Complexity Analysis of Multiparty Session Types: Top Down and Bottom Up Procedures (Extended Version)

Trinity Term 2024

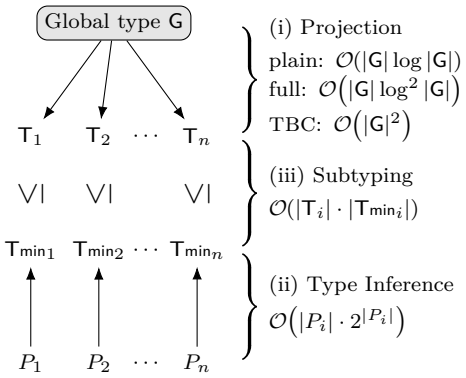
Third Year Project Report

MCompSci Computer Science

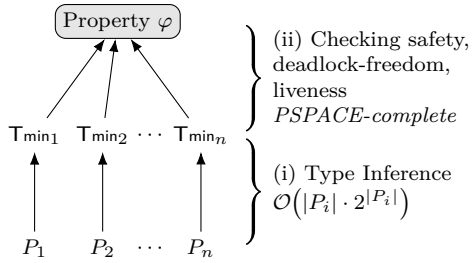
Candidate Number: 1066593

Abstract

Multiparty session types (MPST) is a type discipline for ensuring type and communication safety, deadlock-freedom and liveness for multiple concurrently running participants. This report gives complexity analyses of the two main procedures of MPST. The first procedure is *top-down*, where a programmer specifies a whole view of communications as a *global type*, and each distributed process is locally type-checked against its *projection*. The second is *bottom-up*, where a desired property φ of a set of participants is ensured if the same property φ is true for its typing context. To compare their complexity, (1) we measure complexity of three projection algorithms from the literature [25, 21]; (2) we propose a new subtyping algorithm which is more efficient (quadratic) than the inductive algorithm in [12] (exponential); (3) we develop a novel minimum type inference system from MPST processes; and (4) we measure complexity for checking a typing context property by reducing it to a quantified Boolean formula (QBF) problem. Our analyses reveal that the top-down approach is more efficient (quadratic) than the bottom-up approach (PSPACE-complete); and type inference has exponential cost against the size of a process.



(a) Top-down procedure.
(i) Projection of global types (Section 3);
(ii) Type inference (Section 4);
(iii) Subtype-checking (Section 2).



(b) Bottom-up procedure.
(i) Checking of desirable properties (safety, deadlock-freedom and liveness) (Section 5);
(ii) Type inference (Section 4).

Figure 1: Overview of the top-down and bottom-up procedures.

1 Introduction

Multiparty Session Types [14] is a type discipline for ensuring desirable properties such as safety, deadlock-freedom and liveness for multiple interacting participants. Historically, this is done with a *top-down* approach. First, a *global type* is constructed by a system designer, which specifies the allowed pattern of communications of the interacting processes. It is then *projected* onto a collection of *local types*, and each participant is type-checked locally and independly against its global type.

Scalas and Yoshida [18] proposed an alternative approach for typing MPST, which we call *bottom-up*. It does not use global types; instead, the properties of processes are ensured by checking the properties of *typing contexts* (a collection of local types).

This report gives complexity analyses of the *top-down* and *bottom-up* approaches. While the top-down approach is widely used and implemented in many programming languages and tools [24], the bottom-up approach is more expressive and can type more processes [18]; however, it is believed that the top-down approach is more tractable. Our motivation is to formalise this notion and compare the complexities of the approaches.

Figure 1a gives an overview of the top-down approach with the complexity results to be proved in this report. The procedure starts with a *global type* (Figure 4) that specifies the pattern of communications of participants. For example, the following global type G is

describing the behaviour: first, *participant* p sends to q either *label* l_1 or l_2, then in both branches q sends r an $\mathtt{int}$, and this repeats indefinitely. The construct $\mu\mathbf{t}\ldots\mathbf{t}$, where $\mathbf{t}$ is drawn from a countably infinite set of *recursion variables*, represents a tail recursion: once $\mathbf{t}$ is reached, the type repeats at the point where $\mathbf{t}$ was bound.

$$\mathsf{G} = \mu\mathbf{t}.\mathsf{p} \to \mathsf{q} : \{l_1 : \mathsf{q} \to \mathsf{r} : [\mathtt{int}];\ \mathbf{t}, l_2 : \mathsf{q} \to \mathsf{r} : [\mathtt{int}];\ \mathbf{t}\}$$

Projection. The first step is to *project* G onto a collection of *local types* $\{\mathsf{T}_i\}_{i\in I}$. Local types specify the communication pattern of a *single* participant, and are used to locally type-check processes.

For example, a valid projection (Definition 3.2) of G onto p is the local type (Figure 2) $\mu\mathbf{t}.\mathsf{q}\oplus\{l_1 : \mathbf{t}, l_2 : \mathbf{t}\}$, meaning that p selects either l_1 or l_2 and sends it to q.

Projection for MPST was first introduced by Honda *et al.* [14], using the *plain merge* (Definition 3.3). Later, Bettini *et al.* [4, 7] proposed simplified global types, which was adopted by most works, including ours. Several works and tools [12, 24, 18, 27, 3] use the *full merge* (Definition 3.4) to enlarge the set of projectable of global types. Tirore *et al.* [21] defined a projection on inductive types that is sound and complete with respect to the coinductive projection with plain merging on coinductive types. They mechanised their algorithm and proved correctness in Coq.

We calculate the complexity of the above kinds of projections, each with different levels of expressivity and complexity, and give an analysis of their complexities (Theorems 3.7, 3.11, 3.13).

Minimum Type Inference. Vasconcelos and Honda [23] introduce a principal typing scheme for the polyadic π-calculus, where types are simple tuples of types.

We propose a different inference algorithm for MPST, where the *minimum type* (Definition 4.2) is inferred from a given process P. The minimum type $\mathsf{T}_{\mathsf{min}}$ of P characterises, in a succinct way, the tightest behaviour of P. In Section 4, we give a sound and complete algorithm for finding a minimum type. This inference takes exponential time against the

size of a process (Theorem 4.13). Once a participant is typed by each local type, the composition of processes becomes type-safe, deadlock-free and live [25, 15, 12], and satisfies a subject reduction theorem [12, Theorem 3.21].

Subtyping. The final step is to check that the inferred minimum type is a *subtype* of the projected local type, completing the top-down procedure. A type T_1 is a subtype of T_2 if a process of type T_1 can be safely used in place of T_2.

For subtype-checking, we consider two algorithms: one from [12] based on inductive proof rules, which we prove is exponential in the worst case (Theorem 2.8), and the other extended from [22] which uses a search in the product graph of types. We show that this new subtyping algorithm has quadratic complexity (Theorem 2.10).

Property-checking. Scalas and Yoshida [18] propose a general typing system, which we call the *bottom-up* approach (drawn in Figure 1b), in which a collection of the local types inferred from processes are checked directly whether they satisfy *safety* (Definition 5.2), *deadlock-freedom* (Definition 5.3) and *liveness* (Definition 5.5), to verify that the same properties hold for processes. Figure 1b summarises the results to be proven in this report. We prove in Section 5 that checking these properties of types is PSPACE-complete (Theorems 5.12, 5.14, 5.18).

Section 6 concludes. Although this report is self-contained, many proofs are omitted due to space constraints. An extended appendix with full definitions and proofs, along with an implementation of subtyping algorithms, can be found at `https://github.com/anonymous-1066593/project_appendix`.

2 Complexity Analysis of Multiparty Session Subtyping

We use the synchronous multiparty subtyping relation (denoted by $\mathsf{T} \leq \mathsf{T}'$) following [9, 12, 6] (called *process subtyping* in [11]): this means that a process of type T can be used safely in place of a process of type T'. Formally, Ghilezan *et al.* [12] prove operational and denotational *preciseness* of this subtyping relation.

$$
\begin{array}{lll}
S, S', \cdots ::= & \mathtt{int}, \mathtt{nat}, \mathtt{bool} & \\
\mathsf{T}, \mathsf{T}', \cdots ::= & \mathtt{end} & \text{(inaction)} \\
\mid & \mathsf{p}![S]; \mathsf{T} & \text{(output)} \\
\mid & \mathsf{p}?[S]; \mathsf{T} & \text{(input)} \\
\mid & \mathsf{p} \oplus \{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n\} & \text{(selection)} \\
\mid & \mathsf{p} \& \{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n\} & \text{(branching)} \\
\mid & \mu \mathbf{t}.\mathsf{T} & \text{(recursive type)} \\
\mid & \mathbf{t} & \text{(type variable)}
\end{array}
$$

Figure 2: Local sorts and types.

First, Figure 2 defines the syntax of local types according to Section 1. We will give an equivalent definition of subtyping (similarly to [19, 22]), based on *type simulations* on *type graphs*, which will be more useful when we represent types syntactically as labelled transition systems where each node is a subformula of the type.

Definition 2.1 (Type graph)**.** Define *actions*:

$$
\ell, \ell', \cdots ::= \ \mathsf{p}?[S] \mid \mathsf{p}![S] \mid \mathsf{p} \& l_j \mid \mathsf{p} \oplus l_j \mid \mathsf{end}
$$

We define the transition relation $\mathsf{T} \xrightarrow{\ell} \mathsf{T}'$ where ℓ is an action and T' is extended with the additional type Skip as follows:

$$
\frac{\mathrm{unfold}(\mathsf{T}) = \mathtt{end}}{\mathsf{T} \xrightarrow{\mathsf{end}} \mathsf{Skip}} \qquad \frac{\mathrm{unfold}(\mathsf{T}) = \mathsf{p} \dagger [S]; \mathsf{T}' \ \dagger \in \{!, ?\}}{\mathsf{T} \xrightarrow{\mathsf{p} \dagger [S]} \mathsf{T}'} \qquad \frac{\mathrm{unfold}(\mathsf{T}) = \mathsf{p} \dagger \{l_i : \mathsf{T}_i\}_{i \in I} \ \dagger \in \{\oplus, \&\} \ \ j \in I}{\mathsf{T} \xrightarrow{\mathsf{p} \dagger l_j} \mathsf{T}_j}
$$

where the *unfold* function is defined by: $\mathrm{unfold}\,(\mu \mathbf{t}.\mathsf{T}) = \mathrm{unfold}\,(\mathsf{T}[\mu \mathbf{t}.\mathsf{T}/\mathbf{t}])$, and $\mathrm{unfold}\,(\mathsf{T}) = \mathsf{T}$ otherwise. The type graph is a directed connected graph whose edges are local types or Skip, and an edge is given by $\mathsf{T} \xrightarrow{\ell} \mathsf{T}'$ (from T to T' labelled by ℓ). The type graph for T, denoted by $\mathbb{G}(\mathsf{T})$, is the graph reachable by the above transitions from T. We often write T for $\mathbb{G}(\mathsf{T})$ when it is clear from the context.

Definition 2.2 (Type simulations)**.** A *type simulation* $\mathcal{R}$ is a relation on type graphs such that $\mathsf{T}_1 \mathcal{R} \mathsf{T}_2$ implies:

- If $\mathsf{T}_1 \xrightarrow{\ell} \mathsf{T}_1'$ then $\mathsf{T}_2 \xrightarrow{\ell} \mathsf{T}_2'$, and $\mathsf{T}_1' \mathcal{R} \mathsf{T}_2'$, for $\ell \in \{\mathsf{p} \oplus l, \mathtt{end}, \mathsf{p}?[S], \mathsf{p}![S]\}$.

- If $\mathsf{T}_2 \xrightarrow{\ell} \mathsf{T}_2'$ then $\mathsf{T}_1 \xrightarrow{\ell} \mathsf{T}_1'$, and $\mathsf{T}_1' \mathcal{R} \mathsf{T}_2'$, for $\ell \in \{\mathsf{p}\&l, \mathtt{end}\}$.

Lemma 2.3 (Subtyping as a type simulation)**.** $\mathsf{T}_1 \leq \mathsf{T}_2$ if and only if there exists a type simulation $\mathcal{R}$ such that $\mathsf{T}_1 \mathcal{R} \mathsf{T}_2$.

Definition 2.4 (Size of Local Types)**.** The size of a local type T, denoted $|\mathsf{T}|$, is defined as: $|\mathtt{end}| = |\mathbf{t}| = 1$, $|\mu \mathbf{t}.\mathsf{T}| = |\mathsf{T}| + 1$, $|\mathsf{p}?[S];\mathsf{T}| = |\mathsf{p}![S];\mathsf{T}| = |\mathsf{T}| + 1$ and $|\mathsf{p}\oplus\{l_i : \mathsf{T}_i\}_{i\in I}| = |\mathsf{p}\&\{l_i : \mathsf{T}_i\}_{i\in I}| = \sum_{i\in I} |\mathsf{T}_i| + 1$.

We then bound the size of the type graph of T.

Definition 2.5 (Subformulas of Local Types)**.** The set of subformulas of a local type T, denoted $\mathrm{Sub}(\mathsf{T})$, is defined as: $\mathrm{Sub}(\mathtt{end}) = \{\mathtt{end}\}$, $\mathrm{Sub}(\mathbf{t}) = \{\mathbf{t}\}$, $\mathrm{Sub}(\mu \mathbf{t}.\mathsf{T}) = \{\mu \mathbf{t}.\mathsf{T}\} \cup \{\mathsf{T}'[\mu \mathbf{t}.\mathsf{T}/\mathbf{t}] \mid \mathsf{T}' \in \mathrm{Sub}(\mathsf{T})\}$, $\mathrm{Sub}(\mathsf{p}![S];\mathsf{T}) = \{\mathsf{p}![S];\mathsf{T}\} \cup \mathrm{Sub}(\mathsf{T})$, $\mathrm{Sub}(\mathsf{p}?[S];\mathsf{T}) = \{\mathsf{p}?[S];\mathsf{T}\} \cup \mathrm{Sub}(\mathsf{T})$, $\mathrm{Sub}(\mathsf{p}\oplus\{l_i : \mathsf{T}_i\}_{i\in I}) = \{\mathsf{p}\oplus\{l_i : \mathsf{T}_i\}_{i\in I}\} \cup \bigcup_{i\in I} \mathrm{Sub}(\mathsf{T}_i)$, and $\mathrm{Sub}(\mathsf{p}\&\{l_i : \mathsf{T}_i\}_{i\in I}) = \{\mathsf{p}\&\{l_i : \mathsf{T}_i\}_{i\in I}\} \cup \bigcup_{i\in I} \mathrm{Sub}(\mathsf{T}_i)$.

Lemma 2.6. $|\mathrm{Sub}(\mathsf{T})| = \mathcal{O}(|\mathsf{T}|)$.

Proof. By induction on the structure of T. □

Lemma 2.7. If T' is reachable from T in the type graph $\mathbb{G}(\mathsf{T})$ then $\mathsf{T}' \in \mathrm{Sub}(\mathsf{T})$.

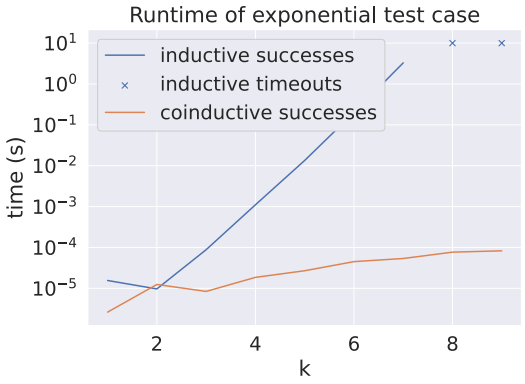
Proof. By induction on the length of the path from T' to T, and matching this with the appropriate rule in Definition 2.5. □

Corollary 2.7.1. The number of nodes in $\mathbb{G}(\mathsf{T})$ is $\mathcal{O}(|\mathsf{T}|)$.

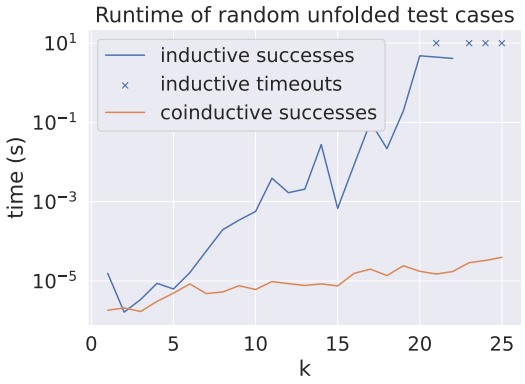
Ghilezan *et al.* give an algorithm for checking subtyping in [12, Table 6] (extending the inductive algorithm for binary session types in [10]). We extend the proof in [22, Theorem 3.6] and prove that their algorithm takes exponential time.

Theorem 2.8. The algorithm to check $\mathsf{T}_1 \leq \mathsf{T}_2$ given in [12, Table 6] takes between $\mathcal{O}\left(n^{n^3}\right)$ and $\Omega((\sqrt{n})!)$ time in the worst case with $n = |\mathsf{T}_1| + |\mathsf{T}_2|$.

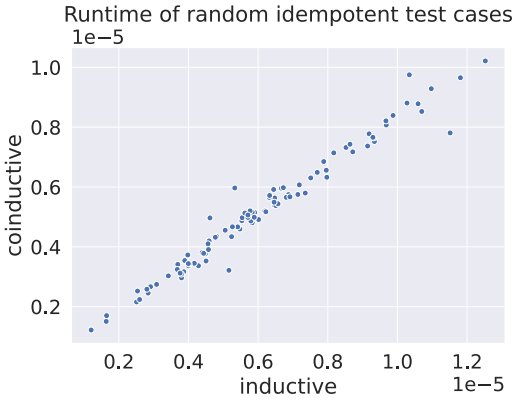
Subtype checking can be performed in quadratic time by checking that the types admit a type simulation (Definition 2.2), which by Corollary 2.7.1 has size $\mathcal{O}(|\mathsf{T}_1| \cdot |\mathsf{T}_2|)$. This



(a) Worst-case.



(b) Random types isomorphic to $\mu \mathbf{t}.l_1 : \mathbf{t}, l_2 : \mathbf{t}$.

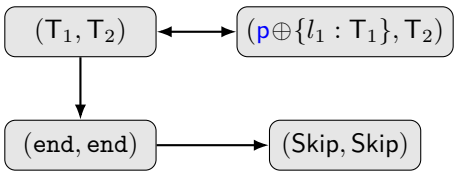


(c) $\mathsf{T} \leq \mathsf{T}$.

Figure 3: Runtimes from empirical analysis of the two subtyping algorithms.

is done by building a product graph of required dependencies for the type simulation relation.

Example 2.9. Let $\mathsf{T}_1 = \mu \mathbf{t}.\mathsf{p} \oplus \{l_1 : \mathsf{p} \oplus \{l_1 : \mathbf{t}\}, l_2 : \mathtt{end}\}$ and $\mathsf{T}_2 = \mu \mathbf{t}.\mathsf{p} \oplus \{l_1 : \mathbf{t}, l_2 : \mathtt{end}\}$. T_1 is a subtype of T_2; the graph for the type simulation is below.



Theorem 2.10. $\mathsf{T}_1 \leq \mathsf{T}_2$ can be checked in $\mathcal{O}(|\mathsf{T}_1| \cdot |\mathsf{T}_2|)$ time in the worst case.

Similarly to Lange *et al.* [16], we implemented the two algorithms for MPST subtyping in C++, and benchmarked them on different test cases (Figure 3).

$$
\begin{array}{lll}
\mathsf{G} ::= \mathtt{end} & & \text{(inaction)} \\
\quad \mid \mathsf{p} \to \mathsf{q} : [S];\ \mathsf{G} & & \text{(message)} \\
\quad \mid \mathsf{p} \to \mathsf{q} : \{l_i : \mathsf{G}_i\}_{i \in I} & & \text{(branching)} \\
\quad \mid \mu \mathbf{t}.\mathsf{G} & & \text{(recursive type)} \\
\quad \mid \mathbf{t} & & \text{(type variable)}
\end{array}
$$

Figure 4: Global types.

3 Complexity Analyses of Three Projections

This section studies the projection of global types. We first define the syntax of global types (Figures 4), based on the intuition from Section 1. We then study inductive projections with two kinds of merging in Section 3.1, and a recent algorithm [21] in Section 3.2.

Definition 3.1 (Size of a global type)**.** The size of a global type G, denoted $|\mathsf{G}|$, is defined as: $|\mathbf{t}| = |\mathtt{end}| = 1$; $|\mu \mathbf{t}.\mathsf{G}| = 1 + |\mathsf{G}|$; $|\mathsf{p} \to \mathsf{q} : [S];\ \mathsf{G}| = 1 + |G|$; and $|\mathsf{p} \to \mathsf{q} : \{l_i : \mathsf{G}_i\}_{i \in I}| = 1 + \sum_{i \in I} |G_i|$.

3.1 Two Inductive Projections

The *projection* of a global type G onto p includes only the communications of G where p is involved (Definition 3.2). However, the problem is when there is a branching not involving p: how can p know which branch the global type is on? This is resolved by *merging*. *Plain merging* [15] (Definition 3.3) requires that both branches have to be identical when projected to p. *Full merging* [26, 18] (Definition 3.4) allows us to merge different branches where p has been notified which branch the global type has taken.

Definition 3.2 (Inductive projection [18])**.** The projection operator on participant p for global type G, denoted by $\mathsf{G}\upharpoonright_{\mathsf{p}}$, is a partial function defined as:

- $(\mathsf{q} \to \mathsf{r} : [S]; \mathsf{G})\upharpoonright_{\mathsf{p}} = \begin{cases} \mathsf{r}![S]; \mathsf{G}\upharpoonright_{\mathsf{p}} & \text{if } \mathsf{p} = \mathsf{q} \\ \mathsf{q}?[S]; \mathsf{G}\upharpoonright_{\mathsf{p}} & \text{if } \mathsf{p} = \mathsf{r} \\ \mathsf{G}\upharpoonright_{\mathsf{p}} & \text{otherwise} \end{cases}$

- $(\mathsf{q} \to \mathsf{r} : \{l_i : \mathsf{G}_i\}_{i\in I})\upharpoonright_{\mathsf{p}} = \begin{cases} \mathsf{r}\oplus\{l_i : \mathsf{G}_i\upharpoonright_{\mathsf{p}}\}_{i\in I} & \text{if } \mathsf{p} = \mathsf{q} \\ \mathsf{q}\&\{l_i : \mathsf{G}_i\upharpoonright_{\mathsf{p}}\}_{i\in I} & \text{if } \mathsf{p} = \mathsf{r} \\ \bigsqcap_{i\in I}\mathsf{G}_i\upharpoonright_{\mathsf{p}} & \text{otherwise} \end{cases}$

- $(\mu \mathbf{t}.\mathsf{G})\upharpoonright_{\mathsf{p}} = \begin{cases} \mathtt{end} & \text{if } \mathsf{p} \notin \mathtt{pt}\{\mathsf{G}\} \text{ and } \mu\mathbf{t}.\mathsf{G} \text{ is closed} \\ \mu\mathbf{t}.\,(\mathsf{G}\upharpoonright_{\mathsf{p}}) & \text{otherwise} \end{cases}$

- $\mathtt{end}\upharpoonright_{\mathsf{p}} = \mathtt{end}$

where the merging operator $\sqcap$ is defined in Definitions 3.3 and 3.4 below. If undefined by the above rules, then $\mathsf{G}\upharpoonright_{\mathsf{p}}$ is undefined.

Definition 3.3 (Plain merging [18])**.** The plain merge is defined by $\mathsf{T} \sqcap \mathsf{T} = \mathsf{T}$, and undefined otherwise.

Definition 3.4 (Full merging [18])**.** The full merge is defined inductively by:

- $\mathsf{p}\,\dagger\,[S];\mathsf{T}_1 \bowtie \mathsf{p}\,\dagger\,[S];\mathsf{T}_2 = \mathsf{p}\,\dagger\,[S];(\mathsf{T}_1 \bowtie \mathsf{T}_2)$ with $\dagger \in \{!, ?\}$;

- $\mathbf{t} \bowtie \mathbf{t} = \mathbf{t}$

- $\mathtt{end} \bowtie \mathtt{end} = \mathtt{end}$

- $\mathsf{p}\oplus\{l_i : \mathsf{T}_i\}_{i\in I} \bowtie \mathsf{p}\oplus\{l_i : \mathsf{T}'_i\}_{i\in I} = \mathsf{p}\oplus\{l_i : \mathsf{T}_i \bowtie \mathsf{T}'_i\}_{i\in I}$

- $\mathsf{p}\&\{l_i : \mathsf{T}_i\}_{i\in I} \bowtie \mathsf{p}\&\{l_j : \mathsf{T}'_j\}_{j\in J}$
$$= \mathsf{p}\&\Big(\{l_k : (\mathsf{T}_k \bowtie \mathsf{T}'_k) \mid k \in I \cap J\} \cup \{l_i : \mathsf{T}_i \mid i \in I \setminus J\} \cup \{l_j : \mathsf{T}'_j \mid j \in J \setminus I\}\Big)$$

and undefined otherwise.

Example 3.5. Continuing the example in Section 1, we may project G onto each participant. For example:

$$\begin{aligned}
\mathsf{G}\upharpoonright_{\mathsf{r}} &= \mu\mathbf{t}.(\mathsf{p} \to \mathsf{q} : \{l_1 : \mathsf{q} \to \mathsf{r} : [\mathtt{int}];\ \mathbf{t}, l_2 : \mathsf{q} \to \mathsf{r} : [\mathtt{int}];\ \mathbf{t}\})\upharpoonright_{\mathsf{r}} \\
&= \mu\mathbf{t}.((\mathsf{q} \to \mathsf{r} : [\mathtt{int}];\ \mathbf{t})\upharpoonright_{\mathsf{r}} \sqcap (\mathsf{q} \to \mathsf{r} : [\mathtt{int}];\ \mathbf{t})\upharpoonright_{\mathsf{r}}) \\
&= \mu\mathbf{t}.((\mathsf{q}?[\mathtt{int}];\mathbf{t}) \sqcap (\mathsf{q}?[\mathtt{int}];\mathbf{t})) \\
&= \mu\mathbf{t}.\mathsf{q}?[\mathtt{int}];\mathbf{t} \qquad \text{((plain) mergeable because both sides are identical)}
\end{aligned}$$

We also have $\mathsf{G}\upharpoonright_{\mathsf{p}} = \mu\mathbf{t}.\mathsf{q}\oplus\{l_1 : \mathbf{t}, l_2 : \mathbf{t}\}$ and $\mathsf{G}\upharpoonright_{\mathsf{q}} = \mu\mathbf{t}.\mathsf{p}\triangleright\{l_1 : \mathsf{r}![\mathtt{int}];\mathbf{t}, l_2 : \mathsf{r}![\mathtt{int}];\mathbf{t}\}$.

The following example illustrates that full merging is more expressive than plain merging.

Example 3.6. Consider the following global type, where r is relaying a label received from q, to p: $\mathsf{G} = \mu\mathbf{t}.\ \mathsf{q} \to \mathsf{r} : \{l_1 : \mathsf{r} \to \mathsf{p} : \{l_1 : \mathbf{t}\}, l_2 : \mathsf{r} \to \mathsf{p} : \{l_2 : \mathtt{end}\}\}$. Then $\mathsf{G}\upharpoonright_{\mathsf{p}} = \mu\mathbf{t}.\ (\mathsf{r} \to \mathsf{p} : \{l_1 : \mathbf{t}\}\upharpoonright_{\mathsf{p}} \sqcap \mathsf{r} \to \mathsf{p} : \{l_2 : \mathtt{end}\}\upharpoonright_{\mathsf{p}}) = \mu\mathbf{t}.\ (\mathsf{r}\&\{l_1 : \mathbf{t}\} \sqcap \mathsf{r}\&\{l_2 : \mathtt{end}\})$.

By plain merging, this is *undefined*; by full merging, it is *defined* as: $\mu\mathbf{t}.\mathsf{r}\&\{l_1 : \mathbf{t}, l_2 : \mathtt{end}\}$.

3.1.1 Complexity of projection with plain merging

We show that projection with plain merging can be performed in $\mathcal{O}(|\mathsf{G}| \log |\mathsf{G}|)$ time. Most cases of inductive projection can be performed in constant time and reduces the total sizes of the types to be projected by a constant amount; the only exception is the projection of $(\mathsf{q} \to \mathsf{r} : \{l_i : \mathsf{G}_i\}_{i\in I})\upharpoonright_{\mathsf{p}}$ where $|I| \geq 2$ and $\mathsf{p} \notin \{\mathsf{q}, \mathsf{r}\}$. In this case the syntactic equality check takes linear time and the size of each projected branch is at most half of the size of the original type. Theorem 3.7 formalises this argument.

Theorem 3.7. Projection with plain merging, $\mathsf{G}\upharpoonright_{\mathsf{p}}$, can be performed in $\mathcal{O}(n \log n)$ time (where $n = |\mathsf{G}|$).

3.1.2 Complexity of projection with full merging: two algorithms

Naïve Algorithm for Full Merging. If we use the same naïve approach to projection with full merging, the time complexity is not optimal, since the branch sizes may be unbalanced.

Example 3.8. Define $\mathsf{G}^{(n)}$ as follows: $\mathsf{G}^{(0)} = \mathsf{q} \to \mathsf{p} : \{l_0 : \mathtt{end}\}$, and $\mathsf{G}^{(n+1)} = \mathsf{q} \to \mathsf{r} : \left\{l_1 : \mathsf{G}^{(n)}, l_2 : \mathsf{q} \to \mathsf{p} : \{l_{n+1} : \mathtt{end}\}\right\}$ for $n > 0$.

Theorem 3.9. In Example 3.8, finding $\mathsf{G}^{(n)}\upharpoonright_{\mathsf{p}}$ under full merging with a naïve representation of types takes $\Theta(n^2)$ time.

Proof. We have $\mathsf{G}^{(n)}\upharpoonright_{\mathsf{p}} = \mathsf{q}\&\{l_i : \mathtt{end}\}_{1 \le i \le n}$. Therefore, when computing $\mathsf{G}^{(n+1)}\upharpoonright_{\mathsf{p}}$ from $\mathsf{G}^{(n)}\upharpoonright_{\mathsf{p}}$, the naïve algorithm will need to compute $\mathsf{q}\&\{l_i : \mathtt{end}\}_{1 \le i \le n} \between \mathsf{q}\&\{l_{n+1} : \mathtt{end}\}$. As the size of the types is linear, and merging takes linear time in the naïve algorithm, the n-th step uses $\Theta(n)$ time, so the total time is $\Theta(n^2)$. □

An Optimised Algorithm for Full Merging. We construct the optimised algorithm using the classical technique of small-to-large merging: by representing branching in terms of binary search trees, we can efficiently merge branching types by adding the branches of one smaller type to the other bigger type. Intuitively, each branch will be "merged" into a larger type only $\mathcal{O}(\log n)$ times, but a careful analysis is needed as merging affects the size of types and is recursive.

Lemma 3.10. If $\mathsf{T}_1 \between \mathsf{T}_2 = \mathsf{T}$ and T is defined then $\mathsf{T}_1 \between \mathsf{T}_2$ can be computed in time $\mathcal{O}(|\mathsf{T}_1| + |\mathsf{T}_2| - |\mathsf{T}| + |\mathsf{T}_2| \cdot \log |\mathsf{T}_1|)$.

Proof. By structural induction on T_1. We show that, up to constant factors, the merge can be computed in time at most $2(|\mathsf{T}_1| + |\mathsf{T}_2| - |\mathsf{T}|) - 1 + (|\mathsf{T}_2| - 1) \cdot \log |\mathsf{T}_1|$. The $\log |\mathsf{T}_1|$ factor stems from searching for matching labels when merging branches. □

Theorem 3.11. Projection with full merging can be performed in $\mathcal{O}\Big(|\mathsf{G}| \log^2 |\mathsf{G}|\Big)$ time.

Proof. We show inductively that (up to constant factors) we can compute $\mathsf{G}\upharpoonright_{\mathsf{p}}$ in time $|\mathsf{G}| \log^2 |\mathsf{G}| + 2|\mathsf{G}| - |\mathsf{T}|$, where $\mathsf{T} = \mathsf{G}\upharpoonright_{\mathsf{p}}$. □

3.2 Tirore *et al.*'s Projection [21]

We briefly analyse the complexity of the most recent projection algorithm proposed by Tirore *et al.* [21], which we call TBC. This algorithm computes a projection on inductive types with plain merge that is sound and complete with respect to its coinductive projection [12], which treats types as infinite trees. Example 3.12 shows that syntactic inductive projections may fail to compute a projection when TBC may not.

