Higher-order Context-free Session Types in System F

Diana Costa   Andreia Mordido   Diogo Poças   Vasco T. Vasconcelos

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

`dfdcosta,afmordido,dmpocas,vmvasconcelos@ciencias.ulisboa.pt`

We present an extension of System F with higher-order context-free session types. The mixture of functional types with session types has proven to be a challenge for type equivalence formalization: whereas functional type equivalence is often rule-based, session type equivalence usually follows a semantic approach based on bisimulations. We propose a unifying approach that handles the equivalence of functional and session types together. We present three notions of type equivalence: a syntactic rule-based version, a semantic bisimulation-based version, and an algorithmic version by reduction to the problem of bisimulation of simple grammars. We prove that the three notions coincide and derive a decidability result for the type equivalence problem of higher-order context-free session types.

1 Introduction

Session types describe the behaviour of structured communication [22, 23, 34]. The behaviour of process `sendInt` can be expressed by the session type $\mathsf{int} \to \mathsf{!int}.b \to b$ asserting that the process is given a value of type int and a channel of type $\mathsf{!int}.b$ and returns a channel of type b. Session types also provide primitives for offering and selecting choices as well as for unbounded behaviour via recursion. A client willing to send a list of integers on a channel can be governed by type $\mathsf{IntList} \doteq \oplus\{\mathsf{Cons}\colon \mathsf{!int.IntList}, \mathsf{Nil}\colon \mathsf{end}\}$ stating that the client can either choose Cons or Nil. In the former case the client must subsequently send an integer value and go back to the choice; in the latter case the protocol is terminated as identified by type end.

Traditional session types have proven particularly useful in the specification of protocols of different natures, provided that they can be characterized by regular languages. Traditional session types are restricted to tail recursion—this specificity is not just a feature, it is rather a limitation: there are numerous protocols whose traces cannot be characterized by regular languages. Context-free session types liberate session types from tail recursion. In the context-free world, clients can send integer trees on channels in a type-safe way, without requiring the exchange of additional channels: $\mathsf{IntTree} \doteq \oplus\{\mathsf{Node}\colon \mathsf{IntTree};\mathsf{!int};\mathsf{IntTree}, \mathsf{Leaf}\colon \mathsf{skip}\}$. Context-free session types provide a sequential composition operator $;$ and the corresponding neutral element, skip. Governed by type $\mathsf{IntTree}$, the client is now able to select Node, send the left subtree, followed by an integer value, followed by the right subtree, as witnessed by the double recursion on type identifier $\mathsf{IntTree}$. The increase in the expressivity of types comes at a price: checking type equivalence becomes a challenge. Thiemann and Vasconcelos proved that type equivalence is decidable for context-free session types [35], but a practical type equivalence algorithm was only provided a few years later by Almeida et al. [4].

All proposals in the literature consider first-order context-free session types: decidability of type equivalence is only guaranteed when basic types (or any other types that can be syntactically compared for equality) are exchanged in messages. This paper promotes context-free session types to the higher-order setting. In this new setting we can define trees with values of non-basic types, such as the type of

a channel on which a binary tree of input-int channels can be sent:

$$\mathsf{InputTree} \doteq \oplus\{\mathsf{Node}\colon \mathsf{InputTree}; !(?\mathsf{int}); \mathsf{InputTree}, \mathsf{Leaf}\colon \mathsf{skip}\}.$$

Higher-order context-free session types can be endowed with impredicative polymorphism. However, some care must be exercised. Allowing polymorphism over arbitrary types may raise complications: should one consider $\forall\alpha.(\alpha; !\mathsf{int})$ a bona fide type? It really depends on what we replace α with: if skip, then we get a genuine type skip; !int; if unit, then we get a bogus type unit; !int. In order to distinguish functional types from session types in the presence of polymorphic types we introduce kinds: T for functional and s for session types, collectively know as κ. The universal type is then annotated with the kind of the bound variable, $\forall\alpha\colon \kappa.T$.

