This correspondence between a branch in the graph and a branch in the tree only holds if the branch in the collecting sim graph does not get stuck.

Corollary 1. *Let $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ (resp. $(N', n'_0, \xrightarrow{\ell}_g, \mathcal{L}')$) be a collecting sim tree (resp. graph) for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$. If $b = n_0 \cdots n_i$ is a branch in the tree $(N, \xrightarrow{\ell}_t)$ then there exists $b' = n'_0 \cdots n'_k$ in the graph $(N', \xrightarrow{\ell}_g)$ with either: $k = i$ or $k < i$ and $n'_k \not\xrightarrow{\ell}_g$. Moreover, $\mathcal{L}(n_j) = (p_j, S_j)$, $\mathcal{L}'(n'_j) = (p_j, S'_j)$ and $S_j \subseteq S'_j$ for all $j \leq k$.*

Example 3. Figure 1 (left) shows an infinite simulation tree (following the notation of [5]) for machines M_1 and M_2 given in the introduction. The corresponding collecting sim tree has the same structure but $T_2 = \langle a : q_0 \rangle$ is substituted with $\{aq_0\}$, $T_3 = \langle a : \langle a : q_0 \rangle \rangle$ with $\{aaq_0\}$, whereas q_0 and q_1 (at and beneath the root of the tree) are replaced with $\{q_0\}$ and $\{q_1\}$ in the collecting sim tree. A (finite) collecting sim graph for M_1 and M_2 is shown in Figure 1 (right). Observe $q_0 \in S$, $q_1 \in S'$, $q_0 \in S$, $aq_0 \in S$, $aq_0 \in S'$, $aaq_0 \in S$, $aq_0 \in S$, $q_0 \in S'$, etc.

The force of collecting sim graphs is that they still act as a vehicle for establishing asynchronous subtyping, as the following result asserts:

Theorem 2 (Soundness). *Let $(N', n'_0, \xrightarrow{\ell}_g, \mathcal{L})$ be a collecting sim graph for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$. Then $M_1 \leq M_2$ if every complete branch in $(N', \xrightarrow{\ell}_g)$ is successful.*

Example 4 informally anticipates how finite representations of infinite executions can be algorithmically computed (using regular expressions) ahead of the detailed presentation and evaluation of the algorithm in the following sections.

Example 4 (Running example: collecting sim graph). Continuing with Example 1 (Figure 2), N_1 and N_2 are examples of machines for which [5] cannot prove subtyping, even though it does hold. In contrast, Figure 4 presents a collecting sim graph showing $N_1 \leq N_2$. The graph is rooted at n_0 where $\mathcal{L}(n_0) = (p_0, S_0)$.

5 Async Subtyping with Regular Expressions

Our work was motivated by the question of whether subtyping can be addressed with a simpler and more general approach. Beyond this conceptual question,

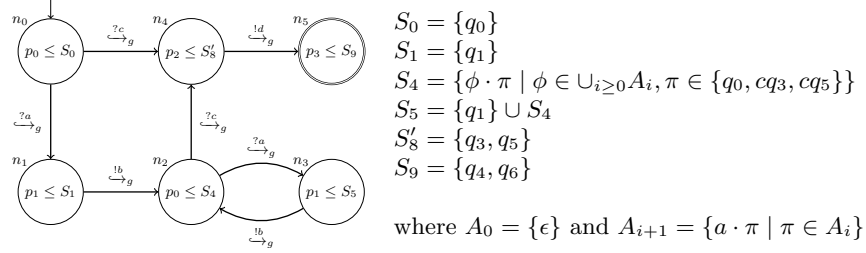


Fig. 4. A collecting sim graph for N_1 and N_2

there is the practical matter of whether our subtyping can algorithmically establish subtyping on more problems than before [4, 5]. To do so, we represent sets of traces using regular expressions and simulate the operations on sets of traces with analogous operations on regular expressions. To derive a finite collecting sim graph, we apply regular expression widening [12].

5.1 Representing sets of traces with regular expressions

A set of traces can be represented as a finite set of regular expressions drawn from the syntactic category Reg_A which is parameterised by alphabet A . Reg_A is inductively defined as $\text{Reg}_A = \epsilon \mid C \mid r \cdot r' \mid r^*$ where $C \subseteq A$, $r, r' \in \text{Reg}_A$, and \cdot is concatenation of words. To specify the language (set of words) represented by a regular expression, recall that Kleene closure W^* of a set of words W is defined as $W^* = \cup_{i=0}^{\infty} W_i$ where $W_0 = \{\epsilon\}$ and $W_{i+1} = \{\omega \cdot \omega' \mid \omega \in W, \omega' \in W_i\}$. Then the language of $r \in \text{Reg}_A$, denoted $\llbracket r \rrbracket$, is defined as $\llbracket \epsilon \rrbracket = \{\epsilon\}$, $\llbracket C \rrbracket = C$, $\llbracket r \cdot r' \rrbracket = \{\omega \cdot \omega' \mid \omega \in \llbracket r \rrbracket, \omega' \in \llbracket r' \rrbracket\}$ and $\llbracket r^* \rrbracket = \llbracket r \rrbracket^*$.

If $r \in \text{Reg}_A$ and $q \in Q$ the pair (r, q) represents the sets of traces $\llbracket (r, q) \rrbracket = \{\pi \cdot q \mid \pi \in \llbracket r \rrbracket\}$. Furthermore, if $R \subseteq \text{Reg}_A \times Q$ then R represents the traces $\llbracket R \rrbracket = \cup \{\llbracket (r, q) \rrbracket \mid (r, q) \in R\}$. Henceforth rq will abbreviate the pair (r, q) .

Example 5. To illustrate, $\llbracket \{a^*q_0, cq_3\} \rrbracket = \{cq_3\} \cup \{\pi \cdot q_0 \mid \pi \in \cup_{i \geq 0} A_i\}$ with A_i defined as in Figure 4.

Our technique uses the existing notion of widening [15, 16] to approximate regular expressions, namely to relax a sequence of regular expressions to derive another sequence which is not strictly increasing (thereby inducing convergence):

Definition 14. An operation $\nabla : \text{Reg}_A \times \text{Reg}_A \rightarrow \text{Reg}_A$ is a widening iff given a sequence $s_0, s_1, \dots \in \text{Reg}_A$ such that $\llbracket s_i \rrbracket \subseteq \llbracket s_{i+1} \rrbracket$ for all $i \geq 0$, the (widened) sequence $w_0 = s_0$ and $w_{i+1} = w_i \nabla s_{i+1}$ satisfies the following properties:

- $\llbracket s_i \rrbracket \subseteq \llbracket w_i \rrbracket$ and $\llbracket w_i \rrbracket \subseteq \llbracket w_{i+1} \rrbracket$ for all $i \geq 0$
- the sequence $\llbracket w_0 \rrbracket, \llbracket w_1 \rrbracket, \dots$ is not strictly increasing

Our approach is parametric on the widening (of which there are many [14]). We provide a primer on (string) widening to keep the presentation self-contained.

5.2 Widening regular expressions (a self-contained primer)

The intuition behind the widening we adopt [12] is to preserve commonality across two regular expressions and resolve any difference using Kleene star for relaxation. The widening scans both expressions left-to-right and, as it does so, it partitions each expression into a prefix p which has been traversed and a suffix s which is yet to be considered. The state of the scan thus represented by a pair (p, s) , with widen_k operating on two such pairs simultaneously:

$$\begin{aligned} \text{widen}_k((p, \epsilon), (p', s')) &= \text{mash}_k(p, p' \cdot s') & \text{widen}_k((p, s), (p', \epsilon)) &= \text{mash}_k(p \cdot s, p') \\ \text{widen}_k((p, q \cdot s), (p', q' \cdot s')) &= \\ &\begin{cases} \text{mash}_k(p, p') \circ q \circ \text{widen}_k((\epsilon, s), (\epsilon, s')) & \text{if } q = q' \text{ and } \text{sh}(q) \leq k \\ \text{widen}_k((p \cdot q, s), (p' \cdot q, s')) & \text{if } q = q' \text{ and } \text{sh}(q) > k \\ \text{widen}_k((p \cdot q, s), (p', q' \cdot s')) & \text{if } q \neq q' \text{ and } |s| > |s'| \\ \text{widen}_k((p, q \cdot s), (p' \cdot q', s')) & \text{if } q \neq q' \text{ and } |s| \leq |s'| \end{cases} \end{aligned}$$

The widening is defined in terms of two notions of size: (1) *star height* defined $\text{sh}(\epsilon) = \text{sh}(C) = 0$, $\text{sh}(r^*) = \text{sh}(r) + 1$ and $\text{sh}(r \cdot s) = \max(\text{sh}(r), \text{sh}(s))$; (2) *star length* defined $|\epsilon| = 0$, $|C| = |r^*| = 1$ and $|r \cdot s| = |r| + |s|$. Given two expressions r and s , the auxiliary $\text{mash}_k(r, s)$ computes a relaxation of r and s such that $\text{sh}(\text{mash}_k(r, s)) \leq k$ where k is a predefined depth bound. Thus $\llbracket r \rrbracket \subseteq \llbracket \text{mash}_k(r, s) \rrbracket$ and $\llbracket s \rrbracket \subseteq \llbracket \text{mash}_k(r, s) \rrbracket$.