Example 3.12. Consider the following global type:

$$\mathsf{G} = \mathsf{q} \to \mathsf{r} : \{l_1 : \mu \mathbf{t}.\mathsf{q} \to \mathsf{p} : [S];\ \mathbf{t}, l_2 : \mu \mathbf{t}'.\mathsf{q} \to \mathsf{p} : [S];\ \mathsf{q} \to \mathsf{p} : [S];\ \mathbf{t}'\}$$

Both branches of the communication $\mathsf{q} \to \mathsf{r}$ represent the same tree, but they are not equal syntactically. Therefore, inductive projection (with either kind of merging) will fail to compute a projection. The TBC projection is defined as these types represent the same infinite trees.

To calculate $\mathsf{G}\upharpoonright_{\mathsf{p}}$, the algorithm discards all branches except one for each branching not involving p, to get a candidate T. Then, the algorithm decides whether T is in fact a projection of G [21, Definition 29]. Two predicates are checked: one for all nodes in a graph representation of G and another for all nodes in a *product graph* of G and T, which has size $|\mathsf{G}| \cdot |\mathsf{T}|$. As $|\mathsf{T}| \leq |\mathsf{G}|$, we arrive at the following theorem.

Theorem 3.13. The projection in [21] takes $\mathcal{O}(|\mathsf{G}|^2)$ time.

4 Minimum Type Inference and its Complexity Analysis

4.1 MPST Calculus

We first introduce the MPST process calculus (from [12]). A *multiparty session* [13] (Figure 5) is the *parallel composition* of *processes*, which can send and receive *messages* and *labels* with other processes in the composition. We illustrate the syntax with the following example.

Expressions	$\mathsf{e}, \mathsf{e}', \cdots ::= \mathsf{n} \mid \mathsf{i}$	(natural number, integer)
	$\mid \mathsf{true} \mid \mathsf{false} \mid x$	(true, false, variable)
	$\mid \mathsf{e} \vee \mathsf{e} \mid \neg\mathsf{e} \mid \mathsf{e} + \mathsf{e}$	(or, not, plus)
	$\mid \mathsf{e} \oplus \mathsf{e} \mid \mathtt{neg}(\mathsf{e})$	(nondet. choice, negation)
Processes	$P, P', \cdots ::= \mathbf{0} \mid \mathsf{p}!\langle\mathsf{e}\rangle.P \mid \mathsf{p}?(x).P \mid$	(inaction, output, input)
	$\mid \mathsf{p} \triangleleft l.P \mid \mathsf{p} \triangleright \{l_i : P_i\}_{i \in I}$	(selection, branch)
	$\mid \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ P \mid \mu X.P \mid X$	(conditional, recursion, proc variable)
Multiparty sessions	$\mathcal{M} ::= \mathsf{p} :: P \mid \mathcal{M} \mid \mathcal{M}$	(role, parallel composition)

Figure 5: Multiparty synchronous session calculus.

Example 4.1. Consider the session which is a parallel composition of three *participants* $\mathsf{p}, \mathsf{q}, \mathsf{r}$.

$$\begin{aligned}\mathcal{M} = {} & \mathsf{p} :: \mu X.\mathsf{if}\ (\mathsf{true} \oplus \mathsf{false})\ \mathsf{then}\ \mathsf{q} \triangleleft l_1.X\ \mathsf{else}\ \mathsf{q} \triangleleft l_2.X \\ & \mid \mathsf{q} :: \mu X.\mathsf{p} \triangleright \{l_1 : \mathsf{r}!\langle 10\rangle.X, l_2 : \mathsf{r}!\langle 20\rangle.X\} \\ & \mid \mathsf{r} :: \mu X.\mathsf{q}?(x).X\end{aligned}$$

First, participant p chooses nondeterministically ($\mathsf{true} \oplus \mathsf{false}$) whether to *select* label l_1 or l_2. Participant q *branches* on this label, and *sends* either 10 or 20 to r, who *receives* this value. The recursive construct $P = \mu X \ldots X$ repeats the process at P once X is reached.

Processes can be *typed* by *local types* [12]. This is done using *typing rules* for expressions and processes (Figure 10). The rule $\Gamma \vdash P : \mathsf{T}$ states that process P is typed by local type T under context Γ (which contains types for variables and recursion variables).

4.2 Minimum Type Inference System

We now develop a *minimum type inference system* and give a complexity analysis. Our system involves two steps: (1) deriving a set of *constraints*, and (2) solving these constraints for a *minimum type*. Using the minimum type, we can use a subtyping check to verify that the processes are typed by the projected local types. This section focuses on the first step: deriving constraints. Section 4.3 will explain the second step.

Let $\mathbf{T}$ and $\mathbf{S}$ denote the set of types and sorts, respectively. First, we extend the syntax of types (resp. sorts) to include a countable set of *type variables* $\mathbf{Tv}$ (resp. *sort variables* $\mathbf{Sv}$), denoted by $\xi, \psi, \ldots$ (resp. $\alpha, \beta, \ldots$).

Given a constraint set $\mathcal{C}$ on a set of free variables $\mathcal{X}$, a *type substitution* $\sigma : \mathbf{Tv} \rightharpoonup \mathbf{T}$ is a partial function from type variables to types. This can be extended in the usual way to a function $\mathbf{T} \to \mathbf{T}$. Similarly, we define sort substitutions $\pi : \mathbf{Sv} \rightharpoonup \mathbf{S}$.

Similarly to Pierce's [17, Section 22.3] treatment of μ-types in the λ-calculus, we describe the *rules for constraint derivation*, in which *type* and *sort constraints* are derived inductively. We write $\uplus$ for disjoint union, and alpha-convert all μ-bound variables to be unique. In Figure 6, we define the judgement $\Gamma \vdash P : \psi \mid_{\mathcal{X}} \mathcal{C}$ which states that P corresponds to the variable ψ, under constraint set $\mathcal{C}$ and context Γ, with free type and sort variables in $\mathcal{X}$. Γ contains judgements of the forms $x : \alpha$ and $X : \xi$. The rules $\Gamma \vdash \mathsf{e} : \alpha \mid_{\mathcal{X}} \mathcal{C}$ (Figure 7) are the analogous rules for sorts which replaces ψ by the sort variable α.

For example, the rule [C-Out] states that, for a process of the form $\mathsf{p}!\langle \mathsf{e} \rangle.P$ to correspond to the variable ξ, the shape of ξ must match the sort of e and the type of P, which is captured in the constraint $\mathsf{p}?[\alpha]; \psi \leq \xi$.

Now we are ready to introduce the main definition, a minimum type of process P.

Definition 4.2 (Minimum types)**.** A *minimum type* of a process P is a local type T such that: (1) $\emptyset \vdash P : \mathsf{T}$; and (2) for all T' such that $\emptyset \vdash P : \mathsf{T}'$, there exists a sort substitution $\pi : \mathbf{Sv} \to \mathbf{S}$ such that $\pi\mathsf{T} \leq \mathsf{T}'$.

Definition 4.3 (Size of a process)**.** The size of a process P (resp. e), denoted $|P|$ (resp. $|\mathsf{e}|$), is inductively defined as: $|\mathbf{0}| = |X| = 1$, $|\mu X.P| = |P| + 1$, $|\mathsf{p}?(x).P| = |\mathsf{p}!\langle \mathsf{e} \rangle.P| = |P| + |\mathsf{e}| + 1$, $|\mathsf{p} \triangleleft l.P| = |P| + 1$, $|\mathsf{p} \triangleright \{l_i : P_i\}_{i \in I}| = \sum_{i \in I} |P_i| + 1$, $|\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2| = |P_1| + |P_2| + |\mathsf{e}| + 1$, $|\mathsf{true}| = |\mathsf{false}| = |\mathsf{n}| = |\mathsf{i}| = |x| = 1$, $|\neg \mathsf{e}| = |\mathtt{neg}(\mathsf{e})| = |\mathsf{e}| + 1$, and $|\mathsf{e} \vee \mathsf{e}'| = |\mathsf{e} + \mathsf{e}'| = |\mathsf{e} \oplus \mathsf{e}'| = |\mathsf{e}| + |\mathsf{e}'| + 1$.

Definition 4.4 (Solutions)**.** Given that $\mathcal{C}$ is a constraint set, we say that (σ, π) is a solution to $\mathcal{C}$ if for all $\xi \leq \psi \in \mathcal{C}$, $\pi\sigma\xi \leq \pi\sigma\psi$, and for all $\alpha = \beta \in \mathcal{C}$, $\pi\alpha = \pi\beta$.

Processes $\Gamma \vdash P : \psi \mid_{\mathcal{X}} \mathcal{C}$

$$\frac{\xi \text{ fresh}}{\Gamma \vdash \mathbf{0} : \xi \mid_{\{\xi\}} \{\mathtt{end} \leq \xi\}} \text{ [C-End]} \qquad \frac{X : \chi \in \Gamma \quad \xi \text{ fresh}}{\Gamma \vdash X : \xi \mid_{\{\xi\}} \{\chi \leq \xi\}} \text{ [C-Var]} \qquad \frac{\Gamma, X : \xi \vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}}{\Gamma \vdash \mu X.P : \xi \mid_{\mathcal{X}} \mathcal{C}} \text{ [C-Rec]}$$

$$\frac{\Gamma, x : \alpha \vdash P : \psi \mid_{\mathcal{X}} \mathcal{C} \quad \alpha \text{ fresh} \quad \xi \text{ fresh} \quad \mathcal{C}' = \mathcal{C} \cup \{\mathsf{p}?[\alpha]; \psi \leq \xi\}}{\Gamma \vdash \mathsf{p}?(x).P : \xi \mid_{\mathcal{X} \cup \{\xi, \alpha\}} \mathcal{C}'} \text{ [C-In]}$$

$$\frac{\Gamma \vdash P : \psi \mid_{\mathcal{X}} \mathcal{C}_1 \quad \Gamma \vdash \mathsf{e} : \alpha \mid_{\mathcal{X}'} \mathcal{C}_2 \quad \xi \text{ fresh} \quad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathsf{p}![\alpha]; \psi \leq \xi\}}{\Gamma \vdash \mathsf{p}!\langle \mathsf{e} \rangle.P : \xi \mid_{\mathcal{X} \cup \mathcal{X}' \cup \{\xi\}} \mathcal{C}'} \text{ [C-Out]}$$

$$\frac{\forall i \in I.\, \Gamma \vdash P_i : \psi_i \mid_{\mathcal{X}_i} \mathcal{C}_i \quad \xi \text{ fresh} \quad \mathcal{C}' = \bigcup_{i \in I} \mathcal{C}_i \cup \{\mathsf{p}\&\{l_i : \psi_i\}_{i \in I} \leq \xi\} \quad \mathcal{X}' = \bigcup_{i \in I} \mathcal{X}_i \cup \{\xi\}}{\Gamma \vdash \mathsf{p} \triangleright \{l_i : P_i\}_{i \in I} : \xi \mid_{\mathcal{X}'} \mathcal{C}'} \text{ [C-Bra]}$$

$$\frac{\Gamma \vdash P : \psi \mid_{\mathcal{X}} \mathcal{C}_i \quad \xi \text{ fresh} \quad \mathcal{C}' = \mathcal{C} \cup \{\mathsf{p} \oplus \{l : \psi\} \leq \xi\} \quad \mathcal{X}' = \mathcal{X} \cup \{\xi\}}{\Gamma \vdash \mathsf{p} \triangleleft l.P : \xi \mid_{\mathcal{X}'} \mathcal{C}'} \text{ [C-Sel]}$$

$$\frac{\begin{array}{c}\Gamma \vdash P_1 : \psi_1 \mid_{\mathcal{X}_1} \mathcal{C}_1 \quad \Gamma \vdash P_2 : \psi_2 \mid_{\mathcal{X}_2} \mathcal{C}_2 \quad \Gamma \vdash \mathsf{e} : \alpha \mid_{\mathcal{X}_3} \mathcal{C}_3 \\ \xi \text{ fresh} \quad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\psi_1 \leq \xi, \psi_2 \leq \xi, \alpha = \mathtt{bool}\}\end{array}}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : \xi \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3 \cup \{\xi\}} \mathcal{C}'} \text{ [C-Cond]}$$

Figure 6: Constraint rules for processes.

Our constraint system proceeds in two stages: first, we find a constraint set for P using the rules above, then we try to solve the constraint set. The following theorems prove soundness and completeness of the first step.

Theorem 4.5 (Soundness of constraints)**.** If $\vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$ is derivable and (σ, π) is a solution to $\mathcal{C}$, then P has type $\pi\sigma\xi$.

Proof. We show that if $\Gamma \vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$ is derivable and (σ, π) is a solution to $\mathcal{C}$, then there is a typing derivation for $\pi\sigma\Gamma \vdash P : \pi\sigma\xi$ (where we define $\pi\sigma\Gamma$ to apply $\pi\sigma$ to each type in Γ). We prove this by induction on the proof tree of the constraint derivations. □

Theorem 4.6 (Completeness of constraints)**.** Let $\vdash P : \mathsf{T}_0$ and $\vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$. Then there exists a solution (σ', π') of $\mathcal{C}$ such that $\pi'\sigma'\xi = \mathsf{T}_0$.

Proof. We show that if $\Gamma \vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$, and $\pi\sigma\Gamma \vdash P : \mathsf{T}_0$, such that $\mathrm{dom}(\sigma) \cap \mathcal{X} = \emptyset$ and $\mathrm{dom}(\pi) \cap \mathcal{X} = \emptyset$, then there exists a solution (σ', π') of $\mathcal{C}$ such that $\pi'\sigma'\xi = \mathsf{T}_0$, $\sigma' \setminus \mathcal{X} = \sigma$,

Expressions $\Gamma \vdash \mathsf{e} : \alpha \mid_{\mathcal{X}} \mathcal{C}$

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha \mid_{\emptyset} \emptyset} \text{ [C-SortVar]}$$

$$\frac{\alpha \text{ fresh}}{\Gamma \vdash \mathsf{true} : \alpha \mid_{\{\alpha\}} \{\alpha = \mathtt{bool}\}} \text{ [C-True]} \qquad \frac{\alpha \text{ fresh}}{\Gamma \vdash \mathsf{false} : \alpha \mid_{\{\alpha\}} \{\alpha = \mathtt{bool}\}} \text{ [C-False]}$$

$$\frac{\alpha \text{ fresh}}{\Gamma \vdash \mathsf{n} : \alpha \mid_{\{\alpha\}} \{\alpha = \mathtt{nat}\}} \text{ [C-Nat]} \qquad \frac{\alpha \text{ fresh}}{\Gamma \vdash \mathsf{i} : \alpha \mid_{\{\alpha\}} \{\alpha = \mathtt{int}\}} \text{ [C-Int]}$$

$$\frac{\Gamma \vdash \mathsf{e} : \alpha \mid_{\mathcal{X}} \mathcal{C} \qquad \mathcal{C}' = \mathcal{C} \cup \{\alpha = \mathtt{int}\}}{\Gamma \vdash \mathtt{neg}(\mathsf{e}) : \alpha \mid_{\mathcal{X}} \mathcal{C}'} \text{ [C-Neg]}$$

$$\frac{\Gamma \vdash \mathsf{e} : \alpha \mid_{\mathcal{X}} \mathcal{C} \qquad \mathcal{C}' = \mathcal{C} \cup \{\alpha = \mathtt{bool}\}}{\Gamma \vdash \neg \mathsf{e} : \alpha \mid_{\mathcal{X}} \mathcal{C}'} \text{ [C-Not]}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \alpha_1 \mid_{\mathcal{X}_1} \mathcal{C}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \alpha_2 \mid_{\mathcal{X}_2} \mathcal{C}_2 \\ \beta \text{ fresh} \qquad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\alpha_1 = \mathtt{bool}, \alpha_2 = \mathtt{bool}, \beta = \mathtt{bool}\}\end{array}}{\Gamma \vdash \mathsf{e}_1 \vee \mathsf{e}_2 : \beta \mid_{\mathcal{X}_1 \uplus \mathcal{X}_2 \uplus \{\beta\}} \mathcal{C}'} \text{ [C-Or]}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \alpha_1 \mid_{\mathcal{X}_1} \mathcal{C}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \alpha_2 \mid_{\mathcal{X}_2} \mathcal{C}_2 \qquad \beta \text{ fresh} \qquad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\alpha_1 = \beta, \alpha_2 = \beta\}}{\Gamma \vdash \mathsf{e}_1 \oplus \mathsf{e}_2 : \beta \mid_{\mathcal{X}_1 \uplus \mathcal{X}_2 \uplus \{\beta\}} \mathcal{C}'} \text{ [C-Nondet]}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \alpha_1 \mid_{\mathcal{X}_1} \mathcal{C}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \alpha_2 \mid_{\mathcal{X}_2} \mathcal{C}_2 \qquad \beta \text{ fresh} \qquad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\alpha_1 = \beta, \alpha_2 = \beta, \beta = \mathtt{int}\}}{\Gamma \vdash \mathsf{e}_1 + \mathsf{e}_2 : \beta \mid_{\mathcal{X}_1 \uplus \mathcal{X}_2 \uplus \{\beta\}} \mathcal{C}'} \text{ [C-Add]}$$

Figure 7: Constraint rules for expressions.

and $\pi' \setminus \mathcal{X} = \pi$. The proof is done by postponing the subsumption rules in the derivations of P. $\square$

Example 4.7 (Constraint derivations of a conditional)**.** As an example, we give the constraint derivation of process $P = \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \mathsf{p}\triangleleft l_1.\mathsf{p}\triangleright\{l_2 : \mathbf{0}\}\ \mathsf{else}\ \mathsf{p}\triangleleft l_1.\mathsf{p}\triangleright\{l_3 : \mathbf{0}\}$. Intuitively, this process should have a type $\mathsf{T} = \mathsf{p}\oplus\{l_1 : \mathsf{p}\&\{l_2 : \mathtt{end}, l_3 : \mathtt{end}\}\}$.

Let $\mathcal{C}'_1 = \{\mathtt{end} \le \xi_1, \mathsf{p}\&\{l_2 : \xi_1\} \le \xi_2\}$, $\mathcal{C}_1 = \mathcal{C}'_1 \cup \{\mathsf{p}\oplus\{l_1 : \xi_2\} \le \xi_3\}$,
$\mathcal{C}'_2 = \{\mathtt{end} \le \xi_4, \mathsf{p}\&\{l_3 : \xi_4\} \le \xi_5\}$, and $\mathcal{C}_2 = \mathcal{C}'_2 \cup \{\mathsf{p}\oplus\{l_1 : \xi_5\} \le \xi_6\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{0} : \xi_1 \mid_{\{\xi_1\}} \{\mathtt{end} \le \xi_1\}}\ [\text{C-End}]}{\vdash \mathsf{p}\triangleright\{l_2 : \mathbf{0}\} : \xi_2 \mid_{\{\xi_1,\xi_2\}} \mathcal{C}'_1}\ [\text{C-Bra}]}{\vdash \mathsf{p}\triangleleft l_1.\mathsf{p}\triangleright\{l_2 : \mathbf{0}\} : \xi_3 \mid_{\{\xi_1,\xi_2,\xi_3\}} \mathcal{C}_1}\ [\text{C-Sel}] \quad \dfrac{\dfrac{\dfrac{}{\vdash \mathbf{0} : \xi_4 \mid_{\{\xi_4\}} \{\mathtt{end} \le \xi_4\}}\ [\text{C-End}]}{\vdash \mathsf{p}\triangleright\{l_3 : \mathbf{0}\} : \xi_5 \mid_{\{\xi_4,\xi_5\}} \mathcal{C}'_2}\ [\text{C-Bra}]}{\vdash \mathsf{p}\triangleleft l_2.\mathsf{p}\triangleright\{l_3 : \mathbf{0}\} : \xi_6 \mid_{\{\xi_4,\xi_5,\xi_6\}} \mathcal{C}_2}\ [\text{C-Sel}] \quad \dfrac{}{\vdash \mathsf{true} : \alpha \mid_{\{\alpha\}} \{\mathtt{bool} = \alpha\}}}{\vdash P : \xi \mid_{\{\xi_1,\xi_2,\xi_3,\xi_4,\xi_5,\xi_6,\alpha,\xi\}} \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathtt{bool} = \alpha, \xi_3 \le \xi, \xi_6 \le \xi\}}$$

The last derivation is by [C-Cond], and the set of constraints is $\mathcal{C} = \{\mathtt{end} \le \xi_1, \mathsf{p}\&\{l_2 : \xi_1\} \le \xi_2, \mathsf{p}\oplus\{l_1 : \xi_2\} \le \xi_3, \mathtt{end} \le \xi_4, \mathsf{p}\&\{l_3 : \xi_4\} \le \xi_5, \mathsf{p}\oplus\{l_1 : \xi_5\} \le \xi_6, \mathtt{bool} = \alpha, \xi_3 \le \xi, \xi_6 \le \xi\}$. We have (σ, π) is a solution to $\mathcal{C}$ with $\sigma = \{\xi, \xi_3, \xi_6 \mapsto \mathsf{T}, \xi_1, \xi_4 \mapsto \mathtt{end}, \xi_2 \mapsto \mathsf{p}\&\{l_2 : \mathtt{end}\}, \xi_5 \mapsto \mathsf{p}\&\{l_3 : \mathtt{end}\}\}$ and $\pi = \{\alpha \mapsto \mathtt{bool}\}$. We have $\pi\sigma\xi = \mathsf{T}$, so by soundness, $\vdash P : \mathsf{T}$.

4.3 Solving the Minimum Types

First, we describe how the sort constraints are solved.

Definition 4.8 (Most general solutions for sorts)**.** Let π solve the sort constraints $\mathcal{C}_\mathsf{S}$ if $\pi S_1 = \pi S_2$ for all $S_1 = S_2 \in \mathcal{C}_\mathsf{S}$. Then π is a most general solution of $\mathcal{C}_\mathsf{S}$ if, for all π' solving $\mathcal{C}_\mathsf{S}$, there exists π'' such that $\pi'\alpha = \pi''\pi\alpha$ for all α.

Note that, given a set of sort constraints, we may use a union-find to identify the equivalence classes of equality constraints. Using this, we can find the most general solution of the constraints.

We prove that there exists an algorithm to find the minimum type for typable P. In short, we will build a pair of a *minimum type graph* and a set of *sort constraints* from the constraint set $\mathcal{C}$ generated by $\vdash P : \xi \mid_\mathcal{X} \mathcal{C}$. We then prove *soundness* and *completeness* of our construction.

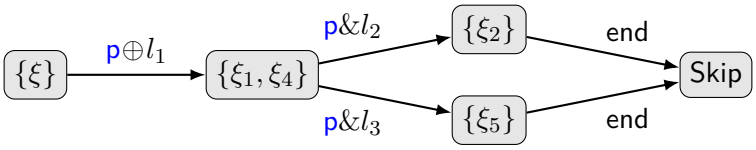


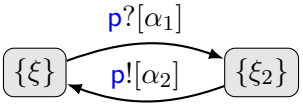
Figure 8: Minimum type graph of Example 4.9.

First, we eliminate all occurences of $\xi \leq \psi$ in $\mathcal{C}$ by applying the substitution $[\xi/\psi]$ to $\mathcal{C}$. We call the resulting set $\mathtt{tr}(\mathcal{C})$. Then, our type graph is a graph whose nodes are subsets of type variables in $\mathcal{C}$, which represent the set of behaviours that the type is required to have. The transitions are generated as the tightest constraints that respect all type variables in the set. We illustrate this with an example.

Example 4.9 (Minimum type graph (1))**.** We continue Example 4.7. We have (up to renaming of type variables) $\mathtt{tr}(\mathcal{C}) = \{\mathtt{end} \leq \xi_1, \mathsf{p}\&\{l_2 : \xi_1\} \leq \xi_2, \mathsf{p}\oplus\{l_1 : \xi_2\} \leq \xi, \mathtt{end} \leq \xi_4, \mathsf{p}\&\{l_3 : \xi_4\} \leq \xi_5, \mathsf{p}\oplus\{l_1 : \xi_5\} \leq \xi, \alpha = \mathtt{bool}\}$. Then the minimum type graph $\mathbb{G}_{\mathtt{tr}(\mathcal{C})}$ is given in Figure 8. This graph represents that after selecting the label l_1, the process can be in either side of the conditional statement, thus it must respect the behaviours of both. This is represented by the node being a two element-set, with one element corresponding to each branch. After branching to the labels l_2 or l_3, this nondeterminism is resolved, as each side of the conditional statement has a disjoint set of branches. Thus we have two separate nodes with one single element, which can now have different behaviours.

Formally, we have that the sort constraints are $\mathcal{C}_\mathsf{S} = \{\alpha = \mathtt{bool}\}$. The most general solution is $\pi = \{\alpha \mapsto \mathtt{bool}\}$, so the minimum type is the type corresponding to $\mathbb{G}_{\mathtt{tr}(\mathcal{C})}$, with π applied to each transition: $\mathsf{T}_{\mathsf{min}} = \mathsf{p}\oplus\{l_1 : \mathsf{p}\&\{l_2 : \mathtt{end}, l_3 : \mathtt{end}\}\}$.

Example 4.10 (Minimum type graph (2))**.** Consider $P = \mu X.\mathsf{p}?(x).\mathsf{p}!\langle x\rangle.X$. We have (up to renaming of type variables) $\mathtt{tr}(\mathcal{C}) = \{\mathsf{p}![\alpha]; \xi \leq \xi_2, \mathsf{p}?[\alpha]; \xi_2 \leq \xi\}$. Then the minimum type graph $\mathbb{G}_\mathcal{C}$ is:



The sort constraints are $\mathcal{C}_\mathsf{S} = \{\alpha = \alpha_1, \alpha = \alpha_2\}$. Solving the sort constraints, the minimum type is $\mathsf{T}_{\mathsf{min}} = \mu\mathbf{t}.\mathsf{p}?[\alpha]; \mathsf{p}![\alpha]; \mathbf{t}$.

Theorem 4.11 (Soundness of minimum type inference)**.** Let $\vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}'$ and $\mathcal{C} = \mathtt{tr}(\mathcal{C}')$. Let the minimum type graph be $\mathbb{G}_{\mathcal{C}}$ with sort constraints $\mathcal{C}_{\mathsf{S}}$. Let $\sigma = \{\psi \mapsto \{\psi\} \mid \psi \in \mathcal{C}\}$ and π be a most general solution of $\mathcal{C}_{\mathsf{S}}$. Then (σ, π) is a solution to $\mathcal{C}$.

Below we write $\pi\mathbb{G}_{\mathcal{C}}$ to be the graph with π applied to all transitions.

Theorem 4.12 (Completeness of minimum type inference)**.** Let $\vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}'$, and $\mathcal{C} = \mathtt{tr}(\mathcal{C}')$. Let (σ, π) be a solution to $\mathcal{C}$. Let $\mathbb{G}_{\mathcal{C}}, \mathcal{C}_{\mathsf{S}}$ be the minimum graph and sort constraints, respectively. If π' is the most general solution to $\mathcal{C}_{\mathsf{S}}$, then $\pi'\mathbb{G}_{\mathcal{C}}$ is a type graph (Definition 2.1) that corresponds to a minimum type (Definition 4.2) $\mathsf{T}_{\mathsf{min}}$ of P.

Proof. Let $\mathsf{T} = \pi\sigma\xi$. By Theorems 4.5 and 4.6, it suffices to show that there exists π'' such that $\pi''\mathsf{T}_{\mathsf{min}} \leq \mathsf{T}$. The proof involves defining a simulation $\sim$ on $V(\pi'\mathbb{G}_{\mathcal{C}}) \times \mathrm{Sub}(\mathsf{T})$ inductively, matching each subformula of T with its set of expected behaviours. We then prove that $\sim$ is a type simulation. □

Theorem 4.13. The minimum type of a process P can be found in $\mathcal{O}(n \cdot 2^n)$ time, where $n = |P|$.

Proof. Finding the minimum type graph requires $\mathcal{O}(n \cdot 2^n)$ time. First, building the set of constraints $\mathcal{C}$ and finding $\mathtt{tr}(C)$ takes polynomial time in n, as there are only polynomially many type constraints. There are at most n type variables, so the type graph has $\mathcal{O}(2^n)$ nodes; generating the transitions takes $\mathcal{O}(n)$ time per node. □

We can show an explicit example where the minimum type has an exponential size.

Theorem 4.14. There exists a process P such that any minimum type of P has size exponential in n.

Proof. Define $P_n = \mu X.\mathsf{p}\triangleright\{l_1 : \ldots \mathsf{p}\triangleright\{l_1 : \mathsf{p}\triangleright\{l_1 : X, l_2 : X\}\}\}$. This has the minimum type T_n such that $\mathsf{T}_n = \mu\mathbf{t}.\mathsf{p}\&\{l_1 : \ldots \mathsf{p}\&\{l_1 : \mathsf{p}\&\{l_1 : \mathbf{t}, l_2 : \mathbf{t}\}\}\}$, where there are n branch constructs in the sort and the type. Consider $P = \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P^{(1)}\ \mathsf{else}\ P^{(2)}$ for some expression e. If the minimum types of $P^{(1)}$ and $P^{(2)}$ are T_n and T_m respectively, then the minimum type of P is $\mathsf{T}_{\mathrm{lcm}(n,m)}$, as l_2 is guaranteed to be safely received only when each cycle is synchronised. Thus, for integers $n_1, \ldots, n_k$, the process P_k, defined by $P^{(1)} = P_{n_1}$

and $P^{(k)} = \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_{n_k}\ \mathsf{else}\ P^{(k-1)}$ has a minimum type of $\mathsf{T}_{\mathrm{lcm}(n_1,\ldots,n_k)}$, which in general has size exponential in $\sum_{i\in\{1,\ldots,k\}} n_i$. □

5 Complexity Analysis of Bottom Up MPST

We now start the analysis of *bottom-up* MPST, where contexts of local types are checked for properties such as safety, liveness and deadlock-freedom. This guarantees that their corresponding processes satisfy the same properties [18].

Define a *typing context* $(\Delta, \Delta', \ldots)$ as $\Delta ::= \emptyset \mid \Delta, \mathsf{p}{:}\mathsf{T}$. We often write Δ as $\prod_{i\in I} \mathsf{p}_i{:}\mathsf{T}_i$ as shorthand for $\{\mathsf{p}_1{:}\mathsf{T}_1, \ldots, \mathsf{p}_n{:}\mathsf{T}_n\}$ where $\prod_{i\in\emptyset} \mathsf{p}_i{:}\mathsf{T}_i = \emptyset$. To define safety, deadlock-freedom and liveness, we first recall the transition relation of typing contexts from [18, Definition 2.8].

Definition 5.1 (LTS of typing environments, Definition 2.8 in [18])**.** We define a *labelled transition relation* $\xrightarrow{\ell}$ over typing contexts by the following rules:

$$\frac{}{\mathsf{p} : \mathsf{q}!\lfloor S\rfloor; \mathsf{T} \xrightarrow{\mathsf{pq}!\lfloor S\rfloor} \mathsf{p} : \mathsf{T}}\ [\textsc{Out}] \qquad \frac{}{\mathsf{p} : \mathsf{q}?\lfloor S\rfloor; \mathsf{T} \xrightarrow{\mathsf{pq}?\lfloor S\rfloor} \mathsf{p} : \mathsf{T}}\ [\textsc{In}] \qquad \frac{k \in I}{\mathsf{p} : \mathsf{q}\&\{l_i : \mathsf{T}_i\}_{i\in I} \xrightarrow{\mathsf{pq}\&l_k} \mathsf{p} : \mathsf{T}_k}\ [\&]$$

$$\frac{k \in I}{\mathsf{p} : \mathsf{q}\oplus\{l_i : \mathsf{T}_i\}_{i\in I} \xrightarrow{\mathsf{pq}\oplus l_k} \mathsf{p} : \mathsf{T}_k}\ [\oplus] \qquad \frac{\Delta_1 \xrightarrow{\mathsf{pq}!\lfloor S\rfloor} \Delta_1' \qquad \Delta_2 \xrightarrow{\mathsf{qp}?\lfloor S\rfloor} \Delta_2'}{\Delta_1, \Delta_2 \xrightarrow{\mathsf{pq}} \Delta_1', \Delta_2'}\ [\textsc{Msg}]$$

$$\frac{\Delta_1 \xrightarrow{\mathsf{pq}\oplus l} \Delta_1' \qquad \Delta_2 \xrightarrow{\mathsf{qp}\&l} \Delta_2'}{\Delta_1, \Delta_2 \xrightarrow{\mathsf{pq}:l} \Delta_1', \Delta_2'}\ [\textsc{Bra}] \qquad \frac{\Delta \xrightarrow{\ell} \Delta'}{\Delta, \mathsf{p} : \mathsf{T} \xrightarrow{\ell} \Delta', \mathsf{p} : \mathsf{T}}\ [\textsc{Comp}] \qquad \frac{\mathsf{p} : \mathsf{T}[\mu\mathbf{t}.\mathsf{T}/\mathbf{t}] \xrightarrow{\ell} \Delta'}{\mathsf{p} : \mu\mathbf{t}.\mathsf{T} \xrightarrow{\ell} \Delta'}\ [\mu]$$

Define the *reduction* $\Delta \to \Delta'$ if $\Delta \xrightarrow{\mathsf{pq}} \Delta'$ or $\Delta \xrightarrow{\mathsf{pq}:l} \Delta'$.