Nominal bound variables cause problems when checking type equivalence. As such, we elide them and use De Bruijn indices [14] to refer to polymorphic variables: the above type is now written $\forall^{\mathsf{s}}0; !\mathsf{int}$, where 0 denotes a type variable bound by the first enclosing $\forall$. Regardless of the nature of the bound variable, should the polymorphic type itself be a session or a functional type? The answer to this question dictates how one composes the type to form larger types. Currently we allow functional polymorphism only.

In order to check the equivalence of polymorphic context-free types we reduce the problem of checking type equivalence into that of checking the bisimilarity of simple grammars, along the lines of Almeida et al. [4]. We have implemented the procedure for checking type equivalence for the monomorphic fragment of the language of this paper in a branch of the FreeST compiler [2]. For example, the simple grammar associated with type InputTree is as follows.

$$X \to \oplus\mathsf{Node}\, X\, X_1\, X \qquad X \to \oplus\mathsf{Leaf} \qquad X_1 \to !_d\, X_2 \bot \qquad X_1 \to !_c$$
$$X_2 \to ?_d\, X_3 \bot \qquad X_2 \to ?_c \qquad X_3 \to \mathsf{int}$$

This work explores the type equivalence problem for higher-order context-free session types. The main contributions are:

- A new formulation of context-free session types that allows a clean integration of session types and functional types, so that session types may exchange both session types and conventional functional types (such as functions or records);

- A syntactic and a semantic definition of equivalence for higher-order session types—a rule-based approach and a labelled transition system—which we prove to coincide;

- A type equivalence algorithm by reduction to the bisimilarity of simple grammars, as well as results of termination, soundness, completeness and decidability of type equivalence.

2 Polymorphic higher-order context-free session types

In this section we introduce an extension of System F [18, 31] with higher-order context-free session types. We rely on non-negative numerals—denoted m and n—to describe polymorphic variables; a set $\mathbb{L}$ of labels—denoted by k and ℓ—to specify labelled choices, records and variant types; and type identifiers—denoted X and Y—to provide for recursive types. A kinding system distinguishes session types (denoted by kind s) from functional types (denoted by T). We use symbol κ to denote either s or T. Kinds for bound variables are kept in a kinding context Δ containing bindings of the form $n\colon \kappa$. The type formation rules are presented in Fig. 1.

Polarity, view, records and quantifiers

$$\sharp ::= \ ? \ | \ ! \qquad\qquad \odot ::= \ \& \ | \ \oplus \qquad\qquad (\!|\cdot|\!) ::= \ \{\cdot\} \ | \ \langle\cdot\rangle \qquad\qquad \exists\forall ::= \ \forall \ | \ \exists$$

Is-terminated (*inductive*) $\boxed{T\checkmark}$ Kind and kinding environment

$$\kappa ::= \ \text{S} \ | \ \text{T}$$

$$\Delta ::= \ \varepsilon \ | \ \Delta, n\colon \kappa$$

$$
\begin{array}{cc}
\checkmark\text{-Skip} &
\dfrac{\checkmark\text{-Seq}}{\dfrac{T\checkmark \quad U\checkmark}{T;U\checkmark}} \qquad
\dfrac{\checkmark\text{-Id}}{\dfrac{X \doteq T \quad T\checkmark}{X\checkmark}}
\end{array}
$$