Now consider scans of the form $(p, q \cdot s)$ and $(p', q' \cdot s')$ where q and q' are sub-expressions of the form C or r^* . If $q = q'$ then the common q is preserved provided $\text{sh}(q) \leq k$ and widening continues with scans (ϵ, s) and (ϵ, s') . Operator \circ is concatenation followed by a normalisation step [12] which ensures that no consecutive stars are introduced. If $\text{sh}(q) > k$ both q and q' are appended onto r and r' to be relaxed subsequently by mash_k . If $q \neq q'$ either q or q' is appended onto its prefix depending on $|s| > |s'|$ so that the remaining suffices are closer in length (which is merely a heuristic for improving their similarity). Analogous to mash_k , $\text{widen}_k((p, s), (p', s'))$ relaxes $p \cdot s$ and $p' \cdot s'$ such that $\text{sh}(\text{widen}_k((p, s), (p', s')) \leq k$. The star height bound ensures $r \nabla s = \text{widen}_k((\epsilon, r), (\epsilon, s))$ yields a sequence which is not strictly increasing [12].

Example 6. For brevity, we refer the reader to [12] for a definition¹ and commentary on the auxiliary $\text{mash}_k(r, s)$ but note that $\text{mash}_k(r, \epsilon) = r^*$ if $\text{sh}(r^*) \leq k$ and conversely $\text{mash}_k(\epsilon, s) = s^*$ if $\text{sh}(s^*) \leq k$. Hence

$$\begin{aligned} (a \cdot c \cdot d) \nabla (a \cdot b \cdot c) &= \text{widen}_1((\epsilon, a \cdot c \cdot d), (\epsilon, a \cdot b \cdot c)) = \epsilon \cdot a \cdot \text{widen}_1((\epsilon, c \cdot d), (\epsilon, b \cdot c)) \\ &= \epsilon \cdot a \cdot \text{mash}_1(\epsilon, b) \cdot c \cdot \text{widen}_1((\epsilon, d), (\epsilon, \epsilon)) \\ &= \epsilon \cdot a \cdot b^* \cdot c \cdot \text{mash}_1(d, \epsilon) = \epsilon \cdot a \cdot b^* \cdot c \cdot d^* \end{aligned}$$

The widening can be lifted from a pair of regular expressions to a pair of sets of regular expressions in a point-wise fashion [12]. In our setting, regular expressions represent traces, where each trace takes the form rq , and thus it is natural to partition a set of traces according to the state q in which they end. Two sets of expressions can be widened point-wise, for each q separately.

¹ Given in Appendix B for the convenience of the reviewer.

Algorithm 1 Algorithm for async subtyping ($\xrightarrow{\ell}$ is defined in Figure 3)

```

1: function SUBTYPE( $M_1, M_2, \Delta$ )           //  $M_1 = (P, p_0, \delta_1)$   $M_2 = (Q, q_0, \delta_2)$ 
2:   for ( $p \in P$ ) do
3:     if ( $\Delta(p) \neq \emptyset \wedge p \leq \Delta(p) \not\hookrightarrow$ ) then return maybe
4:      $R_p := \bigcup_{p' \in P} \{R \mid \exists \ell. p' \leq \Delta(p') \xrightarrow{\ell} p \leq R\}$ 
5:      $\Delta'(p) :=$  if ( $p \in wp$ ) then  $\Delta(p) \nabla R_p$  else  $\Delta(p) \cup R_p$ 
6:   if ( $\Delta' \subseteq \Delta$ ) then return  $\Delta$ 
7:   return SUBTYPE( $M_1, M_2, \Delta'$ )

```

5.3 Computing a collecting sim graph with regular expressions

Before outlining the algorithm, we illustrate it by example. Example 7 revisits Example 4 and shows how the sets of traces in Figure 2 can be algorithmically generated by using regular expressions and widening in tandem.

Example 7. Figure 5 presents a collecting sim graph for $N_1 \leq N_2$. Some nodes are shadowed by grey nodes that elaborate their relaxations by widening or union. The construction of the graph commences at node for $p_0 \leq R_0$ and proceeds iteratively, the number to the top-right of a node indicating the iteration at which that node is added to the graph. Iteration 1 is computed merely using the rules of Figure 3. On iteration 2, $p_0 \leq R_2$ is computed, again using the rules. Since p_0 was visited before, to ensure that p_0 is not revisited ad infinitum, a relaxation is applied, denoted ∇ following [15, 16], which relaxes R_2 using R_0 to obtain R'_2 . Observe how $\llbracket R_0 \rrbracket \subseteq \llbracket R'_2 \rrbracket$ and $\llbracket R_2 \rrbracket \subseteq \llbracket R'_2 \rrbracket$ but crucially the regular expression R'_2 is computed using a (widening) algorithm [12] which ensures that only a finite number of regular expressions are ever generated for p_0 . Not all nodes of Figure 5 need to be relaxed using widening. On iteration 3, p_1 is revisited. In this case, R'_3 is derived from R_3 and R_1 by computing their union. Thus again $\llbracket R_1 \rrbracket \subseteq \llbracket R'_3 \rrbracket$ and $\llbracket R_3 \rrbracket \subseteq \llbracket R'_3 \rrbracket$. The general strategy is to apply widening only as required, namely on a set of nodes which cut any cycle [3]. The machine N_1 of Figure 2 has a single cycle through p_0 and p_1 , thus it is sufficient to widen at either p_0 or p_1 . We elect to widen at p_0 , whereas for all other nodes of N_2 , the relaxation is union. On iteration 5, $p_1 \leq R_5$ is computing as before, the union of R'_3 with R_5 being R_5 . The following $\xrightarrow{?b}$ transition derives a regular expression R which is subsumed by R_4 , that is, $p_1 \leq R_5 \xrightarrow{?b} p_0 \leq R$ where $\llbracket R \rrbracket \subseteq \llbracket R_4 \rrbracket$. Thus the graph is no longer developed along the cycle. Despite employing relaxation, R_9 only contains q_4 and q_6 for which $\text{final}_{N_2}(q_4)$ and $\text{final}_{N_1}(q_6)$ hold. Recall $\text{final}_{N_1}(p_3)$ holds, hence subtyping is demonstrated.

Our SUBTYPE algorithm takes as input two machines $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ and is parametric on: (1) a widening $\nabla : \wp(\text{Reg}_A) \times \wp(\text{Reg}_A) \rightarrow \wp(\text{Reg}_A)$ and (2) a set $wp \subseteq P$ of widening points. At least one state of wp must appear in any cycle of M_1 ; a condition which is sufficient for widening to induce termination [3]. The mapping $\Delta : P \rightarrow \wp(\text{Reg}_A \times Q)$ represents the nodes of an

