Subtyping Context-Free Session Types

Gil Silva
LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal

Andreia Mordido
LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal

Vasco T. Vasconcelos
LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal

Abstract

Context-free session types describe structured patterns of communication on heterogeneously typed channels, allowing the specification of protocols unconstrained by tail recursion. The enhanced expressive power provided by non-regular recursion comes, however, at the cost of the decidability of subtyping, even if equivalence is still decidable. We present an approach to subtyping context-free session types based on a novel kind of observational preorder we call $\mathcal{XYZW}$-simulation, which generalizes $\mathcal{XY}$-simulation (also known as covariant-contravariant simulation) and therefore also bisimulation and plain simulation. We further propose a subtyping algorithm that we prove to be sound, and present an empirical evaluation in the context of a compiler for a programming language. Due to the general nature of the simulation relation upon which it is built, this algorithm may also find applications in other domains.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Concurrency; Software and its engineering → Concurrent programming structures

Keywords and phrases Session types, Subtyping, Simulation, Simple grammars, Non-regular recursion

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.11

Related Version *Technical Report*: `https://arxiv.org/abs/2307.05661` [49]

Funding Support for this research was provided by the Fundação para a Ciência e a Tecnologia through project SafeSessions, ref. PTDC/CCI-COM/6453/2020, and by the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

Acknowledgements We thank Luca Padovani, Diana Costa, Diogo Poças and the anonymous reviewers for their insightful comments.

1 Introduction

Session types, introduced by Honda et al. [31, 32, 50], enhance traditional type systems with the ability to specify and enforce structured communication protocols on bidirectional, heterogeneously typed channels. Typically, these specifications include the type, direction (input or output) and order of the messages, as well as branching points where one participant can choose how to proceed and the other must follow.

Traditional session types are bound by tail recursion and therefore restricted to the specification of protocols described by regular languages. This excludes many protocols of practical interest, with the quintessential example being the serialization of tree-structured data on a single channel. Context-free session types, proposed by Thiemann and Vasconcelos [51], liberate types from tail recursion by introducing a sequential composition operator $(\_;\_)$ with a monoidal structure and a left and right identity in type Skip, representing no action. As their name hints, context-free session types can specify protocols corresponding to (simple deterministic) context-free languages and are thus considerably more expressive than their regular counterparts.

What does it mean for a context-free session type to be a subtype of another? Our answer follows Gay and Hole's seminal work on subtyping for regular session types [25], and Liskov's *principle of safe substitution* [39]: S is a subtype of R if channels governed by type S can take the place of channels governed by type R in whatever context, without violating the guarantees offered by a type system (e.g. progress, deadlock freedom, session fidelity, etc.).

More concretely, subtyping allows increased flexibility in the interactions between participants, namely on the type of the messages (a feature inherited from the subtyped π-calculus [46]) and on the choices available at branching points [25], allowing a channel to be governed by a simpler session type if its context so requires. A practical benefit of this flexibility is that it promotes *modular development*: the behaviour of one participant may be refined, while the behaviour of the other is kept intact.

▶ **Example 1.** Consider the following context-free session types for serializing binary trees.

$$\begin{aligned}
\mathsf{STree} &= \mu s.\oplus\{\mathsf{Nil}\colon \mathsf{Skip}, \mathsf{Node}\colon s;!\mathsf{Int};s\} & \mathsf{SEmpty} &= \oplus\{\mathsf{Nil}\colon \mathsf{Skip}\}\\
\mathsf{DTree} &= \mu s.\&\{\mathsf{Nil}\colon \mathsf{Skip}, \mathsf{Node}\colon s;?\mathsf{Int};s\} & \mathsf{SFullTree0} &= \oplus\{\mathsf{Node}\colon \mathsf{SEmpty};!\mathsf{Int};\mathsf{SEmpty}\}\\
& & \mathsf{SFullTree1} &= \oplus\{\mathsf{Node}\colon \mathsf{SFullTree0};!\mathsf{Int};\mathsf{SFullTree0}\}
\end{aligned}$$

The recursive STree and DTree types specify, respectively, the serialization and deserialization of a possibly infinite arbitrary tree, while the remaining non-recursive types specify the serialization of finite trees of particular configurations. The benefit of subtyping is that it makes the particular types SEmpty, $\mathsf{SFullTree0}$ and $\mathsf{SFullTree1}$ compatible with the general DTree type. Observe that its dual, STree, may safely take the place of any type in the right column. Consider now a function f that generates full trees of height 1 and serializes them on a given channel end. Assigning it type $\mathsf{STree} \to \mathsf{Unit}$ would not statically ensure that the fullness and height of the tree are as specified. Type $\mathsf{SFullTree1} \to \mathsf{Unit}$ would do so, and subtyping would still allow the function to use an STree channel (i.e., communicate with someone expecting an arbitrary DTree tree).

Expressive power usually comes at the cost of decidability. While subtyping for regular session types has been formalized, shown decidable and given an algorithm by Gay and Hole [25], subtyping in the context-free setting has been proven undecidable by Padovani [43]. The proof is given by a reduction from the inclusion problem for simple languages, shown undecidable by Friedman [21]. Remarkably, the equivalence problem for simple languages is known to be decidable, as is the type equivalence of context-free session types [36, 51].

Subtyping context-free session types has until now been considered only in a limited form, where message types must be syntactically equal [43]. Consequently, the interesting co/contravariant properties of input/output types have been left unexplored. In this paper, we propose a more expressive subtyping relation, where the types of messages may vary co/contravariantly, according to the classical subtyping notion of Gay and Hole. To handle the contravariance of output types, we introduce a novel notion of observational preorder, which we call $\mathcal{XYZW}$*-simulation* (by analogy with $\mathcal{XY}$*-simulation* [1]).

While initially formulated in the context of the π-calculus, considerable work has been done to integrate session types in more standard settings, such as functional languages based on the polymorphic λ-calculus with linear types [2, 16, 47]. In this scenario, functional types and session types are not orthogonal: sessions may carry functions, and functions may act on sessions. With this in mind, we promote our theory to a linear functional setting, thereby showing how subtyping for records, variants and (linear and unrestricted [22]) functions, usually introduced by inference rules, can be seamlessly integrated with simulation-based subtyping for context-free session types.

Functional and higher-order context-free session types

$$T, U, V, W ::= \mathsf{Unit} \mid T \xrightarrow{m} U \mid (\!|\ell\colon T|\!)_{\ell \in L} \mid S \mid t \mid \mu t.T$$
$$S, R ::= \sharp T \mid \odot\{\ell\colon T\}_{\ell \in L} \mid \mathsf{Skip} \mid \mathsf{End} \mid S;R \mid s \mid \mu s.S$$

Multiplicities, records/variants, polarities and views

$$m, n ::= 1 \mid * \qquad (\!|\cdot|\!) ::= \{\cdot\} \mid \langle\cdot\rangle \qquad \sharp ::= ? \mid ! \qquad \odot ::= \oplus \mid \&$$

Figure 1 Syntax of types.

Finally, we present a sound algorithm for the novel notion of subtyping, based on the type equivalence algorithm of Almeida et al. [4]. This algorithm works by first encoding the types as words in a simple grammar [36] and then deciding their $\mathcal{XYZW}$-similarity. Being grammar-based and, at its core, agnostic to types, our algorithm may also find applications for other objects with similar non-regular and contravariant properties.

Contributions. We address the subtyping problem for context-free session types, proposing:

- A syntactic definition of subtyping for context-free session types;
- A novel kind of behavioural preorder called $\mathcal{XYZW}$-simulation, and, based on it, a semantic definition of subtyping that coincides with the syntactic one;
- A sound subtyping algorithm based on the $\mathcal{XYZW}$-similarity of simple grammars;
- An empirical evaluation of the performance of the algorithm, and a comparison with an existing type equivalence algorithm.

Overview. The rest of this paper is organized as follows: in Section 2 we introduce types, type formation and syntactic subtyping; in Section 3 we present a notion of semantic subtyping, to be used as a stepping stone to develop our subtyping algorithm; in Section 4 we present the algorithm and show it to be sound with respect to the semantic subtyping relation; in Section 5 we evaluate the performance of our implementation of the algorithm; in Section 6 we present related work; in Section 7 we conclude the paper and trace a path for the work to follow. The reader can find the rules for type formation and proofs for all results in the paper in a technical report on arXiv [49].