$$\text{skip}\checkmark$$

$$\Delta^{+1} \ = \ \{n+1\colon \kappa \mid n\colon \kappa \in \Delta\}$$

Contractivity (*inductive*) $\boxed{T \ \text{contr}}$

$$
\begin{array}{ccccc}
\text{C-Axiom} & & & & \\
\dfrac{T = \text{unit}, T \to U, (\!|\ell\colon T_\ell|\!)_{\ell\in L}}{\text{skip}, \sharp T, \odot\{\ell\colon T_\ell\}_{\ell\in L}, n} &
\dfrac{\text{C-Quant}}{T \ \text{contr}} &
\dfrac{\text{C-Seq}1}{T\checkmark \quad U \ \text{contr}} &
\dfrac{\text{C-Seq}2}{T\ \not\checkmark \quad T \ \text{contr}} &
\dfrac{\text{C-Id}}{X \doteq T \quad T \ \text{contr}} \\[2mm]
T \ \text{contr} & \exists\forall^\kappa T \ \text{contr} & T;U \ \text{contr} & T;U \ \text{contr} & X \ \text{contr}
\end{array}
$$

Type formation (*coinductive*) $\boxed{\Delta \vdash T : \kappa}$

$$
\begin{array}{cccc}
\dfrac{\text{K-Unit}}{\Delta \vdash \text{unit} : \text{T}} &
\dfrac{\text{K-Arrow}}{\dfrac{\Delta \vdash T : \kappa \quad \Delta \vdash V : \kappa'}{\Delta \vdash T \to V : \text{T}}} &
\dfrac{\text{K-Rcd}}{\dfrac{\Delta \vdash T_\ell : \kappa_\ell \quad (\forall \ell \in L)}{\Delta \vdash (\!|\ell\colon T|\!)_{\ell\in L} : \text{T}}} &
\dfrac{\text{K-Quant}}{\dfrac{\Delta^{+1}, 0\colon \kappa \vdash T : \kappa'}{\Delta \vdash \exists\forall^\kappa T : \text{T}}} \qquad
\dfrac{\text{K-Skip}}{\Delta \vdash \text{skip} : \text{S}}
\end{array}
$$

$$
\begin{array}{ccccc}
\dfrac{\text{K-Msg}}{\dfrac{\Delta \vdash T : \kappa}{\Delta \vdash \sharp T : \text{S}}} &
\dfrac{\text{K-Choice}}{\dfrac{\Delta \vdash T_\ell : \text{S} \quad (\forall \ell \in L)}{\Delta \vdash \odot\{\ell\colon T_\ell\}_{\ell\in L} : \text{S}}} &
\dfrac{\text{K-Seq}}{\dfrac{\Delta \vdash T : \text{S} \quad \Delta \vdash U : \text{S}}{\Delta \vdash T;U : \text{S}}} &
\dfrac{\text{K-Index}}{\dfrac{n\colon \kappa \in \Delta}{\Delta \vdash n : \kappa}} &
\dfrac{\text{K-Id}}{\dfrac{X \doteq T \quad T \ \text{contr} \quad \Delta \vdash T : \kappa}{\Delta \vdash X : \kappa}}
\end{array}
$$

Figure 1: Type formation

The first four rules in the figure introduce functional types: the unit type, functions $T \to V$, records $\{\ell\colon T_\ell\}_{\ell\in L}$, variants $\langle\ell\colon T_\ell\rangle_{\ell\in L}$ and polymorphic types $\forall^\kappa T$. This restriction on polymorphic types is crucial to ensure that the translation in Section 5 is well defined and indeed maps types to simple grammars. De Bruijn indices (starting at 0) allow checking polymorphic types against polymorphic types without needing to worry about the concrete names of variables. To ensure the correct formation of types, quantifiers are annotated with the kind of the bound variable—denoted by the superscript κ. With this notation, instead of writing $\forall\alpha\colon \text{T}. \alpha \to \forall\beta\colon \text{S}. !\alpha;\beta \to \beta$ for the send primitive, we write $\forall^\text{T}0 \to \forall^\text{S}!1;0 \to 0$. When crossing a quantifier, the numerals in the kinding context Δ are incremented by 1—denoted by the superscript $+1$ in rule K-Quant.