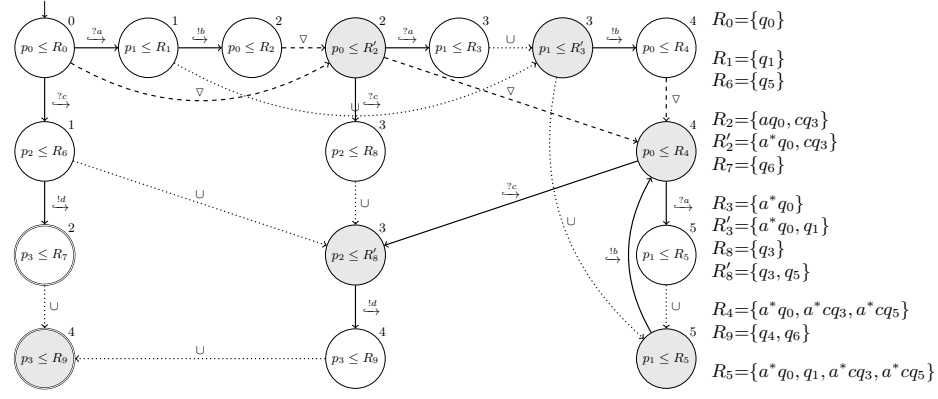


Fig. 5. A collecting simulation graph for proving $N_1 \leq N_2$: reprise

evolving collecting sim graph: $\text{SUBTYPE}(M_1, M_2, \Delta)$ is initially primed with $\Delta = \lambda p. \text{ if } (p = p_0) \text{ then } \{(\epsilon, q_0)\} \text{ else } \emptyset$. In line 3, **maybe** is returned if the simulation gets stuck. Note that $p \leq R \xrightarrow{\ell} p' \leq R'$ abbreviates $p \leq \llbracket R \rrbracket \xrightarrow{\ell} p' \leq \llbracket R' \rrbracket$ and likewise $p \leq R \not\vdash$ abbreviates $p \leq \llbracket R \rrbracket \not\vdash$. In line 4, R_p collects all the (r, q) pairs reachable at p in the current iteration Δ . $\Delta(p)$ is then relaxed to $\Delta'(p)$ applying widening if $P \in wp$ and union otherwise. In line 5, $\Delta' \subseteq \Delta$ iff $\llbracket \Delta(p) \rrbracket \subseteq \llbracket \Delta'(p) \rrbracket$ for all $p \in P$. This check determines whether a fix-point is reached: if so the algorithm returns Δ . **SUBTYPING** is sound and, due to widening, is guaranteed to terminate. In short, if **SUBTYPING** returns Δ then $M_1 \leq M_2$, otherwise it returns **maybe** and the subtyping check is deemed inconclusive.

For complexity, observe that wp can be chosen so that each state of $P \setminus wp$ has at most one incoming edge. Then algorithm 1 updates each state of P at most $(c|Q|)^{|wp|}$ times, updating Δ at most $|P|(c|Q|)^{|wp|}$ times, where c bounds the number of times a regular string can be relaxed. But $c \leq (2|Q|)^{d+2}$ (Appendix C).

5.4 Implementation and benchmarking

The regular expression-based subtyping algorithm has been implemented in Scala 3.2.2 on a laptop running Ubuntu 22.04.2 with 32 GB of DDR3 and a 2.8GHz Intel i7 processor. The code base is 1059 LOC, making use of parser combinators and the mutable and immutable Set libraries. No attempt has been made to improve the iteration strategy (which is normally a source of speedups). The tool and benchmarks are available at <https://anonymous.4open.science/r/AsynchSubtypingRegex-2905/>. The benchmarks² consists of 175 pairs of session types: 83 pairs where one type is known to be a subtype of the other (the

² The suite is based on the benchmark in [4, 5] with the addition of one (positive) case that is used in [4, 5] as a running example.

M_1	M_2	$ M_1 $	$ M_2 $	[5]	regex	time	M_1	M_2	$ M_1 $	$ M_2 $	[5]	regex	time
ctxta1	ctxta2	7	5	✗	✓	110	march3testa1	march3testa2	6	7	✗	✓	222
ctxtb1	ctxtb2	6	7	✗	✓	41	aaaaaab1	aaaaaab2	5	3	✗	✓	43
14may2	14may1	4	7	✗	✓	10	ex1okloop	ex2okloop	10	8	✗	✓	1757
badseq1	badseq2	5	12	✗	✓	1127	march3testa1	march3testb2	6	10	✗	✗	8

Fig. 6. Comparison of subtyping experiments: success rates and execution time (in ms)

positive problems); and 92 pairs which are known not to be in a subtyping relation (the negative problems).

If successful, our tool generates a collecting sim graph (in the form of Δ) which provides a concrete artefact that certifies subtyping. This is a positive outcome. Alternatively, the algorithm terminates with an inconclusive verdict. We have applied our tool to all the subtyping problems in the benchmarking suite. Our tool gave positive outcome for 82 of them, whereas the tool in [4, 5] gave 75 positive outcomes. In addition to certifying all positive cases in [4, 5], the tool could certify 7 “complex accumulation [input tree] patterns” [5] that were inconclusive cases in previous work. All 92 negative problems were (rightly) categorised as inconclusive by our tool.

An analysis of the 7 complex accumulation patterns is summarised in Figure 6. The M_1 (resp. M_2) column give the candidate subtype (resp. type). To convey some indication of the size of the problems, the $|M_1|$ (resp. $|M_2|$) column gives the number of states in M_1 (resp. M_2). The [5] column indicates whether subtyping can be proven using the algorithm of [5] using their distribution. The regex column indicates whether subtyping can be proven using collecting sim graphs instantiated with regular expressions, as proposed in our work. Time is walltime measured in milliseconds, the median of 5 runs. Widening was performed with a maximum star height of just 1 ($k = 1$). The last example in Figure 6, $\text{marchtesta1} \leq \text{marchtestb2}$, is known to be positive but neither our tool nor the one in [4, 5] could prove it. Nevertheless, it is remarkable that the widening of [12] performs so well considering it was originally devised for extracting SQL queries from database application programs.

The certificate produced by the algorithm (in the form of Δ) can be checked against the rules of Figure 3, without using widening or iteration. This could conceivably be performed by a proof assistant for high-assurance applications.

We finally comment on one complex example, $\text{marchtesta1} \leq \text{marchtestb2}$, that neither our tool nor the one in [4, 5] could prove. A post mortem reveals that $p_4 \leq S_4$ gets stuck: traces of S_4 of the form $b\pi q_3$ cannot make any move thus RecvSet does not apply. However, $b\pi q_3$ originates from $\{a, b\}^* q_3$ in $p_0 \leq S_0$ which itself stems from $(\epsilon q_3 \nabla_1 a q_3) \nabla_1 b(\epsilon q_3 \nabla_1 a q_3)$. Setting $k = 2$ (or higher) does not remedy the problem, which suggests that the widening needs tuning. Indeed, replacing $\{a, b\}^* q_3$ in S_0 with a more nuanced relaxation, namely $(a^*(ba)^*a)^* q_3$, is sufficient to establish subtyping. Crucially, this shows that the problem does not lie in collecting sim graph construction itself but in the widening (something which can be tuned without change to the underlying framework).

6 Conclusion and Related Work

We presented an algorithm for (binary) asynchronous session subtyping based on the application of abstract interpretation to session types. Our approach centres on the use of sets of traces to obtain a tractable representation of input trees. Sets of traces allow us to separate the proof for correctness of the core algorithm, from the problem of how to finitely represent and manipulate traces. This separation makes the methodology modular and tunable. As well as providing a conceptually simple approach for proving subtyping, the resulting algorithm, when instantiated with an off-the-shelf string widening, can prove subtyping for rich forms of interaction that were previously out-of-reach [5]. From a large suite of benchmarks, our algorithm was able to verify subtyping all but one problem and, even for that, we have shown that the collecting simulation approach is still adequate for proving subtyping. These results show that abstract interpretation is a clean, useful and powerful vehicle for inferring subtyping. Furthermore, a collecting sim graph once obtained constitutes a *certificate* for validating subtyping. The certificate can be then checked by a third-party, without consideration for how the graph is actually derived (whether algorithmically or manually).

Related work Async subtyping was first explored in [29] where subtyping rules consider a restricted form of permutation on actions. These concepts were then refined [10, 11] to disallow orphan messages, a requirement adopted in [5] and inherited into our study for ease of comparison.

Since async subtyping is undecidable [6, 27], some works proposed decidable safe approximated algorithms. For instance, subtyping can be approximated by k -bounded asynchronous subtyping [7]. The state of the art is [4, 5] that inspired our work. Fragments of session types for which asyn subtyping is decidable include: alternating session types [7] and single-out (resp. single-in) types [7] where internal (resp. external) choices are singletons.

Fair subtyping [9, 33] is an alternative to standard subtyping that preserves the possibility of correct termination. Asynchronous fair subtyping [8] is undecidable, and a sound algorithm has been proposed [8], which extends [5]. We would expect trace relaxation to extend to this setting as well.

The work above mostly focuses on binary sessions. The subtyping algorithm of [17], instead, focuses on the more general case of async *multiparty* subtyping. When restricted to binary types, the algorithm in [17] is less powerful than both [5] and our algorithm. The last case of [17, Table 1], taken from the running example in [5], is undetected with deadlock-free subtyping [17] but is proven by [5] and ourselves (see case ‘sub – runningex \leq sup – runningex’ in <https://anonymous.4open.science/r/AsynchSubtypingRegex-2905/>). [17] is still able to establish subtyping for several realistic protocols. A precise definition of async multiparty subtyping (AMS) has been provided in Ghilezan et al. [22]. This means that AMS in [22] is sound and complete with respect to async multiparty typing with a subsumption rule. Such definition is not obviously useful for algorithmic purposes: it contains quantifications over uncountably infinite sets. Application of our methodology to AMS is an interesting future direction.