2 Types and syntactic subtyping

We base our contributions on a type language that includes both functional types and higher-order context-free session types (i.e., types that allow messages of arbitrary types). The language is shown in Figure 1. As customary in session types for functional languages [26], the language of types is given by two mutually recursive syntactic categories: one for functional types and another for session types. We assume two disjoint and denumerable sets of type references, with the first ranged over by t, u, v, w, the second by r, s and their union by x, y, z. We further assume a set of record, variant and choice labels, ranged over by j, k, ℓ.

The first three productions of the grammar for functional types introduce the Unit type, functions $T \xrightarrow{m} U$, records $\{\ell\colon T_\ell\}_{\ell \in L}$ and variants $\langle \ell\colon T_\ell \rangle_{\ell \in L}$ (which correspond to *datatypes* in ML-like languages). Our system exhibits linear characteristics: function types contain a multiplicity annotation m (also in Figure 1), meaning that they must be used exactly once

if $m = 1$ or without restrictions if $m = *$ (such types can also be found, for instance, in Gay's proposal [26], in System F° [40] and in the FreeST language [2]). Their inclusion in our system is justified by the interesting subtyping properties they exhibit [22].

Session types $!T$ and $?T$ represent the sending and receiving, respectively, of a value of type T (an arbitrary type, making the system higher-order). Internal choice types $\oplus\{\ell\colon S_\ell\}_{\ell\in L}$ allow the selection of a label $k \in L$ and its continuation S_k, while external choice types $\&\{\ell\colon S_\ell\}_{\ell\in L}$ represent the branching on any label $k \in L$ and its continuation S_k. We stipulate that the set of labels for these types must be non empty. Type Skip represents no action, while type End indicates the closing of a channel, after which no more communication can take place. Type $R;S$ denotes the sequential composition of R and S, which is associative, right distributes over choices types, has (left and right) identity Skip and left-absorber End.

The final two productions in both functional and session grammars introduce self-references and the recursion operator. Their inclusion in the two grammars ensures we can have both recursive functional types and recursive session types while avoiding nonsensical types such as $\mu t.\mathsf{Unit} \xrightarrow{*} !\mathsf{Unit};t$ at the syntactical level (avoiding the need for a kinding system).

Still, we do not consider all types generated by these grammars to be *well-formed*. Consider session type $\mu r.r;!\mathsf{Unit}$. No matter how many times we unfold it, we cannot resolve its first communication action. The same could be said of $\mu r.\mathsf{Skip};r;!\mathsf{Unit}$. We must therefore ensure that any self-reference in a sequential composition is preceded by a type constructor representing some meaningful action, i.e., not equivalent to Skip. This is achieved by adapting the conventional notion of contractivity (no subterms of the form $\mu x.\mu x_1.\ \ldots\ \mu x_n.x$) [25] to account for Skip as the identity of sequential composition. This corresponds to the notion of *guardedness* in the theory of process algebra (e.g. [28, 42]).

In addition to contractivity, we must ensure that well-formed types contain no free references. The type formation judgement $\Delta \vdash T$, where Δ is a set of references, combines these requirements. The rules for the judgement can be found in the technical report [49].

We are now set to define our syntactic subtyping relation. We begin by surveying the features it should support:

Input and output subtyping. Input variance and output contravariance are the central features of subtyping for types that govern entities that can be written to or read from, such as channels and references [45]. They are therefore natural features of the subtyping relation for conventional session types as well [25]. Observe that $?\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\} \leq ?\{\mathsf{A}\colon \mathsf{Int}\}$ should be true, for the type of the received value, $\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\}$, safely substitutes the expected type, $\{\mathsf{A}\colon \mathsf{Int}\}$. Observe also that $!\{\mathsf{A}\colon \mathsf{Int}\} \leq !\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\}$ should be true, because the type of the value to be sent, $\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\}$, is a subtype of $\{\mathsf{A}\colon \mathsf{Int}\}$, the type of the messages the substitute channel is allowed to send.

Choice subtyping. If we understand external and internal choice types as, respectively, the input and output of a label, then their subtyping properties are easy to derive: external choices are covariant on their label set, internal choices are contravariant on their label set, and both are covariant on the continuation of the labels (this is known as *width subtyping*). Observe that $\&\{\mathsf{A}\colon ?\mathsf{Int}\} \leq \&\{\mathsf{A}\colon ?\mathsf{Int}, \mathsf{B}\colon !\mathsf{Bool}\}$ should be true, for every branch in the first type can be safely handled by matching on the second type. Likewise, $\oplus\{\mathsf{A}\colon ?\mathsf{Int}, \mathsf{B}\colon !\mathsf{Bool}\} \leq \oplus\{\mathsf{A}\colon ?\mathsf{Int}\}$ should be true, for every choice in the second type can be safely selected in the first.

Sequential composition. In the classical subtyping relation for regular session types, input and output types ($\sharp T.S$) can be characterized as covariant in their continuation. Although the same general intuition applies in the context-free setting, we cannot as easily characterize the variance of the sequential composition constructor ($S;R$) due to its monoidal,

Syntactic subtyping (*coinductive*) $\boxed{T \leq T}$

$$\textsc{S-Unit}\ \ \mathsf{Unit} \leq \mathsf{Unit} \qquad \textsc{S-Arrow}\ \ \frac{U_1 \leq T_1 \qquad T_2 \leq U_2 \qquad m \sqsubseteq n}{T_1 \xrightarrow{m} T_2 \leq U_1 \xrightarrow{n} U_2} \qquad \textsc{S-Rcd}\ \ \frac{K \subseteq L \qquad T_j \leq U_j\ (\forall j \in K)}{\{\ell\colon T_\ell\}_{\ell \in L} \leq \{k\colon U_k\}_{k \in K}}$$

$$\textsc{S-Vrt}\ \ \frac{L \subseteq K \qquad T_j \leq U_j\ (\forall j \in L)}{\langle \ell\colon T_\ell\rangle_{\ell \in L} \leq \langle k\colon U_k\rangle_{k \in K}} \qquad \textsc{S-RecL}\ \ \frac{[\mu x.T/x]T \leq U}{\mu x.T \leq U} \qquad \textsc{S-RecR}\ \ \frac{T \leq [\mu x.U/x]U}{T \leq \mu x.U} \qquad \textsc{S-In}\ \ \frac{T \leq U}{?T \leq ?U}$$

$$\textsc{S-Out}\ \ \frac{U \leq T}{!T \leq !U} \qquad \textsc{S-ExtChoice}\ \ \frac{L \subseteq K \qquad S_j \leq R_j\ (\forall j \in L)}{\&\{\ell\colon S_\ell\}_{\ell \in L} \leq \&\{k\colon R_k\}_{k \in K}} \qquad \textsc{S-IntChoice}\ \ \frac{K \subseteq L \qquad S_j \leq R_j\ (\forall j \in K)}{\oplus\{\ell\colon S_\ell\}_{\ell \in L} \leq \oplus\{k\colon R_k\}_{k \in K}}$$

$$\textsc{S-Skip}\ \ \mathsf{Skip} \leq \mathsf{Skip} \qquad \textsc{S-End}\ \ \mathsf{End} \leq \mathsf{End} \qquad \textsc{S-InSeq1L}\ \ \frac{T \leq U \qquad S \leq \mathsf{Skip}}{?T;S \leq ?U} \qquad \textsc{S-InSeq1R}\ \ \frac{T \leq U \qquad S \leq \mathsf{Skip}}{?T \leq ?U;S}$$

$$\textsc{S-InSeq2}\ \ \frac{T \leq U \qquad S \leq R}{?T;S \leq ?U;R} \qquad \textsc{S-OutSeq1L}\ \ \frac{U \leq T \qquad S \leq \mathsf{Skip}}{!T;S \leq !U} \qquad \textsc{S-OutSeq1R}\ \ \frac{U \leq T \qquad S \leq \mathsf{Skip}}{!T \leq !U;S} \qquad \textsc{S-OutSeq2}\ \ \frac{U \leq T \qquad S \leq R}{!T;S \leq !U;R}$$

$$\textsc{S-ChoiceSeqL}\ \ \frac{\odot\{\ell\colon S_\ell;S\}_{\ell \in L} \leq R}{\odot\{\ell\colon S_\ell\}_{\ell \in L};S \leq R} \qquad \textsc{S-ChoiceSeqR}\ \ \frac{S \leq \odot\{\ell\colon R_\ell;R\}_{\ell \in L}}{S \leq \odot\{\ell\colon R_\ell\}_{\ell \in L};R} \qquad \textsc{S-SkipSeqL}\ \ \frac{S \leq R}{\mathsf{Skip};S \leq R} \qquad \textsc{S-SkipSeqR}\ \ \frac{S \leq R}{S \leq \mathsf{Skip};R}$$

$$\textsc{S-EndSeq1L}\ \ \mathsf{End};S \leq \mathsf{End} \qquad \textsc{S-EndSeq1R}\ \ \mathsf{End} \leq \mathsf{End};R \qquad \textsc{S-EndSeq2}\ \ \mathsf{End};S \leq \mathsf{End};R \qquad \textsc{S-SeqSeqL}\ \ \frac{S_1;(S_2;S_3) \leq R}{(S_1;S_2);S_3 \leq R} \qquad \textsc{S-SeqSeqR}\ \ \frac{S \leq R_1;(R_2;R_3)}{S \leq (R_1;R_2);R_3}$$

$$\textsc{S-RecSeqL}\ \ \frac{([\mu s.S_1/s]S_1);S_2 \leq R}{(\mu s.S_1);S_2 \leq R} \qquad \textsc{S-RecSeqR}\ \ \frac{S \leq ([\mu s.R_1/s]R_1);R_2}{S \leq (\mu s.R_1);R_2}$$