Definition 5.2 (Safety property, Definition 4.1 in [18])**.** We say Δ is a *safe state* if

- $\Delta \xrightarrow{\mathsf{pq}!\lfloor S\rfloor}$ and $\Delta \xrightarrow{\mathsf{qp}?\lfloor S'\rfloor}$ implies $\Delta \xrightarrow{\mathsf{pq}}$; and
- $\Delta \xrightarrow{\mathsf{pq}\oplus l}$ and $\Delta \xrightarrow{\mathsf{qp}\&l'}$ implies $\Delta \xrightarrow{\mathsf{pq}:l}$.

Given a context Δ, ϕ is a *safety property* if, for all Δ such that $\phi(\Delta)$, we have:

- Δ is a safe state;
- $\Delta = \Delta', \mathsf{p} : \mu\mathbf{t}.\mathsf{T}$ implies $\phi(\Delta', \mathsf{p} : \mathsf{T}[\mu\mathbf{t}.\mathsf{T}/\mathbf{t}])$; and
- $\Delta \to \Delta'$ implies $\phi(\Delta')$.

We call Δ *safe* if $\phi(\Delta)$ holds for some safety property ϕ.

A corollary of this definition is that Δ is safe if and only if Δ' is safe for all $\Delta \to^* \Delta'$.

Definition 5.3 (Deadlock-freedom, Figure 5(2) in [18])**.** Δ is deadlock-free if $\Delta \to^* \Delta' \not\to$ implies $\mathsf{unfold}(\mathsf{T}_\mathsf{p}) = \mathtt{end}$ for all $\mathsf{p} : \mathsf{T}_\mathsf{p} \in \Delta'$.

Definition 5.4 (Fair and Live Paths, Definition 17 in [2])**.** A *path* is a sequence of contexts $\mathcal{P} ::= (\Delta_i)_{i \in N}$, where $N = \{0, 1, 2, \dots\}$ is a (finite or infinite) set of consecutive natural numbers, and $\Delta_i \to \Delta_{i+1}$ for all i. Then a path is *fair* if, for all $i \in N$, $\Delta_i \xrightarrow{\mathsf{pq}}$ implies $\exists j \geq i.\, \Delta_j \xrightarrow{\mathsf{pq}} \Delta_{j+1}$, and $\Delta_i \xrightarrow{\mathsf{pq}:l}$ implies $\exists l', j$ such that $j \geq i$ and $\Delta_j \xrightarrow{\mathsf{pq}:l} \Delta_{j+1}$.

A path is *live* if, for all $i \in N$:

- $\Delta_i \xrightarrow{\mathsf{pq}![S]}$ implies $\exists j \geq i.\, \Delta_j \xrightarrow{\mathsf{pq}} \Delta_{j+1}$;
- $\Delta_i \xrightarrow{\mathsf{pq}?[S]}$ implies $\exists j \geq i.\, \Delta_j \xrightarrow{\mathsf{qp}} \Delta_{j+1}$;
- $\Delta_i \xrightarrow{\mathsf{pq}\oplus l}$ implies $\exists j \geq i.\, \Delta_j \xrightarrow{\mathsf{pq}:l} \Delta_{j+1}$; and
- $\Delta_i \xrightarrow{\mathsf{pq}\&l}$ implies $\exists l', j$ such that $j \geq i$ and $\Delta_j \xrightarrow{\mathsf{qp}:l'} \Delta_{j+1}$.

Definition 5.5 (Liveness, Definition 17 in [2])**.** Δ is *live*, denoted $\mathsf{live}(\Delta)$, if $\Delta \to^* \Delta'$ implies that all paths starting with Δ' that are fair are also live.

We check that a process is well-typed using [T-Sess] in Figure 10. Under the bottom-up methodology, to guarantee a good property φ for $\mathcal{M}$, one needs to *additionally* check Δ satisfies the corresponding property $\varphi(\Delta)$.

Theorem 5.6 (Deadlock-freedom and liveness, Theorem 5.15 in [18])**.** Suppose $\vdash \mathcal{M} \triangleright \Delta$ with $\varphi \in \{\mathsf{df}, \mathsf{live}\}$, and $\varphi(\Delta)$. Then $\mathcal{M}$ is φ.

5.1 Complexity of Checking Safety, Deadlock-Freedom and Liveness

Unfortunately, checking the three aforementioned properties is PSPACE-hard. To prove this, we reduce it to the *quantified Boolean formula* (QBF) problem, which is known to be PSPACE-complete [20, Theorem 8.9]. Given a formula in QBF $\mathcal{F} = \mathcal{Q}_1 v_1 \dots \mathcal{Q}_n v_n \mathcal{F}'$, $\mathcal{Q}_i \in \{\exists, \forall\}$, with $\mathcal{F}'$ in conjunctive normal form: $\mathcal{F}' := \bigwedge_{i=1}^{m} C_i$, $C_i = (L_{i1} \vee L_{i2} \vee L_{i3})$, the QBF problem asks whether $\mathcal{F}$ is true.

Our construction is a typing context Δ_{init}, defined in Figure 11, that computes the truth value of $\mathcal{F}$. It contains one participant for each variable (p_i) and clause (r_i), as well as the "controller" (s). Participant p_i computes the truth value of the formula $\mathcal{Q}_i v_i \ldots \mathcal{Q}_n v_n \mathcal{F}'$, and r_i computes the truth value of the clauses C_i onwards. Participant s queries p_1 and enters an undesirable state T_{bad} if $\mathcal{F}$ is false, otherwise the process repeats indefinitely.

Importantly, we construct Δ_{init} such that its reduction paths are *deterministic*: this ensures that all possible behaviours of the context are described by the above explanation, and no unexpected behaviours occur. Furthermore, the computation must not contain any violations of the properties we are checking. For safety, this follows from the reduction path being *safe*, and deadlock-freedom is guaranteed by the existence of a deterministic path.

Definition 5.7 (Deterministic, safe reduction paths)**.** $\Delta \xrightarrow{\text{det}} \Delta'$ if $\Delta \to \Delta_1$ and $\Delta \to \Delta_2$ implies $\Delta_1 = \Delta_2 = \Delta'$. For safe states Δ, define $\Delta \xrightarrow[\text{safe}]{\text{det}} \Delta'$ if $\Delta \xrightarrow{\text{det}} \Delta'$.

We write $\xrightarrow{\text{det}}{}^*$ for the reflexive and transitive closure of $\xrightarrow{\text{det}}$. We write $\Delta \xrightarrow[\text{safe}]{\text{det}}{}^* \Delta'$ if Δ is a safe state and Δ' can be reached from Δ by a sequence of $\xrightarrow[\text{safe}]{\text{det}}$ reductions.

By specifying the reductions of Δ_{init} (Lemmas 5.8, 5.9, 5.10), we can formally prove properties of the behaviour as intuitively described above. We prove that, if $\mathcal{F}$ is true, then Δ_{init} reduces safely and deterministically (in multiple steps) back to itself. Otherwise, Δ_{init} reduces to a state where s has the undesirable type T_{bad}.

Notation. We draw the type graphs of each participant in the composition in Figure 12. We label the nodes of the type graphs to more clearly prove the reduction, and we will use the names of graph nodes and their corresponding types interchangeably.

We use the following notation to refer to the reachable compositions of the participants. Let $(\mathsf{s} : \mathsf{T}^*_{\mathsf{s}}, \mathsf{p}_1 : \mathsf{T}^*_{\mathsf{p}_1}, \ldots, \mathsf{p}_n : \mathsf{T}^*_{\mathsf{p}_n}, \mathsf{r}_1 : \mathsf{T}^*_{\mathsf{r}_1}, \ldots, \mathsf{r}_{m+1} : \mathsf{T}^*_{\mathsf{r}_{m+1}})$ refer to the context $\prod_{\mathsf{p} \in \mathbb{P}} \mathsf{p} : \mathsf{T}^*_{\mathsf{p}}$. We may omit some or all of the prefixes $\mathsf{p} :$ if it is clear from the context.

For an assignment of truth values $\mathcal{A} : \{v_1, \ldots, v_k\} \to \{0, 1\}$ with $k \leq n$, define $V^i = T$ if $\mathcal{A}(v_i) = 1$, and $V^i = F$ otherwise.

Define $(\mathsf{T}^*_{\mathsf{s}}, \mathcal{A}, \mathsf{T}^*_{\mathsf{p}_{k+1}}, \ldots, \mathsf{T}^*_{\mathsf{r}_n}) = (\mathsf{T}^*_{\mathsf{s}}, V^1, \ldots, V^k, \mathsf{T}^*_{\mathsf{p}_{k+1}}, \ldots, \mathsf{T}^*_{\mathsf{r}_n})$, and $[\mathcal{A}] = (W, \mathcal{A}, I_1, I, \ldots, I)$. $\Delta_{\mathsf{init}} = (I, \ldots, I)$ is the starting context.

By analysing the reduction paths of Δ_{init}, we can prove the following lemmas, which formalise the above intuition (each one builds on the last).

Lemma 5.8. [Propagation of $\mathsf{query}_{\mathsf{p}_i}$] Let $\Delta = (W, \mathcal{A}, \mathsf{r}_1 : R, \ldots, \mathsf{r}_{k-1} : R, \mathsf{r}_k : \mathsf{T}, I, \ldots, I)$ for some type T. Let $\mathsf{T} = r_{k-1} \oplus \{\mathsf{query}_{p_i} : \mathsf{T}'\}$, such that $\mathsf{T}' = r_{k-1} \& \{\mathsf{yes} : \mathsf{T}_{\mathsf{yes}}, \mathsf{no} : \mathsf{T}_{\mathsf{no}}\}$. Then:

- If $\mathcal{A}(p_i) = 1$, then $\Delta \xrightarrow[\text{safe}]{\text{det}}{}^* (W, \mathcal{A}, \mathsf{r}_1 : R, \ldots, \mathsf{r}_{k-1} : R, \mathsf{T}_{\mathsf{yes}}, I, \ldots, I)$.
- If $\mathcal{A}(p_i) = 0$, then $\Delta \xrightarrow[\text{safe}]{\text{det}}{}^* (W, \mathcal{A}, \mathsf{r}_1 : R, \ldots, \mathsf{r}_{k-1} : R, \mathsf{T}_{\mathsf{no}}, I, \ldots, I)$.

Lemma 5.9. [Reduction of r_i] For $\mathcal{A} : \{v_1, \ldots, v_n\} \to \{0, 1\}$:

- $(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : I_1, I, \ldots, I) \xrightarrow[\text{safe}]{\text{det}}{}^* (W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : T_r, I, \ldots, I)$ if $\mathcal{A} \models \bigwedge_{i=k}^{m} C_i$.
- $(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : I_1, I, \ldots, I) \xrightarrow[\text{safe}]{\text{det}}{}^* (W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : F_r, I, \ldots, I)$ if $\mathcal{A} \not\models \bigwedge_{i=k}^{m} C_i$.

Lemma 5.10. [Reduction of p_i] For $\mathcal{A} : \{v_1, \ldots, v_k\} \to \{0, 1\}$:

- $[\mathcal{A}] \xrightarrow[\text{safe}]{\text{det}}{}^* (W, \mathcal{A}, T_r, I, \ldots, I)$ if $\mathcal{A} \models \mathcal{Q}_{k+1} v_{k+1} \ldots \mathcal{Q}_n v_n \mathcal{F}'$.
- $[\mathcal{A}] \xrightarrow[\text{safe}]{\text{det}}{}^* (W, \mathcal{A}, F_r, I, \ldots, I)$ if $\mathcal{A} \not\models \mathcal{Q}_{k+1} v_{k+1} \ldots \mathcal{Q}_n v_n \mathcal{F}'$.

Theorem 5.11. Recall that $[\emptyset] = (\mathsf{s}{:}W, \mathsf{p}_1{:}I_1, \mathsf{p}_2{:}I, \ldots, \mathsf{r}_{m+1}{:}I)$. Then:

- $\Delta_{\mathsf{init}} \xrightarrow[\text{safe}]{\text{det}} [\emptyset]$;
- If $\models \mathcal{F}$, then $[\emptyset] \xrightarrow[\text{safe}]{\text{det}}{}^* \Delta_{\mathsf{init}}$; and
- If $\not\models \mathcal{F}$, then $[\emptyset] \xrightarrow[\text{safe}]{\text{det}}{}^* (\mathsf{s}{:}\mathsf{T}_{\text{bad}}, \mathsf{p}_1{:}I, \ldots, \mathsf{r}_{m+1}{:}I)$.

Proof. The first statement is true by inspection of the transition system. The latter two statements are true by Lemma 5.10 with $k = 0$. □

As safety and deadlock-freedom are reachability properties, we only need to exhibit a T_{bad} that violates the properties.

Theorem 5.12. Checking for safety, deadlock-freedom and liveness is PSPACE-hard.

Proof. For safety, set $\mathsf{T}_{\text{bad}} = \mathsf{p}_1 \& \{\texttt{unsafe} : \texttt{end}\}$, which induces an unsafe state. For deadlock-freedom and liveness, set $\mathsf{T}_{\text{bad}} = \texttt{end}$, which induces a deadlocking state (and thus violates liveness). □

We now prove that checking these properties are in PSPACE, to complete the proof of PSPACE-completeness. First, we bound the number of reachable contexts Δ' such that $\Delta \to^* \Delta'$ by the product of the sizes of the types, and thus is exponential in $(\sum_{i \in I} |\mathsf{T}_i|)$.

Theorem 5.13. Let $\Delta = \prod_{i \in I} \mathsf{p}_i : \mathsf{T}_i$. Then $|\{\Delta' \mid \Delta \to^* \Delta'\}| \leq \prod_{i \in I} |\mathsf{T}_i|$.

Proof. By showing that the reachable states $\mathsf{p}_i : \mathsf{T}_i'$ from each $\mathsf{p}_i : \mathsf{T}_i$, are subformulas of T_i. □

This gives rise to a nondeterminstic polynomial-space algorithm for checking the complement of safety, where all states at a distance up to this exponential bound are nondeterministically visited and checked for states that are not safe (Algorithm 1). We use the fact that co-NPSPACE = PSPACE [20, Theorem 8.5]. Deadlock-freedom is similar; instead, we check for deadlocked states.

Theorem 5.14. Checking for safety and deadlock-freedom is in PSPACE.

Algorithm 1 A nondeterministic polynomial-space algorithm for checking if a process is not safe.

```
1:  function NotSafe(Δ)                          ▷ Typing context Δ = ∏_{i∈I} p_i : T_i
2:      M ← ∏_{i∈I} |T_i|
3:      n ← 0                                    ▷ Current path length
4:      Δ' ← Δ
5:      loop
6:          if Δ' is not a safe state then
7:              return accept
8:          end if
9:          n ← n + 1
10:         if n > M then                        ▷ No violation of safety found in M steps
11:             return reject
12:         end if
13:         nondeterministically choose Δ'' such that Δ' → Δ''
14:         Δ' ← Δ''
15:     end loop
16: end function
```

$$\frac{\Delta \xrightarrow{\mathsf{pq}?[S]} \Delta'}{\mathsf{p?q} \in \mathsf{barbs}(\Delta)}\ [\textsc{Barb-In}] \qquad \frac{\Delta \xrightarrow{\mathsf{pq}![S]} \Delta'}{\mathsf{p!q} \in \mathsf{barbs}(\Delta)}\ [\textsc{Barb-Out}]$$

$$\frac{\Delta \xrightarrow{\mathsf{pq}\&l} \Delta'}{\mathsf{p\&q} \in \mathsf{barbs}(\Delta)}\ [\textsc{Barb-Bra}] \qquad \frac{\Delta \xrightarrow{\mathsf{pq}\oplus l} \Delta'}{\mathsf{p}\oplus\mathsf{q}[l] \in \mathsf{barbs}(\Delta)}\ [\textsc{Barb-Sel}]$$

$$\frac{\Delta \xrightarrow{\mathsf{pq}} \Delta'}{\mathsf{q?p} \in \mathsf{observations}(\Delta, \Delta')}\ [\textsc{Obs-In}] \qquad \frac{\Delta \xrightarrow{\mathsf{pq}} \Delta'}{\mathsf{p!q} \in \mathsf{observations}(\Delta, \Delta')}\ [\textsc{Obs-Out}]$$

$$\frac{\Delta \xrightarrow{\mathsf{pq}:l} \Delta'}{\mathsf{q\&p} \in \mathsf{observations}(\Delta, \Delta')}\ [\textsc{Obs-Bra}] \qquad \frac{\Delta \xrightarrow{\mathsf{pq}:l} \Delta'}{\mathsf{p}\oplus\mathsf{q}[l] \in \mathsf{observations}(\Delta, \Delta')}\ [\textsc{Obs-Sel}]$$

Figure 9: Rules for $\mathsf{barbs}(\Delta)$ and $\mathsf{observations}(\Delta, \Delta')$.

The proof that checking for liveness is in PSPACE is more involved: first, we define an equivalent characterisation of liveness. Live paths are characterised by *barbs*, which are the offered transitions in some context, and *observations*, which are the transitions that are matched in some reduction. A path is not live if there is some barb which is never observed. Counterwitnesses (Definition 5.16) are paths which have a barb that is never observed. A counterwitness exists iff the context is not live.

Definition 5.15. The set of *barbs* of Δ and the set of *observations* of Δ, Δ' are defined in Figure 9. For paths $\mathcal{P} = (\Delta_i)_{i \in N}$, define $\mathsf{barbs}(\mathcal{P}) = \bigcup_{i \in N} \mathsf{barbs}(\Delta_i)$ and $\mathsf{observations}(\mathcal{P}) = \bigcup_{i,i+1 \in N} \mathsf{observations}(\Delta_i, \Delta_{i+1})$.

Definition 5.16. A path $(\Delta_i)_{i \in N}$ is a counterwitness of Δ if $\Delta \to^* \Delta_0$ and $\Delta_i \xrightarrow{\ell_i} \Delta_{i+1}$ and:

- There exists $k \in N$ and $a \in \mathsf{barbs}(\Delta_k)$ such that $a \notin \mathsf{observations}((\Delta_i)_{i \geq k})$; and
- For all $k \in N$, $\{\ell_i \mid i \geq k\} = \{\ell \mid \exists i \geq k.\ \Delta_i \xrightarrow{\ell}, \ell \in \{\mathsf{pq}, \mathsf{pq} : l\}\}$.

We give a sufficient bound on the length and periodicity of a counterwitness. The proof involves shortcutting certain cycles that keep the counterwitness fair and not live.

Lemma 5.17. Let $\Delta = \prod_{i \in I} \mathsf{p}_i : \mathsf{T}_i$. Let $n = \sum_{i \in I} |\mathsf{T}_i|$ and $M = (2n + 2) \cdot \prod_{i \in I} |\mathsf{T}_i|$. If Δ is not live, then it has a finite counterwitness of size $\leq M$ or an infinite counterwitness of the form $\mathcal{P}_1 \mathcal{P}_2^\omega$ with $|\mathcal{P}_1|, |\mathcal{P}_2| \leq M$.

Theorem 5.18. Checking for liveness is in PSPACE.

Proof. Using the above bound on the length of a counterwitness, we can give a nondeterministic algorithm that checks for the complement of liveness (Algorithm 2), by checking all paths in the form specified by Lemma 5.17.

At the end of the first loop, U is the set of reductions ℓ such that $\Delta_i \xrightarrow{\ell}$ but there is no $\Delta_j \in \mathcal{P}_1$, $j > i$ such that $\Delta_j \xrightarrow{\ell} \Delta_{j+1}$. $B = \mathsf{barbs}(\Delta_k)$ and $O = \mathsf{observations}((\Delta_i)_{i \geq k})$ in $\mathcal{P}_1$. Thus we are checking if $\mathcal{P}_1$ is a counterwitness with the given value of k.

At the end of the second loop, we add all reductions $\Delta_i \xrightarrow{\ell}$ to U and all observed transitions to T. After the loop, we check that all transitions were observed and not all barbs were matched, so we are checking for a counterwitness for $\mathcal{P}_1 \mathcal{P}_2^\omega$. Thus the algorithm is correct.

Note that, the paths $\mathcal{P}_1$ and $\mathcal{P}_2$ do not need to be stored explicitly. Thus the algorithm is polynomial-space. □

Using these results, we have the following main result.

Corollary 5.18.1. Checking for safety, deadlock-freedom and liveness is PSPACE-complete.

6 Conclusion

To our best knowledge, we gave the first full complexity analysis of top-down and bottom-up procedures of MPST type-checking. A summary of our results is Figure 1.

Specifically, we analysed the complexity of three different kinds of projection. We gave the first complexity analysis of plain, full and TBC merging, and showed that a binary search tree representation of branching global types improves the complexity of full merging, using a small-to-large merging technique. Albeit limited, the projections defined over the inductive syntax stay low in complexity, and are easy to implement in tools. Furthermore, we extended the existing literature on subtyping of binary session types to the

Algorithm 2 A nondeterministic polynomial-space algorithm for checking if a process is not live.

function NotLive(Δ) ▷ Typing context $\Delta = \prod_{i \in I} \mathsf{p}_i : \mathsf{T}_i$
  $n \leftarrow \sum_{i \in I} |\mathsf{T}_i|$
  $M \leftarrow (2n + 2) \cdot \prod_{i \in I} |\mathsf{T}_i|$
  $U \leftarrow \emptyset$ ▷ unfair transitions
  $T \leftarrow \emptyset$ ▷ observed transitions in cycle
  $B \leftarrow \emptyset$ ▷ barbs of Δ_k
  $O \leftarrow \emptyset$ ▷ observed barbs
  $\Delta_0 \leftarrow$ some nondeterministically chosen context reachable from Δ
  for $\Delta', \mathsf{prev}(\Delta')$ in some nondeterministically chosen path $\mathcal{P}_1$ of length $\leq M$ starting at Δ_0 **do**
    $U \leftarrow U \setminus \{\ell'\}$ where $\mathsf{prev}(\Delta') \xrightarrow{\ell'} \Delta'$
    if Δ_k has not been set **then**
      nondeterministically choose to set $\Delta_k \leftarrow \Delta'$ and $B \leftarrow \mathsf{barbs}(\Delta')$
    else
      $O \leftarrow O \cup \mathsf{observations}(\mathsf{prev}(\Delta'), \Delta')$
    end if
    $U \leftarrow U \cup \{\ell \mid \Delta' \xrightarrow{\ell}\}$
  end for
  if Δ_k is set and $U = \emptyset$ and $B \setminus O \neq \emptyset$ **then return** accept ▷ finite case
  end if
  for $\mathsf{prev}(\Delta'), \Delta'$ in some nondeterministically chosen nonempty path $\mathcal{P}_2$ of length $\leq M$ starting at the last element of $\mathcal{P}_1$ **do**
    $T \leftarrow T \cup \{\ell'\}$ where $\mathsf{prev}(\Delta') \xrightarrow{\ell'} \Delta'$
    if Δ_k has not been set **then**
      nondeterministically choose to set $\Delta_k \leftarrow \Delta'$ and $B \leftarrow \mathsf{barbs}(\Delta')$
    else
      $O \leftarrow O \cup \mathsf{observations}(\mathsf{prev}(\Delta'), \Delta')$
    end if
    $U \leftarrow U \cup \{\ell \mid \Delta' \xrightarrow{\ell}\}$
  end for
  if $\mathcal{P}_2$ is not a cycle **then return** reject
  end if
  if Δ_k is set and $U \setminus T = \emptyset$ and $B \setminus O \neq \emptyset$ **then return** accept ▷ cycle case
  else return reject
  end if
end function

MPST setting. We gave the first sound and complete minimum type inference system for MPST, which involves a subset construction to constrain the behaviours of each subtype, and argued that the complexity is exponential. Finally, we proved that the complexity of checking three important properties of typing contexts is PSPACE-complete, by constructing a context that calculates the truth value of an arbitrary quantified Boolean formula, and proved its correctness.

References

[1] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Open compliance in multiparty sessions. In Silvia Lizeth Tapia Tarifa and José Proença, editors, *Formal Aspects of Component Software - 18th International Conference, FACS 2022, Virtual Event, November 10-11, 2022, Proceedings*, volume 13712 of *Lecture Notes in Computer Science*, pages 222–243. Springer, 2022. doi:10.1007/978-3-031-20872-0_13.

[2] Adam Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised Multiparty Session Types with Crash-Stop Failures. In *33rd International Conference on Concurrency Theory*, volume 243 of *LIPIcs*, pages 35:1–35:25. Dagstuhl, 2022. doi:10.4230/LIPIcs.CONCUR.2022.35.

[3] Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised multiparty session types with crash-stop failures (technical report), 2022. URL: https://arxiv.org/abs/2207.02015, doi:10.48550/ARXIV.2207.02015.

[4] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *Proceedings of CONCUR 2008*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[5] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Event structure semantics for multiparty sessions. *J. Log. Algebraic Methods Program.*, 131:100844, 2023. URL: https://doi.org/10.1016/j.jlamp.2022.100844, doi:10.1016/J.JLAMP.2022.100844.

[6] 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), June 2017. doi:10.23638/LMCS-13(2:12)2017.

[7] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS*, 26:238–302, 2015.

[8] Francesco Dagnino, Paola Giannini, and Mariangiola Dezani-Ciancaglini. Deconfined global types for asynchronous sessions. *Log. Methods Comput. Sci.*, 19(1), 2023. URL: https://doi.org/10.46298/lmcs-19(1:3)2023, doi:10.46298/LMCS-19(1:3)2023.

[9] Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *Proceedings of CONCUR 2011*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011.

[10] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, November 2005. doi:10.1007/s00236-005-0177-z.

[11] Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2016. doi:10.1007/978-3-319-30936-1_5.

[12] Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 104:127–173, April 2019. doi:10.1016/j.jlamp.2018.12.002.

[13] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFB0053567.

[14] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY,

USA, January 2008. Association for Computing Machinery. doi:10.1145/1328438.1328472.

[15] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

[16] Julien Lange and Nobuko Yoshida. Characteristic Formulae for Session Types (extended version), October 2015. arXiv:1510.06879.

[17] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, January 2002.

[18] Alceste Scalas and Nobuko Yoshida. Less is more: Multiparty session types revisited. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, January 2019. doi:10.1145/3290343.

[19] Gil Silva, Andreia Mordido, and Vasco T. Vasconcelos. Subtyping Context-Free Session Types. volume 279, pages 11:1–11:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CONCUR.2023.11.

[20] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, Australia, third edition. edition, 2012/2013.

[21] Dawit Tirore, Jesper Bengtson, and Marco Carbone. A Sound and Complete Projection for Global Types. In *ITP'23*, pages 19 pages, 927363 bytes. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ITP.2023.28.

[22] Thien Udomsrirungruang and Nobuko Yoshida. Three Subtyping Algorithms for Binary Session Types and their Complexity Analyses. In *15th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*, volume 401 of *Electronic Proceedings in Theoretical Computer Science*, pages 49–60. Open Publishing Association, 2024. doi:10.48550/arXiv.2404.05480.

[23] Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic pi-calculus. In Eike Best, editor, *CONCUR'93*, Lecture Notes in Computer Science, pages 524–538, Berlin, Heidelberg, 1993. Springer. doi:10.1007/3-540-57208-2_36.

[24] Nobuko Yoshida. Programming language implementations with multiparty session types. In Frank S. de Boer, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Eduard Kamburjan, editors, *Active Object Languages: Current Research Trends*, volume 14360 of *Lecture Notes in Computer Science*, pages 147–165. Springer, 2024. doi:10.1007/978-3-031-51060-1_6.

[25] Nobuko Yoshida and Lorenzo Gheri. A Very Gentle Introduction to Multiparty Session Types. In *Distributed Computing and Internet Technology: 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9–12, 2020, Proceedings*, pages 73–93, Berlin, Heidelberg, January 2020. Springer-Verlag. doi:10.1007/978-3-030-36987-3_5.

[26] Nobuko Yoshida and Ping Hou. Less is more revisit, 2024. Accepted by Cliff B. Jones Festschrift Proceeding. arXiv:2402.16741.

[27] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble Protocol Language. In *8th International Symposium on Trustworthy Global Computing*, volume 8358 of *LNCS*, pages 22–41. Springer, 2013.

A Appendix for Checking Safety, Liveness and Deadlock-freedom

We draw the type graphs of each participant in the composition in Figure 12. We label the nodes of the type graphs to more clearly prove the reduction, and we will use the names of graph nodes and their corresponding types interchangeably.