References

1. Bartoletti, M., Murgia, M., Scalas, A., Zunino, R.: Verifiable Abstractions for Contract-oriented Systems. *J. Log. Algebraic Methods Program.* **86**(1), 159–207 (2017)
2. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring Networks through Multiparty Session Types. *Theoretical Computer Science* **669**, 33–58 (2017), <https://doi.org/10.1016/j.tcs.2017.02.009>
3. Bourdoncle, F.: Efficient Chaotic Iteration Strategies with Widenings. In: *Formal Methods in Programming and Their Applications. Lecture Notes in Computer Science*, vol. 735, pp. 128–141. Springer-Verlag (1993). <https://doi.org/https://doi.org/10.1007/BFb0039704>
4. Bravetti, M., Carbone, M., Lange, J., Yoshida, N., Zavattaro, G.: A Sound Algorithm for Asynchronous Session Subtyping. In: *International Conference on Concurrency Theory. LIPIcs*, vol. 140, pp. 38:1–38:16. Schloss Dagstuhl, Leibniz-Zentrum für Informatik (2019), <http://dx.doi.org/10.4230/LIPIcs.CONCUR.2019.38>
5. Bravetti, M., Carbone, M., Lange, J., Yoshida, N., Zavattaro, G.: A Sound Algorithm for Asynchronous Session Subtyping and its Implementation. *Logical Methods in Computer Science* **17**(1), 1–35 (2021). [https://doi.org/10.23638/LMCS-17\(1:20\)2021](https://doi.org/10.23638/LMCS-17(1:20)2021)
6. Bravetti, M., Carbone, M., Zavattaro, G.: Undecidability of Asynchronous Session Subtyping. *Information and Computation* **256**, 300–320 (2017), <https://doi.org/10.1016/j.ic.2017.07.010>
7. Bravetti, M., Carbone, M., Zavattaro, G.: On the Boundary between Decidability and Undecidability of Asynchronous Session Subtyping. *Theoretical Computer Science* **722**, 19–51 (2018), <https://doi.org/10.1016/j.tcs.2018.02.010>
8. Bravetti, M., Lange, J., Zavattaro, G.: Fair Refinement for Asynchronous Session Types. In: *Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science*, vol. 12650, pp. 144–163. Springer-Verlag (2021). https://doi.org/10.1007/978-3-030-71995-1_8
9. Bravetti, M., Zavattaro, G.: A Foundational Theory of Contracts for Multi-party Service Composition. *Fundamenta Informaticae* **89**(4), 451–478 (2008)
10. Chen, T.C., Dezani-Ciancaglini, M., Scalas, A., Yoshida, N.: On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science* **13**(2), 1–61 (2017). [https://doi.org/10.23638/LMCS-13\(2:12\)2017](https://doi.org/10.23638/LMCS-13(2:12)2017)
11. Chen, T.C., Dezani-Ciancaglini, M., Yoshida, N.: On the Preciseness of Subtyping in Session Types. In: *Principles and Practice of Declarative Programming*. pp. 135–146. ACM Press (2014). <https://doi.org/10.1145/2643135.2643138>
12. Choi, T., Lee, O., Kim, H., Doh, K.: A Practical String Analyzer by the Widening Approach. In: *Asian Symposium on Programming and Systems. Lecture Notes in Computer Science*, vol. 4279, pp. 374–388. Springer-Verlag (2006). https://doi.org/10.1007/11924661_23
13. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: *Static Analysis Symposium. Lecture Notes in Computer Science*, vol. 2694, pp. 1–18. Springer-Verlag (2003). https://doi.org/10.1007/3-540-44898-5_1
14. Costantini, G., Ferrara, P., Cortesi, A.: A Suite of Abstract Domains for Static Analysis of String Values. *Software Practice and Experience* **45**, 245–287 (2015)
15. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In:

- Principles of Programming Languages. pp. 238–252. ACM Press (1977), doi: 10.1145/512950.512973
16. Cousot, P., Cousot, R.: Comparing the Galois connection and Widening/Narrowing approaches to Abstract Interpretation. In: Programming Language Implementation and Logic Programming. pp. 269–295. No. 631 in Lecture Notes in Computer Science, Springer-Verlag (1992), doi:10.1007/3-540-55844-6_142
 17. Cutner, Z., Yoshida, N., Vassor, M.: Deadlock-Free Asynchronous Message Re-ordering in Rust with Multiparty Session Types. In: Symposium on Principles and Practice of Parallel Programming. pp. 246–261. ACM Press (2022). <https://doi.org/10.1145/3503221.3508404>
 18. Demangeon, R., Honda, K.: Full Abstraction in a Subtyped pi-Calculus with Linear Types. In: International Conference on Concurrency Theory. Lecture Notes in Computer Science, vol. 6901, pp. 280–296. Springer-Verlag (2011). https://doi.org/10.1007/978-3-642-23217-6_19
 19. Deniélou, P.M., Yoshida, N.: Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In: International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 7966, pp. 174–186. Springer-Verlag (2013). https://doi.org/10.1007/978-3-642-39212-2_18
 20. Gay, S., Hole, M.: Types and Subtypes for Client-Server Interactions. In: European Symposium on Programming. Lecture Notes in Computer Science, vol. 1576, pp. 74–90. Springer-Verlag (1999)
 21. Gay, S., Hole, M.: Subtyping for Session Types in the Pi Calculus. *Acta Informatica* **42**, 191–225 (2005). <https://doi.org/10.1007/s00236-005-0177-z>
 22. Ghilezan, S., Pantovic, J., Prokic, I., Scalas, A., Yoshida, N.: Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.* **5**(POPL), 1–28 (2021). <https://doi.org/10.1145/3434297>
 23. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: European Symposium on Programming. Lecture Notes in Computer Science, vol. 1381, pp. 22–138. Springer-Verlag (1998). <https://doi.org/10.1007/BFb0053567>
 24. Hu, R., Yoshida, N.: Hybrid Session Verification Through Endpoint API Generation. In: Formal Aspects of Software Engineering. Lecture Notes in Computer Science, vol. 9633, pp. 401–418. Springer-Verlag (2016). https://doi.org/10.1007/978-3-662-49665-7_24
 25. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, vol. 5142, pp. 516–541. Springer-Verlag (2008). https://doi.org/10.1007/978-3-540-70592-5_22
 26. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In: European Conference on Object-Oriented Programming. vol. 222, pp. 4:1–4:29. Schloss Dagstuhl, Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPIcs.ECOOP.2022.4>
 27. Lange, J., Yoshida, N.: On the Undecidability of Asynchronous Session Subtyping. In: Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science, vol. 10203, pp. 441–457. Springer-Verlag (2017), https://link.springer.com/chapter/10.1007/978-3-662-54458-7_26
 28. Lindley, S., Morris, J.G.: Embedding session types in Haskell. In: International Symposium on Haskell. pp. 133–145. ACM Press (2016). <https://doi.org/10.1145/2976002.2976018>

29. Mostrous, D., Yoshida, N., Honda, K.: Global Principal Typing in Partially Commutative Asynchronous Sessions. In: European Symposium on Programming. Lecture Notes in Computer Science, vol. 5502, pp. 316–332. Springer-Verlag (2009). https://doi.org/10.1007/978-3-642-00590-9_23
30. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A Session Type Provider: Compile-time API Generation of Distributed Protocols with Interaction Refinements in F#. In: Compiler Construction. pp. 128–138. ACM Press (2018). <https://doi.org/10.1145/3178372.3179495>
31. Ng, N., Yoshida, N., Honda, K.: Multipart Session C: Safe Parallel Programming with Message Optimisation. In: Objects, Models, Components, Patterns. Lecture Notes in Computer Science, vol. 7304, pp. 202–218. Springer-Verlag (2012). https://doi.org/https://doi.org/10.1007/978-3-642-30561-0_15
32. Orchard, D., Yoshida, N.: Session Types with Linearity in Haskell. In: Behavioural Types: from Theory to Tools. pp. 219–241. River Publishers (2017). <https://doi.org/https://doi.org/10.13052/rp-9788793519817>
33. Padovani, L.: Fair subtyping for multi-party session types. In: Coordination Models and Languages. Lecture Notes in Computer Science, vol. 6721, pp. 127–141. Springer-Verlag (2011). https://doi.org/10.1007/978-3-642-21464-6_9
34. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing System. In: Parallel Architectures and Languages Europe. Lecture Notes in Computer Science, vol. 817, pp. 398–413. Springer-Verlag (1994). https://doi.org/10.1007/3-540-58184-7_118

A Supporting Lemmata and Proofs (for collecting simulation trees and graphs)

Proof (for proposition 1).