Preorder on multiplicities $\boxed{m \sqsubseteq m}$

$$m \sqsubseteq m \qquad * \sqsubseteq 1$$

Figure 2 Syntactic subtyping.

distributive and absorbing properties. For instance, consider types $S_1;S_2$ and $R_1;R_2$, with $S_1 = !\mathsf{Int};!\mathsf{Bool}$, $S_2 = ?\mathsf{Int}$, $R_1 = !\mathsf{Int}$ and $R_2 = !\mathsf{Bool};?\mathsf{Int}$. Although it should be true that $S_1;S_2 \leq R_1;R_2$, we can have neither $S_1 \leq R_1$ nor $S_2 \leq R_2$.

Functional subtyping. The subtyping properties of function, record and variant types are well known, and we refer the readers to Pierce's book for the reasoning behind them [45]. Succinctly, the function type constructor is contravariant on the domain and covariant on the range, and the variant and record constructors are both covariant on the type of the fields, but respectively covariant and contravariant on their label sets.

Multiplicity subtyping. Using an unrestricted ($*$) resource where a linear (1) one is expected does not compromise safety, provided that, multiplicities aside, the type of the former may safely substitute the type of the latter. We can express this relationship between multiplicities through a preorder captured by inequality $* \sqsubseteq 1$. In our system, function types may be either linear or unrestricted. Thus, type $T_1 \xrightarrow{m} T_2$ can be considered a subtype of $U_1 \xrightarrow{n} U_2$ if U_1 and T_2 are subtypes, respectively, of T_1 and U_2 and if $m \sqsubseteq n$ (thus we can characterize the function type constructor as covariant on its multiplicity).

The rules for our syntactic subtyping relation, interpreted coinductively, are shown in Figure 2. Rules S-Unit, S-Arrow, S-Rcd, S-Vrt, S-RecL and S-RecR establish the classical subtyping properties associated with both functional and equi-recursive types, with S-Arrow additionally encoding subtyping between linear and unrestricted functions, relying on a preorder on multiplicities also defined in Figure 2. Rules S-End, S-In, S-Out, S-ExtChoice and S-IntChoice bring to the context-free setting the classical subtyping properties expected from session types, as put forth by Gay and Hole [25].

The remaining rules account for sequential composition, which distributes over choice and exhibits a monoidal structure with its neutral element in Skip and left-absorbing element in End. We include, for each session type constructor S, a left rule (denoted by suffix L) of the form $S;R \leq S'$ and a right rule (denoted by suffix R) of the form $S' \leq S;R$. An additional rule is necessary for each constructor over which sequential composition does not distribute, associate or neutralize (S-InSeq2, S-OutSeq2 and S-EndSeq2). Since we are using a coinductive proof scheme, we include rules to "move" sequential composition down the syntax. Thus, given a type $S;R$, we inspect S to decide which rule to apply next.

▶ **Theorem 2.** *The syntactic subtyping relation $\leq$ is a preorder on types.*

▶ **Example 3.** Let us briefly return to Example 1. It is now easy to see that $\mathsf{STree} \leq \mathsf{SFullTree1}$: we unfold the left-hand side and apply rule S-IntChoice. Then we apply the distributivity rules as necessary until reaching an internal choice with no continuation, at which point we can apply S-IntChoice again, or until reaching a type with !Int at the head, at which point we apply S-InSeq2. We repeat this process until reaching $\mathsf{STree} \leq \mathsf{SFullTree0}$, and proceed similarly until reaching $\mathsf{STree} \leq \mathsf{SEmpty}$, which follows from S-IntChoice and S-Skip.

Despite clearly conveying the intended meaning of the subtyping relation, the rules suggest no obvious algorithmic intepretation: on the one hand, the presence of bare metavariables makes the system not syntax-directed; on the other hand, rules S-RecL, S-RecSeqL and their right counterparts lead to infinite derivations which are not solvable by a conventional fixed-point construction [25, 45]. In the next section we develop an alternative, semantic approach to subtyping, which we use as a stepping stone to develop our subtyping algorithm.

3 Semantic subtyping

Semantic equivalence for context-free session types is usually based on *observational equivalence* or *bisimilarity*, meaning that two session types are considered equivalent if they exhibit exactly the same communication behaviour [51]. An analogous notion of *semantic subtyping* should therefore rely on an *observational preorder*. In this section we develop such a preorder.

We define the behaviour of types via a labelled transition system (LTS) by establishing relation $T \xrightarrow{a} U$ ("type T transitions by action a to type U"). We follow Costa et al. [16] in attributing behaviour to functional types, allowing them to be encompassed in our observational preorder. The rules defining the transition relation, as well as the grammar that generates all possible transition actions, are shown in Figure 3.

Labelled transition system $\boxed{T \xrightarrow{a} T}$

$$\textsc{L-Unit}\ \ \mathsf{Unit} \xrightarrow{\mathsf{Unit}} \mathsf{Skip} \qquad \textsc{L-ArrowDom}\ \ (T \xrightarrow{m} U) \xrightarrow{\to\mathsf{d}} T \qquad \textsc{L-ArrowRng}\ \ (T \xrightarrow{m} U) \xrightarrow{\to\mathsf{r}} U \qquad \textsc{L-LinArrow}\ \ (T \xrightarrow{1} U) \xrightarrow{\to 1} \mathsf{Skip}$$

$$\textsc{L-RcdVrtField}\ \ \frac{k \in L}{(\!|\ell\colon T_\ell|\!)_{\ell\in L} \xrightarrow{(\!|\!|_k} T_k} \qquad \textsc{L-RcdVrt}\ \ (\!|\ell\colon T_\ell|\!)_{\ell\in L} \xrightarrow{(\!|\!|} \mathsf{Skip} \qquad \textsc{L-Rec}\ \ \frac{[\mu x.T/x]T \xrightarrow{a} U}{\mu x.T \xrightarrow{a} U} \qquad \textsc{L-Msg1}\ \ \sharp T \xrightarrow{\sharp\mathsf{p}} T \qquad \textsc{L-Msg2}\ \ \sharp T \xrightarrow{\sharp\mathsf{c}} \mathsf{Skip}$$

$$\textsc{L-Choice}\ \ \odot\{\ell\colon S_\ell\}_{\ell\in L} \xrightarrow{\odot} \mathsf{Skip} \qquad \textsc{L-ChoiceField}\ \ \frac{k \in L}{\odot\{\ell\colon S_\ell\}_{\ell\in L} \xrightarrow{\odot_k} S_k} \qquad \textsc{L-End}\ \ \mathsf{End} \xrightarrow{\mathsf{End}} \mathsf{Skip} \qquad \textsc{L-MsgSeq1}\ \ \sharp T;S \xrightarrow{\sharp\mathsf{p}} T \qquad \textsc{L-MsgSeq2}\ \ \sharp T;S \xrightarrow{\sharp\mathsf{c}} S$$

$$\textsc{L-ChoiceSeq}\ \ \odot\{\ell\colon S_\ell\}_{\ell\in L};R \xrightarrow{\odot} \mathsf{Skip} \qquad \textsc{L-SkipSeq}\ \ \frac{S \xrightarrow{a} T}{\mathsf{Skip};S \xrightarrow{a} T} \qquad \textsc{L-EndSeq}\ \ \mathsf{End};S \xrightarrow{\mathsf{End}} \mathsf{Skip} \qquad \textsc{L-SeqSeq}\ \ \frac{S_1;(S_2;S_3) \xrightarrow{a} T}{(S_1;S_2);S_3 \xrightarrow{a} T}$$

$$\textsc{L-ChoiceFieldSeq}\ \ \frac{k \in L}{\odot\{\ell\colon S_\ell\}_{\ell\in L};R \xrightarrow{\odot_k} S_k;R} \qquad \textsc{L-RecSeq}\ \ \frac{([\mu s.S/s]S);R \xrightarrow{a} T}{(\mu s.S);R \xrightarrow{a} T} \qquad \text{(no rule for } \mathsf{Skip}\text{)}$$

Actions

$$a ::= \mathsf{Unit} \mid \to\mathsf{d} \mid \to\mathsf{r} \mid \to 1 \mid \mathsf{End} \mid (\!|\!|_\ell \mid (\!|\!| \mid \sharp\mathsf{p} \mid \sharp\mathsf{c} \mid \odot \mid \odot_\ell$$

Figure 3 Labelled transition system. Letters d, r, p, c in labels stand for "domain", "range", "payload" and "continuation".