The next four rules in Fig. 1 introduce session types: the skip type, output of arbitrary types $!T$, input of arbitrary types $?T$, internal choice $\oplus\{\ell\colon T_\ell\}_{\ell\in L}$, external choice $\&\{\ell\colon T_\ell\}_{\ell\in L}$ and the sequential composition $T;U$. The last two rules in the figure introduce numerals n as polymorphic type variables and type identifiers X, defined by equations of the form $X \doteq T$.

A signature Σ is a finite collection of equations $X \doteq T$ where no type identifier X occurs twice at the left of an equation. Whenever a signature Σ is clear from context, we write $X \doteq T$ to mean an entry in Σ. The right-hand sides of equations must be *contractive*. Contractivity ensures that a type eventually

rewrites to a type constructor, eschewing non-types such as X with equations $X \doteq Y$ and $Y \doteq X$. A more elaborate example of a (non-contractive, hence) non-type is X with $X \doteq \mathsf{skip};Y$ and $Y \doteq X;Y$.

The notion of contractivity relies on the *is-terminated* predicate, which is true on types comprising only skip, sequential composition and type identifiers. Terminated types have a simple characterisation, which justifies the inclusion of predicate $T \not\checkmark$ in rule C-SEQ2 [4]. The is-terminated and contractivity predicates are inductively defined, whereas type formation is coinductive.

We can easily show that kinds are unique: any syntactic object has at most one kind, in which case we call the object a type. We say that T is a type when there are Δ and κ such that $\Delta \vdash T : \kappa$.

3 Syntactic type equivalence

The rules for type equivalence are shown in Fig. 2. The novelty lies in the rules for sequential composition. Intuitively, sequential composition has a monoidal structure—$(T;U);V \simeq T;(U;V)$—with skip being the (left and right) neutral element—$\mathsf{skip};T \simeq T;\mathsf{skip} \simeq T$. In addition, sequential composition must distribute with choice—$\oplus\{\ell\colon T_\ell\}_{\ell \in L};U \simeq \oplus\{\ell\colon T_\ell;U\}_{\ell \in L}$. The first eight rules, from E-UNIT to E-INDEX, are the congruence rules for all type constructors (i.e., without type identifiers and sequential composition). Rules E-IDL and E-IDR interpret recursive types equi-recursively. The rules in the last three lines are the rules for sequential composition. For each session type constructor T one finds a left rule (of the form $T;U \simeq V$) and a right rule ($V \simeq T;U$). Since sequential composition does not distribute with message passing or indices, we require an additional rule for these constructors (rules E-MSGSEQ2 and E-INDEXSEQ2). As we are using a coinductive proof scheme, we have rules that 'move' the sequential composition operator 'down' the syntax (or, to put it in another way, that 'move' type constructors that actually produce something 'up' the syntax). This is why for types of the form $T;U$ we look at the structure of T to decide which rule to apply next.

Theorem 1 (Agreement for type equivalence). *If $\Delta \vdash T : \kappa$, $\Delta \vdash U : \kappa'$ and $T \simeq U$, then $\kappa = \kappa'$.*

Type equivalence is unkinded (yet we call it type equivalence). There are objects in the equivalence relation that are not types. Object skip;unit is equivalent to unit yet it is not a type, that is, there are no Δ and κ such that $\Delta \vdash \mathsf{skip};\mathsf{unit} : \kappa$. This precludes a stronger agreement result, namely, $T \simeq U$ implies $\Delta \vdash T : \kappa$.

Theorem 2 (Equivalence relation). *$\simeq$ is an equivalence relation on types.*

4 Semantic type equivalence

Following Gay and Hole [16], we build on a type bisimulation to provide a semantic definition for type equivalence. For this purpose, we extend the original labelled transition system for context-free session types [4, 35] and introduce labelled transitions for functional and higher-order types. The definition of the labelled transition system (LTS) is in Fig. 3.