- Suppose $p \leq T \xrightarrow{?a} p' \leq T'$. If there exists $\pi \in S$ such that $p \leq \pi \xrightarrow{?b}$ for all $b \in A$ then $p \leq S \xrightarrow{?a}$. Otherwise

$$T' = \{\pi' \mid \pi \in T, p \leq \pi \xrightarrow{?a} p' \leq \pi'\} \subseteq \{\pi' \mid \pi \in S, p \leq \pi \xrightarrow{?a} p' \leq \pi'\} = S'$$

Observe $S' \neq \emptyset$ since $T' \neq \emptyset$ hence $p \leq S \xrightarrow{?a} p' \leq S'$.

- Suppose $p \leq T \xrightarrow{!a} p' \leq T'$. If for all $a \in A$ there exists $\pi \in S$ such that $p \leq \pi \xrightarrow{!a}$ then $p \leq S \xrightarrow{!a}$. Otherwise

$$T' = \{\pi' \mid \pi \in T, p \leq \pi \xrightarrow{!a} p' \leq \pi'\} \subseteq \{\pi' \mid \pi \in S, p \leq \pi \xrightarrow{!a} p' \leq \pi'\} = S'$$

Observe $S' \neq \emptyset$ since $T' \neq \emptyset$ hence $p \leq S \xrightarrow{!a} p' \leq S'$.

Lemma 1. *Let $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ be a collecting simulation tree for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$. If every complete branch in $(N, \xrightarrow{\ell}_t)$ is successful then there exists $\mathcal{R}_0, \dots, \mathcal{R}_i \subseteq P \times T_Q$ such that*

- if $b = n_0 \dots n_i$ a branch in the tree then $\mathcal{L}(n_j) = (p_j, \text{tr}(t_j))$ and $(p_j, t_j) \in \mathcal{R}_j$ for all $j \leq i$
- $\cup_{j=0}^{\infty} \mathcal{R}_j$ is an asynchronous subtyping relation for M_1 and M_2

Proof (for lemma 1). Let $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ be a collecting simulation tree for M_1 and M_2 . Proof by induction. Put $\mathcal{R}_0 = \{(p_0, q_0)\}$ and consider $(p_i, t_i) \in \mathcal{R}_i$. Then, by induction, there exists a branch $b = n_0 \dots n_i$ in the tree such that $\mathcal{L}(n_j) = (p_j, \text{tr}(t_j))$ and $(p_j, t_j) \in \mathcal{R}_j$ for all $j \leq i$.

- Suppose $n_i \not\hookrightarrow_t$. Since b is successful, it follows $\text{final}_{M_1}(p_i)$ and $\text{final}_{M_2}(q_i)$ where $t_i = q_i$. Thus \mathcal{R}_i is consistent with case (1) of definition 6.
- Suppose $n_i \hookrightarrow_t n_{i+1}$ where $\mathcal{L}(n_{i+1}) = (p_{i+1}, S_{i+1})$. Define $\mathcal{R}_{i+1} \subseteq P \times T_Q$ to be the least set constructed as follows:
 - Suppose RecvSet holds. Then $S_{i+1} = \{\pi' \mid \pi \in S_i, p \leq \pi \xrightarrow{!a} p' \leq \pi'\}$ for some $a \in A$.
 - * Suppose $t_i = q$ for some $q \in Q$. Then $S_i = \{q\}$ and Recv applies. Observe $\text{in}_{M_2}(q) \subseteq \text{in}_{M_1}(p_i)$ and let $q \xrightarrow{?a} q'$. Since $a \in \text{in}_{M_2}(q)$, $a \in \text{in}_{M_1}(p)$ hence $p_i \xrightarrow{?a} p'$ for some $p' \in P$. Observe $p_{i+1} = \delta_1(p_i, ?a) = p'$ and $S_{i+1} = \{q'\}$. Put $(p', q') \in \mathcal{R}_{i+1}$ to comply with case (2.1) of definition 6.

- * Suppose $t_i = \langle a_i : t'_i \mid i \in I \rangle$ for $a_i \in A$ and $t'_i \in T_Q$. Then $S_i = \text{tr}(t_i)$. Let $\omega \cdot q \in S_i$. Then $\omega \cdot q \in \text{tr}(t_i)$ hence $\omega \neq \epsilon$. Thus RecvTr applies. Let $j \in I$ and $\omega = a_j \cdot \pi \cdot q \in S_i$. Since RecvTr applies, it follows $a_j \in \text{in}_{M_1}(p_i)$. Hence $p_i \xrightarrow{?a_j} p'$ for some $p' \in P$. Then $p_{i+1} = \delta_1(p_i, ?a_j) = p'$. Observe $a_j \cdot \pi \cdot q \in S_i$ iff $\pi \cdot q \in S_{i+1}$ where $S_{i+1} \neq \emptyset$ and $S_{i+1} = \text{tr}(t'_j)$. Therefore put $(p', t'_j) \in \mathcal{R}_{i+1}$ to comply with case (2.2) of definition 6.
- Suppose SendSet holds. Then $S_{i+1} = \{\pi' \mid \pi \in S_i, p \leq \pi \xrightarrow{?a} p' \leq \pi'\}$ for some $a \in A$.
 - * Suppose $t_i = q$ for some $q \in Q$. Then $S_i = \{q\}$ and suppose Send applies. Then $\text{out}_{M_1}(p_i) \subseteq \text{out}_{M_2}(q)$ and $p_i \xrightarrow{!a} p'$. Since $a \in \text{out}_{M_2}(q)$ let $q \xrightarrow{!a} q'$. Observe $S_{i+1} = \{\delta(q, !a)\} = \{q'\}$. By the absence of mixed states, recall that if SendTr applies then likewise $S_{i+1} = \{q'\}$. Either way, put $(p', q') \in \mathcal{R}_{i+1}$ to comply with case (3.1) of definition 6.
 - * Suppose $\text{leaf}(t_i) = \{q_j \mid j \in J\}$ and SendTr applies. Then $S_i = \text{tr}(t_i)$. Let $j \in J$. There exists $\omega \cdot q_j \in \text{tr}(t_i) = S_i$. Because SendTr holds, $t_j^* = \text{inTree}_{M_2}(q_j) \neq \perp$ for $j \in J$. Define $\kappa = \{q_j \mapsto \theta(t_j^*) \mid j \in J\}$ where $\theta = \{q \mapsto q' \mid q \in Q, q \xrightarrow{!a} q'\}$. To show $S_{i+1} \subseteq \text{tr}(\kappa(t_i))$. Let $\pi \cdot q \in \text{tr}(t_i) = S_i$. Then $q \in \text{leaf}(t_i)$ hence $q = q_j$ for some $j \in J$. Let $\pi_k \cdot q_k \in \text{tr}(\text{inTree}_{M_2}(q)) = \text{tr}(\text{inTree}_{M_2}(q_j)) = \text{tr}(t_j^*)$. Then $\pi_k \cdot \theta(q_k) \in \text{tr}(\theta(t_j^*)) = \text{tr}(\kappa(q_j)) = \text{tr}(\kappa(q))$. Hence $\pi \cdot \pi_k \cdot \theta(q_k) \in \text{tr}(\kappa(t_i))$. To show $\text{tr}(\kappa(t_i)) \subseteq S_{i+1}$. Let $\pi \cdot \pi_k \cdot \theta(q_k) \in \text{tr}(\kappa(t_i))$ where $\pi \cdot q \in \text{tr}(t_i) = S_i$ and $\pi_k \cdot q_k \in \text{tr}(\text{inTree}_{M_2}(q))$. But $q \in \text{leaf}(t_i)$ hence $q = q_j$ for some $j \in J$. Therefore $\pi_k \cdot \theta(q_k) \in \text{tr}(\theta(\text{inTree}_{M_2}(q))) = \text{tr}(\theta(\text{inTree}_{M_2}(q_j))) = \text{tr}(\theta(t_j^*)) = \text{tr}(\kappa(q_j)) = \text{tr}(\kappa(q))$. Since SendSet applies and $\pi \cdot q \in S_i$ it follows $p \leq \pi \cdot q \xrightarrow{!a}$ thus $\pi \cdot \pi_k \cdot \theta(q_k) \in S_{i+1}$. Therefore $S_{i+1} = \text{tr}(\kappa(t_i))$. Let $q \in \text{leaf}(t_j^*)$ for some $j \in J$. Thus there exists $\pi \cdot q_j \in \text{tr}(t_i)$. Since $p \leq \pi \cdot q_j \xrightarrow{!a}$ holds $\text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q)$ therefore $q \in \text{dom}(\theta)$ thus $\text{leaf}(t_j^*) \subseteq \text{dom}(\theta)$. Finally put $(p', \kappa(t_i)) \in \mathcal{R}_{i+1}$ to comply with case (3.2) of definition 6.