Processes: $\Gamma \vdash P : \mathsf{T}$

$$\frac{\Gamma \vdash P : \mathsf{T} \qquad \mathsf{T} \leq \mathsf{T}'}{\Gamma \vdash P : \mathsf{T}'} \text{ [T-Sub]} \qquad \frac{}{\Gamma \vdash \mathbf{0} : \mathsf{end}} \text{ [T-Inact]}$$

$$\frac{\Gamma, X : \mathsf{T} \vdash P : \mathsf{T}}{\Gamma \vdash \mu X.P : \mathsf{T}} \text{ [T-Rec]} \qquad \frac{X : \mathsf{T} \in \Gamma}{\Gamma \vdash X : \mathsf{T}} \text{ [T-Var]}$$

$$\frac{\Gamma, x : S \vdash P : \mathsf{T}}{\Gamma \vdash \mathsf{p}?(x).P : \mathsf{p}?[S]; \mathsf{T}} \text{ [T-In]} \qquad \frac{\Gamma \vdash P : \mathsf{T} \qquad \Gamma \vdash \mathsf{e} : S}{\Gamma \vdash \mathsf{p}!\langle \mathsf{e} \rangle.P : \mathsf{p}![S]; \mathsf{T}} \text{ [T-Out]}$$

$$\frac{\forall i \in I.\, \Gamma \vdash P_i : \mathsf{T}_i}{\Gamma \vdash \mathsf{p} \triangleright \{l_i : P_i\}_{i \in I} : \mathsf{p} \& \{l_i : \mathsf{T}_i\}_{i \in I}} \text{ [T-Bra]} \qquad \frac{\Gamma \vdash P : \mathsf{T}}{\Gamma \vdash \mathsf{p} \triangleleft l.P : \mathsf{p} \oplus \{l : \mathsf{T}\}} \text{ [T-Sel]}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathtt{bool} \qquad \Gamma \vdash P : \mathsf{T} \qquad \Gamma \vdash P' : \mathsf{T}}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ P' : \mathsf{T}} \text{ [T-Cond]}$$

Expressions: $\Gamma \vdash P : \mathsf{T}$

$$\frac{x : S \in \Gamma}{\Gamma \vdash x : S} \text{ [T-SortVar]} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathtt{bool}} \text{ [T-True]} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathtt{bool}} \text{ [T-False]}$$

$$\frac{}{\Gamma \vdash \mathsf{n} : \mathtt{nat}} \text{ [T-Nat]} \qquad \frac{}{\Gamma \vdash \mathsf{i} : \mathtt{int}} \text{ [T-Int]} \qquad \frac{\Gamma \vdash \mathsf{e} : \mathtt{int}}{\Gamma \vdash \mathtt{neg}(\mathsf{e}) : \mathtt{int}} \text{ [T-Neg]}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathtt{bool}}{\Gamma \vdash \neg \mathsf{e} : \mathtt{bool}} \text{ [T-Not]} \qquad \frac{\Gamma \vdash \mathsf{e}_1 : \mathtt{bool} \qquad \Gamma \vdash \mathsf{e}_2 : \mathtt{bool}}{\Gamma \vdash \mathsf{e}_1 \vee \mathsf{e}_2 : \mathtt{bool}} \text{ [T-Or]}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : S \qquad \Gamma \vdash \mathsf{e}_2 : S}{\Gamma \vdash \mathsf{e}_1 \oplus \mathsf{e}_2 : S} \text{ [T-Nondet]}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \mathtt{int} \qquad \Gamma \vdash \mathsf{e}_2 : \mathtt{int}}{\Gamma \vdash \mathsf{e}_1 + \mathsf{e}_2 : \mathtt{int}} \text{ [T-Add]}$$

Top-down

$$\frac{\forall i \in I.\ \vdash P_i : \mathsf{G}\upharpoonright_{\mathsf{p}_i} \qquad \mathtt{pt}\{\mathsf{G}\} \subseteq \{\mathsf{p}_i \mid i \in I\}}{\vdash \prod_{i \in I} \mathsf{p}_i :: P_i \triangleright \mathsf{G}} \text{ [T-Sess]}$$

Bottom-up

$$\frac{\forall i \in I.\, \Gamma \vdash P_i : \mathsf{T}_i \qquad \mathsf{safe}(\prod_{i \in I} \mathsf{p}_i : \mathsf{T}_i)}{\vdash \prod_{i \in I} \mathsf{p}_i :: P_i \triangleright \prod_{i \in I} \mathsf{p}_i : \mathsf{T}_i} \text{ [T-safe]}$$

Figure 10: Typing rules for processes.

$$\mathsf{T}_{\mathsf{s}} = \mu \mathbf{t}.\mathsf{p}_1![\mathtt{int}];\mathsf{p}_1\&\{\mathsf{tt} : \mathbf{t}, \mathsf{ff} : \mathsf{T}_{\mathrm{bad}}\}$$

$$\mathsf{T}_{\mathsf{p}_i} = \mu \mathbf{t}_1.\mathsf{p}_{i-1}?[\mathtt{int}];\mathsf{p}_{i+1}![\mathtt{int}];$$
$$\mu \mathbf{t}_2.\mathsf{p}_{i+1}\&\left\{\begin{array}{ll}
\mathsf{query}_t : \mathsf{p}_{i-1}\oplus\left\{\mathsf{query}_t : \mathsf{p}_{i-1}\&\left\{\begin{array}{l}\mathsf{yes} : \mathsf{p}_{i+1}\oplus\{\mathsf{yes} : \mathbf{t}_2\},\\ \mathsf{no} : \mathsf{p}_{i+1}\oplus\{\mathsf{no} : \mathbf{t}_2\}\end{array}\right\}\right\}, & (t \neq \mathsf{p}_i)\\
\mathsf{query}_{\mathsf{p}_i} : \mathsf{p}_{i+1}\oplus\{\mathsf{no} : \mathbf{t}_2\}, & \\
\mathrm{resolve}(\mathcal{Q}_i) : \mathsf{p}_{i-1}\oplus\{\mathrm{resolve}(\mathcal{Q}_i) : \mathbf{t}_1\}, & \\
\mathrm{resolve}^\dagger(\mathcal{Q}_i) : \mathsf{p}_{i+1}![\mathtt{int}]; & \\
\quad \mu \mathbf{t}_3.\mathsf{p}_{i+1}\&\left\{\begin{array}{ll}
\mathsf{query}_t : \mathsf{p}_{i-1}\oplus\left\{\mathsf{query}_t : \mathsf{p}_{i-1}\&\left\{\begin{array}{l}\mathsf{yes} : \mathsf{p}_{i+1}\oplus\{\mathsf{yes} : \mathbf{t}_3\},\\ \mathsf{no} : \mathsf{p}_{i+1}\oplus\{\mathsf{no} : \mathbf{t}_3\}\end{array}\right\}\right\}, & (t \neq \mathsf{p}_i)\\
\mathsf{query}_{\mathsf{p}_i} : \mathsf{p}_{i+1}\oplus\{\mathsf{yes} : \mathbf{t}_3\}, & \\
\mathsf{ff} : \mathsf{p}_{i-1}\oplus\{\mathsf{ff} : \mathbf{t}_1\}, & \\
\mathsf{tt} : \mathsf{p}_{i-1}\oplus\{\mathsf{tt} : \mathbf{t}_1\} &
\end{array}\right.
\end{array}\right.$$

$$\mathsf{T}_{\mathsf{r}_i} = \mu \mathbf{t}_1.\mathsf{r}_{i-1}?[\mathtt{int}];\mathsf{r}_{i+1}![\mathtt{int}]; \qquad (i \leq m)$$
$$\mathsf{r}_{i+1}\&\left\{\begin{array}{l}
\mathsf{query}_t : \mathsf{r}_{i-1}\oplus\left\{\mathsf{query}_t : \mathsf{r}_{i-1}\&\left\{\begin{array}{l}\mathsf{yes} : \mathsf{r}_{i+1}\oplus\{\mathsf{yes} : \mathbf{t}_1\},\\ \mathsf{no} : \mathsf{r}_{i+1}\oplus\{\mathsf{no} : \mathbf{t}_1\}\end{array}\right\}\right\},\\
\mathsf{ff} : \mathsf{r}_{i-1}\oplus\{\mathsf{ff} : \mathbf{t}_1\},\\
\mathsf{tt} : \mathsf{r}_{i-1}\oplus\{\mathsf{query}_{p_{j_{i1}}} : \mathsf{r}_{i-1}\&\{\exp^\dagger(L_{i1}) : \mathsf{r}_{i-1}\oplus\{\mathsf{ff} : \mathbf{t}_1\},\\
\qquad \exp(L_{i1}) : \mathsf{r}_{i-1}\oplus\{\mathsf{query}_{p_{j_{i2}}} : \mathsf{r}_{i-1}\&\{\exp^\dagger(L_{i2}) : \mathsf{r}_{i-1}\oplus\{\mathsf{ff} : \mathbf{t}_1\},\\
\qquad\quad \exp(L_{i2}) : \mathsf{r}_{i-1}\oplus\{\mathsf{query}_{p_{j_{i3}}} : \mathsf{r}_{i-1}\&\{\exp^\dagger(L_{i3}) : \mathsf{r}_{i-1}\oplus\{\mathsf{ff} : \mathbf{t}_1\},\\
\qquad\qquad \exp(L_{i3}) : \mathsf{r}_{i-1}\oplus\{\mathsf{tt} : \mathbf{t}_1\}\}\}\}\}\}
\end{array}\right.$$

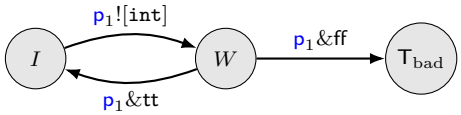
$$\mathsf{T}_{\mathsf{r}_{m+1}} = \mu \mathbf{t}_1.\mathsf{r}_m?[\mathtt{int}];\mathsf{r}_m\oplus\{\mathsf{ff} : \mathbf{t}_1\}$$

$$\mathrm{resolve}(\mathcal{Q}_i) = \begin{cases}\mathsf{tt} & \text{if } \mathcal{Q}_i = \exists\\ \mathsf{ff} & \text{if } \mathcal{Q}_i = \forall\end{cases}$$
$$\mathrm{resolve}^\dagger(\mathcal{Q}_i) = \begin{cases}\mathsf{ff} & \text{if } \mathcal{Q}_i = \exists\\ \mathsf{tt} & \text{if } \mathcal{Q}_i = \forall\end{cases}$$

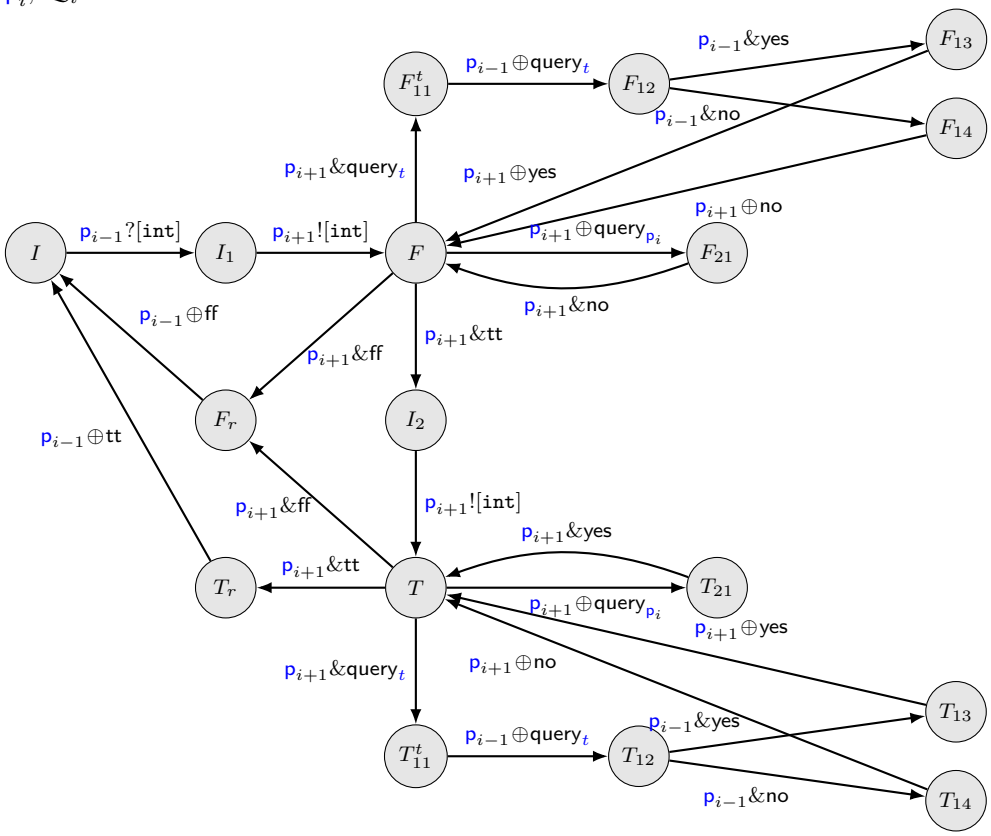
$$\exp(v_i) = \mathsf{yes} \qquad \exp^\dagger(v_i) = \mathsf{no}$$
$$\exp(\neg v_i) = \mathsf{no} \qquad \exp^\dagger(\neg v_i) = \mathsf{yes}$$

Figure 11: Reduction from QBF to checking safety. Let $v_{j_{iw}}$ be the variable in L_{iw}.

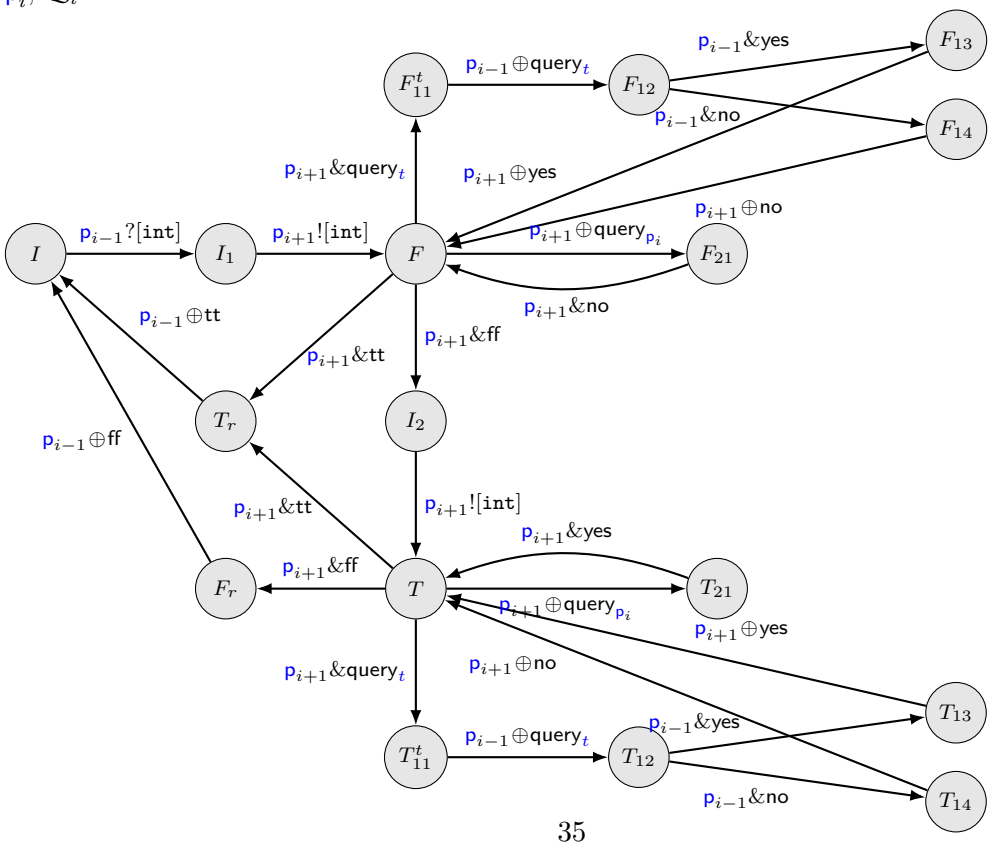
s:



$\mathsf{p}_i,\ \mathcal{Q}_i = \forall$:



$\mathsf{p}_i,\ \mathcal{Q}_i = \exists$:



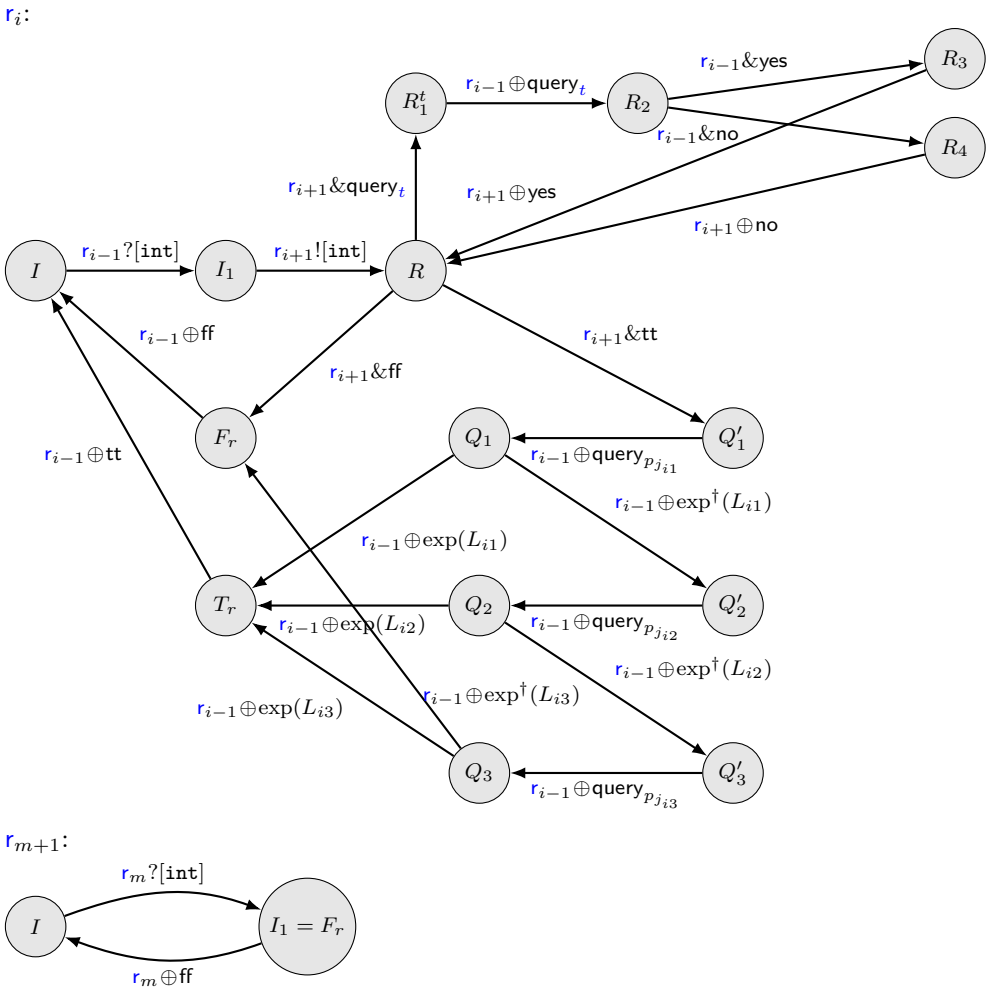


Figure 12: Graphs of the participants in Figure 11. Note that the graphs for p_i differ only on the edges incident to T_r and F_r.

Extended Appendices

B Appendix for Complexity Analysis of Multiparty Session Subtyping

Definition B.1 (Subtyping)**.** Let us denote a *tree representation* of T as $\mathbb{T}$ (see the detailed definition in [12, Notation 3.5]). The subtyping relation $\leq$ is the largest relation between session type trees coinductively defined by the following rules:

$$[\text{SUB-END}]\ \frac{-}{\mathtt{end} \leq \mathtt{end}} \qquad [\text{SUB-IN}]\ \frac{\mathbb{T} \leq \mathbb{T}'}{\mathsf{p}?[S];\mathbb{T} \leq \mathsf{p}?[S'];\mathbb{T}'} \qquad [\text{SUB-OUT}]\ \frac{\mathbb{T} \leq \mathbb{T}'}{\mathsf{p}![S];\mathbb{T} \leq \mathsf{p}![S'];\mathsf{T}'}$$

$$[\text{SUB-SEL}]\ \frac{\forall i \in I.\quad \mathbb{T}_i \leq \mathbb{T}'_i \quad I \subseteq I'}{\mathsf{p}\oplus\{l_i : \mathbb{T}_i\}_{i\in I} \leq \mathsf{p}\oplus\{l_i : \mathbb{T}'_i\}_{i\in I'}} \qquad [\text{SUB-BRA}]\ \frac{\forall i \in I.\quad \mathbb{T}_i \leq \mathbb{T}'_i \quad I' \subseteq I}{\mathsf{p}\&\{l_i : \mathbb{T}_i\}_{i\in I} \leq \mathsf{p}\&\{l_i : \mathbb{T}'_i\}_{i\in I'}}$$

We lift $\leq$ to syntactic types: i.e., $\mathsf{T} \leq \mathsf{T}'$ iff $\mathbb{T} \leq \mathbb{T}'$.

Lemma 2.3 (Subtyping as a type simulation)**.** $\mathsf{T}_1 \leq \mathsf{T}_2$ if and only if there exists a type simulation $\mathcal{R}$ such that $\mathsf{T}_1 \mathcal{R} \mathsf{T}_2$.

Proof. Note that $\leq$ is the largest type simulation, as all type simulations are consistent with the rules in Definition B.1. Thus the forward direction is immediate. For the backward direction, if $\mathsf{T}_1 \mathcal{R} \mathsf{T}_2$ then $\mathcal{R}$ is consistent with the subtyping rules, so $\mathsf{T}_1 \leq \mathsf{T}_2$. □

Lemma 2.6. $|\mathrm{Sub}(\mathsf{T})| = \mathcal{O}(|\mathsf{T}|)$.

Proof. By induction on the structure of T. We will show that $|\mathrm{Sub}(\mathsf{T})| \leq |\mathsf{T}|$.

Case $\mathsf{T} = \mathtt{end}$ or $\mathsf{T} = \mathbf{t}$: trivial.

Case $\mathsf{T} = \mu \mathbf{t}.\mathsf{T}'$: $|\mathrm{Sub}(\mathsf{T})| = |\{\mu \mathbf{t}.\mathsf{T}'\} \cup \{\mathsf{T}''[\mu \mathbf{t}.\mathsf{T}'/\mathbf{t}] \mid \mathsf{T}'' \in \mathrm{Sub}(\mathsf{T}')\}| \leq 1 + |\mathrm{Sub}(\mathsf{T}')| = |\mathrm{Sub}(\mathsf{T})|$.

Case $\mathsf{T} = \mathsf{p}![S]; \mathsf{T}'$: $|\mathrm{Sub}(\mathsf{T})| = |\{\mathsf{p}![S]; \mathsf{T}'\} \cup \mathrm{Sub}(\mathsf{T}')| \leq 1 + |\mathrm{Sub}(\mathsf{T}')| = |\mathrm{Sub}(\mathsf{T})|$.

Case $\mathsf{T} = \mathsf{p} \oplus \{\mathsf{T}_1, \ldots, \mathsf{T}_n\}$: $\mathrm{Sub}(\mathsf{T}) = |\{\mathsf{p} \oplus \{\mathsf{T}_1, \ldots, \mathsf{T}_n\}\} \cup \bigcup_{i=1}^{n} \mathrm{Sub}(\mathsf{T}_i)| \leq 1 + \sum_{i=1}^{n} |\mathrm{Sub}(\mathsf{T}_i)| \leq 1 + \sum_{i=1}^{n} |\mathsf{T}_i| = |\mathrm{Sub}(\mathsf{T})|$.

Other cases: similar. □

B.1 Proof of Theorem 2.8

To help with the proofs of the complexity results in this section, we first define *nesting depth*:

Definition B.2 (Nesting depth)**.** The nesting depth of a type is defined inductively:

$$\mathrm{nd}(\mathtt{end}) = \mathrm{nd}(\mathbf{t}) = 1$$

$$\mathrm{nd}(\mu \mathbf{t}.\mathsf{T}) = \mathrm{nd}(\mathsf{T}) + 1$$

$$\mathrm{nd}(\mathsf{p}?[S]; \mathsf{T}) = \mathrm{nd}(\mathsf{p}![S]; \mathsf{T}) = \mathrm{nd}(\mathsf{T}) + 1$$

$$\mathrm{nd}(\mathsf{p} \oplus \{l_i : \mathsf{T}_i\}_{i \in I}) = \mathrm{nd}(\mathsf{p} \& \{l_i : \mathsf{T}_i\}_{i \in I}) = \max(\{\mathrm{nd}(\mathsf{T}_i) \mid i \in I\}) + 1$$

We first give an algorithm which is worst-case exponential, taken from Ghilezan *et al.* [12], which is the algorithm in [10] adapted to the multiparty setting. We adapt the syntax to more closely match ours: in particular, we have separate constructs for sending and selection, as well as receiving and branching. The algorithm involves rules in Figure 13. These rules prove judgements of the form $\Theta \vdash \mathsf{T} \leq \mathsf{T}'$, which intuitively means "assuming the relations in Θ, we may conclude that T is a subtype of T'".

Ghilezan *et al.* [12, Theorem 3.26] prove soundness and completeness of this type system:

Theorem B.3. $\emptyset \vdash \mathsf{T} \leq \mathsf{T}'$ iff $\mathsf{T} \leq \mathsf{T}'$.

Using this theorem, we can give an algorithm for subtyping of session types. The algorithm tries to build the proof tree bottom-up, using rule [Alg-Assump] if possible. Note that the

$$\frac{(\mathsf{T},\mathsf{T}') \in \Theta}{\Theta \vdash \mathsf{T} \leq \mathsf{T}'}\ [\textsc{Alg-Assump}] \qquad \frac{}{\Theta \vdash \mathtt{end} \leq \mathtt{end}}\ [\textsc{Alg-End}]$$

$$\frac{\Theta \cup \{\mu \mathbf{t}.\mathsf{T}, \mathsf{T}'\} \vdash T[\mu \mathbf{t}.\mathsf{T}/\mathbf{t}] \leq \mathsf{T}'}{\Theta \vdash \mu \mathbf{t}.\mathsf{T} \leq \mathsf{T}'}\ [\textsc{Alg-RecL}] \qquad \frac{\Theta \cup \{\mathsf{T}, \mu \mathbf{t}.\mathsf{T}'\} \vdash \mathsf{T} \leq \mathsf{T}'[\mu \mathbf{t}.\mathsf{T}'/\mathbf{t}]}{\Theta \vdash \mathsf{T} \leq \mu \mathbf{t}.\mathsf{T}'}\ [\textsc{Alg-RecR}]$$

$$\frac{\Theta \vdash \mathsf{T}_1 \leq \mathsf{T}_1'}{\Theta \vdash \mathsf{p}?[S];\mathsf{T}_1 \leq \mathsf{p}?[S];\mathsf{T}_1'}\ [\textsc{Alg-In}] \qquad \frac{\Theta \vdash \mathsf{T}_1 \leq \mathsf{T}_1'}{\Theta \vdash \mathsf{p}![S];\mathsf{T}_1 \leq \mathsf{p}![S];\mathsf{T}_1'}\ [\textsc{Alg-Out}]$$

$$\frac{I \subseteq J \qquad \forall i \in I.\, \Theta \vdash \mathsf{T}_i \leq \mathsf{T}_i'}{\Theta \vdash \mathsf{p}\&\{l_j : \mathsf{T}_j\}_{j \in J} \leq \mathsf{p}\&\{l_i : \mathsf{T}_i\}_{i \in I}}\ [\text{AS-Bra}] \qquad \frac{I \subseteq J \qquad \forall i \in I.\, \Theta \vdash \mathsf{T}_i \leq \mathsf{T}_i'}{\Theta \vdash \mathsf{p}\oplus\{l_i : \mathsf{T}_i\}_{i \in I} \leq \mathsf{p}\oplus\{l_j : \mathsf{T}_j\}_{j \in J}}\ [\text{AS-Sel}]$$

Figure 13: Algorithmic rules for subtyping, from [12, Table 6] (syntax slightly modified).

rule taken at each step is determinstic (if we give [Alg-RecL] higher priority that [Alg-RecR]). By the above theorem, this algorithm is correct.

We adapt the results in [22] to prove the following complexity results of this algorithm. Taking $n = |\mathsf{T}| + |\mathsf{T}|'$ where the input is T, T':

Theorem B.4. The time complexity of the subtyping algorithm in [12] is $\mathcal{O}\left(n^{n^3}\right)$.

Proof. We proceed similarly to [22, Theorem 3.6]. First, note that all types appearing in the proof tree of $\emptyset \vdash \mathsf{T}_1 \leq \mathsf{T}_2$ are members of $\mathrm{Sub}(\mathsf{T}_1) \cup \mathrm{Sub}(\mathsf{T}_2)$. Furthermore, if $\Theta'' \vdash \mathsf{T}_1'' \leq \mathsf{T}_2''$ appears above $\Theta' \vdash \mathsf{T}_1' \leq \mathsf{T}_2'$, then either $|\Theta''| > |\Theta'|$ or $\mathrm{nd}(\mathsf{T}_1'') < \mathrm{nd}(\mathsf{T}_1')$; this can be proved by induction on the proof tree. Thus, in any vertical path in the proof tree, the pairs (Θ', T_1') are distinct. Therefore, in any judgement $\Theta \vdash \mathsf{T}_1' \leq \mathsf{T}_2'$ in the proof tree, there are only $\mathcal{O}(n^2)$ possible pairs that can be in Θ. Furthermore, there are only $\mathcal{O}(n)$ possible values of T_1', and thus $\mathcal{O}(n)$ possible values of $\mathrm{nd}(\mathsf{T}_1')$. Therefore, the height of the proof tree is bounded by $\mathcal{O}(n^3)$. As the branching factor is $\mathcal{O}(n)$, we conclude that the total number of nodes in the proof tree is $\mathcal{O}\left(n^{n^3}\right)$. Each node requires $\mathcal{O}(1)$ time to process. □

Theorem B.5. The worst-case complexity of the subtyping algorithm in [12] is $\Omega((\sqrt{n})!)$.

Proof. Consider the following example, which is adapted from the types in [22, Example 3.8], where this theorem is proved for binary session types:

$$\mathsf{T}_k \leq \mathsf{T}_{k+1} \quad \text{where} \quad \mathsf{T}_k := \mu \mathbf{t}.\mathsf{p}\oplus\left\{\begin{array}{l} l_1 : \mathsf{p}\oplus\left\{\begin{array}{ll} l_1 : & \ldots \quad \mathsf{p}\oplus\left\{\begin{array}{l} l_1 : \mathbf{t}, \\ l_2 : \mu \mathbf{t}'_0.\mathsf{T}^k_0 \end{array}\right\}, \\ l_2 : \mu \mathbf{t}'_{k-2}.\mathsf{T}^k_{k-2} & \end{array}\right\}, \\ l_2 : \mu \mathbf{t}'_{k-1}.\mathsf{T}^k_{k-1} \end{array}\right\}$$

$$\text{and} \quad \mathsf{T}^k_j := \mathsf{p}\oplus\left\{\begin{array}{l} l_1 : \mathsf{p}\oplus\left\{\begin{array}{ll} l_1 : & \ldots \quad \mathsf{p}\oplus\left\{\begin{array}{l} l_1 : \mathbf{t}, \\ l_2 : \mu \mathbf{t}''.\mathsf{p}\oplus\{l_1 : \mathbf{t}'', l_2 : \mathbf{t}''\} \end{array}\right\}, \\ l_2 : \mu \mathbf{t}''.\mathsf{p}\oplus\{l_1 : \mathbf{t}'', l_2 : \mathbf{t}''\} & \end{array}\right\}, \\ l_2 : \mu \mathbf{t}''.\mathsf{p}\oplus\{l_1 : \mathbf{t}'', l_2 : \mathbf{t}''\} \end{array}\right\}$$

where T^k_j contains j nested selections before $\mathbf{t}$. This is identical to the type in the original example, but adapted to a multiparty setting and using selection instead of the receive construct: this is needed because in our system types can only send and receive sorts, not types. However, the subtyping rules are identical.

We may then use the same argument as in [22, Theorem 3.11] to prove that the complexity of the subtyping algorithm is $\Omega((\sqrt{n})!)$, by replacing all instances of $?[\mathsf{T}_1];\mathsf{T}_2$ with $\mathsf{p}\oplus\{l_1 : \mathsf{T}_1, l_2 : \mathsf{T}_2\}$. □

Thus, the algorithm as described in [12] takes exponential time in the worst case. Therefore Theorem 2.8 is proven.

Theorem 2.8. The algorithm to check $\mathsf{T}_1 \leq \mathsf{T}_2$ given in [12, Table 6] takes between $\mathcal{O}\left(n^{n^3}\right)$ and $\Omega((\sqrt{n})!)$ time in the worst case with $n = |\mathsf{T}_1| + |\mathsf{T}_2|$.

B.2 Proof of Theorem 2.10

[22] gives an $\mathcal{O}(n^2)$ algorithm for subtyping of binary synchronous session types, based on type simulations (Definition 2.2). We adapt this algorithm to the multiparty synchronous calculus.

Definition B.6 (Product subtyping graph)**.** Define the subtyping graph $\mathbb{G}(\mathsf{T}_1^*, \mathsf{T}_2^*)$ as the following transition system on vertices $\mathbb{G}(\mathsf{T}_1^*) \times \mathbb{G}(\mathsf{T}_2^*)$, with transitions satisfying the following condition: if $\mathsf{T}_1 \xrightarrow{\alpha} \mathsf{T}_1'$ and $\mathsf{T}_2 \xrightarrow{\alpha} \mathsf{T}_2'$, then $(\mathsf{T}_1, \mathsf{T}_2) \to (\mathsf{T}_1', \mathsf{T}_2')$.

Definition B.7 (Inconsistent nodes)**.** In $\mathbb{G}(\mathsf{T}_1^*, \mathsf{T}_2^*)$, define a state $(\mathsf{T}_1, \mathsf{T}_2)$ *inconsistent* if it immediately violates Definition 2.2. Formally, $(\mathsf{T}_1, \mathsf{T}_2)$ is inconsistent iff at least one of the following hold:

- $\mathsf{T}_1 \xrightarrow{\ell} \mathsf{T}_1'$ and $\mathsf{T}_2 \not\xrightarrow{\ell}$, for $\ell \in \{\mathsf{p}\&l, \mathtt{end}, \mathsf{p}?[S], \mathsf{p}![S]\}$.
- $\mathsf{T}_2 \xrightarrow{\ell} \mathsf{T}_2'$ and $\mathsf{T}_1 \not\xrightarrow{\ell}$, for $\ell \in \{\mathsf{p}\oplus l, \mathtt{end}\}$.

Using this notion of product graph and inconsistent nodes, we can state the quadratic algorithm for subtyping.

Theorem 2.10. $\mathsf{T}_1 \leq \mathsf{T}_2$ can be checked in $\mathcal{O}(|\mathsf{T}_1| \cdot |\mathsf{T}_2|)$ time in the worst case.

Proof. The algorithm proceeds as follows:

- Construct the graph $\mathbb{G}(\mathsf{T}_1, \mathsf{T}_2)$.
- For each node $(\mathsf{T}_1', \mathsf{T}_2')$ in the graph, check if it is inconsistent.
- If there is an inconsistent node, then $\mathsf{T}_1 \not\leq \mathsf{T}_2$. Otherwise, $\mathsf{T}_1 \leq \mathsf{T}_2$.

If there are no inconsistent nodes in the graph, then the set of nodes defines a type simulation (by definition). Furthermore, if $\mathsf{T}_1 \leq \mathsf{T}_2$, then $\mathbb{G}(\mathsf{T}_1, \mathsf{T}_2)$ has no inconsistent nodes (by induction, all reachable nodes of $\mathbb{G}(\mathsf{T}_1, \mathsf{T}_2)$ follow the subtyping relation). Therefore, the algorithm is correct. The graph has $\mathcal{O}(|\mathsf{T}_1| \cdot |\mathsf{T}_2|)$ nodes and edges (a proof of this is found in [22, Lemmas 4.2 and 4.3]), therefore the algorithm runs in $\mathcal{O}(|\mathsf{T}_1| \cdot |\mathsf{T}_2|)$ time. □

B.3 Empirical Analysis

We give measurements and benchmarks of the two subtyping algorithms mentioned above, with a focus on worst-case performance. Each algorithm was run once on each input with a timeout of 10 seconds. The algorithms were implemented in C++ and run on a machine with an AMD Ryzen 5 5600H at 3300 MHz and 16GB of RAM.