In general, each functional type constructor generates a transition for each of its fields (Unit and End, which have none, transition to Skip). Linear functions exhibit an additional transition to represent their restricted use (L-LinArrow), and records/variants include a default transition that is independent of their fields (L-RcdVrt). The behaviour of session types is more complex, since it must account for their algebraic properties. Message types exhibit a transition for their payload (L-Msg1, L-MsgSeq1) and another for their continuation, which is Skip by omission (L-Msg2, L-MsgSeq2). Choices behave much like records/variants when alone, but are subject to distributivity when composed (L-ChoiceFieldSeq). Type End, which absorbs its continuation, transitions to Skip (L-End, L-EndSeq). Rules L-SeqSeq, L-SkipSeq account for associativity and identity, and rules L-Rec and L-RecSeq dictate that recursive types behave just like their unfoldings. Notice that Skip has no transitions.

With the behaviour of types established, we now look for an appropriate notion of observational preorder. Several such notions have been studied in the literature. *Similarity*, defined as follows, is arguably the simplest of them [41, 44].

▶ **Definition 4.** *A type relation $\mathcal{R}$ is said to be a simulation if, whenever $T\mathcal{R}U$, for all a and T' with $T \xrightarrow{a} T'$ there is U' such that $U \xrightarrow{a} U'$ and $T'\mathcal{R}U'$*

Similarity, written $\preceq$, is the union of all simulation relations. We say that a type U simulates type T if $T \preceq U$.

Unfortunately, plain similarity is of no use to us. A small example shows why: type $\oplus\{\mathsf{A}\colon \mathsf{End}, \mathsf{B}\colon \mathsf{End}\}$ both simulates and is a subtype of $\oplus\{\mathsf{A}\colon \mathsf{End}\}$, while type $\&\{\mathsf{A}\colon \mathsf{End}\}$ does not simulate yet is a subtype of $\&\{\mathsf{A}\colon \mathsf{End}, \mathsf{B}\colon \mathsf{End}\}$. Reversing the direction of the simulation would be of no avail either, as it would leave us with the reverse problem.

It is apparent that a more refined notion of simulation is necessary, where the direction of the implication depends on the transition labels. Aarts and Vaandrager provide just such a notion in the form of $\mathcal{XY}$*-simulation* [1], a simulation relation parameterized by two subsets of actions, $\mathcal{X}$ and $\mathcal{Y}$, such that actions in $\mathcal{X}$ are simulated from left to right and those in $\mathcal{Y}$ are simulated from right to left, selectively combining the requirements of simulation and reverse simulation.

▶ **Definition 5.** *Let $\mathcal{X}, \mathcal{Y} \subseteq \mathcal{A}$. A type relation $\mathcal{R}$ is said to be an $\mathcal{XY}$-simulation if, whenever $T\mathcal{R}U$, we have:*
1. *for each $a \in \mathcal{X}$ and each T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ with $T'\mathcal{R}U'$;*
2. *for each $a \in \mathcal{Y}$ and each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ with $T'\mathcal{R}U'$.*
$\mathcal{XY}$-similarity, written $\preceq^{\mathcal{XY}}$, is the union of all $\mathcal{XY}$-simulation relations. We say that a type T is $\mathcal{XY}$-similar to type U if $T \preceq^{\mathcal{XY}} U$.

Similar or equivalent notions have appeared throughout the literature: *modal refinement* [38], *alternating simulation* [7] and, perhaps more appropriately named (for our purposes), *covariant-contravariant simulation* [20]. Padovani's original subtyping relation for context-free session types [43] can also be understood as a refined form of $\mathcal{XY}$-simulation.

We can tentatively define a semantic subtyping relation $\lesssim'$ as $\mathcal{XY}$-similarity, where $\mathcal{X}$ and $\mathcal{Y}$ are the label sets generated by the following grammars for $a_{\mathcal{X}}$ and $a_{\mathcal{Y}}$, respectively.

$$a_{\mathcal{X}} ::= a_{\mathcal{XY}} \mid \langle\rangle_\ell \mid \&_\ell \qquad a_{\mathcal{XY}} ::= \mathsf{Unit} \mid \to\mathsf{d} \mid \to\mathsf{r} \mid (\!|\,|\!) \mid \sharp\mathsf{p} \mid \sharp\mathsf{c} \mid \odot \mid \mathsf{End}$$
$$a_{\mathcal{Y}} ::= a_{\mathcal{XY}} \mid \to\mathsf{1} \mid \{\}_\ell \mid \oplus_\ell$$

This would indeed give us the desired result for our previous example, but we still cannot account for the contravariance of output and function types: we want $T = !\{\mathsf{A}\colon \mathsf{Int}\}$ to be a subtype of $U = !\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\}$, yet $T \lesssim' U$ does not hold (in fact, we have $U \lesssim' T$, a clear violation of run-time safety). The same could be said for types $\{\mathsf{A}\colon \mathsf{Int}\} \xrightarrow{*} \mathsf{Int}$ and $\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\} \xrightarrow{*} \mathsf{Int}$. In short, our simulation needs the $!\mathsf{p}$ and $\to\mathsf{d}$-derivatives to be related in the direction opposite to that of the initial types. Thus we need to selectively apply a strong form of *contrasimulation* as well [48, 52] (the original notion is defined with weak transitions, a sort of transition we do not address).

To allow this, we generalize the definition of $\mathcal{XY}$-simulation by parameterizing it on two further subsets of actions and including two more clauses where the direction of the relation between the derivatives is reversed. By analogy with $\mathcal{XY}$-simulation, we call the resulting notion $\mathcal{XYZW}$-simulation.

▶ **Definition 6.** *Let $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{W} \subseteq \mathcal{A}$. A type relation $\mathcal{R}$ is a $\mathcal{XYZW}$-simulation if, whenever $T\mathcal{R}U$, we have:*
1. *for each $a \in \mathcal{X}$ and each T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ with $T'\mathcal{R}U'$;*
2. *for each $a \in \mathcal{Y}$ and each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ with $T'\mathcal{R}U'$;*
3. *for each $a \in \mathcal{Z}$ and each T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ with $U'\mathcal{R}T'$;*
4. *for each $a \in \mathcal{W}$ and each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ with $U'\mathcal{R}T'$.*
$\mathcal{XYZW}$-similarity, written $\preceq^{\mathcal{XYZW}}$, is the union of all $\mathcal{XYZW}$-simulation relations. We say that a type T is $\mathcal{XYZW}$-similar to type U if $T \preceq^{\mathcal{XYZW}} U$.

$\mathcal{XYZW}$-simulation generalizes several existing observational relations: $\mathcal{XY}$-simulation can be defined as an $\mathcal{XY}\emptyset\emptyset$-simulation, bisimulation as $\mathcal{AA}\emptyset\emptyset$-simulation (alternatively, $\emptyset\emptyset\mathcal{AA}$-simulation or $\mathcal{AAAA}$-simulation), and plain simulation as $\mathcal{A}\emptyset\emptyset\emptyset$-simulation.

▶ **Theorem 7.** *For any $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{W}$, $\preceq^{\mathcal{XYZW}}$ is a preorder relation on types.*

Equipped with the notion of $\mathcal{XYZW}$-similarity, we are ready to define the semantic subtyping relation for functional and higher-order context-free session types as follows.