By case (3) of definition 11 only the above rules need to be considered. This completes the construction of \mathcal{R}_{i+1} .

It follows that $\cup_{j=0}^{\infty} \mathcal{R}_j$ is an asynchronous subtyping relation for M_1 and M_2 .

Lemma 2. *Let $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ be a collecting simulation tree for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$. If there exists an asynchronous subtyping relation \mathcal{R} for M_1 and M_2 with $(p_0, q_0) \in \mathcal{R}$ and $b = n_0 \cdots n_i$ is a branch in the tree then:*

- $\mathcal{L}(n_j) = (p_j, \text{tr}(t_j))$ and $(p_j, t_j) \in \mathcal{R}$ for all $j \leq i$
- either $n_i \xrightarrow{\ell}_t n_{i+1}$ where $\mathcal{L}(n_{i+1}) = (p_{i+1}, \text{tr}(t_{i+1}))$ or $t_i = q$ for some $q \in Q$ where $\text{final}_{M_1}(p_i)$ and $\text{final}_{M_2}(q)$

Proof (for lemma 2). Suppose $\mathcal{R} \subseteq P \times T_Q$ is an asynchronous subtyping relation such that $(p_0, q_0) \in \mathcal{R}$. Let $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ be a collecting simulation tree for M_1 and M_2 and let $b = n_0 \cdots n_i$ be a branch in the tree. By induction, suppose $\mathcal{L}(n_j) = (p_j, \text{tr}(t_j))$ and $(p_j, t_j) \in \mathcal{R}$ for all $j \leq i$. Now consider $(p_i, t_i) \in \mathcal{R}$:

- Suppose $\text{final}_{M_1}(p_i)$. By case (1) of definition 6 there exists $q \in Q$ such that $t_i = q$ and $\text{final}_{M_2}(q)$. Thus b is successful.
- Suppose $\text{recv}_{M_1}(p_i)$ and $t_i = q$ for some $q \in Q$. Let $q \xrightarrow{?a} q'$. By case (2.1) of definition 6 it follows $p_i \xrightarrow{?a} p'$ and $(q', p') \in \mathcal{R}$. Thus $\text{in}_{M_2}(q) \subseteq \text{in}_{M_1}(p)$. By the **Recv** rule, it follows $p_i \leq \{q\} \xrightarrow{?a} \delta_1(p_i, ?a) \leq \{q'\}$ where $\delta_1(p_i, ?a) = p'$. By **RecvSet**, $p_i \leq \{q\} \xrightarrow{?a} p' \leq \{q'\}$ hence $\mathcal{L}(n_{i+1}) = (p', \{q'\})$ and $n_i \xrightarrow{?a} n_{i+1}$ for some $n_{i+1} \in N$.
- Suppose $\text{recv}_{M_1}(p_i)$ and $t_i = \langle a_i : t'_i \mid i \in I \rangle$. Let $k \in I$. By case (2.2) of definition 6, $p_i \xrightarrow{?a_k} p'$ and $(p', t'_i) \in \mathcal{R}$. Thus $a_k \in \text{in}_{M_1}(p_i)$. Let $a_k \cdot \pi \in S_i$. By the **RecvTr** rule, it follows $p_i \leq a_k \cdot \pi \xrightarrow{?a_k} \delta_1(p_i, ?a_k) \leq \pi$ where $\delta_1(p_i, ?a_k) = p'$. By **RecvSet**, $\pi \in S_{i+1}$. Conversely, also by **RecvSet**, $a_k \cdot \pi \in S_i$ if $\pi \in S_{i+1}$. Therefore $S_{i+1} = \text{tr}(t'_i)$. Thus $\mathcal{L}(n_{i+1}) = (p', \text{tr}(t'_i))$ and $n_i \xrightarrow{?a_k} n_{i+1}$ for some $n_{i+1} \in N$.
- Suppose $\text{send}_{M_1}(p_i)$ and $t_i = q$ for some $q \in Q$. Let $p_i \xrightarrow{!a} p'$. By case (3.1) of definition 6, $q \xrightarrow{!a} q'$ and $(p', q') \in \mathcal{R}$. Thus $\text{out}_{M_1}(p_i) \subseteq \text{out}_{M_2}(q)$. By the **Send** rule, $p_i \leq \{q\} \xrightarrow{!a} p' \leq \{\delta_2(q, !a)\}$ where $\delta_2(q, !a) = q'$. By **SendSet** $p_i \leq \{q\} \xrightarrow{!a} p' \leq \{q'\}$ hence $\mathcal{L}(n_{i+1}) = (p', \{q'\})$ and $n_i \xrightarrow{!a} n_{i+1}$ for some $n_{i+1} \in N$.
- Suppose $\text{send}_{M_1}(p_i)$ and $\text{leaf}(t_i) = \{q_j \mid j \in J\}$. Let $p_i \xrightarrow{!a} p'$. Put $\theta = \{q \mapsto q' \mid q \in Q, q \xrightarrow{!a} q'\}$. By case (3.2) of definition 6, it follows $\neg\text{cycle}_{M_1}(!, p)$, $t_j^* = \text{inTree}_{M_2}(q_j)$ for all $j \in J$, $\text{leaf}(t_j^*) \subseteq \text{dom}(\theta)$ for all $j \in J$ and $(p', \kappa(t_i)) \in \mathcal{R}$ where $\kappa = \{q_j \mapsto \theta(t_j^*) \mid j \in J\}$. Let $\pi \cdot q \in S_i = \text{tr}(t_i)$. Then $q = q_j$ for some $j \in J$. Let $q^* \in \text{leaf}(t_j^*)$. Then $q^* \in \text{dom}(\theta)$ hence $q^* \xrightarrow{!a} q'$ for some $q' \in Q$ thus $a \in \text{out}_{M_2}(q^*)$. Thus $\text{out}_{M_1}(p_i) \subseteq \text{out}_{M_2}(q^*)$. Therefore rule **SendTr** applies and $p_i \leq \pi \cdot q \xrightarrow{!a} p' \leq \pi \cdot \pi^* \cdot q'$ where $\pi^* \cdot q^* \in \text{tr}(t_j^*)$. Repeating the argument in lemma 1 it follows $S_{i+1} = \text{tr}(\kappa(t_i))$. Thus $\mathcal{L}(n_{i+1}) = (p', \text{tr}(\kappa(t_i)))$ and $n_i \xrightarrow{!a} n_{i+1}$ for some $n_{i+1} \in N$.

Proof (for theorem 1). Suppose that every complete branch in $(N, \xrightarrow{\ell}_t)$ is successful. Then, by lemma 1, there exists an asynchronous subtyping relation \mathcal{R} hence $M_1 \leq M_2$. Now suppose $M_1 \leq M_2$ hence there exists an asynchronous subtyping relation, thus by lemma 2, every complete branch in $(N, \xrightarrow{\ell}_t)$ is successful.

Proof (for corollary 1). Proof by induction.

- Suppose $b = n_0$. Then $\mathcal{L}(n_0) = (p_0, \{q_0\})$. Put $b' = n'_0$ where $\mathcal{L}'(n'_0) = (p_0, \{q_0\})$. The result follows.
- Suppose $b = n_0 \cdots n_m$ for $m < i$. By induction either:
 - There exists $b' = n'_0 \cdots n'_m$ such that $\mathcal{L}(n_j) = (p_j, S_j)$, $\mathcal{L}'(n'_j) = (p_j, S'_j)$ and $S_j \subseteq S'_j$ for all $j \leq m$. Now consider $n_m \xrightarrow{\ell}_t n_{m+1}$ where $\mathcal{L}(n_{m+1}) = (p_{m+1}, S_{m+1})$. By proposition 1 and case (3) of definition 13, either $n'_m / \xrightarrow{\ell}_g$, and the result follows immediately, or $(p_m \leq S'_m) \xrightarrow{\ell} (p_{m+1} \leq R)$ where $S_{m+1} \subseteq R$. Then $n'_m \xrightarrow{\ell}_g n'_{m+1}$ where $\mathcal{L}'(n'_{m+1}) = (p_{m+1}, S'_{m+1})$ and $S_{m+1} \subseteq R \subseteq S'_{m+1}$.
 - There exists $b' = n'_0 \cdots n'_k$ for $k < i$ and $n'_k \not\xrightarrow{\ell}_g$ such that $\mathcal{L}(n_j) = (p, S)$, $\mathcal{L}'(n'_j) = (p, T)$ and $S \subseteq T$ for all $j \leq k$. The result is immediate.