We start with functional types. Type unit transitions to skip via label unit. Function types induce two transitions: one via label $\rightarrow_d$ to the domain of the function, the other via $\rightarrow_r$ to the range. Records and variants step to each component k via labels $\{\}_k$ and $\langle\rangle_k$, respectively. Polymorphic types transition via the respective label, $\forall^\kappa$ or $\exists^\kappa$, to its body.

For session types, choices follow the original proposal [35] and step via $\oplus_k$ and $\&_k$ to the continuation type, for each labelled choice k. However, message exchanges for higher-order types now feature two distinct transitions: one to the type exchanged in the message (via label $!_d$, d for data) and the other

Type equivalence (*coinductive*) $\boxed{T \simeq T}$

$$\text{E-UNIT} \quad \frac{\text{E-ARROW}}{\text{unit} \simeq \text{unit}} \quad \frac{T \simeq U \quad V \simeq W}{T \to V \simeq U \to W} \quad \frac{\text{E-RCD}}{(\!(\ell\colon T_\ell)\!)_{\ell \in L} \simeq (\!(\ell\colon U_\ell)\!)_{\ell \in L}} \quad \frac{\text{E-QUANT}}{\exists \forall^\kappa T \simeq \exists \forall^\kappa U} \quad \frac{\text{E-SKIP}}{\text{skip} \simeq \text{skip}} \quad \frac{\text{E-MSG}}{\sharp T \simeq \sharp U}$$

$$\frac{\text{E-CHOICE}}{\odot\{\ell\colon T_\ell\}_{\ell \in L} \simeq \odot\{\ell\colon U_\ell\}_{\ell \in L}} \quad \frac{\text{E-INDEX}}{n \simeq n} \quad \frac{\text{E-IDL}}{X \simeq U} \quad \frac{X \doteq T \quad T \text{ contr} \quad T \simeq U}{X \simeq U} \quad \frac{\text{E-IDR}}{T \simeq X}$$

$$\frac{\text{E-SKIPSEQL}}{\text{skip};T \simeq U} \quad \frac{\text{E-SKIPSEQR}}{T \simeq \text{skip};U} \quad \frac{\text{E-MSGSEQ1L}}{\sharp T;V \simeq \sharp U} \quad \frac{\text{E-MSGSEQ1R}}{\sharp T \simeq \sharp U;V} \quad \frac{\text{E-MSGSEQ2}}{\sharp T;V \simeq \sharp U;W} \quad \frac{\text{E-CHOICESEQL}}{\odot\{\ell\colon T_\ell\}_{\ell \in L};U \simeq V}$$

$$\frac{\text{E-CHOICESEQR}}{U \simeq \odot\{\ell\colon T_\ell\}_{\ell \in L};V} \quad \frac{\text{E-SEQSEQL}}{(T;U);V \simeq W} \quad \frac{\text{E-SEQSEQR}}{T \simeq (U;V);W} \quad \frac{\text{E-INDEXSEQ1L}}{n;T \simeq n} \quad \frac{\text{E-INDEXSEQ1R}}{n \simeq n;T}$$

$$\frac{\text{E-INDEXSEQ2}}{n;T \simeq n;U} \quad \frac{\text{E-IDSEQL}}{X;V \simeq U} \quad \frac{X \doteq U \quad U \text{ contr} \quad T \simeq U;V}{T \simeq X;V}$$

Figure 2: Type equivalence

to the continuation type (via label $!_c$, c for continuation). Type skip does not exhibit any transition. Indices n transition by label n to type skip (rule L-INDEX) and type identifiers inherit the transitions from the associated type (rule L-ID). Finally, sequential composition $T;U$ distinguishes cases for all type constructors in T (rules L-SKIPSEQ to L-IDSEQ).

The labelled transition system does not preserve kinding. There are types in transition relation whose (only) kinds do not match. One example is unit of kind T that transitions to skip of kind S.