▶ **Definition 8.** *The semantic subtyping relation for functional and higher-order context-free session types $\lesssim$ is defined by $T \lesssim U$ when $T \preceq^{\mathcal{XYZW}} U$ such that $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$ and $\mathcal{W}$ are defined as the label sets generated by the following grammars for $a_{\mathcal{X}}$, $a_{\mathcal{Y}}$, $a_{\mathcal{Z}}$ and $a_{\mathcal{W}}$, respectively.*

$$a_{\mathcal{X}} ::= a_{\mathcal{XY}} \mid \to 1 \mid \langle\rangle_\ell \mid \&_\ell \qquad a_{\mathcal{Z}}, a_{\mathcal{W}} ::= !\mathsf{p} \mid \to \mathsf{d}$$
$$a_{\mathcal{Y}} ::= a_{\mathcal{XY}} \mid \{\}_\ell \mid \oplus_\ell \qquad a_{\mathcal{XY}} ::= \mathsf{Unit} \mid \to \mathsf{r} \mid (\!|\,|\!) \mid ?\mathsf{p} \mid \sharp\mathsf{c} \mid \odot \mid \mathsf{End}$$

Notice the correspondence between the placement of the labels and the variance of their respective type constructors. Labels arising from covariant positions of the arrow and input type constructors are placed in both the $\mathcal{X}$ and $\mathcal{Y}$ sets, while those arising from the contravariant positions of the arrow and output type constructors are placed in both the $\mathcal{Z}$ and $\mathcal{W}$ sets. Labels arising from the fields of constructors exhibiting width subtyping are placed in a single set, depending on the variance of the constructor on the label set: $\mathcal{X}$ for covariance (external choice and variant constructors), $\mathcal{Y}$ for contravariance (internal choice and record constructors). The function type constructor is covariant on its multiplicity, thus the linear arrow label is placed in $\mathcal{X}$. Finally, default record/variant/choice labels and those arising from nullary constructors are placed in $\mathcal{X}$ and $\mathcal{Y}$, but they could alternatively be placed in $\mathcal{Z}$ and $\mathcal{W}$ or in all four sets (notice the parallel with bisimulation, that can be defined as $\mathcal{AA}\emptyset\emptyset$-simulation, $\emptyset\emptyset\mathcal{AA}$-simulation, or $\mathcal{AAAA}$-simulation).

▶ **Example 9.** Let us go back once again to our tree serialization example from Section 1. Here it is also easy to see that $\mathsf{STree} \lesssim \mathsf{SFullTree1}$. Observe that, on the side of STree, transitions by $\oplus_{\mathsf{Nil}}$ and $\oplus_{\mathsf{Node}}$ always appear together, while on the side of $\mathsf{SFullTree1}$ types transition first by $\oplus_{\mathsf{Node}}$ and then by $\oplus_{\mathsf{Nil}}$. Since $\oplus_{\mathsf{Nil}}$ and $\oplus_{\mathsf{Node}}$ belong exclusively to $\mathcal{Y}$, STree is always able to match $\mathsf{SFullTree1}$ on these labels (as in all the others in $\mathcal{Y} \cup \mathcal{W}$, and *vice-versa* for $\mathcal{X} \cup \mathcal{Z}$).

▶ **Theorem 10** (Soundness and completeness for subtyping relations)**.** *Let $\vdash T$ and $\vdash U$. Then $T \leq U$ iff $T \lesssim U$.*

4 A subtyping algorithm

The notion of subtyping we have outlined is undecidable. This follows from the fact that our system, albeit different, contains all the features necessary to reconstruct Padovani's proof of undecidability [43]. Using just external choices, sequential composition, the Skip type and recursion, one is able to encode simple grammars [36] as context-free session types, in a way that language strings correspond to complete LTS traces of types. By exploiting the covariant width-subtyping in external choices, one can show that subtyping for these types corresponds to language inclusion, which is known to be undecidable for simple languages [21].

Despite the undecidability of our subtyping problem, we are still able to devise a sound (but necessarily incomplete) algorithm for it. In this section we present this algorithm, an adaptation of the equivalence algorithm of Almeida et al. [4]. At its core, it determines the $\mathcal{XYZW}$-similarity of simple grammars. Its application to context-free session types is

facilitated by a translation function to properly encode types as grammars. The algorithm may likewise be adapted to other domains. Much like the original, our algorithm can be succinctly described in three distinct phases:

1. translate the given types to a simple grammar [36] and two starting words;
2. prune unreachable symbols from productions;
3. explore an expansion tree rooted at a node containing the initial words, alternating between expansion and simplification operations until either an empty node is found (decide **True**) or all nodes fail to expand (decide **False**).

Phase 1. The first phase consists of translating the two types to a grammar in *Greibach normal form* (GNF) [27], i.e., a grammar where all productions have the form $Y \to a\vec{Z}$, and two starting words $(\vec{X}, \vec{Y})$. A word is defined as a sequence of non-terminal symbols. We can check the $\mathcal{XYZW}$-similarity of words in GNF grammars because they naturally induce a labelled transition system, where states are words $\vec{X}$, actions are terminal symbols a and the transition relation is defined as $X\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Z}\vec{Y}$ when $X \to a\vec{Z} \in \mathcal{P}$. We denote the bisimilarity and $\mathcal{XYZW}$-similarity of grammars by, respectively, $\sim_{\mathcal{P}}$ and $\preceq_{\mathcal{P}}^{\mathcal{XYZW}}$, where $\mathcal{P}$ is the set of productions. We also let $\lesssim_{\mathcal{P}}$ denote grammar $\mathcal{XYZW}$-similarity with label sets as in Definition 8. The deterministic nature of context-free session types allows their corresponding grammars to be simple [36]: for each non-terminal Y and terminal symbol a, we have at most one production of the form $Y \to a\vec{Z}$.

The grammar translation procedure *grm* remains unchanged from the original equivalence algorithm [4], and for this reason we omit its details (which include generating productions for all μ-subterms in types). However, this procedure relies on two auxiliary definitions which must be adapted: the *unr* function (Definition 11), which normalizes the head of session types and unravels recursive types until reaching a type constructor, and the *word* procedure (Definition 12), which builds a word from a session type while updating a set $\mathcal{P}$ of productions.

▶ **Definition 11.** *The* unraveling *of a type T is defined by induction on the structure of T:*

$$\begin{aligned} unr(\mu x.T) &= unr([\mu x.T/x]T) & unr(\mathsf{Skip};S) &= unr(S) \\ unr(\mathsf{End};S) &= \mathsf{End} & unr((\mu s.S);R) &= unr(([\mu s.S/s]S);R) \\ unr(\odot\{\ell\colon S_\ell\}_{\ell\in L};R) &= \odot\{\ell\colon S_\ell; R\}_{\ell\in L} & unr((S_1;S_2);S_3) &= unr(S_1;(S_2;S_3)) \end{aligned}$$

and in all other cases by $unr(T) = T$.

▶ **Definition 12.** *The word corresponding to a well-formed type T, $word(T)$, is built by descending on the structure of T while updating a set $\mathcal{P}$ of productions:*

$$\begin{aligned} word(\mathsf{Unit}) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \mathsf{Unit}\} \\ word(U \xrightarrow{1} V) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \to\mathsf{d}word(U), Y \to \to\mathsf{r}word(V), Y \to \to\mathsf{1}\} \\ word(U \xrightarrow{*} V) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \to\mathsf{d}word(U), Y \to \to\mathsf{r}word(V)\} \\ word((\!|\ell\colon T_\ell|\!)_{\ell\in L}) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to (\!|\bot|\!)\} \cup \{Y \to (\!|\!)_k word(T_k) \mid k \in L\} \\ word(\mathsf{Skip}) &= \varepsilon \\ word(\mathsf{End}) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \mathsf{End}\bot\} \\ word(\sharp U) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \sharp\mathsf{p}word(U)\bot, Y \to \sharp\mathsf{c}\} \\ word(\odot\{\ell\colon S_\ell\}_{\ell\in L}) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \odot\bot\} \cup \{Y \to \odot_\mathsf{k} word(S_k) \mid k \in L\} \\ word(S_1; S_2) &= word(S_1)word(S_2) \\ word(\mu x.U) &= X \end{aligned}$$

where, in each equation, Y is understood as a fresh non-terminal symbol, X as the non-terminal symbol corresponding to type reference x, and $\bot$ as a non-terminal symbol without productions.