Proof (for theorem 2). Let $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ be a collecting simulation tree for M_1 and M_2 . Let $b = n_0 \cdots n_i$ be a branch in the tree $(N, \xrightarrow{\ell}_t)$. By corollary 1 there exists $b' = n'_0 \cdots n'_k$ in $(N', \xrightarrow{\ell}_g)$ where $k \leq i$ such that $\mathcal{L}(n_j) = (p, S)$, $\mathcal{L}'(n'_j) = (p, T)$ and $S \subseteq T$ for all $j \leq k$. For the sake of a contradiction, suppose b' is complete and $k < i$. Then $n'_k \xrightarrow{\ell}_g$. Since b' is successful, $\mathcal{L}'(n'_k) = (p, F)$ where $\text{final}_{M_1}(p)$ and $\text{final}_{M_2}(q)$ for all $q \in F$. By rules **SendSet** and **RecvSet**, it follows $\mathcal{L}(n_k) = (p, \{q\})$ for some $q \in F$. But $n_k \xrightarrow{\ell}_t n_{k+1}$ which is a contradiction, hence $k = i$.

Now suppose $n_i \xrightarrow{\ell}_t$ where $\mathcal{L}(n_i) = (p, S)$ and $\mathcal{L}'(n'_i) = (p, S')$. Then $S \neq \emptyset$ and $S \subseteq S'$. Because **RecvSet** is not applicable, there exists $\pi \in S$ such that $p \leq \pi \not\xrightarrow{?b}$ for all $b \in A$. But $\pi \in S'$. Because **SendSet** is not applicable, for all $a \in A$ there exists $\pi \in S$ such that $p \leq \pi \not\xrightarrow{!a}$. But $\pi \in S'$. Therefore $n'_i \xrightarrow{\ell}_g$. Since b' is successful, $\mathcal{L}'(n'_i) = (p, F)$ where $\text{final}_{M_1}(p)$ and $\text{final}_{M_2}(q)$ for all $q \in F$. It follows $\mathcal{L}(n_i) = (p, \{q\})$ for some $q \in F$ hence b is successful. It follows by theorem 1 that $M_1 \leq M_2$.

B Auxiliary relaxation for widening regular expressions

For completeness, we give the auxiliary too, but refer to [12] for a commentary:

$$\text{mash}_k(r, s) = \begin{cases} r^* & \text{if } s = \epsilon \text{ and } \text{sh}(r^*) \leq k \\ s^* & \text{if } r = \epsilon \text{ and } \text{sh}(s^*) \leq k \\ \text{widen}_{k-1}((\epsilon, r'), (\epsilon, s'))^* & \text{if } r = r'^*, s = s'^* \text{ and } k \geq 2 \\ \text{widen}_{k-1}((\epsilon, r'), (\epsilon, s))^* & \text{if } r = r'^*, s \neq s'^* \text{ and } k \geq 2 \\ \text{widen}_{k-1}((\epsilon, r), (\epsilon, s'))^* & \text{if } r \neq r'^*, s = s'^* \text{ and } k \geq 2 \\ \{a \mid a \text{ appears in } r \text{ or } s\}^* & \text{otherwise} \end{cases}$$

The auxiliary does little more than call widen_{k-1} recursively if there is sufficient star height, and otherwise resorts to a C^* -style regular expression.

C Supporting Lemmata and Proofs (for chain length)

To derive a bound on chain length for regular strings, we introduce an order relation on regular strings, which is itself defined in terms of a structural congruence relation on regular strings:

$$\begin{array}{c}
\frac{}{w \equiv w} [\text{Sym}] \quad \frac{}{w_1 \cdot (w_2 \cdot w_3) \equiv (w_1 \cdot w_2) \cdot w_3} [\text{Assoc}] \\
\frac{w_1 \equiv w_2}{w[w_1] \equiv w[w_2]} [\text{Hole}] \quad \frac{w_1 \equiv w_2}{w_2 \equiv w_1} [\text{Swap}] \quad \frac{w_1 \equiv w_2}{w_1 \leq_0 w_2} [\text{Equiv}] \\
\frac{w_1 \leq_{\delta_1} w_2 \quad w_2 \leq_{\delta_2} w_3}{w_1 \leq_{\max(\delta_1, \delta_2)} w_3} [\text{Trans}] \quad \frac{w_1 \leq_{\delta_1} w'_1 \quad w_2 \leq_{\delta_2} w'_2}{w_1 \cdot w_2 \leq_{\max(\delta_1, \delta_2)} w'_1 \cdot w'_2} [\text{Concat}] \\
\frac{}{\epsilon \leq_1 w^*} [\text{Empty}] \quad \frac{}{w \leq_1 w^*} [\text{Star}] \quad \frac{w_1 \leq_{\delta} w_2}{(w_1)^* \leq_{\delta} (w_2)^*} [\text{Lift}] \\
\frac{C_1 \subseteq C_2}{C_1 \leq_{\min(|C_2| - |C_1|, 1)} C_2} [\text{Char}] \quad \frac{C = \text{occ}(w)}{w \leq_1 C^*} [\text{Mash}]
\end{array}$$

Fig. 7. Structural congruence and ordering rules on Reg_A

Definition 15. *The structural congruence relation $\equiv \subseteq \text{Reg}_A \times \text{Reg}_A$ is the least binary relation satisfying the rules of Figure 7, where $w_1 \equiv w_2$ abbreviates the predicate $(w_1, w_2) \in \equiv$.*

In the rule **Mash**, $\text{occ}(w)$ denotes the alphabet symbols of A which occur in w .

Definition 16. *The ordering relation $\leq \subseteq \text{Reg}_A \times \mathbb{N}_0 \times \text{Reg}_A$ is the least 3-place relation satisfying the rules of Figure 7 where $w_1 \leq_d w_2$ abbreviates the predicate $(w_1, d, w_2) \in \leq$.*

Proposition 2. *If $w_1 \leq_d w_2$ then $\llbracket w_1 \rrbracket \subseteq \llbracket w_2 \rrbracket$*

Proof. Observe $\llbracket w_1 \rrbracket = \llbracket w_2 \rrbracket$ if $w_1 \equiv w_2$. Now consider the rules of \leq in turn:

- Consider **Equiv**: If $w_1 \equiv w_2$ then $\llbracket w_1 \rrbracket \subseteq \llbracket w_2 \rrbracket$.
- Consider **Trans**: If $w_1 \leq_{\delta_1 + \delta_2} w_3$ then $w_1 \leq_{\delta_1} w_2$ and $w_2 \leq_{\delta_2} w_3$. Hence, by induction, $\llbracket w_1 \rrbracket \subseteq \llbracket w_2 \rrbracket$ and $\llbracket w_2 \rrbracket \subseteq \llbracket w_3 \rrbracket$ therefore $\llbracket w_1 \rrbracket \subseteq \llbracket w_3 \rrbracket$.
- Consider **Concat**: If $w_1 \cdot w_2 \leq_{\max(\delta_1, \delta_2)} w'_1 \cdot w'_2$ then $w_1 \leq_{\delta_1} w'_1$ and $w_2 \leq_{\delta_2} w'_2$. Hence, by induction, $\llbracket w_1 \rrbracket \subseteq \llbracket w'_1 \rrbracket$ and $\llbracket w_2 \rrbracket \subseteq \llbracket w'_2 \rrbracket$ thus $\llbracket w_1 \cdot w_2 \rrbracket \subseteq \llbracket w'_1 \cdot w'_2 \rrbracket$.
- Consider **Empty**: Observe $\epsilon \in \llbracket w^* \rrbracket$.
- Consider **Star**: Observe $w \in \llbracket w^* \rrbracket$ hence $\llbracket w \rrbracket \subseteq \llbracket w^* \rrbracket$.
- Consider **Lift**: If $(w_1)^* \leq_{\delta} (w_2)^*$ then $w_1 \leq_{\delta} w_2$. By induction $\llbracket w_1 \rrbracket \subseteq \llbracket w_2 \rrbracket$ therefore $\llbracket (w_1)^* \rrbracket \subseteq \llbracket (w_2)^* \rrbracket$.
- Consider **Char**: If $C_1 \leq_{\delta} C_2$ then $C_1 \subseteq C_2$ then $\llbracket C_1 \rrbracket \subseteq \llbracket C_2 \rrbracket$.
- Consider **Mash**: Observe $w \in \llbracket C^* \rrbracket$ because $C = \text{occ}(w)$ thus $\llbracket w \rrbracket \subseteq \llbracket C^* \rrbracket$.

Definition 17.