We benchmark our algorithm on three test suites. In these suites, we consider test cases where the subtyping relation holds, i.e. inputs T_1 and T_2 where $\mathsf{T}_1 \leq \mathsf{T}_2$, because in these cases all possible reachable branches in the algorithm have to be taken and checked, which is the worst-case scenario for our algorithms. Our first test suite is based on the worst-case construction in Theorem B.5, where the inductive algorithm is proven to be exponential; the second suite is based on randomly generating larger types coinductively equal to $\mu\mathbf{t}.\mathsf{p}\oplus\{l_1 : \mathbf{t}, l_2 : \mathbf{t}\}$. Finally, we run the algorithms on idempotent test cases (i.e. $\mathsf{T} \leq \mathsf{T}$), showing that there is no difference between the algorithms on this test suite.

In Figure 3a, we run the two algorithms on the test case in Theorem 2.10: $\mathsf{T}_k \leq \mathsf{T}_{k+1}$. The results show that, as expected, the algorithm in [12] grows exponentially, while the quadratic algorithm in 2.10 grows much more slowly. In particular, this effect is already apparent at $k = 3$, where the inductive algorithm takes 10 times longer than the quadratic algorithm. This runtime gap grows as k increases, with the inductive algorithm timing out at $k = 8$.

In Figure 3b, instead of using T_k, we experiment with other types that are also isomorphic to $\mu\mathbf{t}.\mathsf{p} \oplus \{l_1 : \mathbf{t}, l_2 : \mathbf{t}\}$. We generate random types of this form with size $\mathcal{O}(k)$. The procedure is as follows: a random binary tree was generated with k leaves. This was then converted to a type by starting with the recursive construct $\mu\mathbf{t}$ at the root, then replacing each leaf with $\mathbf{t}$, and each internal node with a selection, with each branch pointing to the type of the corresponding children. This has the effect of unfolding the above type many times. Similar results to Figure 3a are observed, but with more noise due to the random nature of the input: we observe that the quadratic algorithm runs very quickly, while the exponential algorithm is much slower, timing out at types with around 20 leaves. This shows that the exponential blowup found in Theorem B.5 also empirically apply to other types that exhibit self-similarity.

We also ran the algorithm on instances of the form $\mathsf{T} \leq \mathsf{T}$, where T is randomly generated; the results are in Figure 3c. We would expect no difference in the runtime of the two algorithms, as only nodes of the form $(\Gamma \vdash)\mathsf{T}_1 \leq \mathsf{T}_1$ have to be checked. Random instances of size 10^4 were generated and the algorithms were run 10^4 times; average runtime per iteration is shown. As expected, the runtimes are nearly identical in all test cases.

C Appendix for Complexity of Projection

Definition C.1 (Subformulas of global types)**.** The set of *subformulas of a global type* G, denoted $\mathrm{Sub}(\mathsf{G})$, is defined as: $\mathrm{Sub}(\mathtt{end}) = \mathtt{end}$; $\mathrm{Sub}(\mathsf{p} \to \mathsf{q} : [S]; \mathsf{G}) = \{\mathsf{p} \to \mathsf{q} : [S]; \mathsf{G}\} \cup \mathrm{Sub}(\mathsf{G})$; $\mathrm{Sub}(\mathsf{p} \to \mathsf{q} : \{l_i : \mathsf{G}_i\}_{i \in I}) = \{\mathsf{p} \to \mathsf{q} : \{l_i : \mathsf{G}_i\}_{i \in I}\} \cup \bigcup_{i \in I} \mathrm{Sub}(\mathsf{G}_i)$; and $\mathrm{Sub}(\mu \mathbf{t}.\mathsf{G}) = \{\mu \mathbf{t}.\mathsf{G}\} \cup \{\mathsf{G}'[\mu \mathbf{t}.\mathsf{G}/\mathbf{t}] \mid \mathsf{T}' \in \mathrm{Sub}(\mathsf{G})\}$.

Lemma C.2. $|\mathrm{Sub}(\mathsf{G})| = \mathcal{O}(|\mathsf{G}|)$.

Proof. By structural induction on G, similarly to Lemma 2.6. □

C.1 Appendix for Complexity of Inductive Projection with Plain Merging

Lemma C.3. Under plain merging, $|\mathsf{G}\upharpoonright_{\mathsf{p}}| \leq |\mathsf{G}|$.

Proof. Straightforward; by induction on the structure of G. □

Theorem 3.7. Projection with plain merging, $\mathsf{G}\upharpoonright_{\mathsf{p}}$, can be performed in $\mathcal{O}(n \log n)$ time (where $n = |\mathsf{G}|$).

Proof. First, note that by keeping track of the free variables, we may check every syntactic subterm of G for closedness in $\mathcal{O}(n)$ time. We perform this as a first pass. Then, our algorithm will perform the projection in the same recursive sense as in Definition 3.2, except for the aforementioned checking of closedness. In particular, computing $\bigsqcap_{i \in I} \mathsf{G}_i\upharpoonright_{\mathsf{p}}$ will be a syntactic equality check. First, we prove the result for global types where the projection is defined.

Claim C.4. The algorithm computes $\mathsf{G}\upharpoonright_{\mathsf{p}}$ in $2|\mathsf{G}| \log |\mathsf{G}|$ time, if $\mathsf{G}\upharpoonright_{\mathsf{p}}$ is defined.

Proof. We prove that the algorithm computes $\mathsf{G}\upharpoonright_{\mathsf{p}}$ in $2|\mathsf{G}|(\log |\mathsf{G}|+1)$ time, if $\mathsf{G}\upharpoonright_{\mathsf{p}}$ is defined. We use base-2 logarithms for convenience:

Case $\mathsf{G} = \mathsf{q} \to \mathsf{r} : [S]; \mathsf{G}'$. We have $|\mathsf{G}| = |\mathsf{G}'| + 1$; by the inductive hypothesis the time taken, including a constant time overhead, is $2|\mathsf{G}'|(\log |\mathsf{G}'| + 1) + 1 \leq 2|\mathsf{G}|(\log |\mathsf{G}| + 1)$.

Case $\mathsf{G} = \mathsf{q} \to \mathsf{r} : \{l_i : \mathsf{G}_i\}_{i \in I}$, $\mathsf{p} \notin \{\mathsf{q}, \mathsf{r}\}$. If $I = \{i\}$ then by the same logic as above, we compute $\mathsf{G}\upharpoonright_{\mathsf{p}}$ in time $2|\mathsf{G}'|(\log|\mathsf{G}'|+1)+1 \leq 2|\mathsf{G}|(\log|\mathsf{G}|+1)$.

Otherwise, $|I| \geq 2$. Let $n_i = |\mathsf{G}_i|$ and $\mathsf{T}_i = \mathsf{G}_i\upharpoonright_{\mathsf{p}}$. Then the projections of G_i take time $2n_i \log n_i$; the merging is a syntactic equality check which takes time $\sum_{i\in I}|\mathsf{T}_i|$. We are guaranteed that $\mathsf{T}_i = \mathsf{T}$ because of the plain merge.

Let $x \in I$ be such that n_x is maximum. Thus, for $i \neq x$, $n_i \leq \frac{n}{2}$ so $\log n_i + 1 \leq \log n$. The time taken for recursive projection is $\sum_{i\in I}(2n_i(\log n_i+1))$ and merging is $\sum_{i\in I}|\mathsf{T}_i|$. Hence we have:

$$
\begin{aligned}
&\underbrace{\sum_{i\in I}(2n_i(\log n_i+1))}_{\text{recursive projection}} + \underbrace{\sum_{i\in I}|\mathsf{T}_i|}_{\text{merging}} \\
&= \sum_{i\in I}(2n_i(\log n_i+1)) + |I|\cdot|\mathsf{T}| \\
&\leq 2n_x(\log n_x+1) + \sum_{i\in I\setminus\{x\}}(2n_i(\log n_i+1)+2|\mathsf{T}|) && \text{(because } |I|\geq 2\text{)} \\
&\leq 2n_x(\log n_x+1) + \sum_{i\in I\setminus\{x\}}(2n_i(\log n_i+1)+2n_i) && \text{(by Lemma C.3)} \\
&\leq 2n_x(\log n+1) + \sum_{i\in I\setminus\{x\}}(2n_i(\log n+1)) \\
&= 2(\log n+1)\sum_{i\in I} n_i \\
&\leq 2n(\log n+1)
\end{aligned}
$$

Case $\mathsf{G} = \mathsf{q} \to \mathsf{r} : \{l_i : \mathsf{G}_i\}_{i \in I}$, $\mathsf{p} \in \{\mathsf{q}, \mathsf{r}\}$. Then the time taken for the recursive projections (using the same notation as the above case) is $\sum_{i\in I}(2n_i(\log n_i+1)) \leq 2n(\log n+1)$.

Case $\mathsf{G} = \mu\mathbf{t}.\mathsf{G}'$. Let $\mathsf{T}' = \mathsf{G}'\upharpoonright_{\mathsf{p}}$. We have $|\mathsf{T}| \leq |\mathsf{T}'|+1$. Checking for closedness was precomputed, thus the total time is $2|\mathsf{T}'|(\log|\mathsf{T}'|+1)+1 \leq 2|\mathsf{T}|(\log|\mathsf{T}|+1) \leq 2|\mathsf{G}|(\log|\mathsf{G}|+1)$.

Case $\mathsf{G} = \mathtt{end}$ and $\mathsf{G} = \mathbf{t}$. Both are projectable in constant time, and so it is bounded by $\mathsf{G}(\log|\mathsf{G}|+1) = 1$. ■

When $\mathsf{G}\upharpoonright_\mathsf{p}$ is not defined, we can show that the time taken to compute $\mathsf{G}\upharpoonright_\mathsf{p}$ is bounded by $2|\mathsf{G}|\log|\mathsf{G}| + |\mathsf{G}|$. Most reasoning is identical to the above proof, except for the case where an unprojectability error is found, i.e. $\mathsf{G} = \mathsf{q} \to \mathsf{r} : \{l_i : \mathsf{G}_i\}_{i\in I}$, $\mathsf{p} \notin \{\mathsf{q}, \mathsf{r}\}$, $|I| \geq 2$, $\mathsf{G}_i\upharpoonright_\mathsf{p}$ is defined for all i, and $\bigsqcap_{i\in I}\mathsf{G}_i\upharpoonright_\mathsf{p}$ is undefined. In this case we cannot rely on the fact that $|\mathsf{T}_j| < n_i$ for all i, j, which was necessary for the above proof. Instead we prove the following bound: $\sum_{i\in I}(2n_i \log n_i) + \sum_{i\in I}|\mathsf{T}_i| \leq 2\log n \cdot \sum_{i\in I} n_i + \sum_{i\in I}|\mathsf{G}_i\upharpoonright_\mathsf{p}| \leq 2n\log n + \sum_{i\in I}|\mathsf{G}_i\upharpoonright_\mathsf{p}| \leq 2n\log n + n$. □

C.2 Appendix for Complexity of Inductive Projection with Full Merging

Lemma C.5. If $\mathsf{T}_1 \boxtimes \mathsf{T}_2 = \mathsf{T}$ and T is defined then $|\mathsf{T}| < |\mathsf{T}_1| + |\mathsf{T}_2|$.

Proof. By structural induction on T_1.

Case $\mathsf{T}_1 = \mathbf{t}$ or $\mathsf{T}_1 = \mathtt{end}$. Then $\mathsf{T}_2 = \mathsf{T}$, so $|\mathsf{T}| = 1 < 2 = |\mathsf{T}_1| + |\mathsf{T}_2|$.

Case $\mathsf{T}_1 = \mathsf{p}?[S]; \mathsf{T}_1'$. Then $\mathsf{T}_2 = \mathsf{p}?[S]; \mathsf{T}_2'$, so $|\mathsf{T}| = 1 + |\mathsf{T}_1' \boxtimes \mathsf{T}_2'| < 1 + |\mathsf{T}_1'| + |\mathsf{T}_2'| < |\mathsf{T}_1| + |\mathsf{T}_2|$.

Case $\mathsf{T}_1 = \mathsf{p}![S]; \mathsf{T}_1'$. Similar to the above.

Case $\mathsf{T}_1 = \mathsf{p}\oplus\{l_i : \mathsf{T}_1^{(i)}\}_{i\in I}$. Then $\mathsf{T}_2 = \mathsf{p}\oplus\{l_i : \mathsf{T}_2^{(i)}\}_{i\in I}$, so $|\mathsf{T}| = 1 + \sum_{i\in I}|\mathsf{T}_1^{(i)} \boxtimes \mathsf{T}_2^{(i)}| < 1 + \sum_{i\in I}(|\mathsf{T}_1^{(i)}| + |\mathsf{T}_2^{(i)}|) < 1 + |\mathsf{T}_1| + |\mathsf{T}_2|$.

Case $\mathsf{T}_1 = \mathsf{p}\&\{l_i : \mathsf{T}_1^{(i)}\}_{i\in I}$. Then $\mathsf{T}_2 = \mathsf{p}\oplus\{l_i : \mathsf{T}_2^{(j)}\}_{j\in J}$, so $|\mathsf{T}| = 1 + \sum_{i\in I\cap J}\left(\mathsf{T}_1^{(i)} \boxtimes \mathsf{T}_2^{(i)}\right) + \sum_{i\in I\setminus J}\left(\mathsf{T}_1^{(i)}\right) + \sum_{i\in J\setminus I}\left(\mathsf{T}_2^{(i)}\right) < 1 + \sum_{i\in I}(\mathsf{T}_1^{(i)}) + \sum_{i\in J}(\mathsf{T}_2^{(i)}) < |\mathsf{T}_1| + |\mathsf{T}_2|$. □

By repeated application of the above lemma, we have:

Corollary C.5.1. If $\boxtimes_{i\in I}\mathsf{T}_i = \mathsf{T}$ and T is defined then $|\mathsf{T}| < \sum_{i\in I}|\mathsf{T}_i|$.

Lemma C.6. Under full merging, $|\mathsf{G}\upharpoonright_\mathsf{p}| \leq |\mathsf{G}|$.

Proof. By structural induction on G.

Case $\mathsf{G} = (\mathsf{q} \to \mathsf{r} : \{l_i : \mathsf{G}_i\}_{i\in I})$ where $\mathsf{p} \notin \{\mathsf{q}, \mathsf{r}\}$. Then, using Corollary C.5.1 and the inductive hypothesis, $|\mathsf{G}\upharpoonright_\mathsf{p}| = 1 + \boxtimes_{i\in I}\mathsf{G}_i\upharpoonright_\mathsf{p} < 1 + \sum_{i\in I}|\mathsf{G}_i\upharpoonright_\mathsf{p}| < 1 + \sum_{i\in I}|\mathsf{G}_i| = |\mathsf{G}|$.

All other cases are trivial. □

Lemma 3.10. If $\mathsf{T}_1 \boxtimes \mathsf{T}_2 = \mathsf{T}$ and T is defined then $\mathsf{T}_1 \boxtimes \mathsf{T}_2$ can be computed in time $\mathcal{O}(|\mathsf{T}_1| + |\mathsf{T}_2| - |\mathsf{T}| + |\mathsf{T}_2| \cdot \log |\mathsf{T}_1|)$.

Proof. To allow for the complexity logarithmic in $|\mathsf{T}_1|$, in our algorithm (and other algorithms that use this lemma), we will represent local types as follows:

- $\mathbf{t}$ and $\mathtt{end}$ are primitive.
- $\mathsf{p}?[S];\mathsf{T}$ and $\mathsf{p}![S];\mathsf{T}$ contain a pointer to T.
- $\mathsf{p}\oplus\{l_i : \mathsf{T}_i\}_{i\in I}$ is represented as a (sorted) list of $\langle l_i, \text{pointer to } \mathsf{T}_i\rangle$ pairs.
- $\mathsf{p}\&\{l_i : \mathsf{T}_i\}_{i\in I}$ is represented as a balanced binary search tree (BST) with keys l_i and values as pointers to T_i. This is done by first fixing a total ordering on labels consistently across the algorithm.

Global types will be represented in the usual way, as syntactic trees.

By structural induction on T_1. We will show that, up to constant factors, the merge can be computed in time at most $2(|\mathsf{T}_1| + |\mathsf{T}_2| - |\mathsf{T}|) - 1 + (|\mathsf{T}_2| - 1) \cdot \log |\mathsf{T}_1|$. Note that this is positive by Lemma C.5.

Case $\mathsf{T}_1 = \mathbf{t}$ or $\mathsf{T}_1 = \mathtt{end}$. Then $\mathsf{T}_2 = \mathsf{T}$. This can be checked in time $1 = 2(|\mathsf{T}_1| + |\mathsf{T}_2| - |\mathsf{T}|) - 1 + (|\mathsf{T}_2| - 1) \cdot \log |\mathsf{T}_1|$.

Case $\mathsf{T}_1 = \mathsf{p}?[S];\mathsf{T}_1'$, $\mathsf{T}_2 = \mathsf{p}?[S];\mathsf{T}_2'$. Then it suffices to find $\mathsf{T}_1' \boxtimes \mathsf{T}_2'$. By the inductive hypothesis this takes $2(|\mathsf{T}_1'| + |\mathsf{T}_2'| - |\mathsf{T}_1' \boxtimes \mathsf{T}_2'|) - 1 + (|\mathsf{T}_2'| - 1) \cdot \log |\mathsf{T}_1'|$ time. Adding a constant overhead of 1, the total time is $2(|\mathsf{T}_1'| + |\mathsf{T}_2'| - |\mathsf{T}_1' \boxtimes \mathsf{T}_2'|) + (|\mathsf{T}_2'| - 1) \cdot \log |\mathsf{T}_1'| = 2(|\mathsf{T}_1| + |\mathsf{T}_2| - |\mathsf{T}|) - 2 + (|\mathsf{T}_2'| - 1) \cdot \log |\mathsf{T}_1'| < 2(|\mathsf{T}_1| + |\mathsf{T}_2| - |\mathsf{T}|) - 1 + (|\mathsf{T}_2| - 1) \cdot \log |\mathsf{T}_1|$.

Case $\mathsf{T}_1 = \mathsf{p}![S];\mathsf{T}_1'$, $\mathsf{T}_2 = \mathsf{p}![S];\mathsf{T}_2'$. Similar to the above.

Case $\mathsf{T}_1 = \mathsf{p}\oplus\{l_i : \mathsf{T}_1^{(i)}\}_{i\in I}$, $\mathsf{T}_2 = \mathsf{p}\oplus\{l_i : \mathsf{T}_2^{(i)}\}_{i\in I}$. Then it suffices to find $\mathsf{T}_1^{(i)} \bowtie \mathsf{T}_2^{(i)}$ for each $i \in I$. By the inductive hypothesis this takes time:

$$
\begin{aligned}
&\sum_{i\in I}\left(2(|\mathsf{T}_1^{(i)}| + |\mathsf{T}_2^{(i)}| - |\mathsf{T}_1^{(i)} \bowtie \mathsf{T}_2^{(i)}|) - 1 + (|\mathsf{T}_2^{(i)}| - 1)\cdot\log|\mathsf{T}_1^{(i)}|\right)\\
&= 2\left(\sum_{i\in I}|\mathsf{T}_1^{(i)}| + \sum_{i\in I}|\mathsf{T}_2^{(i)}| - \sum_{i\in I}|\mathsf{T}_1^{(i)} \bowtie \mathsf{T}_2^{(i)}|\right) - |I| + \sum_{i\in I}(|\mathsf{T}_2^{(i)}| - 1)\cdot\log|\mathsf{T}_1^{(i)}|\\
&\le 2(|\mathsf{T}_1| + |\mathsf{T}_2| - |\mathsf{T}|) - |I| - 2 + \sum_{i\in I}(|\mathsf{T}_2^{(i)}| - 1)\cdot\log|\mathsf{T}_1|\\
&< 2(|\mathsf{T}_1| + |\mathsf{T}_2| - |\mathsf{T}|) - |I| - 1 + (|\mathsf{T}_2| - 1)\cdot\log|\mathsf{T}_1|
\end{aligned}
$$

Adding a linear overhead in the number of branches, $|I|$, yields the result.

Case $\mathsf{T}_1 = \mathsf{p}\&\{l_i : \mathsf{T}_1^{(i)}\}_{i\in I}$, $\mathsf{T}_2 = \mathsf{p}\&\{l_i : \mathsf{T}_2^{(j)}\}_{j\in J}$. We do the following:

- Start with the BST for T_1, containing $\{l_i \mapsto \mathsf{T}_1^{(i)} \mid i \in I\}$. Call this tree $\mathbb{T}$.
- Iterate over the BST for T_2. For each item $l_j \mapsto \mathsf{T}_2^{(j)}$:
  - If the key l_j is present in $\mathbb{T}$, with value $\mathsf{T}_1^{(j)}$, then overwrite the value with $\mathsf{T}_1^{(j)} \bowtie \mathsf{T}_2^{(j)}$.
  - Otherwise, add a new key-value pair $l_j \mapsto \mathsf{T}_2^{(j)}$ to $\mathbb{T}$.

It follows that the tree $\mathbb{T}$ at the end of this process is the tree representing the type $\mathsf{T} = \mathsf{T}_1 \bowtie \mathsf{T}_2$. The time taken to perform this operation is the sum of the time taken for the merging of branches in $\{l_i \mid i \in I \cap J\}$, as well as the overhead of the $|J|$ tree

operations:

$$\underbrace{\sum_{i\in I\cap J}\Big(2(|\mathsf{T}_1^{(i)}|+|\mathsf{T}_2^{(i)}|-|\mathsf{T}_1^{(i)}\bowtie\mathsf{T}_2^{(i)}|)-1+(|\mathsf{T}_2^{(i)}|-1)\cdot\log|\mathsf{T}_1^{(i)}|\Big)}_{\text{merging}}+\underbrace{|J|\log|\mathsf{T}_1|}_{\text{overhead}}$$

$$\leq \sum_{i\in I\cap J}\Big(2(|\mathsf{T}_1^{(i)}|+|\mathsf{T}_2^{(i)}|-|\mathsf{T}_1^{(i)}\bowtie\mathsf{T}_2^{(i)}|)-1+(|\mathsf{T}_2^{(i)}|-1)\cdot\log|\mathsf{T}_1|\Big)+|J|\log|\mathsf{T}_1|$$

$$=2\left(\sum_{i\in I\cap J}|\mathsf{T}_1^{(i)}|+\sum_{i\in I\cap J}|\mathsf{T}_2^{(j)}|-\sum_{i\in I\cap J}|\mathsf{T}_1^{(i)}\bowtie\mathsf{T}_2^{(i)}|\right)$$
$$-|I\cap J|+\log|\mathsf{T}_1|\cdot\left(\sum_{i\in I\cap J}|\mathsf{T}_2^{(i)}|-|I\cap J|+|J|\right)$$

We have

$$\sum_{i\in I\cap J}|\mathsf{T}_1^{(i)}|+\sum_{i\in I\cap J}|\mathsf{T}_2^{(j)}|-\sum_{i\in I\cap J}|\mathsf{T}_1^{(i)}\bowtie\mathsf{T}_2^{(i)}|$$
$$=\sum_{i\in I}|\mathsf{T}_1^{(i)}|+\sum_{i\in J}|\mathsf{T}_2^{(i)}|-\left(\sum_{i\in I\cap J}|\mathsf{T}_1^{(i)}\bowtie\mathsf{T}_2^{(i)}|+\sum_{i\in I\setminus J}|\mathsf{T}_1^{(i)}|+\sum_{i\in J\setminus I}|\mathsf{T}_2^{(i)}|\right)$$
$$=|\mathsf{T}_1|+|\mathsf{T}_2|-|\mathsf{T}|-1$$

and

$$\sum_{i\in I\cap J}|\mathsf{T}_2^{(i)}|-|I\cap J|+|J|$$
$$=\sum_{i\in I\cap J}|\mathsf{T}_2^{(i)}|+|J\setminus I|$$
$$\leq\sum_{i\in J}|\mathsf{T}_2^{(i)}| \qquad \text{because } |T_2^{(i)}|\geq 1$$
$$\leq|\mathsf{T}_2|-1$$

Thus the total time is less than $2(|\mathsf{T}_1|+|\mathsf{T}_2|-|\mathsf{T}|)-2-|I\cap J|+(|\mathsf{T}_2|-1)\cdot\log|\mathsf{T}_1|$, as desired. □

Corollary C.6.1. If $\bowtie_{i\in I}\mathsf{T}_i=\mathsf{T}$ and T is defined then T can be computed in time $\mathcal{O}\Big(\sum_{i\in I}|\mathsf{T}_i|-|\mathsf{T}|+\log\left(\sum_{i\in I}|\mathsf{T}_i|\right)\cdot\sum_{i\in I\setminus\{j\}}|\mathsf{T}_i|\Big)$, for any $j\in I$.

Proof. Our algorithm proceeds as follows:

$\mathsf{T} \leftarrow \mathsf{T}_j$

for $i \in I \setminus \{j\}$ **do**

  $\mathsf{T} \leftarrow \mathsf{T} \boxplus \mathsf{T}_i$

end for

return T

Taking $I = \{j, a_1, \ldots, a_k\}$, label the m-th intermediate result $\mathsf{T}'_m = \boxplus_{j \in \{x, a_1, \ldots, a_m\}} \mathsf{T}_i$. By Lemma C.6, $|\mathsf{T}'_m| \leq (\sum_{i \in I} |\mathsf{T}_i|)$. Then by the above lemma, calculating $\mathsf{T}'_{m-1} \boxplus \mathsf{T}_m$ takes time proportional to

$$|\mathsf{T}'_{m-1}| + |\mathsf{T}_m| - |\mathsf{T}'_m| + |\mathsf{T}_m| \log |\mathsf{T}'_{m-1}| \leq |\mathsf{T}'_{m-1}| + |\mathsf{T}_m| - |\mathsf{T}'_m| + \log \left(\textstyle\sum_{i \in I} |\mathsf{T}_i|\right) \cdot |\mathsf{T}_m|$$

Using the facts that $\mathsf{T}'_0 = \mathsf{T}_j$ and $\mathsf{T}'_k = \mathsf{T}$, summing the above for $1 \leq m \leq k$ yields the desired result. □

Theorem 3.11. Projection with full merging can be performed in $\mathcal{O}\Big(|\mathsf{G}| \log^2 |\mathsf{G}|\Big)$ time.

Proof. Similarly to the plain merging algorithm, we start by checking every syntactic subterm for closedness in linear time. Then, we show inductively that (up to constant factors) we can compute $\mathsf{G}\restriction_{\mathsf{p}}$ in time $|\mathsf{G}| \log^2 |\mathsf{G}| + 2|\mathsf{G}| - |\mathsf{T}|$, where $\mathsf{T} = \mathsf{G}\restriction_{\mathsf{p}}$.

For convenience, define $|\mathsf{T}| = 0$ if T is undefined. We use base-2 logarithms for ease of explanations.

Case $\mathsf{G} = \mathsf{q} \to \mathsf{r} : [S]; \mathsf{G}'$. We have $|\mathsf{G}'| + 1 = |\mathsf{G}|$. Define $\mathsf{T}' = \mathsf{G}'\restriction_{\mathsf{p}}$; we have $|\mathsf{T}| \leq |\mathsf{T}'| + 1$. Then the time taken is $1 + |\mathsf{G}'| \log^2 |\mathsf{G}'| + 2|\mathsf{G}'| - |\mathsf{T}'| \leq |\mathsf{G}'| \log^2 |\mathsf{G}'| + 2|\mathsf{G}| - |\mathsf{T}| \leq |\mathsf{G}| \log^2 |\mathsf{G}| + 2|\mathsf{G}| - |\mathsf{T}|$.

Case $\mathsf{G} = \mathsf{q} \to \mathsf{r} : \{l_i : \mathsf{G}_i\}_{i \in I}$, $\mathsf{p} \notin \{\mathsf{q}, \mathsf{r}\}$. Denote $n_i = |\mathsf{G}_i|$. Let $m \in I$ be such that $|G_m|$ is maximised. Let $\mathsf{T}_i = \mathsf{G}_i{\upharpoonright}_{\mathsf{p}}$. Then the time taken is

$$
\begin{aligned}
&\underbrace{\sum_{i \in I} \left(n_i \log^2 n_i + 2n_i - |\mathsf{T}_i| \right)}_{\text{recursive projection}} + \underbrace{\sum_{i \in I} |\mathsf{T}_i| - |\mathsf{T}| + \log \left(\sum_{i \in I} |\mathsf{T}_i| \right) \cdot \log |\mathsf{T}_m| \sum_{i \in I \setminus \{m\}} |\mathsf{T}_i|}_{\text{merging}} \\
&= \sum_{i \in I} \left(n_i \log^2 n_i \right) + \log \left(\sum_{i \in I} |\mathsf{T}_i| \right) \cdot \left(\sum_{i \in I \setminus \{m\}} |\mathsf{T}_i| \right) + 2(|\mathsf{G}| - 1) - |\mathsf{T}| \\
&\leq \log n \cdot \left(\sum_{i \in I \setminus \{m\}} (n_i \log n_i + n_i) + n_m \log n \right) + 2(|\mathsf{G}| - 1) - |\mathsf{T}| \\
&\leq \log n \cdot \left(\sum_{i \in I \setminus \{m\}} (n_i \log n) + n_m \log n \right) + 2(|\mathsf{G}| - 1) - |\mathsf{T}| \\
&= \log n \cdot \sum_{i \in I} (n_i \log n) + 2(|\mathsf{G}| - 1) - |\mathsf{T}| \\
&= n \log^2 n + 2(|\mathsf{G}| - 1) - |\mathsf{T}|
\end{aligned}
$$

using the fact that $\frac{n_i}{n} \leq \frac{1}{2}$ for $i \neq m$ (therefore $\log n - \log n_i \geq 1$).

Case $\mathsf{G} = \mathsf{q} \to \mathsf{r} : \{l_i : \mathsf{G}_i\}_{i \in I}$, $\mathsf{p} \in \{\mathsf{q}, \mathsf{r}\}$. Similar to the above case, with no merging. Thus this case can be performed in less time.

Case $\mathsf{G} = \mu \mathbf{t}.\mathsf{G}'$. Let $\mathsf{T}' = \mathsf{G}'{\upharpoonright}_{\mathsf{p}}$. We have $|\mathsf{T}| \leq |\mathsf{T}'| + 1$. Checking for closedness was precomputed, thus this is a constant-time check and a recursion: $1 + |\mathsf{G}'| \log^2 |\mathsf{G}'| + 2|\mathsf{G}'| - |\mathsf{T}'| \leq |\mathsf{G}| \log^2 |\mathsf{G}| + 2|\mathsf{G}| - |\mathsf{T}|$.

Case $\mathsf{G} = \mathbf{t}, \mathtt{end}$. This takes constant time; $|\mathsf{G}| \log^2 |\mathsf{G}| + 2|\mathsf{G}| - |\mathsf{T}| = 1$ is positive. □

C.3 Appendix for Complexity of Tirore *et al.*'s Binder-Agnostic Projection [21]

We prove Theorem 3.13, recalling all definitions needed from [21].

Adapted to our syntax, the function trans is defined identically to inductive projection (Definition 3.2) with the following merging function: $\mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_1$. Clearly, trans is not a sound projection function on its own, so we must rely on the checking of $\mathsf{G}{\downharpoonright}_{\mathsf{p}}\ \mathsf{trans}(\mathsf{G})$.