▶ **Example 13.** Consider again the types for tree serialization in Section 1. Suppose we want to know whether $\mathsf{SFullTree0} \xrightarrow{*} \mathsf{Unit} \lesssim \mathsf{STree} \xrightarrow{1} \mathsf{Unit}$. We know that the grammar generated for these types is as follows, with X_0 and Y_0 as their starting words.

$$
\begin{array}{llllll}
X_0 \to \rightarrow \mathsf{d} X_1 & X_2 \to \oplus_{\mathsf{Empty}} & X_4 \to \mathsf{Int} & Y_0 \to \rightarrow \mathsf{d} Y_1 & Y_1 \to \oplus \bot \\
X_0 \to \rightarrow \mathsf{r} X_5 & X_2 \to \oplus \bot & X_5 \to \mathsf{Unit} & Y_0 \to \rightarrow \mathsf{r} X_5 & Y_1 \to \oplus_{\mathsf{Empty}} \\
X_1 \to \oplus_{\mathsf{Node}} X_2 X_3 X_2 & X_3 \to !\mathsf{p} X_4 \bot & & Y_0 \to \rightarrow 1 & Y_1 \to \oplus_{\mathsf{Node}} Y_1 X_3 Y_1 \\
X_1 \to \oplus \bot & X_3 \to !\mathsf{c} & & &
\end{array}
$$

For the rest of this section let $\vdash T$, $\vdash U$, $(\vec{X}_T, \mathcal{P}') = grm(T, \emptyset)$ and $(\vec{X}_U, \mathcal{P}) = grm(U, \mathcal{P}')$.

▶ **Theorem 14** (Soundness for grammars). *If $\vec{X}_T \lesssim_{\mathcal{P}} \vec{X}_U$, then $T \lesssim U$.*

Phase 2. The grammars generated by procedure *grm* may contain unreachable words, which can be ignored by the algorithm. Intuitively, these words correspond to communication actions that cannot be fulfilled, such as subterm ?Bool in type $(\mu s.!\mathsf{Int};s);?\mathsf{Bool}$. Formally, these words appear in productions following what are known as *unnormed words*.

▶ **Definition 15.** *Let $\vec{a}$ be a non-empty sequence of non-terminal symbols $a_1, \ldots, a_n$. Write $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$ when $\vec{Y} \xrightarrow{a_1}_{\mathcal{P}} \ldots \xrightarrow{a_n}_{\mathcal{P}} \vec{Z}$. We say that a word $\vec{Y}$ is normed if $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \varepsilon$ for some $\vec{a}$, and unnormed otherwise. If $\vec{Y}$ is normed and $\vec{a}$ is the shortest path such that $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \varepsilon$, then $\vec{a}$ is called the minimal path of $\vec{Y}$, and its length is the norm of $\vec{Y}$, denoted $|\vec{Y}|$.*

It is known that any unnormed word $\vec{Y}$ is bisimilar to its concatenation with any other word, i.e., if $\vec{Y}$ is unnormed, then $\vec{Y} \sim_{\mathcal{P}} \vec{Y}\vec{X}$. It is also easy to show that $\sim_{\mathcal{P}} \subseteq \lesssim_{\mathcal{P}}$, and hence that $\vec{Y} \lesssim_{\mathcal{P}} \vec{Y}\vec{X}$. In this case, $\vec{X}$ is said to be unreachable and can be safely removed from the grammar. We call the procedure of removing all unreachable symbols from a grammar *pruning*, and denote the pruned version of a grammar $\mathcal{P}$ by $prune(\mathcal{P})$.

▶ **Lemma 16** (Pruning preserves $\mathcal{XYZW}$-similarity). *$\vec{X} \preceq_{\mathcal{P}}^{\mathcal{XYZW}} \vec{Y}$ iff $\vec{X} \preceq_{prune(\mathcal{P})}^{\mathcal{XYZW}} \vec{Y}$*

Phase 3. In its third and final phase, the algorithm explores an *expansion tree*, alternating between expansion and simplification steps. An expansion tree is a tree whose nodes are sets of pairs of words, whose root is the singleton set containing the pair of starting words under test, and where every child is an *expansion* of its parent. A branch is deemed *successful* if it is infinite or has an empty leaf, and deemed *unsuccessful* otherwise. The original definition of expansion ensures that the union of all nodes along a successful branch (without simplifications) constitutes a bisimulation [35]. We adapt this definition to ensure that such a union yields an $\mathcal{XYZW}$-simulation instead.

▶ **Definition 17.** *The* $\mathcal{XYZW}$-expansion *of a node N is defined as the minimal set N' such that, for every pair $(\vec{X}, \vec{Y})$ in N, it holds that:*

1. *if $\vec{X} \to a\vec{X}'$ and $a \in \mathcal{X}$ then $\vec{Y} \to a\vec{Y}'$ with $(\vec{X}', \vec{Y}') \in N'$*
2. *if $\vec{Y} \to a\vec{Y}'$ and $a \in \mathcal{Y}$ then $\vec{X} \to a\vec{X}'$ with $(\vec{X}', \vec{Y}') \in N'$*
3. *if $\vec{X} \to a\vec{X}'$ and $a \in \mathcal{Z}$ then $\vec{Y} \to a\vec{Y}'$ with $(\vec{Y}', \vec{X}') \in N'$*
4. *if $\vec{Y} \to a\vec{Y}'$ and $a \in \mathcal{W}$ then $\vec{X} \to a\vec{X}'$ with $(\vec{Y}', \vec{X}') \in N'$*

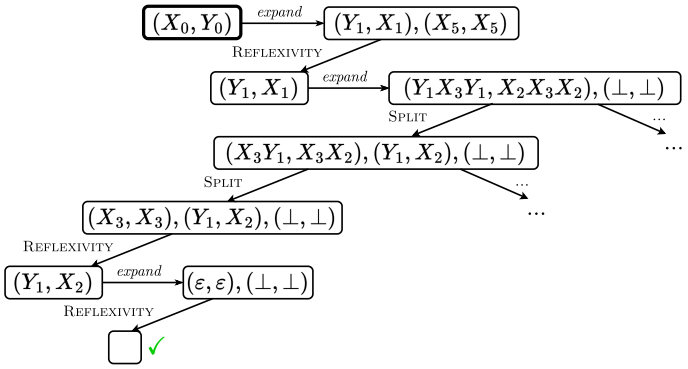


Figure 4 An $\mathcal{XYZW}$-expansion tree for Example 13, exhibiting a finite successful branch.

▶ **Lemma 18** (Safeness property for $\mathcal{XYZW}$-simulation). *Given a set of productions $\mathcal{P}$, $\vec{X} \preceq_{\mathcal{P}}^{\mathcal{XYZW}} \vec{Y}$ iff the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$ has a successful branch.*

The simplification stage consists of applying rules that safely modify the expansion tree during its construction, in an attempt to keep some branches finite. The rules are iteratively applied to each node until a fixed point is reached, at which point we can proceed with expansion. To each node N we apply three simplification rules, adapted from the equivalence algorithm [4]:

1. Reflexivity: omit pairs of the form $(\vec{X}, \vec{X})$;
2. Preorder: omit pairs belonging to the least preorder containing the ancestors of N;
3. Split: if $(X_0\vec{X}, Y_0\vec{Y}) \in N$ and X_0 and Y_0 are normed, then:
   - Case $|X_0| \leq |Y_0|$: Let $\vec{a}$ be a minimal path for X_0 and $\vec{Z}$ the word such that $Y_0 \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Add a sibling node for N including pairs $(X_0\vec{Z}, Y_0)$ and $(\vec{X}, \vec{Z}\vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$;
   - Otherwise: Let $\vec{a}$ be a minimal path for Y_0 and $\vec{Z}$ the word such that $X_0 \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Add a sibling node for N including pairs $(X_0, Y_0\vec{Z})$ and $(\vec{Z}\vec{X}, \vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$.

When a node is simplified, we keep track of the original node in a sibling, thus ensuring that along the tree we keep an "expansion-only" branch.

The algorithm explores the tree by breadth-first search using a queue of node-ancestors pairs, thus avoiding getting stuck in infinite branches, and alternates between expansion and simplification steps until it terminates with **False** if all nodes fail to expand or with **True** if an empty node is reached. The following pseudo-code illustrates the procedure.

$$
\begin{array}{l}
\mathit{subG}(\vec{X}, \vec{Y}, \mathcal{P}) = \mathit{explore}(\mathit{singletonQueue}((\{(\vec{X}, \vec{Y})\}, \emptyset), \mathcal{P}) \\
\quad \textbf{where } \mathit{explore}(q, \mathcal{P}) = \\
\quad\quad \textbf{if } \mathit{empty}(q) \textbf{ then False } \% \text{ all nodes failed to expand} \\
\quad\quad \textbf{else let } (n, a) = \mathit{front}(q) \textbf{ in} \\
\quad\quad\quad \textbf{if } \mathit{empty}(n) \textbf{ then True } \% \text{ empty node reached} \\
\quad\quad\quad \textbf{else if } \mathit{hasExpansion}(n, \mathcal{P}) \% \text{ then expand, simplify and recur} \\
\quad\quad\quad\quad \textbf{then } \mathit{explore}(\mathit{simplify}(\mathit{expand}(n, \mathcal{P}), a \cup n, \mathit{dequeue}(q)), \mathcal{P}) \\
\quad\quad\quad\quad \textbf{else } \mathit{explore}(\mathit{dequeue}(q), \mathcal{P}) \% \text{ otherwise, discard node}
\end{array}
$$

▶ **Example 19.** The $\mathcal{XYZW}$-expansion tree for Example 13 is illustrated in Figure 4.