- $w_1 \leq w_2$ iff $w_1 \leq_d w_2$ for some $d > 0$
- $w_1 < w_2$ iff $w_1 \not\equiv w_2$ and $w_1 \leq w_2$

As stated in [12], albeit formulated with the reverse ordering, widening preserves \leq in the following sense:

Proposition 3.

- $w_1 \leq \text{mash}_k(w_1, w_2)$ for all $w_1, w_2 \in \text{Reg}_A$
- $w_1 \leq w_1 \nabla_k w_2$ for all $w_1, w_2 \in \text{Reg}_A$

Proof.

- Case 1 of $\text{mash}_k(r, s)$: Observe $r \leq r^*$ by the **Star** rule.
- Case 2 of $\text{mash}_k(r, s)$: Because $r = \epsilon$ it follows $r \leq s^*$ by the **Empty** rule.
- Case 3 of $\text{mash}_k(r, s)$: By induction $r' \leq r' \nabla_k s'$ hence $(r')^* \leq (r' \nabla_k s')^*$ by the **Lift** rule.
- Case 4 of $\text{mash}_k(r, s)$: By induction $r' \leq r' \nabla_k s$ hence $(r')^* \leq (r' \nabla_k s)^*$ by the **Lift** rule.
- Case 5 of $\text{mash}_k(r, s)$: By induction $r \leq r \nabla_k s'$ hence $r^* \leq (r \nabla_k s')^*$ by the **Lift** rule. But $r \leq r^*$ by **Star** hence $r \leq (r \nabla_k s')^*$ by the **Trans** rule.
- Case 6 of $\text{mash}_k(r, s)$: By the **Mash**, $r \leq \text{occ}(r)^*$ and by **Char** and **Lift** it follows $\text{occ}(r)^* \leq (\text{occ}(r) \cup \text{occ}(s))^*$ therefore $r \leq (\text{occ}(r) \cup \text{occ}(s))^*$ by **Trans**.
- Case 1 of $(p \cdot q \cdot s) \nabla_k (p' \cdot q' \cdot s')$: Then $q = q'$. By induction $s \leq s \nabla_k s'$. By **Equiv** $q \leq q$ hence by **Trans** $q \cdot s \leq q \cdot (s \nabla_k s')$. By induction $p \leq \text{mash}_k(p, p')$ thus by **Trans** it follows $p \cdot q \cdot s \leq \text{mash}_k(p, p') \cdot q \cdot (s \nabla_k s')$.
- Case 2 of $(p \cdot q \cdot s) \nabla_k (p' \cdot q' \cdot s')$: Observe $p \cdot (q \cdot s) \equiv (p \cdot q) \cdot s$ and $p' \cdot (q' \cdot s') \equiv (p' \cdot q') \cdot s'$ hence the result holds by induction.
- Case 3 of $(p \cdot q \cdot s) \nabla_k (p' \cdot q' \cdot s')$: Similar to the previous case.
- Case 4 of $(p \cdot q \cdot s) \nabla_k (p' \cdot q' \cdot s')$: Similar to the previous case.

The consequence of the above result is $\llbracket w_1 \rrbracket \subseteq \llbracket w_1 \nabla_k w_2 \rrbracket$ and likewise it follows $\llbracket w_1 \rrbracket \subseteq \llbracket \text{mash}_k(w_1, w_2) \rrbracket$. Although ∇_k relaxes a string, its star length does not increase: $|w_1 \nabla_k w_2| \leq |w_1|$ [12, Theorem 10].

Although it is possible to construct strictly ascending chains in Reg_A which are arbitrarily long, Reg_A does not contain infinite chains which are strictly ascending. The proposition asserts that this is not a paradox. To state the result succinctly a generalised notation of star length on the words of Reg_A is introduced:

Definition 18. $\dim^* : \text{Reg}_A \rightarrow \mathbb{N}_0$ is defined:

$$\begin{aligned} \dim^*(\epsilon) &= 0 \\ \dim^*(C) &= 0 \\ \dim^*(w^*) &= \dim^*(w) \\ \dim^*(w_1 \cdot w_2) &= \max\{\dim^*(w_1), \dim^*(w_2), 2(|w_1| + |w_2|) + 1\} \end{aligned}$$

where $|w|$ denotes star length

Proposition 4. *Let $w_1 < w_2 < \dots \in \text{Reg}_A$ be a strictly ascending chain (which is not necessarily finite). If $\text{sh}(w_i) \leq d$ for all $i \geq 1$ then:*

- *there exists a bound $b \geq 0$ such that $\dim^*(w_i) \leq b$ for all $i \geq 1$*
- *$|\{w_i \mid i \geq 1\}| \leq l(A, b, d)$ where $l(A, b, d) = b^d|A| + \sum_{i=1}^d b^i$*

Proof (for proposition 4). Proof by induction.

- Suppose $d = 0$. Put $b = 0$. Then $\dim^*(w_i) \leq b$ for all $i \geq 1$. Consider $w_i = C_i$ where $C_i \subseteq C_{i+1}$ and $C_i \neq C_{i+1}$ for all $1 \leq i < n$. Then $n \leq |A| = 0^0|A| + \sum_{i=1}^d b^i = l(A, b, d)$ where $0^0 = 1$.
- Suppose $d > 0$. Let $w_1 \equiv w_{1,1} \cdot \dots \cdot w_{1,m}$ where $|w_{1,j}| \leq 1$ for all $1 \leq j \leq m$. Let $j \in \{1, \dots, m\}$. Consider $w_{1,j} \leq w_{2,j} \leq \dots$ where $w_i \equiv w_i^p \cdot w_{i,j} \cdot w_i^s$ for all $1 \leq i$. Thus $w_1^p = w_{1,1} \cdot \dots \cdot w_{1,j-1}$ and $w_1^s = w_{1,j+1} \cdot \dots \cdot w_{1,m}$. Suppose $w_{i,j} = \epsilon$ for all $1 \leq i \leq \ell$ and $w_{i,j} = (\omega_{i-m})^*$ for all $\ell < i$. It follows $\omega_1 \leq \omega_2 \leq \dots$ is an increasing chain which is not strictly increasing. Put $\Omega = \{\omega_i \mid i > 1\}$ and construct the strictly increasing chain $\omega'_1 < \omega'_2 < \dots$ from the set Ω . Observe $\text{sh}(\omega'_i) \leq d - 1$ for all $i \geq 1$. By induction there exists $b_j \geq 0$ such that $\dim^*(\omega'_i) \leq b_j$ for all $i \geq 0$. Moreover $|\Omega| \leq l(A, b_j, d - 1)$. Let $B = \{b_1, \dots, b_m\}$ denote the set all such bounds. Put $b = \max(\{2|w_1| + 1\} \cup B)$ and observe $m \leq 2|w_1| + 1 \leq b$. Observe $\dim^*(w_i) \leq b$ for all $i \geq 1$. Since $l(A, b, 0) = l(A, 0, 0)$ it follows

$$\begin{aligned}
 |\{w_i \mid i \geq 1\}| &\leq \sum_{j=1}^m (1 + l(A, b_j, d - 1)) \\
 &\leq \sum_{j=1}^m (1 + l(A, b, d - 1)) \\
 &< b(1 + l(A, b, d - 1)) \\
 &= b(1 + b^{d-1}|A| + \sum_{i=1}^{d-1} b^i) \\
 &= b^d|A| + b + b \sum_{i=1}^{d-1} b^i \\
 &= l(A, b, d)
 \end{aligned}$$

The force of the proposition is revealed by hypothesising that there exists a chain $w_1 < w_2 < \dots$ which is infinite. The result asserts the existence of a bound $b \geq 0$ such that $\dim^*(w_i) \leq b$ for all $i \geq 0$. With the bound on $\dim^*(w_i)$, fixed d and fixed A , $l(A, b, d)$ then bounds the number of distinct w_i , implying the hypothesis is false. As well as asserting finiteness, it also bounds chain length, albeit in terms of an unknown b .

However b itself can be bounded by studying how the widening is deployed in our setting. Observe that the number of input actions in any branch of an input tree is bounded by $|Q|$, and likewise for any trace of Tr_Q . Thus, in the above argument, $|w_1| \leq |Q| - 1$ and $2|w_1| + 1 \leq 2|Q| - 1$. Repeating this argument, $b \leq 2|Q| - 1$. Without loss of generality, suppose $|A| \leq 2|Q|$ then:

Proposition 5. *If $b \leq 2|Q| - 1$ then $l(A, b, d) < (2|Q|)^{d+2}$*

Note that, to our knowledge, this is the first statement that on the complexity of a sound, asynchronous subtyping algorithm (previous work [6, 27] has presented an algorithm but not a complexity bound).