Definition C.7. The graph of G has nodes $\mathrm{Sub}(\mathsf{G})$ and has edges defined by the following rules:

$$\frac{\mathrm{unfold}(\mathsf{G}) = \mathsf{p} \to \mathsf{q} : [S];\ \mathsf{G}'}{\mathsf{G} \to \mathsf{G}'} \qquad \frac{\mathrm{unfold}(\mathsf{G}) = \mathsf{p} \to \mathsf{q} : \{l_i : \mathsf{G}_i\}_{i \in I}}{\mathsf{G} \to \mathsf{G}_i}$$

Definition C.8. The graph of (G, T), with respect to p, has nodes $\mathrm{Sub}(\mathsf{G}) \times \mathrm{Sub}(\mathsf{T})$, with edges defined by the following rules:

$$\frac{\mathrm{unfold}(\mathsf{G}) = \mathsf{q} \to \mathsf{r} : [S];\ \mathsf{G}' \qquad \mathrm{unfold}(\mathsf{T}) = \mathsf{p}?[S]; \mathsf{T}' \vee \mathrm{unfold}(\mathsf{T}) = \mathsf{p}![S]; \mathsf{T}' \qquad \mathsf{p} \in \{\mathsf{q}, \mathsf{r}\}}{(\mathsf{G}, \mathsf{T}) \to (\mathsf{G}', \mathsf{T}')}$$

$$\frac{\begin{array}{c}\mathrm{unfold}(\mathsf{T}) = \mathsf{p} \oplus \{l_i : \mathsf{T}_i\}_{i \in J} \vee \mathrm{unfold}(\mathsf{T}) = \mathsf{p} \& \{l_i : \mathsf{T}_i\}_{i \in J} \\ \mathrm{unfold}(\mathsf{G}) = \mathsf{q} \to \mathsf{r} : \{l_i : \mathsf{G}_i\}_{i \in I} \qquad \mathsf{p} \in \{\mathsf{q}, \mathsf{r}\} \qquad k \in I \cap J\end{array}}{(\mathsf{G}, \mathsf{T}) \to (\mathsf{G}_k, \mathsf{T}_k)}$$

$$\frac{\mathsf{G} \xrightarrow{\mathsf{G}} \mathsf{G}' \qquad \mathsf{p} \text{ not involved in head of unfold}(G)}{(\mathsf{G}, \mathsf{T}) \to (\mathsf{G}', \mathsf{T})}$$

Definition C.9. For a graph $G = (V, E)$ and predicate P, $\mathsf{sat}_P(\{\}, v)$ is a predicate that holds if P holds for all w such that w is reachable from v:

$$\mathsf{sat}_P(V, v) = \begin{cases} 1 & \text{if } v \in V \\ P(v) \wedge \bigwedge_{v \to u \in E} \mathsf{sat}_P(V, u) & \text{otherwise} \end{cases}$$

Definition C.10. The predicate $\mathsf{ProjPred}_{\mathsf{p}}(\mathsf{G}, \mathsf{T})$ is defined as follows:

$\mathsf{ProjPred}_{\mathsf{p}}(\mathsf{G}, \mathsf{T}) =$

$$\begin{cases} \mathsf{PL}_{\mathsf{p}}(\mathsf{LG}(\mathsf{G})) = \mathsf{LT}(\mathsf{T}) \wedge d_g(\mathsf{G}) = d_l(\mathsf{T}) & \text{if } \mathsf{PL}_{\mathsf{p}}(\mathsf{LG}(\mathsf{G})) \text{ is defined} \\ 0 < d_g(\mathsf{G}) & \text{if } \mathsf{partOf}_{\mathsf{p}}(\mathsf{G}) \text{ and } \mathsf{guarded}_{\mathsf{p}}(\mathsf{G}) \\ \mathsf{sat}_{\mathsf{UnravelPred}}(\{\}, \mathsf{G}) \wedge \neg \mathsf{partOf}_{\mathsf{p}}(\mathsf{G}) \wedge \mathrm{unfold}(\mathsf{T}) = \mathtt{end} & \text{otherwise} \end{cases}$$

All terms on the right-hand side of $\mathsf{ProjPred}_\mathsf{p}$, except for $\mathsf{sat}_{\mathsf{UnravelPred}}(\{\}, \mathsf{G})$, are simple constant-time checks (possibly with some linear-time precomputation), which we omit. Also, UnravelPred itself is a constant-time check. The following theorem is proven in [21]:

Theorem C.11. $\mathsf{G}\lfloor_\mathsf{p}\ \mathsf{T}$ if $\mathsf{sat}_{\mathsf{ProjPred}_\mathsf{p}}(\{\}, (\mathsf{G}, \mathsf{T}))$.

Therefore checking $\mathsf{sat}_{\mathsf{ProjPred}_\mathsf{p}}(\{\}, (\mathsf{G}, \mathsf{trans}(\mathsf{G})))$ is enough to decide projection.

C.3.1 Complexity

trans is easy to compute: merging takes constant time thus it can be computed in $\mathcal{O}(|\mathsf{G}|)$. Thus the main complexity of the algorithm comes from checking $\mathsf{sat}_{\mathsf{ProjPred}_\mathsf{p}}(\{\}, (\mathsf{G}, \mathsf{T}))$. The two functions to consider are $\mathsf{sat}_{\mathsf{ProjPred}_\mathsf{p}}$ and $\mathsf{sat}_{\mathsf{UnravelPred}}$. We have the following observations:

Observation C.12. For any graph $G = (V, E)$, $\mathsf{sat}_P(\{\}, v)$ can be computed in $\mathcal{O}\left(|V| \cdot T + |E|\right)$ time, where T is the time taken to compute $P(v)$.

Proof. Obvious; any graph search algorithm works. □

Observation C.13. The size of the graph of G is $\mathcal{O}(|\mathsf{G}|)$; the size of the graph of (G, T) is $\mathcal{O}(|\mathsf{G}| \cdot |\mathsf{T}|)$.

Proof. The number of nodes is bound by $|\mathrm{Sub}(\mathsf{G})|$ and $|\mathrm{Sub}(\mathsf{G})| \cdot |\mathrm{Sub}(\mathsf{T})|$ respectively; the number of edges for each graph can be bounded using structural induction. □

Thus we can show that checking the projection takes quadratic time:

Lemma C.14. Checking $\mathsf{sat}_{\mathsf{ProjPred}_\mathsf{p}}(\{\}, (\mathsf{G}, \mathsf{T}))$ can be done in $\mathcal{O}(|\mathsf{G}| \cdot |\mathsf{T}|)$ time.

Proof. First, computing $\mathsf{sat}_{\mathsf{UnravelPred}}(\{\}, \mathsf{G}')$ for all $\mathsf{G}' \in \mathrm{Sub}(\mathsf{G})$ takes $\mathcal{O}(|\mathsf{G}|)$ time as UnravelPred is a constant-time predicate (then using Observation C.12.)

After this precomputation, $\mathsf{ProjPred}_\mathsf{p}$ takes constant time to compute. So, checking $\mathsf{sat}_{\mathsf{ProjPred}_\mathsf{p}}(\{\}, (\mathsf{G}, \mathsf{T}))$ takes $\mathcal{O}(|\mathrm{Sub}(\mathsf{G}) \times \mathrm{Sub}(\mathsf{T})|) \leq \mathcal{O}(|\mathsf{G}| \cdot |\mathsf{T}|)$ time. □

Theorem 3.13. The projection in [21] takes $\mathcal{O}(|\mathsf{G}|^2)$ time.

Proof. We have shown above that checking $\mathsf{G}\downarrow_{\mathsf{p}}\ \mathsf{T}$ takes $\mathcal{O}(|\mathsf{G}| \cdot |\mathsf{T}|)$ time (Lemma C.14, Theorem C.11). Additionally, checking trans takes $\mathcal{O}(|\mathsf{G}|)$ time. Using the fact that $|\mathsf{T}| \leq |\mathsf{G}|$ (analogue of Lemma C.6 for trans), we have the result. □

D Appendix for Minimum Typing System

D.1 Proofs for Soundness and Completeness of Constraint Derivations

Lemma D.1 (Uniqueness of constraints)**.** Given a process P, there is a unique constraint set $\mathcal{C}$ (up to renaming) such that $\Gamma \vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$ for some $\Gamma, \xi, \mathcal{X}$. That is, if $\mathcal{C}$ and $\mathcal{C}'$ are two such sets, then there exists a one-to-one $\sigma : \mathbf{Sv} \to \mathbf{Sv}$, $\pi : \mathbf{Tv} \to \mathbf{Tv}$ such that $\pi\sigma\mathcal{C} = \mathcal{C}'$.

Proof. By inversion, note that there is only one rule that can be used for any P, thus the trees have the same shape. We map the fresh variables from one tree to the corresponding fresh variables in the other. Formally, we prove this by induction on the proof trees of $\Gamma \vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$ and $\Gamma' \vdash P : \xi' \mid_{\mathcal{X}'} \mathcal{C}'$.

Case [C-End]. Then $\mathcal{C} = \mathcal{C}' = \emptyset$. Take $\sigma = \emptyset$, $\pi = \emptyset$.

Case [C-Bra]. Then $P = \mathsf{p}\triangleright\{l_i : P_i\}_{i \in I}$.

Then we have proof trees for $\Gamma \vdash P_i : \xi_i \mid_{\mathcal{X}_i} \mathcal{C}_i$, $\Gamma' \vdash P_i : \xi'_i \mid_{\mathcal{X}'_i} \mathcal{C}'_i$, for all $i \in I$. By the inductive hypothesis, there exist one-to-one functions σ_i, π_i such that $\pi_i\sigma_i\mathcal{C}_i = \mathcal{C}'_i$. Let $\sigma = \bigcup_{i \in I} \sigma_i \cup \{\xi \mapsto \xi'\}$ and $\pi = \bigcup_{i \in I} \pi_i$. Then, as $\mathcal{X}_i, \xi, \xi'$ are disjoint, σ and π are one-to-one. Furthermore, $\pi\sigma\mathcal{C} = \mathcal{C}'$.

Other cases: Similar. □

Definition D.2 (Size of an expression)**.** The size of an expression e, denoted $|\mathsf{e}|$, is inductively defined as: $|\mathsf{true}| = |\mathsf{false}| = |\mathsf{n}| = |\mathsf{i}| = |x| = 1$, $|\neg\mathsf{e}| = |\mathtt{neg}(\mathsf{e})| = |\mathsf{e}| + 1$, and $|\mathsf{e} \vee \mathsf{e}'| = |\mathsf{e} + \mathsf{e}'| = |\mathsf{e} \oplus \mathsf{e}'| = |\mathsf{e}| + |\mathsf{e}'| + 1$.

Example D.3 (Constraint derivation of a process (2))**.** Let $P = \mu X.\mathsf{p}?(x).\mathsf{p}!\langle x\rangle.X$. Then the constraint derivation of P is the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{X:\xi, x:\alpha \vdash X:\xi_1 \mid_{\{\xi_1\}} \{\xi \le \xi_1\}}\ [\text{C-Var}] \quad \dfrac{}{X:\xi, x:\alpha \vdash x:\alpha \mid_{\emptyset} \emptyset}\ [\text{C-SortVar}]}{X:\xi, x:\alpha \vdash \mathsf{p}!\langle x\rangle.X:\xi_2 \mid_{\{\xi_1,\xi_2\}} \{\xi \le \xi_1, \mathsf{p}![\alpha]; \xi_1 \le \xi_2\}}\ [\text{C-Out}]}{X:\xi \vdash \mathsf{p}?(x).\mathsf{p}!\langle x\rangle.X:\xi \mid_{\{\xi_1,\xi_2,\xi_3,\alpha\}} \mathcal{C}}\ [\text{C-In}]}{\vdash P:\xi \mid_{\{\xi,\xi_1,\xi_2,\xi_3,\alpha\}} \mathcal{C}}\ [\text{C-Rec}]$$

where the generated set of constraints $\mathcal{C} = \{\xi \le \xi_1, \mathsf{p}![\alpha]; \xi_1 \le \xi_2, \mathsf{p}?[\alpha]; \xi_2 \le \xi\}$. (σ, π) is a solution to $\mathcal{C}$ where $\sigma = \{\xi, \xi_2 \mapsto \mu\mathbf{t}.\mathsf{p}?[\alpha]; \mathsf{p}![\alpha]; \mathbf{t}, \xi_1 \mapsto \mu\mathbf{t}.\mathsf{p}![\alpha]; \mathsf{p}?[\alpha]; \mathbf{t}\}$ and $\pi = \{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{int}\}$. Soundness implies that $\vdash P : \mathsf{T}$ where $\mathsf{T} = \pi\sigma\xi = \mu\mathbf{t}.\mathsf{p}?[\texttt{int}]; \mathsf{p}![\texttt{int}]; \mathbf{t}$; however, note that T is not the only type of P (e.g. $\mu\mathbf{t}.\mathsf{p}?[\texttt{bool}]; \mathsf{p}?[\texttt{bool}]; \mathbf{t}$ is another type of P).

Lemma D.4. The number of judgements in a proof tree of $\Gamma \vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$ is $\mathcal{O}(|P|)$.

Proof. By induction on the proof tree. Note that we never unfold recursions in the judgement and the sum of the sizes of the processes in the premises is strictly less than the size of P. Thus, the number of judgements in the proof tree is bounded by $|P|$. □

Corollary D.4.1. $|\mathcal{C}| = \mathcal{O}(|P|)$.

Proof. Note that the number of constraints added per judgement in the proof tree is bounded by a constant. □

Theorem 4.5 (Soundness of constraints)**.** If $\vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$ is derivable and (σ, π) is a solution to $\mathcal{C}$, then P has type $\pi\sigma\xi$.

Proof. We show that if $\Gamma \vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$ is derivable and (σ, π) is a solution to $\mathcal{C}$, then there is a typing derivation for $\pi\sigma\Gamma \vdash P : \pi\sigma\xi$ (where we define $\pi\sigma\Gamma$ to apply $\pi\sigma$ to each type in Γ). Also, we prove this for sorts: if $\Gamma \vdash \mathsf{e} : \alpha \mid_{\mathcal{X}} \mathcal{C}$ is derivable then $\pi\sigma\Gamma \vdash \mathsf{e} : \pi\sigma\alpha$.

We prove this by induction on the proof tree of the constraint derivations.

Case [C-End]. Then we can use [T-Inact] followed by [T-Sub].

Case [C-Var]. Then we can use [T-Var].

Case [C-Rec]. Then by the inductive hypothesis there is a typing derivation for $\pi\sigma\Gamma, \mathbf{t} : \pi\sigma\xi \vdash P : \pi\sigma\xi$. We can use [T-Rec] to get $\pi\sigma\Gamma \vdash \mu\mathbf{t}.P : \pi\sigma\xi$.

Case [C-In]. If $\pi\sigma$ is a solution to $\mathcal{C}'$ then it is also a solution of $\mathcal{C}$. By the inductive hypothesis, there is a typing derivation for $\pi\sigma\Gamma, x : \pi\alpha \vdash P : \pi\sigma\psi$. We also have $\mathsf{p}?[\pi\alpha]; \pi\sigma\psi \leq \pi\sigma\xi$. We can then do the following to create the required type derivation:

$$\dfrac{\dfrac{\pi\sigma\Gamma, x : \pi\alpha \vdash P : \pi\sigma\psi}{\pi\sigma\Gamma \vdash \mathsf{p}?(x).P : \mathsf{p}?[\pi\sigma S]; \pi\sigma\psi}\ [\text{T-In}] \qquad \mathsf{p}?[\pi\alpha]; \pi\sigma\psi \leq \pi\sigma\xi}{\pi\sigma\Gamma \vdash \mathsf{p}?(x).P : \pi\sigma\xi}\ [\text{T-Sub}]$$

Case [C-Out]. We have that $\pi\sigma$ is a solution of $\mathcal{C}_1$ and $\mathcal{C}_2$. By the inductive hypothesis, there are typing derivations for $\pi\sigma\Gamma \vdash P : \pi\sigma\psi$ and $\pi\sigma\Gamma \vdash \mathsf{e} : \pi\alpha$. As (σ, π) is a solution to $\mathcal{C}'$, $\pi\sigma\Gamma \vdash \mathsf{p}!\langle\mathsf{e}\rangle.P : \pi\sigma\xi$. Proceed with the following derivation:

$$\dfrac{\dfrac{\pi\sigma\Gamma \vdash P : \pi\sigma\xi \qquad \pi\sigma\Gamma \vdash \mathsf{e} : \pi\alpha}{\pi\sigma\Gamma \vdash \mathsf{p}!\langle\mathsf{e}\rangle.P : \mathsf{p}![\pi\alpha]; \pi\sigma\psi}\ [\text{T-Out}] \qquad \mathsf{p}![\pi\alpha]; \pi\sigma\psi \leq \pi\sigma\xi}{\pi\sigma\Gamma \vdash \mathsf{p}!\langle\mathsf{e}\rangle.P : \pi\sigma\xi}\ [\text{T-Sub}]$$

Case [C-Bra], [C-Sel]. Similar to the above: use rules [T-Bra] then [T-Sub], and [T-Sel] then [T-Sub] respectively.

Case [C-Cond]. We have that (σ, π) is a solution of $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$. By the inductive hypothesis, there are typing derivations for $\pi\sigma\Gamma \vdash P : \pi\sigma\psi_1$, $\pi\sigma\Gamma \vdash P' : \pi\sigma\psi_2$ and $\pi\sigma\Gamma \vdash \mathsf{e} : \alpha$, with $\alpha = \texttt{bool} \in \mathcal{C}_3$. Thus $\pi\alpha = \texttt{bool}$.

$$\dfrac{\dfrac{\pi\sigma\Gamma \vdash P : \pi\sigma\psi_1 \qquad \pi\sigma\psi_1 \leq \pi\sigma\xi}{\pi\sigma\Gamma \vdash P : \pi\sigma\psi} \qquad \dfrac{\pi\sigma\Gamma \vdash P' : \pi\sigma\psi_2 \qquad \pi\sigma\psi_2 \leq \pi\sigma\xi}{\pi\sigma\Gamma \vdash P' : \pi\sigma\xi} \qquad \pi\sigma\Gamma \vdash \mathsf{e} : \texttt{bool}}{\pi\sigma\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ P' : \pi\sigma\xi}\ [\text{T-Cond}]$$

(where [T-Sub] is used in the top subtrees).

Case [C-SortVar]. We have $x : \alpha \in \Gamma$. Therefore we can use [T-SortVar]:

$$\dfrac{x : \pi\alpha \in \pi\sigma\Gamma}{\pi\sigma\Gamma \vdash x : \pi\alpha}\ [\text{T-SortVar}]$$

Case [C-True]. We have $\texttt{bool} = \alpha \in \mathcal{C}$. Then we can use [T-True]:

$$\dfrac{}{\pi\sigma\Gamma \vdash \mathsf{true} : \pi\alpha}\ [\text{T-True}]$$

Case [C-Nat]. Then we can use [T-Nat].

Case [C-Int]. Then we can use [T-Int].

Case [C-Not]. By the inductive hypothesis, there is a typing derivation for $\pi\sigma\Gamma \vdash \mathsf{e} : \texttt{bool}$. We can use [T-Not], noting that $\pi\alpha = \texttt{bool}$:

$$\frac{\pi\sigma\Gamma \vdash \mathsf{e} : \mathtt{bool}}{\pi\sigma\Gamma \vdash \neg\mathsf{e} : \mathtt{bool}}\ [\text{T-Not}]$$

Case [C-Or], [C-Add], [C-Neg]. Similar to the above.

Case [C-Nondet]. We have that (σ, π) is a solution of $\mathcal{C}$, so $\pi\alpha_1 = \pi\alpha_2 = \pi\beta$. By the inductive hypothesis, there are typing derivations for $\pi\sigma\Gamma \vdash \mathsf{e}_1 : \pi\alpha_1$ and $\pi\sigma\Gamma \vdash \mathsf{e}_2 : \pi\alpha_2$. We can use the following derivation:

$$\frac{\pi\sigma\Gamma \vdash \mathsf{e}_1 : \pi\alpha_1 \qquad \pi\sigma\Gamma \vdash \mathsf{e}_2 : \pi\alpha_2}{\pi\sigma\Gamma \vdash \mathsf{e}_1 \oplus \mathsf{e}_2 : \pi\beta}\ [\text{T-Nondet}]$$ □

Lemma D.5. Let $\Gamma \vdash P : \mathsf{T}$ be derivable. Then there exists a derivation of $\Gamma' \vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$ such that $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$.

Proof. By matching each typing rule with the corresponding constraint rule. □

Corollary D.5.1. Let $\Gamma \vdash P : \mathsf{T}$ and $\Gamma' \vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$ be derivable. Then $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$.

Proof. Direct consequence of Lemmas D.5 and D.1. □

Theorem 4.6 (Completeness of constraints)**.** Let $\vdash P : \mathsf{T}_0$ and $\vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$. Then there exists a solution (σ', π') of $\mathcal{C}$ such that $\pi'\sigma'\xi = \mathsf{T}_0$.

Proof. We show that if $\Gamma \vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$, and $\pi\sigma\Gamma \vdash P : \mathsf{T}_0$, such that $\mathrm{dom}(\sigma) \cap \mathcal{X} = \emptyset$ and $\mathrm{dom}(\pi) \cap \mathcal{X} = \emptyset$, then there exists a solution (σ', π') of $\mathcal{C}$ such that $\pi'\sigma'\xi = \mathsf{T}_0$, $\sigma' \setminus \mathcal{X} = \sigma$, and $\pi' \setminus \mathcal{X} = \pi$.

We also show the analogous statement for sorts, under the same conditions as the above statement for types: if $\Gamma \vdash \mathsf{e} : \alpha \mid_{\mathcal{X}} \mathcal{C}$, and $\pi\sigma\Gamma \vdash \mathsf{e} : S$, then there exists a solution (σ', π') of $\mathcal{C}$ such that $\pi'\alpha = S$, $\sigma' \setminus \mathcal{X} = \sigma$, and $\pi' \setminus \mathcal{X} = \pi$.

We do this by induction on the proof tree of the type derivation. For the following sort cases, we will consider the first rule in the derivation.

Case [T-SortVar]. Then $\mathtt{e} = x$, $x : \alpha \in \pi\sigma\Gamma$. By inversion we have that [C-SortVar] was used, and so $\mathcal{C} = \emptyset$, $\mathcal{X} = \emptyset$. Take $\sigma' = \sigma, \pi' = \pi$. Then (σ', π') is a solution of $\mathcal{C}$ because $\pi\alpha = S$.

Case [T-True]. Then $\mathtt{e} = \mathtt{true}$, $S = \mathtt{bool}$. By inversion we have that [C-True] was used, and so $\mathcal{C} = \{\mathtt{bool} = \alpha\}$, $\mathcal{X} = \{\alpha\}$. Take $\sigma' = \sigma, \pi' = \pi \cup \{\alpha \mapsto \mathtt{bool}\}$. Then (σ', π') is a solution of $\mathcal{C}$ because $\pi\alpha = \mathtt{bool}$.

Case [T-Nondet]. Then $\mathtt{e} = \mathtt{e}_1 \oplus \mathtt{e}_2$, $\pi\sigma\Gamma \vdash \mathtt{e}_1 : S$, $\Gamma \vdash \mathtt{e}_2 : S$. By inversion we have that [C-Nondet] was used, and so $\Gamma \vdash \mathtt{e}_1 : \alpha_1 \mid_{\mathcal{X}_1} \mathcal{C}_1$, $\Gamma \vdash \mathtt{e}_2 : \alpha_2 \mid_{\mathcal{X}_2} \mathcal{C}_2$, $\mathcal{X} = \mathcal{X}_1 \uplus \mathcal{X}_2 \uplus \{\beta\}$, $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\alpha_1 = \beta, \alpha_2 = \beta\}$.

By the inductive hypothesis, there exist solutions (σ_1, π_1) of $\mathcal{C}_1$ and (σ_2, π_2) of $\mathcal{C}_2$ such that $\sigma_1 \setminus \mathcal{X}_1 = \sigma$, $\sigma_2 \setminus \mathcal{X}_2 = \sigma$, $\pi_1 \setminus \mathcal{X}_1 = \pi_1$, $\pi_2 \setminus \mathcal{X}_2 = \pi_2$, $\pi_1\alpha_1 = S$ and $\pi_2\alpha_2 = S$. Take $\sigma' = \sigma_1 \cup \sigma_2, \pi' = \pi_1 \cup \pi_2 \cup \cup\{\beta \mapsto S\}$. We have that $\pi'\beta = S$ as desired.

Case [T-False], [T-Int], [T-Nat], [T-Not], [T-Or], [T-Add], [T-Neg]. Similar to the above.

For the type cases, we will consider the first non-[T-Sub] rule used in the derivation, so the tree covered is of shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\pi\sigma\Gamma \vdash P : \mathsf{T}_k}\,[\mathrm{R}] \qquad \mathsf{T}_k \leq \mathsf{T}_{k-1}}{\pi\sigma\Gamma \vdash P : \mathsf{T}_{k-1}} \qquad \mathsf{T}_{k-1} \leq \mathsf{T}_{k-2}}{\vdots} \qquad \mathsf{T}_1 \leq \mathsf{T}_0}{\pi\sigma\Gamma \vdash P : \mathsf{T}_0}$$

Thus we have that $\pi\sigma\Gamma \vdash P : \mathsf{T}_k$ is derivable and $\mathsf{T}_k \leq \mathsf{T}_0$ by transitivity of subtyping.

We will now consider the rule [R] used to derive $\pi\sigma\Gamma \vdash P : \mathsf{T}_k$.

Case [T-Inact]. Then $P = \mathbf{0}$, $\mathsf{T}_k = \mathtt{end}$. By inversion we have that [C-End] was used, and so $\mathcal{C} = \{\mathtt{end} \leq \xi\}$, $\mathcal{X} = \{\xi\}$. Take $\sigma' = \sigma \cup \{\xi \mapsto \mathsf{T}_0\}, \pi' = \pi$. Then (σ', π') is a solution of $\mathcal{C}$ because $\mathtt{end} = \mathsf{T}_k \leq \mathsf{T}_0$.

Case [T-Var]. Then $P = X$, $X : \mathsf{T}_k \in \pi\sigma\Gamma$. By inversion we have that [C-Var] was used, and so $\mathcal{C} = \emptyset$, $X : \xi \in \Gamma$.

Therefore $\pi\sigma\xi = \mathsf{T}_k$. Take $\sigma' = \sigma \cup \{\xi \mapsto \mathsf{T}_0\}, \pi' = \pi$. Then (σ', π') is a solution of $\mathcal{C}$ because $\mathsf{T}_k \leq \mathsf{T}_0$. Furthermore, $\pi'\sigma'\xi = \mathsf{T}_0$ as desired.

Case [T-Rec]. We can assume that no [T-Sub] rule was used directly below [T-Rec], as we could have done the subtyping directly above instead. So $\mathsf{T}_k = \mathsf{T}_0$.

Then $P = \mu X.P'$, $\pi\sigma\Gamma, X : \mathsf{T}_k \vdash P' : \mathsf{T}_k$. By inversion we have that [C-Rec] was used, and so $\Gamma, \mathbf{t} : \xi \vdash P' : \xi \mid_{\mathcal{X}} \mathcal{C}$.

By the inductive hypothesis there exists a solution (σ_1, π_1) of $\mathcal{C}$ such that $\sigma_1 \setminus \mathcal{X} = \sigma$, $\pi_1 \setminus \mathcal{X} = \pi$, $\pi_1\sigma_1\xi = \mathsf{T}_k$. Taking $\sigma' = \sigma_1, \pi' = \pi_1$ works because $\mathsf{T}_k = \mathsf{T}_0$.

Case [T-In]. Then $P = \mathsf{p}?(x).P'$, $\mathsf{T}_k = \mathsf{p}?[S]; \mathsf{T}'$, $\pi\sigma\Gamma, x : S \vdash P' : \mathsf{T}'$. By inversion we have that [C-In] was used: $\Gamma, x : \alpha \vdash P' : \psi \mid_{\mathcal{X}'} \mathcal{C}'$, $\mathcal{X} = \mathcal{X}' \uplus \{\xi, \alpha\}$, $\mathcal{C} = \mathcal{C}' \cup \{\mathsf{p}?[\alpha]; \psi \leq \xi\}$, α, ψ fresh.

Let $\sigma'' = \sigma$ and $\pi'' = \pi \uplus \{\alpha \mapsto S\}$ and $\Gamma' = \Gamma, x : \alpha$. Then $\pi''\sigma''\Gamma' \vdash P' : \mathsf{T}'$. By the inductive hypothesis, there exists a solution (σ_1, π_1) of $\mathcal{C}'$ such that $\sigma_1 \setminus \mathcal{X}' = \sigma''$, $\pi_1 \setminus \mathcal{X}' = \pi''$, $\pi_1\sigma_1\psi = \mathsf{T}'$. Take $\sigma' = \sigma_1 \cup \{\xi \mapsto \mathsf{T}_0\}, \pi' = \pi_1$. Then $\pi'\sigma'\xi = \mathsf{T}_0$ and $\pi'\alpha = S$ and $\mathsf{p}?[S]; \mathsf{T}' = \mathsf{T}_k \leq \mathsf{T}_0$. Therefore (σ', π') is a solution of $\mathcal{C}$ as desired.

Also, $\sigma' \setminus \mathcal{X} = (\sigma_1 \cup \{\xi \mapsto \mathsf{T}_0\}) \setminus (\mathcal{X}' \uplus \{\xi, \alpha\}) = \sigma_1 \setminus \mathcal{X} = \sigma$ and $\pi' \setminus \mathcal{X} = (\pi_1 \cup \{\alpha \mapsto S\}) \setminus (\mathcal{X}' \uplus \{\xi, S'\}) = \pi'' \setminus \mathcal{X} = \pi$, as desired.

Case [T-Out]. Then $P = \mathsf{p}!\langle \mathsf{e} \rangle.P'$, $\mathsf{T}_k = \mathsf{p}![S]; \mathsf{T}'$, $\pi\sigma\Gamma \vdash P' : \mathsf{T}''$ and $\pi\sigma\Gamma \vdash \mathsf{e} : S$. By inversion we have that [C-Out] was used, and so $\Gamma \vdash P' : \psi \mid_{\mathcal{X}_1} \mathcal{C}_1$, $\Gamma \vdash \mathsf{e} : \alpha \mid_{\mathcal{X}_2} \mathcal{C}_2$, $\mathcal{X} = \mathcal{X}_1 \uplus \mathcal{X}_2 \uplus \{\xi\}$, $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathsf{p}![\alpha]; \psi \leq \xi\}$.

By the inductive hypothesis, there exist solutions (σ_1, π_1) of $\mathcal{C}_1$ and (σ_2, π_2) of $\mathcal{C}_2$ such that $\sigma_1 \setminus \mathcal{X}_1 = \sigma$, $\sigma_2 \setminus \mathcal{X}_2 = \sigma$, $\pi_1\sigma_1\psi = \mathsf{T}'$, $\pi_2\alpha = S$. Take $\sigma' = \sigma_1 \cup \sigma_2 \cup \{\xi \mapsto \mathsf{T}_0\}$ and $\pi' = \pi$. σ' is a valid mapping because σ_1 and σ_2 only differ on $\mathcal{X}_1$ and $\mathcal{X}_2$ (which are disjoint due to freshness), and ξ is fresh.

We have that $\pi'\sigma'\xi = \mathsf{T}_0$, and $\mathsf{p}![S]; \mathsf{T}' = \mathsf{T}_k \leq \mathsf{T}_0$. Therefore (σ', π') is a solution of $\mathcal{C}$ as desired. Also, $\sigma' \setminus \mathcal{X} = (\sigma_1 \setminus \mathcal{X}_1) \cup (\sigma_2 \setminus \mathcal{X}_2) = \sigma$ and $\pi' \setminus \mathcal{X} = (\pi_1 \setminus \mathcal{X}_1) \cup (\pi_2 \setminus \mathcal{X}_2) = \pi$ by freshness of $\xi, \mathcal{X}_1, \mathcal{X}_2$.

Case [T-Bra], [T-Sel], [T-Cond]. Similar to the above. □

Observation D.6. All constraints generated from the rules of Figure 6 are of the form:

- $\mathtt{end} \leq \xi$
- $\xi \leq \psi$
- $\mathsf{p}?[\beta]; \xi \leq \psi$
- $\mathsf{p}![\beta]; \xi \leq \psi$
- $\mathsf{p}\&\{l_i : \xi_i\}_{i \in I} \leq \psi$
- $\mathsf{p}\oplus\{l_i : \xi_i\}_{i \in I} \leq \psi$
- $S = S'$ for $S, S' \in \mathbf{S}$

We call the last type of constraint a *sort constraint*, and the rest *type constraints*.

D.2 Appendix for Solving the Minimum Type

Definition D.7 (Constraint variables)**.** For a set of constraints $\mathcal{C}$, let tvars($\mathcal{C}$) denote the type variables occuring in $\mathcal{C}$. We often write $\varphi \in \mathcal{C}$ if $\varphi \in \text{tvars}(\mathcal{C})$.

We can now define a procedure to find the minimum type of a process.

Lemma D.8. For constraint set $\mathcal{C}$ created from Figure 6 containing $\xi \leq \psi$, if (σ, π) is a solution to $\mathcal{C}' = \mathcal{C}[\xi/\psi]$ then $(\sigma_{[\psi \mapsto \sigma(\xi)]}, \pi)$ is a solution to $\mathcal{C}$.

Proof. By considering each rule in $\mathcal{C}$, and using transitivity of subtyping. □

Definition D.9. Define $\mathtt{tr}(\mathcal{C})$ to be the constraint set created by the following procedure: while $\xi \leq \psi$ occurs in $\mathcal{C}$, perform the substitution $[\xi/\psi]$ on $\mathcal{C}$.