Finally, function $subT$ puts all the pieces of the algorithm together:

$$subT(T,U) = \textbf{let } (\vec{X},\mathcal{P}') = grm(T,\emptyset), (\vec{Y},\mathcal{P}) = grm(U,\mathcal{P}') \textbf{ in } subG(\vec{X},\vec{Y},prune(\mathcal{P}))$$

It receives two well-formed types T and U, computes their grammar and respective starting words $\vec{X}$ and $\vec{Y}$, prunes the productions of the grammar and, lastly, uses function $subG$ to determine whether $\vec{X} \lesssim_{\mathcal{P}} \vec{Y}$.

The following result shows that algorithm $subT$ is sound with respect to semantic subtyping relation on functional and higher-order context-free session types.

▶ **Theorem 20** (Soundness). *If $subT(T,U)$ returns* **True**, *then $T \lesssim U$.*

5 Evaluation

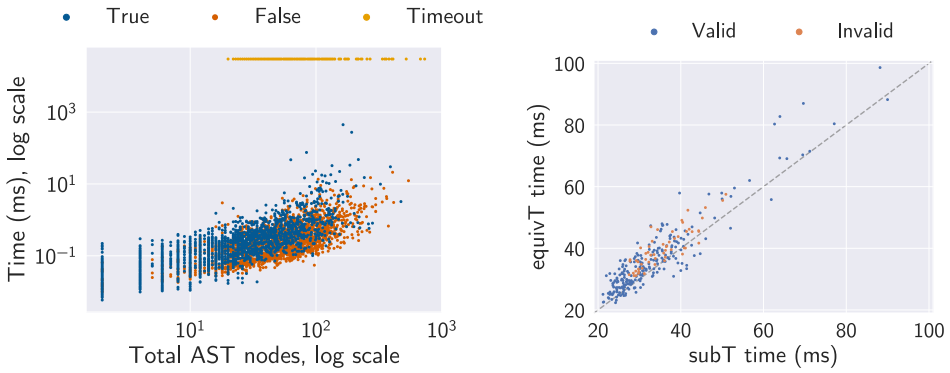
We have implemented our subtyping algorithm in Haskell and integrated it in the freely available compiler for FreeST, a statically typed functional programming language featuring message-passing channels governed by context-free session types [2, 3, 6]. The FreeST compiler features a running implementation of the type equivalence algorithm of Almeida et al. [4]. With our contributions, FreeST effectively gains support for subtyping at little to no cost in performance. In this section we present an empirical study to support this claim.

We employed three test suites to evaluate the performance of our algorithm: a suite of handwritten pairs of types, a suite of randomly generated pairs of types, and a suite of handwritten FreeST programs. We focus on the last two, since they allow a more robust and realistic analysis. All data was collected on a machine featuring an Intel Core i5-6300U at 2.4GHz with 16GB of RAM.

To build our randomly generated suite we employed a type generation module, implemented using the Quickcheck library [15] and following an algorithm induced from the properties of subtyping, much like the one induced by Almeida et al. [4] from the properties of bisimilarity. It includes generators for valid and invalid subtyping pairs. We conducted our evaluation by taking the running time of the algorithm on 2000 valid pairs and 2000 invalid pairs, ranging from 2 to 730 total AST nodes, with a timeout of 30s (ensuring it terminates with either **True**, **False** or **Unknown**). The results are plotted in Figure 5a. Despite the incompleteness of the algorithm, we encountered no false negatives, but obtained 188 timeouts. We found, as expected, that the running time increases considerably with the number of nodes. When a result was produced, valid pairs took generally longer.

Randomly generated types allow for a robust analysis, but they typically do not reflect the types encountered by a subtyping algorithm in its most obvious practical application, a compiler. For this reason, we turn our attention to our suite of FreeST programs, comprised of 286 valid and invalid programs collected throughout the development of the FreeST language. Programs range from small examples demonstrating particular features of the language to concurrent applications simulating, for example, an FTP server.

We began by integrating the algorithm in the FreeST compiler, placing next to every call to the original algorithm [4] (henceforth equivT) a call to subT on the same pairs of types. We then ran each program in our suite 10 times, collecting and averaging the accumulated running time of both algorithms on the same pairs of types. We then took the difference between the average accumulated running times of subT and equivT, obtaining an average difference of -3.85ms, with a standard deviation of 7.08ms, a minimum difference of -71.29ms and a maximum difference of 8.03ms (subT performed faster, on average). Figure 5b illustrates this comparison by plotting against each other the accumulated running times (for clarity, those in the 20-100ms range) of both algorithms during the typechecking phase of each.



(a) Performance on valid and invalid subtyping pairs

(b) Performance comparison against the original equivalence algorithm

Figure 5 Performance evaluation and comparison.

The data collected in this evaluation suggests that replacing the original equivalence algorithm [4] with the subtyping algorithm in the FreeST typechecker generally does not incur an overhead, while providing additional expressive power for programmers.

6 Related work

Session types emerged as a formalism to express communication protocols and statically verify their implementations [31, 32]. Initial formulations allowed only pairwise, tail-recursive protocols, earning such types the "binary" and "regular" epithets. Since then, considerable efforts have been made to extend the theory of session types beyond the binary and regular realms: multiparty session types allow sessions with multiple participants [33], while context-free session types [51] and nested session types [18] allow non-regular communication patterns. Our work is centered on context-free session types, which have seen considerable development since their introduction, most notably their integration in System F [2, 47], an higher-order formulation [16], as well as proposals for kind and type inference [5, 43].

Subtyping is a standard feature of many type systems, and the literature on the topic is vast [8, 10, 13, 14, 17, 19, 37]. Its conventional interpretation, based on the notion of substitutability, originates from the work of Liskov [39]. Multiple approaches to subtyping for regular session types have been proposed, and they can be classified according to the objects they consider substitutable: channels *versus* processes (the difference being most notable in the variance of type constructors). The earliest approach, subscribing to the substitutability of channels, is that of Gay and Hole [25]. It is also the one we follow. A later formulation, proposed by Carbone et al. [12], subscribes to the substitutability of processes. A survey of both interpretations is given by Gay [24]. The interaction between subtyping and polymorphism for regular session types, in the form of bounded quantification, has been investigated by Gay [23]. Horne and Padovani study subtyping under the linear logic interpretation of regular session types [34], showing that it preserves termination of processes.

Subtyping for session types has spread beyond the regular realm. Das et al. [18] introduce subtyping for nested session types, show the problem to be undecidable and present a sound but incomplete algorithm. In the context-free setting, the first and, to the best of our knowledge, only formulation before our work is that of Padovani [43]. It proposes a simulation-based subtyping relation, proves the undecidability of the subtyping problem and provides a sound but incomplete algorithm. This undecidability proof also applies to our

system, as it possesses all the required elements: width-subtyping on choices, sequential composition and recursion. The subtyping relation proposed by Padovani contemplates neither input/output subtyping nor functional subtyping. Furthermore, its implementation relies on the subtyping features of OCaml, the implementation language. In contrast, we propose a more expressive relation, featuring input/output subtyping, as well as functional subtyping. Furthermore, we provide an also sound algorithm that is independent of the implementation language.

Our subtyping relation is based on a novel form of observational preorder, $\mathcal{XYZW}$-simulation. There is, as far as we know, no analogue in the literature. It is a generalization of $\mathcal{XY}$-simulation, introduced by Aarts and Vaandrager in the context of learning automata [1] but already known, under slightly different forms, as modal refinement [38], alternating simulation [7] and covariant-contravariant simulation [20]. The contravariance on the derivatives introduced by $\mathcal{XYZW}$-simulation is also prefigured in contrasimulation [48, 52], but the former uses strong transitions whereas the latter uses weak ones. There is a vast literature on other observational relations, to which Sangiorgi's book provides an overview [48].

Our algorithm decides the $\mathcal{XYZW}$-similarity of simple grammars [36]. It is an adaptation of the bisimilarity algorithm for simple grammars of Almeida et al. [4]. To our knowledge, these are the only running algorithms of their sort. Henry and Sénizergues [29] proposed an algorithm to decide the language equivalence problem on deterministic pushdown automata. On the related topic of basic process algebra (BPA), BPA processes have been shown to be equivalent to grammars in GNF [9], of which simple grammars are a particular case. This makes results and algorithms for BPA processes applicable to grammars in GNF, and *vice-versa*. A bisimilarity algorithm for general BPA processes, of doubly-exponential complexity, has been proposed by Burkart et al. [11], while an analogous polynomial-time algorithm for the special case of normed BPA processes has been proposed by Hirschfield et al. [30].