A bisimulation is defined in the usual way from the labelled transition system [32]. We say that a type relation $\mathscr{R}$ is a *bisimulation* if for all $(T,U) \in \mathscr{R}$ and for all a we have:

1. for each T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ and $(T',U') \in \mathscr{R}$, and

2. for each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ and $(T',U') \in \mathscr{R}$.

We say that two types are bisimilar, $T \sim U$, if there is a bisimulation $\mathscr{R}$ such that $(T,U) \in \mathscr{R}$.

We can easily check that the type for a function send that first receives the type of the message, then sends the value and only afterwards receives the type for the continuation, $\forall^\mathsf{T} 0 \to \forall^\mathsf{S} !1;0 \to 0$, is not equivalent to the type of a function send' that starts by receiving the type of value to be exchanged and the type of the continuation channel, $\forall^\mathsf{T}\forall^\mathsf{S} 1 \to !1;0 \to 0$. We have that $\forall^\mathsf{T} 0 \to \forall^\mathsf{S} !1;0 \to 0 \not\sim \forall^\mathsf{T}\forall^\mathsf{S} 1 \to !1;0 \to 0$ because, even if both types exhibit a transition by label $\forall^\mathsf{T}$, only the first type then exhibits a transition by label 0.

We can easily check that type bisimulation is deterministic (hence finitely branching) and image finite. It features infinite transition sequences, as well as transition sequences that visit infinitely many different states [35].

Transition labels

$$a ::= \text{unit} \mid \to_d \mid \to_r \mid (\![\cdot]\!)_\ell \mid \exists \forall^\kappa \mid \sharp_d \mid \sharp_c \mid \odot_\ell \mid n$$

Labelled transition system (*inductive*)
$$\boxed{T \xrightarrow{a} U}$$

L-UNIT
$$\text{unit} \xrightarrow{\text{unit}} \text{skip}$$

L-ARROW1
$$T \to U \xrightarrow{\to_d} T$$

L-ARROW2
$$T \to U \xrightarrow{\to_r} U$$

L-RCD
$$\frac{k \in L}{(\![\ell : T]\!)_{\ell \in L} \xrightarrow{(\![]\!)_k} T_k}$$

L-QUANT
$$\exists \forall^\kappa T \xrightarrow{\exists \forall^\kappa} T$$

L-MSG1
$$\sharp T \xrightarrow{\sharp_d} T$$

L-MSG2
$$\sharp T \xrightarrow{\sharp_c} \text{skip}$$

L-CHOICE
$$\frac{k \in L}{\odot \{\ell : T_\ell\}_{\ell \in L} \xrightarrow{\odot_k} T_k}$$

L-INDEX
$$n \xrightarrow{n} \text{skip}$$

L-ID
$$\frac{X \doteq T \quad T \xrightarrow{a} U}{X \xrightarrow{a} U}$$

L-SKIPSEQ
$$\frac{T \xrightarrow{a} U}{\text{skip}; T \xrightarrow{a} U}$$

L-MSGSEQ1
$$\sharp T; U \xrightarrow{\sharp_d} T$$

L-MSGSEQ2
$$\sharp T; U \xrightarrow{\sharp_c} U$$

L-CHOICESEQ
$$\frac{k \in L}{\odot \{\ell : T_\ell\}_{\ell \in L}; U \xrightarrow{\odot_k} T_k; U}$$

L-SEQSEQ
$$\frac{T; (U; V) \xrightarrow{a} W}{(T; U); V \xrightarrow{a} W}$$

L-INDEXSEQ
$$n; U \xrightarrow{n} U$$

L-IDSEQ
$$\frac{X \doteq T \quad T; U \xrightarrow{a} V}{X; U \xrightarrow{a} V}$$

Figure 3: Labelled transition system

Theorem 3 (Soundness and completeness). *Let T, U be types. Then $T \simeq U$ iff $T \sim U$.*

The proviso that T and U are types is important. Objects unit; skip and skip are bisimilar (no transition applies to either), yet they cannot be shown equivalent.