Lemma D.8 tells us that it is sufficient to find a solution to $\mathtt{tr}(\mathcal{C})$, which can be used to reconstruct a solution to $\mathcal{C}$.

Definition D.10. Define the dependencies of T in $\mathcal{C}$, $\mathtt{dep}_{\mathcal{C}}(\mathsf{T})$, to be the set $\{\mathsf{T}' \mid \mathsf{T}' \leq \mathsf{T} \in \mathcal{C}\}$. Extend this to sets of type variables: $\mathtt{dep}_{\mathcal{C}}(\mathcal{S}) = \bigcup_{\psi \in \mathcal{S}} \mathtt{dep}(\psi)$.

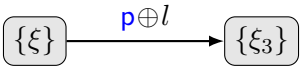
Definition D.11 (Minimum type graph)**.** Let $\vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}'$. Let $\mathcal{C} = \mathtt{tr}(\mathcal{C}')$. We define the following procedure for generating the minimum type graph of P, $\mathbb{G}_{\mathcal{C}}$, as well as the sort constraints $\mathcal{C}_{\mathsf{S}}$. Initially, $\mathcal{C}_{\mathsf{S}}$ contains all sort constraints of $\mathcal{C}$. The vertices of $\mathbb{G}_{\mathcal{C}}$ are subsets of tvars($\mathcal{C}$), and edges are generated by the following rules.

- If $\mathsf{T}_i = \mathsf{p}?[\beta_i];\xi_i$ for all T_i in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S})$, then $\mathcal{S} \xrightarrow{\mathsf{p}?[\alpha]} \{\xi_i \mid \mathsf{T}_i \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S})\}$; add $\alpha = \beta_i$ to $\mathcal{C}_{\mathsf{S}}$.
- If $\mathsf{T}_i = \mathsf{p}![\beta_i];\xi_i$ for all T_i in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S})$, then $\mathcal{S} \xrightarrow{\mathsf{p}![\alpha]} \{\xi_i \mid \mathsf{T}_i \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S})\}$; add $\alpha = \beta_i$ to $\mathcal{C}_{\mathsf{S}}$.
- If $\mathsf{T}_i = \mathsf{p}\&\{l_j : \xi_{ij}\}_{j\in J_i}$ for all T_i in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S})$, then $\mathcal{S} \xrightarrow{\mathsf{p}\&l_j} \{\xi_{ij} \mid \mathsf{T}_i \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S})\}$ for each $j \in \bigcap_i J_i$. If $\bigcap_i J_i = \emptyset$, then the graph is undefined.
- If $\mathsf{T}_i = \mathsf{p}\oplus\{l_j : \xi_{ij}\}_{j\in J_i}$ for all T_i in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S})$, then $\mathcal{S} \xrightarrow{\mathsf{p}\oplus l_j} \{\xi_{ij} \mid \mathsf{T}_i \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}) \wedge j \in J_i\}$ for each $j \in \bigcup_i J_i$.
- If $\mathsf{T}_i = \mathtt{end}$ for all T_i in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S})$, then $\mathcal{S} \xrightarrow{\mathsf{end}} \mathsf{Skip}$.

We only consider nodes that are reachable from $\{\xi\}$. If none of the above rules apply, the graph is undefined.

The minimum type graph is a type graph (Definition 2.1) corresponding to some valid type. We will sometimes write nodes $\mathcal{S}$ to stand for their corresponding types.

Example D.12 (A minimum type graph of an untypable process)**.**
Let $P = \mathsf{p}\triangleleft l.\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ \mathsf{p}!\langle 1\rangle.\mathbf{0}\ \mathsf{else}\ \mathsf{p}?(x).\mathbf{0}$. Intutively, this is not typable because the the branches of the conditional statement are not subtypes of a common type. The constraints are $\mathtt{tr}(\mathcal{C}) = \{\alpha_1 = \mathtt{bool}, \alpha_2 = \mathtt{int}, \mathsf{p}![\alpha_2];\xi_1 \leq \xi_3, \mathsf{p}?[\alpha_3];\xi_2 \leq \xi_3, \mathsf{p}\oplus\{l : \xi_3\} \leq \xi\}$, with $\vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}$. If we try to build a minimum type graph:



but $\mathtt{dep}_{\mathtt{tr}(\mathcal{C})}(\{\xi_3\}) = \{\mathsf{p}![\alpha_2];\xi_1, \mathsf{p}?[\alpha_3];\xi_2\}$ has mismatching communications, so by Definition D.11, the minimum type graph is undefined. Thus no minimum type exists.

Lemma D.13. Let $\mathbb{G}_{\mathcal{C}}$ be a minimum type graph. Let π be a most general solution of $\mathcal{C}_{\mathsf{S}}$. If $\mathcal{S}_1 \subseteq \mathcal{S}_2$, then either $\mathcal{S}_2$ is undefined or $\pi\mathcal{S}_1 \leq \pi\mathcal{S}_2$.

Proof. Intuitively, each type variable in the set $\mathcal{S}$ specifies some set of behaviours that must be followed in node $\mathcal{S}$: thus $\mathcal{S}_1$ is less constrained. Formally, we show that $\sim = \{(\mathcal{S}, \mathcal{S}') \mid \mathcal{S} \subseteq \mathcal{S}'\} \cup \{(\mathsf{Skip}, \mathsf{Skip})\}$ is a type simulation. Let $\mathcal{S}_1 \subseteq \mathcal{S}_2$ be defined.

- If $\mathsf{T}_i = \mathsf{p}?[\beta_i];\xi_i$ for all T_i in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1)$, then $\mathcal{S}_1 \xrightarrow{\mathsf{p}?[\alpha_1]} \{\xi_i \mid \mathsf{T}_i \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1)\}$, and $\beta_i = \alpha_1 \in \mathcal{C}_{\mathsf{S}}$. Then, as $\mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1) \subseteq \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)$, we have that $\mathcal{S}_2 \xrightarrow{\mathsf{p}?[\alpha_2]} \{\xi_j \mid \mathsf{T}_j \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)\}$, and $\beta_j = \alpha_2 \in \mathcal{C}_{\mathsf{S}}$. Thus we have $\pi\alpha_1 = \pi\alpha_2 = \beta_j$ for all $j \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)$. Also, $\{\xi_j \mid \mathsf{T}_j \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1)\} \sim \{\xi_i \mid \mathsf{T}_i \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)\}$, so it is consistent.
- $\mathsf{T}_j = \mathsf{p}![\beta_i];\xi_j$ for all T_j in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1)$ is symmetric.
- If $\mathsf{T}_k = \mathsf{p}\&\{l_j : \xi_{kj}\}_{j\in J_k}$ for all T_k in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)$, then $\mathcal{S}_2 \xrightarrow{\mathsf{p}\&l_j} \{\xi_{kj} \mid \mathsf{T}_j \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)\}$. Thus $j \in \bigcap_k J_k$, so for all T_i in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1)$, $j \in J_i$. Therefore, $\mathcal{S}_1 \xrightarrow{\mathsf{p}\&l_j} \{\xi_{ij} \mid \mathsf{T}_j \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1)\}$. Also, $\{\xi_j \mid \mathsf{T}_j \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1)\} \sim \{\xi_i \mid \mathsf{T}_i \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)\}$, so it is consistent.
- If $\mathsf{T}_i = \mathsf{p}\oplus\{l_j : \xi_{ij}\}_{j\in J_i}$ for all T_i in $\mathtt{dep}_{C}(\mathcal{S}_1)$, then $\mathcal{S}_1 \xrightarrow{\mathsf{p}\oplus l_j} \{\xi_{ij} \mid \mathsf{T}_j \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1)\}$. Thus $j \in \bigcup J_i$, so there exists T_k in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)$ such that $j \in J_k$. Therefore, $\mathcal{S}_2 \xrightarrow{\mathsf{p}\oplus l_j} \{\xi_{kj} \mid \mathsf{T}_k \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)\}$. Also, $\{\xi_j \mid \mathsf{T}_j \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1)\} \sim \{\xi_i \mid \mathsf{T}_i \in \mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)\}$, so it is consistent.
- If $\mathsf{T}_i = \mathtt{end}$ for all T_i in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S}_1)$, then $\mathcal{S}_1 \xrightarrow{\mathsf{end}} \mathsf{Skip}$. Then there exists $\mathsf{T}_j = \mathtt{end}$ in $\mathtt{dep}_{\mathcal{C}}(\mathcal{S}_2)$, so $\mathcal{S}_2 \xrightarrow{\mathsf{end}} \mathsf{Skip}$.

Thus $\sim$ is a type simulation. □

Theorem 4.11 (Soundness of minimum type inference)**.** Let $\vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}'$ and $\mathcal{C} = \mathtt{tr}(\mathcal{C}')$. Let the minimum type graph be $\mathbb{G}_{\mathcal{C}}$ with sort constraints $\mathcal{C}_{\mathsf{S}}$. Let $\sigma = \{\psi \mapsto \{\psi\} \mid \psi \in \mathcal{C}\}$ and π be a most general solution of $\mathcal{C}_{\mathsf{S}}$. Then (σ, π) is a solution to $\mathcal{C}$.

Proof. All sort constraints are satisfied because π solves $\mathcal{C}_{\mathsf{S}}$ which is a superset of the sort constraints of $\mathcal{C}$. We will show that all type constraints are satisfied.

Case $\mathtt{end} \leq \xi$. Then $\pi\sigma\mathtt{end} = \mathtt{end}$, $\pi\sigma\xi = \pi\{\xi\}$. By Definition D.11, $\{\xi\} \xrightarrow{\mathsf{end}}$ so $\pi\{\xi\} = \mathtt{end}$. Thus $\pi\sigma\mathtt{end} \leq \pi\sigma\xi$.

Case $\mathsf{p}?[\alpha];\xi \leq \psi$. Then $\pi\sigma(\mathsf{p}?[\alpha];\xi) = \mathsf{p}?[\pi\alpha];\pi\{\xi\}$. We have $\pi\sigma\psi = \pi\{\psi\} \xrightarrow{\mathsf{p}?[\pi\alpha]} \pi\mathcal{S}$. Then by Definition D.11, $\pi\{\xi\} \subseteq \mathcal{S}$. Therefore, by Lemma D.13, the constraint is satisfied.

Other cases are similar. □

We write $\pi\mathbb{G}_{\mathcal{C}}$ to be the graph with π applied to all transitions.

Theorem 4.12 (Completeness of minimum type inference)**.** Let $\vdash P : \xi \mid_{\mathcal{X}} \mathcal{C}'$, and $\mathcal{C} = \mathtt{tr}(\mathcal{C}')$. Let (σ, π) be a solution to $\mathcal{C}$. Let $\mathbb{G}_{\mathcal{C}}, \mathcal{C}_{\mathsf{S}}$ be the minimum graph and sort constraints, respectively. If π' is the most general solution to $\mathcal{C}_{\mathsf{S}}$, then $\pi'\mathbb{G}_{\mathcal{C}}$ is a type graph (Definition 2.1) that corresponds to a minimum type (Definition 4.2) $\mathsf{T}_{\mathsf{min}}$ of P.

Proof. Let $\mathsf{T} = \pi\sigma\xi$. By Theorems 4.5 and 4.6, it suffices to show that there exists π'' such that $\pi''\mathsf{T}_{\mathsf{min}} \leq \mathsf{T}$.

Let $V(\pi'\mathbb{G}_{\mathcal{C}})$ denote the vertices of the type graph $\pi'\mathbb{G}_{\mathcal{C}}$. Define the simulation $\sim$ on $V(\pi'\mathbb{G}_{\mathcal{C}}) \times \mathrm{Sub}(\mathsf{T})$ as the least relation satisfying the following rules. We will eventually prove that $\sim$ is a type simulation; $\mathcal{S} \sim \mathsf{T}$ intuitively means "T must respect the behaviours of all $\psi \in \mathcal{S}$". Initially, $\{\xi\} \sim \mathsf{T}$ means that T must respect the behaviour of ξ, which is the variable corresponding to $\mathsf{T}_{\mathsf{min}}$.

$$\frac{}{\{\xi\} \sim \mathsf{T}} \text{ [Sim-Init]}$$

$$\frac{\mathcal{S}_1 \xrightarrow{\mathsf{p}?[\pi'\beta]} \mathcal{S}_2 \qquad \mathsf{T}_1 \xrightarrow{\mathsf{p}?[S]} \mathsf{T}_2 \qquad \mathcal{S}_1 \sim \mathsf{T}_1}{\mathcal{S}_2 \sim \mathsf{T}_2} \text{ [Sim-In]}$$

$$\frac{\mathcal{S}_1 \xrightarrow{\mathsf{p}![\pi'\beta]} \mathcal{S}_2 \qquad \mathsf{T}_1 \xrightarrow{\mathsf{p}![S]} \mathsf{T}_2 \qquad \mathcal{S}_1 \sim \mathsf{T}_1}{\mathcal{S}_2 \sim \mathsf{T}_2} \text{ [Sim-Out]}$$

$$\frac{\mathcal{S}_1 \xrightarrow{\mathsf{p}\&l} \mathcal{S}_2 \qquad \mathsf{T}_1 \xrightarrow{\mathsf{p}\&l} \mathsf{T}_2 \qquad \mathcal{S}_1 \sim \mathsf{T}_1}{\mathcal{S}_2 \sim \mathsf{T}_2} \text{ [Sim-Bra]}$$

$$\frac{\mathcal{S}_1 \xrightarrow{\mathsf{p}\oplus l} \mathcal{S}_2 \qquad \mathsf{T}_1 \xrightarrow{\mathsf{p}\oplus l} \mathsf{T}_2 \qquad \mathcal{S}_1 \sim \mathsf{T}_1}{\mathcal{S}_2 \sim \mathsf{T}_2} \text{ [Sim-Sel]}$$

Claim D.14. For all $\mathcal{S}, \mathsf{T}'$, if $\mathcal{S} \sim \mathsf{T}'$ then $\pi\sigma\psi \leq \mathsf{T}'$ for all $\psi \in \mathcal{S}$. Furthermore, π can be extended to solve all constraints in $\mathcal{C}_{\mathsf{S}}$.

Proof. By induction on the proof tree of $\mathcal{S} \sim \mathsf{T}'$.

Case [Sim-Init]. We have $\pi\sigma\xi = \mathsf{T}$.

Case [Sim-In]. By the inductive hypothesis we have $\pi\sigma\psi \leq \mathsf{T}_1$ for all ψ in $\mathcal{S}_1$. Let $\psi_2 \in \mathcal{S}_2$. By definition of $\mathbb{G}_\mathcal{C}$, there exists $\psi_1 \in \mathcal{S}_1$ such that $\mathsf{p}?[\alpha];\psi_2 \leq \psi_1 \in \mathcal{C}$. So $\pi\sigma\psi_1 \leq \mathsf{T}_1$ and $\pi\sigma\psi_1 \xrightarrow{\mathsf{p}?[\pi\alpha]} \pi\sigma\psi_2$. By definition of subtyping, we conclude that $\pi\sigma\psi_2 \leq \mathsf{T}_2$.

Furthermore, for all $\mathsf{p}?[\alpha^i];\psi_2^i \in \mathtt{dep}_\mathcal{C}(\mathcal{S}_1)$, we have that $\mathsf{p}?[\pi\alpha^i];\pi\sigma\psi_2^i \leq \pi\sigma\psi_1 \leq \mathsf{T}_1$ for some $\psi_1 \in \mathcal{S}_1$. Also, $\alpha^i = \beta \in \mathcal{C}$ for all i. Observe that $\pi\alpha^i = S$, so we may extend π with $\beta \mapsto S$.

Case [Sim-Out]. Similar.

Case [Sim-Sel]. By the inductive hypothesis we have $\pi\sigma\psi \leq \mathsf{T}_1$ for all $\psi \in \mathcal{S}_1$. Let $\psi_2 \in \mathcal{S}_2$. By definition of $\mathbb{G}_\mathcal{C}$, there exists $\psi_1 \in \mathcal{S}_1$ such that $\mathsf{p} \oplus \{l : \psi_2\} \leq \psi_1 \in \mathcal{C}$. Therefore, $\pi\sigma\psi_1 \leq \mathsf{T}_1$ and $\pi\sigma\psi_1 \xrightarrow{\mathsf{p}\oplus l} \pi\sigma\psi_2$. By definition of subtyping, we conclude that $\pi\sigma\psi_2 \leq \mathsf{T}_2$.

Case [Sim-Bra]. Similar. ■

Claim D.15. There exists π'' such that $\sim$ is a type simulation on $V(\pi''\pi'\mathbb{G}_\mathcal{C}) \times \mathrm{Sub}(\mathsf{T})$.

Proof. Extend π to solve all constraints of $\mathcal{C}_\mathsf{S}$. As π' is a most general solution on $\mathcal{C}_\mathsf{S}$, there exists π'' such that $\pi\alpha = \pi''\pi'\alpha$ for all α.

Let $\mathcal{S} \sim \mathsf{T}'$. We check the conditions for type simulation:

- If $\mathcal{S} \xrightarrow{\mathsf{end}}$, choose an arbitrary $\psi \in \mathcal{S}$. Then by Definition D.11, $\mathtt{end} \leq \psi \in \mathcal{C}$. By Claim D.14, $\pi\sigma\psi \leq \mathsf{T}'$. Therefore, $\mathtt{end} \leq \mathsf{T}'$, i.e. $\mathsf{T}' \xrightarrow{\mathsf{end}}$.
- If $\mathcal{S} \xrightarrow{\mathsf{p}?[\pi\beta]} \mathcal{S}_1$, then by Definition D.11, $\mathsf{p}?[\alpha_i];\xi_i \leq \psi_i \in \mathcal{C}$ for all $\psi_i \in \mathcal{S}_1$, with $\beta = \alpha_i \in \mathcal{C}$. By Claim D.14, $\pi\sigma\psi_i \leq \mathsf{T}'$. Also, $\pi\beta = \pi''\pi'\beta = \pi''\pi'\alpha$. Therefore, $\mathsf{T}' \xrightarrow{\mathsf{p}?[\pi\beta]} \mathsf{T}_1$ for some T_1. By definition of $\leq$, $\mathcal{S}_1 \leq \mathsf{T}_1$.
- If $\mathcal{S} \xrightarrow{\mathsf{p}![\pi\beta]} \mathcal{S}_1$: symmetric.
- If $\mathsf{T}' \xrightarrow{\mathsf{p}\&l_k} \mathsf{T}_1$, then by Claim D.14, $\pi\sigma\psi \leq \mathsf{T}'$ for all $\psi \in \mathcal{S}$. Therefore, by definition of subtyping, $\pi\sigma\psi \xrightarrow{\mathsf{p}\&l_k}$. By definition of $\mathbb{G}_\mathcal{C}$, $\mathcal{S} \xrightarrow{\mathsf{p}\&l_k} \mathcal{S}_1$ for some $\mathcal{S}_1$, therefore $\mathcal{S}_1 \sim \mathsf{T}_1$.

- If $\mathcal{S} \xrightarrow{\mathsf{p}\oplus l_k} \mathcal{S}_1$, then by Definition D.11, there exists $\psi \in \mathcal{S}_1$ such that $\mathsf{p}\oplus\{l_j : \xi_j\}_{j\in J} \leq \psi \in \mathcal{C}$ such that $k \in J$ and $\xi_k = \psi'$. Then by Claim D.14, $\pi\sigma\psi \leq \mathsf{T}'$. Therefore, $\mathsf{T}' \xrightarrow{\mathsf{p}\oplus l_k} \mathsf{T}_1$ for some T_1. By definition of $\sim$, $\mathcal{S}_1 \sim \mathsf{T}_1$.

All conditions are valid and so $\sim$ is a type simulation. ■

Therefore $\{\xi\} \sim \mathsf{T}$ implies that $\pi''\mathsf{T}_{\mathsf{min}} \leq \mathsf{T}$ as desired. □

D.3 A Type System with Linear-Time Inference

The high complexity of the type inference stems from the conditional (if-then-else construct): it forces types to implicitly branch without recording the taken choice. This requires our inference system to handle arbitrary unifications of types, which can result in an exponential blowup in the worst-case (Theorem 4.14). Recently, Castellani and Dezani *et al.* [5, 8, 1] study the MPST calculus which has no conditional expressions but *multiple selection processes* ($\mathsf{p}\triangleleft\{l_i : P_i\}_{i\in I}$) (instead of a single selection $\mathsf{p}\triangleleft l.P$). In this subsection, we will investigate the complexity of the type inference if we use this system.

The processes we consider is a slight modification of the ones in original system, but with separated message and label passing constructs (similarly to the ones in this paper). The main difference from our system is that selection is now non-deterministic, and the conditional is removed. This still allows for non-determinism via internal choices, but the choices taken must be explicitly stated in the type. The syntax of types is unchanged, but requires a typing rule for this extended selection construct.

Definition D.16 (Processes of [5, 8])**.** Processes are defined by the following syntax:

$$
\begin{array}{lll}
P, P', P_i ::= \mathbf{0} & & \text{(inaction)} \\
\quad \mid \mathsf{p}!\langle e \rangle.P & & \text{(output)} \\
\quad \mid \mathsf{p}?(x).P & & \text{(input)} \\
\quad \mid \mathsf{p} \triangleleft \{l_i : P_i\}_{i \in I} & & \text{(selection)} \\
\quad \mid \mathsf{p} \triangleright \{l_i : P_i\}_{i \in I} & & \text{(branching)} \\
\quad \mid \mu X.P & & \text{(recursive process)} \\
\quad \mid X & & \text{(process variable)}
\end{array}
$$

Definition D.17 (Typing system of [5, 8])**.** The typing rules are identical to Figure 10, except for [T-Sel]:

$$
\frac{\forall i \in I.\, \Gamma \vdash P_i : \mathsf{T}_i}{\Gamma \vdash \mathsf{p} \triangleleft \{l_i : P_i\}_{i \in I} : \mathsf{p} \& \{l_i : \mathsf{T}_i\}_{i \in I}} \text{ [T-Sel]}
$$

Using this typing system, we can then generate constraints in a form similar to Figure 6, with the following change to the rule for selection:

$$
\frac{\begin{array}{c} \forall i \in I.\, \Gamma \vdash P_i : \psi_i \mid_{\mathcal{X}_i} \mathcal{C}_i \qquad \xi \text{ fresh} \\ \mathcal{C}' = \bigcup_{i \in I} \mathcal{C}_i \cup \{\mathsf{p} \oplus \{l_i : \psi_i\}_{i \in I} \leq \xi\} \qquad \mathcal{X}' = \bigcup_{i \in I} \mathcal{X}_i \cup \{\xi\} \end{array}}{\Gamma \vdash \mathsf{p} \triangleleft \{l_i : P_i\}_{i \in I} : \xi \mid_{\mathcal{X}'} \mathcal{C}'} \text{ [C-Sel]}
$$

Analogues of Theorems 4.5 and 4.6 can be derived for this system, and thus the type described in Definition D.11 defines a minimum type by an analogue of Theorem 4.12.

However, because there is no conditional, we can show that type inference can be performed in linear time.

Theorem D.18 (Complexity of type inference with the system in [5, 8])**.** Type inference in the system can be performed in $\mathcal{O}(n)$ time.

Proof. By inspecting the constraint rules of Figure 6, we conclude that for each $\psi \in \mathbf{Tv}$, there is at most one constraint of the form $\mathsf{T} \leq \psi$, which means that $|\mathtt{dep}_{\mathcal{C}}(\mathcal{S})| \leq \mathcal{S}$.

Therefore, all reachable nodes from $\{\xi\}$ in the type graph have cardinality one. So our algorithm only needs to consider $\mathcal{O}(n)$ nodes in the type graph, and solving for the constraints takes linear time. □

By inspecting the structure of constraints further, we can further see that the transformation is syntactic (at the type level). We will illustrate this with an example.

Example D.19. The minimum type of P is T, where:

$$P = \mu X.\mathsf{p}?(x).\mathsf{p}!\langle x\rangle.\mathsf{p}\triangleleft\{l_1 : X, l_2 : \mathbf{0}\}$$
$$\mathsf{T} = \mu \mathbf{t}.\mathsf{p}?(\alpha).\mathsf{p}!\langle \alpha\rangle.\mathsf{p}\triangleleft\{l_1 : \mathbf{t}, l_2 : \mathtt{end}\}$$

E Appendix for Bottom Up MPST

E.1 Reduction from the Quantified Boolean Formula Problem

Lemma E.1. Δ is safe if and only if Δ' is a safe state for all $\Delta \to^* \Delta'$.

Proof. ($\Longrightarrow$) Fix a safety property ϕ such that $\phi(\Delta)$. By definition, Δ is a safe state. Let $\Delta \to^* \Delta'$. By induction on the length of the reduction sequence, then applying the second condition of safety properties, we have that Δ' is a safe state.

($\Longleftarrow$) Define $\Delta', \mu\mathbf{t}.\mathsf{T} \xrightarrow{\mathsf{unf}} \Delta', \mathsf{T}[\mu\mathbf{t}.\mathsf{T}/\mathbf{t}]$ for all Δ', T. Note that, if $\Delta_1 \xrightarrow{\mathsf{unf}} \Delta_2$ then $\Delta_1 \xrightarrow{\ell} \Delta' \iff \Delta_2 \xrightarrow{\ell} \Delta'$. Let Δ' be a safe state for all $\Delta \to^* \Delta'$. Let ϕ be the minimum relation containing Δ and closed under $\to$ and $\xrightarrow{\mathsf{unf}}$.

By definition, the latter two conditions for safety hold. It suffices to show that Δ' is safe for all $\phi(\Delta')$. Consider the shortest sequence of $\to$ and $\xrightarrow{\mathsf{unf}}$ reductions from Δ to Δ'. If $\Delta_1 \xrightarrow{\mathsf{unf}}{}^* \to \Delta_2$ then $\Delta_1 \to \Delta_2$, so the shortest sequence of reductions has the form $\Delta \to^* \Delta'' \xrightarrow{\mathsf{unf}}{}^* \Delta'$. But Δ'' is a safe state by the claim, and Δ'', Δ' have the same outgoing transitions. So Δ' is a safe state. Therefore ϕ is a safety property. □

Intuition. We further explain the roles in Δ_{init}.

- s is the "controller" of the session: it queries the other participants to check if the QBF is true; if it is not true, it enters the undesirable state T_{bad}, which we will define separately for each property we check. Otherwise, the QBF is queried in an infinite loop.

- p_i handles the truth value of v_i, and finds the truth value of $\mathcal{Q}_i v_i \dots \mathcal{Q}_n v_n \mathcal{F}'$. It does this by trying both values of v_i: first, it lets $v_i = 0$ (represented by being in state F in Figure 12) and asks p_{i+1} for the truth value of $\mathcal{Q}_{i+1} v_{i+1} \dots \mathcal{Q}_n v_n$. During this, p_i is able to respond to requests with label query_t, which are querying the truth value of the variable represented by participant t. If needed, it then lets $v_i = 1$ (represented by being in state T) and repeats the process. The responses from p_{i+1} are then combined (depending on $\mathcal{Q}_i$) into a response sent to p_{i-1}: either tt or ff.

- Once all p_i have their truth values initialised, r_i handles the truth value of the i-th clause of $\mathcal{F}'$, namely $L_{i1} \vee L_{i2} \vee L_{i3}$. First, the participant queries the truth value from r_{i+1}. If this is false, then it immediately returns ff to the previous participant. Otherwise, it queries (using query_t) the truth value of each of its literals in turn, and returns the appropriate value. An exception is r_{m+1}, which starts the process by returning ff.

E.1.1 Reduction Behaviour of the Composition

We will now describe the behaviour of the composition in Figure 11, primarily to show that it has a unique and safe reduction path that calculates the truth value of the QBF.

Intuitively, the following lemma states that processes handling clauses can "query" the truth value of variables by selecting the label $\mathsf{query}_{\mathsf{p}_i}$ and will get the truth value in the form of a selection of yes or no.

Lemma 5.8. [Propagation of $\mathsf{query}_{\mathsf{p}_i}$] Let $\Delta = (W, \mathcal{A}, \mathsf{r}_1 : R, \dots, \mathsf{r}_{k-1} : R, \mathsf{r}_k : \mathsf{T}, I, \dots, I)$ for some type T. Let $\mathsf{T} = r_{k-1} \oplus \{\mathsf{query}_{p_i} : \mathsf{T}'\}$, such that $\mathsf{T}' = r_{k-1} \& \{\mathsf{yes} : \mathsf{T}_{\mathsf{yes}}, \mathsf{no} : \mathsf{T}_{\mathsf{no}}\}$. Then:

- If $\mathcal{A}(p_i) = 1$, then $\Delta \xrightarrow[\text{safe}]{\text{det}}{}^{*} (W, \mathcal{A}, \mathsf{r}_1 : R, \dots, \mathsf{r}_{k-1} : R, \mathsf{T}_{\mathsf{yes}}, I, \dots, I)$.

- If $\mathcal{A}(p_i) = 0$, then $\Delta \xrightarrow[\text{safe}]{\text{det}}{}^{*} (W, \mathcal{A}, \mathsf{r}_1 : R, \dots, \mathsf{r}_{k-1} : R, \mathsf{T}_{\mathsf{no}}, I, \dots, I)$.

Proof. We focus on the case where $\mathcal{A}(p_i) = 1$; the other case is symmetric. Defining $(V^j)^t_{11} = T^t_{11}$ if $V^j = T$, and $(V^j)^t_{11} = F^t_{11}$ if $V^j = F$, *etc.*, observe that:

$$
\begin{aligned}
&\Delta \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, \mathsf{r}_1 : R, \mathsf{r}_2 : R, \ldots, \mathsf{r}_{k-3} : R, \mathsf{r}_{k-2} : R, \mathsf{r}_{k-1} : R_1^{\mathsf{p}_i}, \mathsf{r}_k : \mathsf{T}', I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, \mathsf{r}_1 : R, \mathsf{r}_2 : R, \ldots, \mathsf{r}_{k-3} : R, \mathsf{r}_{k-2} : R_1^{\mathsf{p}_i}, \mathsf{r}_{k-1} : R_2, \mathsf{r}_k : \mathsf{T}', I, \ldots, I) \\
&\quad \ldots \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, \mathsf{r}_1 : R_1^{\mathsf{p}_i}, \mathsf{r}_2 : R_2, \ldots, \mathsf{r}_{k-3} : R_2, \mathsf{r}_{k-2} : R_2, \mathsf{r}_{k-1} : R_2, \mathsf{r}_k : \mathsf{T}', I, \ldots, I) \\
&= (W, V^1, \ldots, V^{i-1}, \mathsf{p}_i : T, V^{i+1}, V^{i+2}, \ldots, V^{n-1}, V^n, R_1^{\mathsf{p}_i}, R_2, \ldots, R_2, \mathsf{T}', I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, V^1, \ldots, V^{i-1}, \mathsf{p}_i : T, V^{i+1}, V^{i+2}, \ldots, V^{n-1}, (V^n)^{\mathsf{p}_i}_{11}, R_2, R_2, \ldots, R_2, \mathsf{T}', I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, V^1, \ldots, V^{i-1}, \mathsf{p}_i : T, V^{i+1}, V^{i+2}, \ldots, (V^{n-1})^{\mathsf{p}_i}_{11}, V^n_{12}, R_2, R_2, \ldots, R_2, \mathsf{T}', I, \ldots, I) \\
&\quad \ldots \\
&\xrightarrow[\text{safe}]{\det}(W, V^1, \ldots, V^{i-1}, \mathsf{p}_i : T, (V^{i+1})^{\mathsf{p}_i}_{11}, V^{i+2}_{12}, \ldots, V^{n-1}_{12}, V^n_{12}, R_2, R_2, \ldots, R_2, \mathsf{T}', I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, V^1, \ldots, V^{i-1}, \mathsf{p}_i : T_{21}, V^{i+1}_{12}, V^{i+2}_{12}, \ldots, V^{n-1}_{12}, V^n_{12}, R_2, R_2, \ldots, R_2, \mathsf{T}', I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, V^1, \ldots, V^{i-1}, \mathsf{p}_i : T, V^{i+1}_{13}, V^{i+2}_{12}, \ldots, V^{n-1}_{12}, V^n_{12}, R_2, R_2, \ldots, R_2, \mathsf{T}', I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, V^1, \ldots, V^{i-1}, \mathsf{p}_i : T, V^{i+1}, V^{i+2}_{13}, \ldots, V^{n-1}_{12}, V^n_{12}, R_2, R_2, \ldots, R_2, \mathsf{T}', I, \ldots, I) \\
&\quad \ldots \\
&\xrightarrow[\text{safe}]{\det}(W, V^1, \ldots, V^{i-1}, \mathsf{p}_i : T, V^{i+1}, V^{i+2}, \ldots, V^{n-1}, V^n_{13}, R_2, R_2, \ldots, R_2, \mathsf{T}', I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, V^1, \ldots, V^{i-1}, \mathsf{p}_i : T, V^{i+1}, V^{i+2}, \ldots, V^{n-1}, V^n, R_3, R_2, \ldots, R_2, \mathsf{T}', I, \ldots, I) \\
&= (W, \mathcal{A}, \mathsf{r}_1 : R_3, \mathsf{r}_2 : R_2, \ldots, \mathsf{r}_{k-2} : R_2, \mathsf{r}_{k-1} : R_2, \mathsf{r}_k : \mathsf{T}', I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, \mathsf{r}_1 : R, \mathsf{r}_2 : R_3, \ldots, \mathsf{r}_{k-2} : R_2, \mathsf{r}_{k-1} : R_2, \mathsf{r}_k : \mathsf{T}', I, \ldots, I) \\
&\quad \ldots \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, \mathsf{r}_1 : R, \mathsf{r}_2 : R, \ldots, \mathsf{r}_{k-2} : R, \mathsf{r}_{k-1} : R_3, \mathsf{r}_k : \mathsf{T}', I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, \mathsf{r}_1 : R, \mathsf{r}_2 : R, \ldots, \mathsf{r}_{k-2} : R, \mathsf{r}_{k-1} : R, \mathsf{r}_k : \mathsf{T}_{\mathsf{yes}}, I, \ldots, I)
\end{aligned}
$$

Intuitively, the message $\mathsf{query}_{\mathsf{p}_i}$ is being propagated through the chain of participants, starting at r_k and ending at p_i, and then the "return value" of yes is propagated backwards through the same chain. The states of the participants store the necessary information to uniquely specify this reduction path. □

The next lemma details how r_i finds the truth value of the conjunction of all clauses starting from $\mathcal{C}_i$. As the participants r_i use the query_t labels to find the truth values of their variables, we use Lemma 5.8 to specify the results of this behaviour. As one might expect, r_i enters state T_r, ready to send a tt to r_{i-1} if the conjunction is true, otherwise it will enter state F_r and instead send ff.