7 Conclusion and future work

We have proposed an intuitive notion of subtyping for context-free session types, based on a novel form of observational preorder, $\mathcal{XYZW}$-simulation. This preorder inverts the direction of the simulation in the derivatives covered by its $\mathcal{W}$ and $\mathcal{Z}$ parameters, allowing it to handle co/contravariant features of input/output types. We take advantage of the fact that $\mathcal{XYZW}$-simulation generalizes bisimulation to derive a sound subtyping algorithm from an existing type equivalence algorithm.

Despite its unavoidable incompleteness, stemming from the undecidability of our notion of subtyping, our algorithm has not yielded any false negatives. Thus, we conjecture that is partially correct: it may not halt, but, when it does, the answer is correct. We cannot, however, back this claim without a careful analysis of completeness and termination, which we leave for future work. We believe such an analysis will advance the understanding of the subtyping problem by clarifying the practical reasons for its undecidability.

As shown by Thiemann and Vasconcelos [51], support for polymorphism and polymorphic recursion is paramount in practical applications of context-free session types. Exploring the interaction between polymorphism and subtyping in the context-free setting, possibly in the form of *bounded quantification*, is therefore another avenue for future work.

References

1 Fides Aarts and Frits W. Vaandrager. Learning I/O automata. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010. doi:10.1007/978-3-642-15375-4_6.

2 Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. Polymorphic lambda calculus with context-free session types. *Inf. Comput.*, 289(Part):104948, 2022. doi:10.1016/j.ic.2022.104948.

3 Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. FreeST: Context-free session types in a functional language. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 12–23, 2019. doi:10.4204/EPTCS.291.2.

4 Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Deciding the bisimilarity of context-free session types. In *TACAS@ 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2020. doi:10.1007/978-3-030-45237-7_3.

5 Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Kind inference for the FreeST programming language. *CoRR*, abs/2304.06396, 2023. doi:10.48550/arXiv.2304.06396.

6 Bernardo Almeida, Andreia Mordido, and Vasco Thudichum Vasconcelos. FreeST, a concurrent programming language with context-free session types, 2019. URL: https://freest-lang.github.io/.

7 Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998. doi:10.1007/BFb0055622.

8 Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. doi:10.1145/155183.155231.

9 Jos C. M. Baeten, Jan A. Bergstra, and Jan Willem Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *J. ACM*, 40(3):653–682, 1993. doi:10.1145/174130.174141.

10 Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, 1998. doi:10.3233/FI-1998-33401.

11 Olaf Burkart, Didier Caucal, and Bernhard Steffen. An elementary bisimulation decision procedure for arbitrary context-free processes. In Jirí Wiedermann and Petr Hájek, editors, *Mathematical Foundations of Computer Science 1995, 20th International Symposium, MFCS'95, Prague, Czech Republic, August 28 - September 1, 1995, Proceedings*, volume 969 of *Lecture Notes in Computer Science*, pages 423–433. Springer, 1995. doi:10.1007/3-540-60246-1_148.

12 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007. doi:10.1007/978-3-540-71316-6_2.

13 Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, pages 198–199. ACM, 2005. doi:10.1145/1069774.1069793.

14 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 5–17, 2014. doi:10.1145/2535838.2535840.

15 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000. doi:10.1145/351240.351266.

16 Diana Costa, Andreia Mordido, Diogo Poças, and Vasco T. Vasconcelos. Higher-order context-free session types in system F. In Marco Carbone and Rumyana Neykova, editors, *Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022, Munich, Germany, 3rd April 2022*, volume 356 of *EPTCS*, pages 24–35, 2022. doi:10.4204/EPTCS.356.3.

17 Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In *10th International Conference on Mathematics of Program Construction (MPC 2010)*, pages 100–118, Québec City, Canada, June 2010. Springer LNCS 6120. doi:10.1007/978-3-642-13321-3_8.

18 Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Nested session types. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 178–206. Springer, 2021. doi:10.1007/978-3-030-72019-3_7.

19 Stephen Dolan. *Algebraic Subtyping: Distinguished Dissertation 2017*. BCS, Swindon, GBR, 2017. URL: https://www.cs.tufts.edu/~nr/cs257/archive/stephen-dolan/thesis.pdf.

20 Ignacio Fábregas, David de Frutos-Escrig, and Miguel Palomino. Non-strongly stable orders also define interesting simulation relations. In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings*, volume 5728 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2009. doi:10.1007/978-3-642-03741-2_16.

21 Emily P. Friedman. The inclusion problem for simple languages. *Theor. Comput. Sci.*, 1(4):297–316, 1976. doi:10.1016/0304-3975(76)90074-8.

22 Simon Gay. Subtyping between standard and linear function types. Technical report, University of Glasgow, 2006.

23 Simon J. Gay. Bounded polymorphism in session types. *Math. Struct. Comput. Sci.*, 18(5):895–930, 2008. doi:10.1017/S0960129508006944.

24 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.

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

26 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

27 Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965. doi:10.1145/321250.321254.

28 Jan Friso Groote and Hans Hüttel. Undecidable equivalences for basic process algebra. *Inf. Comput.*, 115(2):354–371, 1994. doi:10.1006/inco.1994.1101.

29 Patrick Henry and Géraud Sénizergues. Lalblc a program testing the equivalence of dpda's. In Stavros Konstantinidis, editor, *Implementation and Application of Automata*, pages 169–180, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

30 Yoram Hirshfeld, Mark Jerrum, and Faron Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theor. Comput. Sci.*, 158(1&2):143–159, 1996. doi:10.1016/0304-3975(95)00064-X.

31 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

32 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.

33 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

34 Ross Horne and Luca Padovani. A logical account of subtyping for session types. *CoRR*, abs/2304.06398, 2023. doi:10.48550/arXiv.2304.06398.

35 Petr Jancar and Faron Moller. Techniques for decidability and undecidability of bisimilarity. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 1999. doi:10.1007/3-540-48320-9_5.

36 A. J. Korenjak and John E. Hopcroft. Simple deterministic languages. In *7th Annual Symposium on Switching and Automata Theory, Berkeley, California, USA, October 23-25, 1966*, pages 36–46. IEEE Computer Society, 1966. doi:10.1109/SWAT.1966.22.

37 Zeeshan Lakhani, Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Polarized subtyping. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings*, pages 431–461. Springer International Publishing Cham, 2022.

38 Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 203–210. IEEE Computer Society, 1988. doi:10.1109/LICS.1988.5119.

39 Barbara Liskov. Keynote address - data abstraction and hierarchy. In Leigh R. Power and Zvi Weiss, editors, *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 1987 Addendum, Orlando, Florida, USA, October 4-8, 1987*, pages 17–34. ACM, 1987. doi:10.1145/62138.62141.

40 Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In Andrew Kennedy and Nick Benton, editors, *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*, pages 77–88. ACM, 2010. doi:10.1145/1708016.1708027.

41 Robin Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*, pages 481–489. William Kaufmann, 1971. URL: http://ijcai.org/Proceedings/71/Papers/044.pdf.

42 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.

43 Luca Padovani. Context-free session type inference. *ACM Trans. Program. Lang. Syst.*, 41(2):9:1–9:37, 2019. doi:10.1145/3229062.

44 David Michael Ritchie Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981. doi:10.1007/BFb0017309.

45 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

46 Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Math. Struct. Comput. Sci.*, 6(5):409–453, 1996. doi:10.1017/s096012950007002x.

47 Diogo Poças, Diana Costa, Andreia Mordido, and Vasco T. Vasconcelos. System F^{μ}_{ω} with context-free session types. In Thomas Wies, editor, *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, volume 13990 of *Lecture Notes in Computer Science*, pages 392–420. Springer, 2023. doi:10.1007/978-3-031-30044-8_15.

48 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011. doi:10.1017/CBO9780511777110.

49 Gil Silva, Andreia Mordido, and Vasco T. Vasconcelos. Subtyping context-free session types. *CoRR*, abs/2307.05661, 2023. doi:10.48550/arXiv.2307.05661.

50 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994. doi:10.1007/3-540-58184-7_118.

51 Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 462–475. ACM, 2016. doi:10.1145/2951913.2951926.

52 Rob J. van Glabbeek. The linear time - branching time spectrum II. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993. doi:10.1007/3-540-57208-2_6.