Lemma 5.9. [Reduction of r_i] For $\mathcal{A} : \{v_1, \ldots, v_n\} \to \{0, 1\}$:

- $(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : I_1, I, \ldots, I) \xrightarrow[\text{safe}]{\text{det}}{}^{*} (W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : T_r, I, \ldots, I)$
  if $\mathcal{A} \models \bigwedge_{i=k}^{m} C_i$.
- $(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : I_1, I, \ldots, I) \xrightarrow[\text{safe}]{\text{det}}{}^{*} (W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : F_r, I, \ldots, I)$
  if $\mathcal{A} \not\models \bigwedge_{i=k}^{m} C_i$.

Proof. Let $\mathcal{F}'_k = \bigwedge_{i=k}^{m} C_i$. The proof is by induction on k. For $k = m$, F and I_1 are the same state, so the result follows immediately. For $k < m$, we have two cases to consider.

Case $\mathcal{A} \models \mathcal{F}'_k$. We have that $\mathcal{A} \models \mathcal{F}'_{k+1}$ and at least one of $\mathcal{A}(L_{k1})$, $\mathcal{A}(L_{k2})$ and $\mathcal{A}(L_{k3})$ is 1.

If $\mathcal{A}(L_{k1}) = 1$:

$$
\begin{aligned}
&(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : I_1, I, \ldots, I) \\
\xrightarrow[\text{safe}]{\text{det}}&(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : R, \mathsf{r}_{k+1} : I_1, I \ldots, I) \\
\xrightarrow[\text{safe}]{\text{det}}{}^{*}&(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : R, \mathsf{r}_{k+1} : T_r, I, \ldots, I) && \text{by inductive hypothesis} \\
\xrightarrow[\text{safe}]{\text{det}}&(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : Q'_1, I, \ldots, I) \\
\xrightarrow[\text{safe}]{\text{det}}{}^{*}&(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : T_r, I, \ldots, I) && \text{by Lemma 5.8}
\end{aligned}
$$

If $\mathcal{A}(L_{k1}) = 0$ and $\mathcal{A}(L_{k2}) = 1$:

$$
\begin{aligned}
&(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : I_1, I, \ldots, I) \\
\xrightarrow[\text{safe}]{\det} &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : R, \mathsf{r}_{k+1} : I_1, I \ldots, I) \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : R, \mathsf{r}_{k+1} : T_r, I, \ldots, I) && \text{by inductive hypothesis} \\
\xrightarrow[\text{safe}]{\det} &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : Q'_1, I, \ldots, I) \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : Q'_2, I, \ldots, I) && \text{by Lemma 5.8} \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : T_r, I, \ldots, I) && \text{by Lemma 5.8}
\end{aligned}
$$

If $\mathcal{A}(L_{k1}) = 0$ and $\mathcal{A}(L_{k2}) = 0$ and $\mathcal{A}(L_{k3}) = 1$:

$$
\begin{aligned}
&(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : I_1, I, \ldots, I) \\
\xrightarrow[\text{safe}]{\det} &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : R, \mathsf{r}_{k+1} : I_1, I \ldots, I) \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : R, \mathsf{r}_{k+1} : T_r, I, \ldots, I) && \text{by inductive hypothesis} \\
\xrightarrow[\text{safe}]{\det} &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : Q'_1, I, \ldots, I) \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : Q'_2, I, \ldots, I) && \text{by Lemma 5.8} \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : Q'_3, I, \ldots, I) && \text{by Lemma 5.8} \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : T_r, I, \ldots, I) && \text{by Lemma 5.8}
\end{aligned}
$$

Case $\mathcal{A} \not\models \mathcal{F}'_k$. We have that $\mathcal{A} \not\models \mathcal{F}'_{k+1}$ or $\mathcal{A}(L_{k1}) = \mathcal{A}(L_{k2}) = \mathcal{A}(L_{k3}) = 0$.

If $\mathcal{A} \not\models \mathcal{F}'_{k+1}$:

$$
\begin{aligned}
&(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : I_1, I, \ldots, I) \\
\xrightarrow[\text{safe}]{\det} &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : R, \mathsf{r}_{k+1} : I_1, I \ldots, I) \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : R, \mathsf{r}_{k+1} : F_r, I, \ldots, I) && \text{by inductive hypothesis} \\
\xrightarrow[\text{safe}]{\det} &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : F_r, I, \ldots, I)
\end{aligned}
$$

If $\mathcal{A} \models \mathcal{F}'_{k+1}$ and $\mathcal{A}(L_{k1}) = \mathcal{A}(L_{k2}) = \mathcal{A}(L_{k3}) = 0$:

$$
\begin{aligned}
&(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : I_1, I, \ldots, I) \\
\xrightarrow[\text{safe}]{\det} &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : R, \mathsf{r}_{k+1} : I_1, I \ldots, I) \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : R, \mathsf{r}_{k+1} : T_r, I, \ldots, I) && \text{by inductive hypothesis} \\
\xrightarrow[\text{safe}]{\det} &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : Q'_1, I, \ldots, I) \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : Q'_2, I, \ldots, I) && \text{by Lemma 5.8} \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : Q'_3, I, \ldots, I) && \text{by Lemma 5.8} \\
\xrightarrow[\text{safe}]{\det}{}^* &(W, \mathcal{A}, R, \ldots, R, \mathsf{r}_k : F_r, I, \ldots, I) && \text{by Lemma 5.8}
\end{aligned}
$$

Each case yields the desired result, so the lemma holds. □

Next, we show the behaviour of p_i, which behaves similarly to r_i, but instead handles the quantified variables.

Lemma 5.10. [Reduction of p_i] For $\mathcal{A} : \{v_1, \ldots, v_k\} \to \{0, 1\}$:

- $[\mathcal{A}] \xrightarrow[\text{safe}]{\det}{}^* (W, \mathcal{A}, T_r, I, \ldots, I)$ if $\mathcal{A} \models \mathcal{Q}_{k+1} v_{k+1} \ldots \mathcal{Q}_n v_n \mathcal{F}'$.
- $[\mathcal{A}] \xrightarrow[\text{safe}]{\det}{}^* (W, \mathcal{A}, F_r, I, \ldots, I)$ if $\mathcal{A} \not\models \mathcal{Q}_{k+1} v_{k+1} \ldots \mathcal{Q}_n v_n \mathcal{F}'$.

Proof. Let $\mathcal{F}'_k = \mathcal{Q}_{k+1} v_{k+1} \ldots \mathcal{Q}_n v_n \mathcal{F}'$. The proof is by induction on k. For $k = 0$, use Lemma 5.9. For $k > 0$, we consider the case $\mathcal{Q}_{k+1} = \forall$; the $\exists$ case is similar.

If $\mathcal{A} \models \mathcal{F'}_k$, we have $\mathcal{A}_{[v_{k+1} \mapsto 0]} \models \mathcal{F'}_k$ and $\mathcal{A}_{[v_{k+1} \mapsto 1]} \models \mathcal{F'}_k$.

$$
\begin{aligned}
&[\mathcal{A}] \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, F, I_1, \ldots, I) = [\mathcal{A}_{[v_{k+1} \mapsto 0]}] \\
&\xrightarrow[\text{safe}]{\det}{}^*(W, \mathcal{A}_{[v_{k+1} \mapsto 0]}, T_r, I, \ldots, I) = (W, \mathcal{A}, F, T_r, I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, I_2, I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, T, I_1, \ldots, I) = [\mathcal{A}_{[v_{k+1} \mapsto 1]}] \\
&\xrightarrow[\text{safe}]{\det}{}^*(W, \mathcal{A}_{[v_{k+1} \mapsto 1]}, T_r, I, \ldots, I) = (W, \mathcal{A}, T, T_r, I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, T_r, I, \ldots, I)
\end{aligned}
$$

If $\mathcal{A} \not\models \mathcal{F'}_k$, then there are two cases: first, (i) $\mathcal{A}_{[v_{k+1} \mapsto 0]} \not\models \mathcal{F'}_k$:

$$
\begin{aligned}
&[\mathcal{A}] \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, F, I_1, \ldots, I) = [\mathcal{A}_{[v_{k+1} \mapsto 0]}] \\
&\xrightarrow[\text{safe}]{\det}{}^*(W, \mathcal{A}_{[v_{k+1} \mapsto 0]}, F_r, I, \ldots, I) = (W, \mathcal{A}, F, F_r, I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, F_r, I, \ldots, I)
\end{aligned}
$$

Second, (ii) $\mathcal{A}_{[v_{k+1} \mapsto 0]} \models \mathcal{F'}_k$ and $\mathcal{A}_{[v_{k+1} \mapsto 1]} \not\models \mathcal{F'}_k$:

$$
\begin{aligned}
&[\mathcal{A}] \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, F, I_1, \ldots, I) = [\mathcal{A}_{[v_{k+1} \mapsto 0]}] \\
&\xrightarrow[\text{safe}]{\det}{}^*(W, \mathcal{A}_{[v_{k+1} \mapsto 0]}, T_r, I, \ldots, I) = (W, \mathcal{A}, F, T_r, I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, I_2, I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, T, I_1, \ldots, I) = [\mathcal{A}_{[v_{k+1} \mapsto 1]}] \\
&\xrightarrow[\text{safe}]{\det}{}^*(W, \mathcal{A}_{[v_{k+1} \mapsto 1]}, F_r, I, \ldots, I) = (W, \mathcal{A}, T, F_r, I, \ldots, I) \\
&\xrightarrow[\text{safe}]{\det}(W, \mathcal{A}, F_r, I, \ldots, I)
\end{aligned}
$$

Each $\xrightarrow[\text{safe}]{\text{det}}$ can be checked directly, and each $\xrightarrow[\text{safe}]{\text{det}}{}^*$ is from the inductive hypothesis. Therefore, the lemma follows by induction. □

We can now use the above lemmas to prove PSPACE-hardness for checking safety, liveness and deadlock-freedom.

E.2 PSPACE-hardness of safety

Lemma E.2. If $\Delta \to \Delta_1 \xrightarrow[\text{safe}]{\text{det}}{}^* \Delta$ and $\Delta \to^* \Delta'$ and Δ is a safe state and Δ' is not a safe state then $\Delta \to \Delta_2 \to^* \Delta'$ for some $\Delta_2 \neq \Delta_1$.

Proof. By contradiction. Assume that Δ, Δ_1, Δ' are as described, but there is no $\Delta_2 \neq \Delta_1$ such that $\Delta \to \Delta_2 \to^* \Delta'$. Fix the shortest reduction sequence $\Delta = \Delta^0 \to \Delta^1 \cdots \to \Delta^n = \Delta'$. As Δ is a safe state and Δ' is not a safe state, $\Delta \neq \Delta'$ and so $n > 0$. From our contradiction assumption, $\Delta^1 = \Delta_1$. As $\Delta_1 \xrightarrow[\text{safe}]{\text{det}}{}^* \Delta$, fix $\Delta_i^\dagger$ such that $\Delta_1^\dagger \xrightarrow[\text{safe}]{\text{det}} \Delta_2^\dagger \xrightarrow[\text{safe}]{\text{det}} \ldots \xrightarrow[\text{safe}]{\text{det}} \Delta_m^\dagger = \Delta$. By the definition of $\xrightarrow[\text{safe}]{\text{det}}$ (Definition 5.7), $\Delta_i^\dagger = \Delta^i$ and Δ^i is a safe state for all $i \in \{1, \ldots, m\}$, so $\Delta^m = \Delta$. But then $\Delta = \Delta^m \to \cdots \to \Delta^n = \Delta'$ is a shorter reduction sequence, contradicting our assumption of minimality. □

Theorem E.3. Checking for safety is PSPACE-hard.

Proof. Set $\mathsf{T}_{\text{bad}} = \mathsf{p}_1 \& \{\mathsf{unsafe} : \mathsf{end}\}$.

- If $\models \mathcal{F}$, then by Corollary 5.11, $[\emptyset] \xrightarrow[\text{safe}]{\text{det}}{}^* \Delta_{\mathsf{init}}$. Assume for contradiction that $\Delta_{\mathsf{init}} \to \Delta'$ for some Δ' that is not a safe state. We have $\Delta_{\mathsf{init}} \to [\emptyset]$, so by Lemma E.2, $\Delta_{\mathsf{init}} \to \Delta_2$ for some $\Delta_2 \neq [\emptyset]$. But no such Δ_2 exists, so Δ_{init} is safe by Lemma E.1.
- If $\not\models \mathcal{F}$, then by Corollary 5.11, $\Delta_{\mathsf{init}} \to [\emptyset] \xrightarrow[\text{safe}]{\text{det}}{}^* (\mathsf{T}_{\text{bad}}, I, \ldots, I)$. But $(\mathsf{T}_{\text{bad}}, I, \ldots, I) \xrightarrow{\mathsf{sp}_1 \& \mathsf{unsafe}}$ and $(\mathsf{T}_{\text{bad}}, I, \ldots, I) \not\xrightarrow{\mathsf{p}_1 \mathsf{s}:\mathsf{unsafe}}$, so it is not a safe state. Thus Δ_{init} is not safe.

Therefore, Δ_{init} is safe if and only if $\models \mathcal{F}$. As QBF is PSPACE-complete, we conclude that checking for safety is PSPACE-hard. □

E.3 PSPACE-hardness of deadlock freedom

The proof that deadlock-freedom is PSPACE-hard is similar to the above proof for safety, as unique reduction paths also guarantee deadlock-freedom. First, we prove the analogue of Lemma E.2 for deadlock-freedom.

Lemma E.4. If $\Delta \to \Delta_1 \xrightarrow{\text{det}}{}^* \Delta$ and $\Delta \to^* \Delta' \not\to$ then $\Delta \to \Delta_2 \to^* \Delta'$ for some $\Delta_2 \neq \Delta_1$.

Proof. By contradiction. Assume that Δ, Δ_1, Δ' are as described, but there is no $\Delta_2 \neq \Delta_1$ such that $\Delta \to \Delta_2 \to^* \Delta'$. Fix the shortest reduction sequence $\Delta = \Delta^0 \to \Delta^1 \cdots \to \Delta^n = \Delta'$. As Δ is a safe state and Δ' is not a safe state, $\Delta \neq \Delta'$ and so $n > 0$. From our contradiction assumption, $\Delta^1 = \Delta_1$. As $\Delta_1 \xrightarrow{\text{det}}{}^* \Delta$, fix $\Delta_i^\dagger$ such that $\Delta_1^\dagger \xrightarrow{\text{det}} \Delta_2^\dagger \xrightarrow{\text{det}} \ldots \xrightarrow{\text{det}} \Delta_m^\dagger = \Delta$. By the definition of $\xrightarrow{\text{det}}$ (Definition 5.7), $\Delta_i^\dagger = \Delta^i$ and $\Delta^i \to$ for all $i \in \{1, \ldots, m\}$, so $\Delta^m = \Delta$. But then $\Delta = \Delta^m \to \cdots \to \Delta^n = \Delta'$ is a shorter reduction sequence, contradicting our assumption of minimality. □

Theorem E.5. Checking for deadlock-freedom is PSPACE-hard.

Proof. Set $\mathsf{T}_{\text{bad}} = \texttt{end}$.

- If $\models \mathcal{F}$, then by Corollary 5.11, $[\emptyset] \xrightarrow[\text{safe}]{\text{det}}{}^* \Delta_{\mathsf{init}}$. Assume for contradiction that $\Delta_{\mathsf{init}} \to \Delta'$ for some $\Delta' \not\to$. We have $\Delta_{\mathsf{init}} \to [\emptyset]$, so by Lemma E.2, $\Delta_{\mathsf{init}} \to \Delta_2$ for some $\Delta_2 \neq [\emptyset]$. But no such Δ_2 exists, so Δ_{init} is deadlock-free by Lemma E.4.
- If $\not\models \mathcal{F}$, then by Corollary 5.11, $\Delta_{\mathsf{init}} \to [\emptyset] \xrightarrow[\text{safe}]{\text{det}}{}^* (\mathsf{T}_{\text{bad}}, I, \ldots, I)$. But $(\mathsf{T}_{\text{bad}}, I, \ldots, I) \not\to$, so Δ_{init} is not deadlock-free (because $\text{unfold}(I) \neq \texttt{end}$).

Therefore, Δ_{init} is deadlock-free if and only if $\models \mathcal{F}$. As QBF is PSPACE-complete, we conclude that checking for deadlock-freedom is PSPACE-hard. □

E.4 PSPACE-hardness of liveness

First, we prove some properties about liveness that will be useful in our proof.

Lemma E.6. If Δ is live then Δ is deadlock-free.

Proof. Let Δ not be deadlock-free. Then there is a finite path $(\Delta_i)_{i \in N}$ with $N = \{0, 1, \ldots, k\}$ such that $\Delta_k \not\to$ and $\mathrm{unfold}(\Delta_k(\mathsf{p})) \neq \mathtt{end}$ for some p. But then Δ_k has some outgoing transition that is never matched, so Δ is not live. □

Definition E.7 (Single-Selection Types)**.** Call a type T a *single-selection type* if all selections in T have one possible branch, i.e. all unfolded subterms of T which are selections are of the form $\mathsf{p} \oplus \{l : \mathsf{T}'\}$.

Definition E.8 (Participants of Reductions)**.** Define the participants of a reduction: $\mathtt{pt}\{\xrightarrow{\mathsf{pq}}\} = \{\mathsf{p}, \mathsf{q}\}$ and $\mathtt{pt}\{\xrightarrow{\mathsf{pq}:l}\} = \{\mathsf{p}, \mathsf{q}\}$.

Lemma E.9 (Liveness of Single-Selection Types by Movement of Participants)**.** Let $(\Delta_i)_{i \in \mathbb{N}}$ be an infinite path such that $\Delta_i \xrightarrow{\ell_i} \Delta_{i+1}$ for all i. If all types in $\Delta = \Delta_0$ are single-selection types and $\bigcup_{i \geq j} \mathtt{pt}\{\ell_i\} = \mathtt{pt}\{\Delta\}$ for all $j \in \mathbb{N}$, then $(\Delta_i)_{i \in \mathbb{N}}$ is a live path.

Proof. Let $i \in \mathbb{N}$, and let $\mathsf{p} \in \mathtt{pt}\{\Delta\}$. Let $\Delta = \mathsf{p} : \mathsf{T}, \Delta'$. By assumption there exists $j \geq i$ such that $\mathsf{p} \in \mathtt{pt}\{\ell_j\}$. Choose the smallest such j. Then, no transition before ℓ_j affects p, so $\Delta_j(\mathsf{p}) = \mathsf{T}$. We consider the syntax of T:

Case $\mathsf{T} = \mathtt{end}$. Then there are no transitions involving p.

Case $\mathsf{T} = \mathsf{q}?[S]; \mathsf{T}'$. By considering the rules of Definition 5.1, $\ell_j = \xrightarrow{\mathsf{qp}}$, as desired.

Case $\mathsf{T} = \mathsf{q}![S]; \mathsf{T}'$. Similar to the above: $\ell_j = \xrightarrow{\mathsf{pq}}$.

Case $\mathsf{T} = \mathsf{q}\&\{l_i : \mathsf{T}_i\}_{i \in I}$. We conclude that $\ell_j = \xrightarrow{\mathsf{qp}:l_j}$ for some $j \in I$.

Case $\mathsf{T} = \mathsf{p} \oplus \{l : \mathsf{T}'\}$. We conclude that $\ell_j = \xrightarrow{\mathsf{qp}:l}$ as there is only one possible branch.

All transitions in the definition of live path fall into one of the above cases, so the path is live. □

Theorem E.10. Checking for liveness is PSPACE-hard.

Proof. We use the same construction as the deadlock-freedom reduction (Theorem E.5): set $\mathsf{T}_{\mathrm{bad}} = \mathtt{end}$.

- If $\models \mathcal{F}$, then by the same argument as Theorem E.5, $\Delta \xrightarrow{\text{det}} [\emptyset] \xrightarrow{\text{det}}{}^{*} \Delta$. Thus there is a unique infinite path $\Delta \xrightarrow{\text{det}}{}^{\omega}$. We will show that this path is live.

  First, note that Δ is a single-selection type (Definition E.7). Furthermore, the reduction sequence $\Delta \to (W, I_1, I, \ldots, I) \xrightarrow{\text{det}} \ldots \xrightarrow{\text{det}} (W, F, \ldots, F, R, \ldots, R)$ involves all participants in at least one transition. As $\Delta_j \xrightarrow{\text{det}}{}^{*} \Delta$ for all Δ_j, this reduction sequence occurs after Δ_j, therefore $\bigcup_{i \geq j} \mathtt{pt}\{\ell_i\} = \mathtt{pt}\{\Delta\}$ for all $j \in \mathbb{N}$. We can then apply Lemma E.9 to conclude that $\Delta \xrightarrow{\text{det}}{}^{\omega}$ is a live path. As all reachable compositions have outgoing transitions (due to the path being deterministic), there are no finite fair paths. Furthermore, we have shown that the only infinite path is live, so all fair paths are live. We conclude that Δ is live.

- If $\not\models \mathcal{F}$, then by the proof of Theorem E.5, Δ is not deadlock-free. By Lemma E.6, Δ is not live.

Therefore, Δ is live if and only if $\models \mathcal{F}$. As QBF is PSPACE-complete, we conclude that checking for liveness is PSPACE-hard. □

E.5 PSPACE-Completeness of Safety and Deadlock-Freedom

Lemma E.11. Let $\Delta = \prod_{i \in I} \mathsf{p}_i : \mathsf{T}_i$. Let $\Delta \to^{*} \Delta'$, such that $\Delta' = \prod_{i \in I} \mathsf{p}_i : \mathsf{T}'_i$. Then $\mathsf{T}'_i \in \text{Sub}(\mathsf{T}_i)$.

Proof. By inspection of the typing rules of Definition 5.1, the possible reductions of $\mathsf{p}_i : \mathsf{T}_i$ are exactly the possible reductions of $\mathbb{G}(\mathsf{T}_i)$. By Lemma 2.7, this is a subset of $\text{Sub}(\mathsf{T}_i)$. □

Theorem 5.13. Let $\Delta = \prod_{i \in I} \mathsf{p}_i : \mathsf{T}_i$. Then $|\{\Delta' \mid \Delta \to^{*} \Delta'\}| \leq \prod_{i \in I} |\mathsf{T}_i|$.

Proof. By the above lemma, applied to each $\mathsf{p}_i : \mathsf{T}_i$. We have that $\{\Delta' \mid \Delta \to^{*} \Delta'\} \subseteq \{\prod_{i \in I} \mathsf{p}_i : \mathsf{T}'_i \mid \mathsf{T}'_i \in \text{Sub}(\mathsf{T}_i)\}$, so $|\{\Delta' \mid \Delta \to^{*} \Delta'\}| \leq \prod_{i \in I} |\mathsf{T}_i|$. □

We can use a similar algorithm for deadlock-freedom.

Theorem E.12. Checking for deadlock-freedom is in PSPACE.

Proof. The proof is identical to the proof for safety, but with the condition in line 6 replaced with $\Delta' \not\to \implies \forall \mathsf{p} : \mathsf{T}_\mathsf{p} \in \Delta'.\text{unfold}(\mathsf{T}_\mathsf{p}) = \texttt{end}$, which is also checkable in polynomial time. □

E.5.1 A More Practical Algorithm for Safety and Deadlock-Freedom

Even though there must exist a deterministic polynomial-space algorithm for checking safety and deadlock-freedom by the reduction from CO-NPSPACE to PSPACE, this might not be practical due to the high time complexity of the reduction. We instead illustrate a determinstic algorithm that might be more practical, which instead does an explicit breadth-first search over all reachable states (Algorithm 3).

Algorithm 3 A deterministic algorithm for checking if a process is safe. The input is a composition $\Delta = \prod_{i \in I} \mathsf{p}_i : \mathsf{T}_i$.

```
1:  function Safe(Δ)
2:      V ← {Δ}                                  ▷ Set of found states
3:      Q ← [Δ]                                  ▷ Queue of states to visit
4:      while Q is nonempty do
5:          Δ′, Q ← Q
6:          if Δ′ is not a safe state then
7:              return reject
8:          end if
9:          V ← V ∪ {Δ′}
10:         for all Δ″ such that Δ′ → Δ″ do
11:             if Δ″ ∉ V then
12:                 V ← V ∪ Δ″
13:                 Q ← Q, Δ″
14:             end if
15:         end for
16:     end while
17:     return accept
18: end function
```

Even though this requires exponential memory in the worst case, it takes linear time and memory in the number of reachable compositions. In practice, this number is often much smaller than the bound of $\prod_{i \in I} |\mathsf{T}_i|$ that we had at the beginning. It also gives rise to a bounded model checking algorithm, where we only check for safety up to a certain length of reductions. This can also be adapted for deadlock-freedom in the same way as the other algorithm.

E.6 PSPACE-Completeness of Liveness

Lemma E.13. A path $(\Delta_i)_{i\in N}$ is live iff $\mathsf{barbs}(\Delta_k) \subseteq \mathsf{observations}((\Delta_i)_{i\geq k})$ for all k.

Proof. By comparing each rule in Definition 5.4 with the rules in Definition 5.15. For example, the rule for $\mathsf{pq}![S]$ is handled by rules [Barb-Out] and [Obs-Out]. □

Lemma E.14. A counterwitness exists for Δ if and only if Δ is not live.

Proof. In Definition 5.16, the first condition holds when the path is not live by the previous lemma. The second condition holds when the path is fair. □

Lemma E.15. Let $\Delta = \prod_{i\in I} \mathsf{p}_i : \mathsf{T}_i$. Let $n = \sum_{i\in I} |\mathsf{T}_i|$. Then there are at most $2n$ possible reductions $\ell \in \{\mathsf{pq}, \mathsf{pq} : l\}$ such that $\Delta \rightarrow^* \xrightarrow{\ell}$.

Proof. By considering the type graphs of T_i.

- If $\ell = \xrightarrow{\mathsf{p}_i\mathsf{p}_j}$, then $\mathbb{G}(\mathsf{T}_{\mathsf{p}_i})$ must contain an edge $\mathsf{p}_j![S]$, and $\mathbb{G}(\mathsf{T}_{\mathsf{p}_j})$ must contain an edge $\mathsf{p}_i?[S]$ for some S.
- If $\ell = \xrightarrow{\mathsf{p}_i\mathsf{p}_j:l}$, then $\mathbb{G}(\mathsf{T}_{\mathsf{p}_i})$ must contain an edge $\mathsf{p}_j \oplus l$, and $\mathbb{G}(\mathsf{T}_{\mathsf{p}_j})$ must contain an edge $\mathsf{p}_i \& l$.

As each reduction gives rise to unique edges, and there are less than or equal to $2|\mathsf{T}|$ edges in a type graph of T [22, Lemma 4.3], we conclude that there are $2n$ possible unique reductions. □

The following lemma shows that there must exists finite or periodic counterwitnesses to liveness if the composition is not live. We write $\mathcal{P}_1\mathcal{P}_2^\omega$ to denote an infinite path that concatenates $\mathcal{P}_1$ and infinitely many copies of $\mathcal{P}_2$.

Lemma 5.17. Let $\Delta = \prod_{i\in I} \mathsf{p}_i : \mathsf{T}_i$. Let $n = \sum_{i\in I} |\mathsf{T}_i|$ and $M = (2n+2) \cdot \prod_{i\in I} |\mathsf{T}_i|$. If Δ is not live, then it has a finite counterwitness of size $\leq M$ or an infinite counterwitness of the form $\mathcal{P}_1\mathcal{P}_2^\omega$ with $|\mathcal{P}_1|, |\mathcal{P}_2| \leq M$.

Proof. Fix some counterwitness $(\Delta_i)_{i\in N}$ of Δ. By definition, there exists $k \in N$ and $a \in \mathsf{barbs}(\Delta_k)$ such that $a \notin \bigcup_{i\geq k} \mathsf{observations}(\Delta_i)$.

Case N is finite. Then we can *shortcut* some cycles: if $\{\ell_i \mid i \geq j_1\} = \{\ell_i \mid i \geq v\}$ such that $j_1 < j_2$ and $k \notin \{j_1, j_1+1, \ldots, j_2\}$ then we may remove $\Delta_{j_1+1}, \ldots, \Delta_{j_2}$ from the path. As $\{\ell_i \mid i \geq w\} \subseteq \{\ell \mid \exists i \geq w.\ \Delta_i \xrightarrow{\ell}\}$ for all w, the subsets stay equal, and the first condition is not violated because we are removing elements from $\mathsf{observations}$. We can continue this process until there are no cycles with the same value of $|\{\ell_i \mid i \geq j\}|$ throughout. As there are only $2n$ possible values of ℓ_i by Lemma E.15, and we cannot shortcut cycles containing k, there are at most $2n+2$ sections, each with no repeating states, so the resulting path is of size M.

Case N is infinite. We first need to make the path periodic. This can be done by taking the first $u \geq k$ such that $\{\ell_i \mid k \leq i \leq u\} = \{\ell_i \mid k \leq i\}$. Then, take the first $u' > u$ such that $\Delta_{u'} = \Delta_v$ for some $v < u'$. For any j, $\{\ell_i \mid j \leq i \leq u'\} = \{\ell_i \mid j \leq i\} = \{\ell \mid \exists j \leq i.\ \Delta_i \xrightarrow{\ell}\} \supseteq \{\ell \mid \exists j \leq i \leq u'.\ \Delta_i \xrightarrow{\ell}\}$. But $\{\ell_i \mid j \leq i \leq u'\} \subseteq \{\ell \mid \exists j \leq i \leq u'.\ \Delta_i \xrightarrow{\ell}\}$ by definition, so the two sets are equal. Also, $\bigcup_{k\leq i\leq u'} \mathsf{observations}(\Delta_i) \subseteq \bigcup_{k\leq i} \mathsf{observations}(\Delta_i)$.

Thus, $\Delta' = \mathcal{P}_1\mathcal{P}_2^\omega$ such that $\mathcal{P}_1 = \Delta_0, \ldots, \Delta_{v-1}$ and $\mathcal{P}_2 = \Delta_v, \ldots, \Delta_{u'-1}$ is a valid counterwitness. Then, we will shortcut cycles similarly to the finite case. Before Δ_v, the strategy for finite paths still works. For the cycle $(\Delta_v, \ldots, \Delta_{u'-1})$, we will shortcut $j_1 < j_2$ such that $\{\ell_i \mid j_1 \leq i < u'\} = \{\ell_i \mid j_2 \leq i < u'\}$, by removing $\Delta_{j_1+1}, \ldots, \Delta_{j_2}$. As all the removed transitions in $\{j_1+1, \ldots, j_2\}$ are still found in $\{j_2, \ldots, u'-1\}$, the second condition is not violated. Furthermore, the first condition is not violated by the same reason as above.

Therefore, by the same reasoning as in the finite case, the resulting path is of size M. □