5 Bisimulation for simple grammars

A grammar is given by a tuple $(\mathscr{T}, \mathscr{N}, X, \mathscr{R})$ where: $\mathscr{T}$ is a set of terminal symbols, denoted by a, b, c, $\mathscr{N}$ is a set of nonterminal symbols, denoted by X, Y, Z, nonterminal $X \in \mathscr{N}$ is the starting symbol and $\mathscr{P} \subseteq \mathscr{N} \times (\mathscr{T} \cup \mathscr{N})^*$ is a set of productions. Greek letters σ and τ denote (possibly empty) words of terminal and nonterminal symbols; greek letters γ and δ denote (possibly empty) words of nonterminal symbols only. Each production is written as $X \to \sigma$. It is well-known that every grammar can be converted into an equivalent grammar in Greibach normal form [19]. A grammar is in Greibach normal form if $\mathscr{P} \subseteq \mathscr{N} \times \mathscr{T} \times \mathscr{N}^*$, in other words, when every production is of the form $X \to a\gamma$. A grammar in Greibach normal form is said to be simple [26] if, for every nonterminal X and every terminal a, there is at most one production of the form $X \to a\gamma$.

We define a notion of bisimulation for grammars in Greibach normal form via a labelled transition system. The system comprises a set of states $\mathscr{N}^*$ corresponding to words of nonterminal symbols. For each production $X \to a\gamma$ and each word of nonterminal symbols δ, we have a labelled transition $X\delta \xrightarrow{a} \gamma\delta$. We let $\approx$ denote the bisimulation relation for grammars in Greibach normal form.

Our next step is to explain how to convert a type into a simple grammar. We do this in the two steps outlined below. For any type T, let subterms(T) denote the set of subterms of T.

Step 1. Construct a grammar Suppose we have a type T defined by means of a signature $\Sigma = \{X_i \doteq T_i\}_{i=1...m}$. For every subterm U appearing in T as well as in the equations in Σ, let X_U denote a fresh nonterminal symbol. Moreover, let $\bot$ denote a nonterminal symbol distinct from all X_U. We define

the grammar $(\mathscr{T}, \mathscr{N}, X_T, \mathscr{R})$ where $\mathscr{T}$ is the language of the transition labels in Fig. 3, $\mathscr{N}$ is the set

$$\{\perp\} \cup \{X_U : U \in \mathrm{subterms}(T)\} \cup \bigcup\nolimits_{i=1\ldots m} \{X_U : U \in \mathrm{subterms}(T_i)\}$$

and the productions in $\mathscr{R}$ for nonterminal X_U are defined according to the syntax of U as follows

Type U	Productions for X_U	Type U	Productions for X_U
unit	$X_U \to \mathrm{unit}$	$\sharp V$	$X_U \to \sharp_d X_V \perp, \quad X_U \to \sharp_c$
$V \to W$	$X_U \to \to_d X_V, \quad X_U \to \to_r X_W$	$\odot\{\ell : U_\ell\}_{\ell \in L}$	$X_U \to \odot_k X_{U_k}$, for each $k \in L$
$(\!(\ell : U_\ell)\!)_{\ell \in L}$	$X_U \to (\!(\!)\!)_k X_{U_k}$, for each $k \in L$	$V;W$	$X_U \to X_V X_W$
$\exists \forall^\kappa V$	$X_U \to \exists \forall^\kappa X_V$	n	$X_U \to n$
skip	$X_U \to \varepsilon$	X_i	$X_U \to X_{T_i}$

Notice that symbol $\perp$ has no production.

Step 2. Transform into a simple grammar   In this step we convert the grammar constructed in the previous step into Greibach normal form. We need to take care of the productions $X \to \varepsilon$, $X \to Y$, and $X \to YZ$ which are not in Greibach normal form. First, suppose we have a production $X \to \varepsilon$; by construction, this is the only such production for X. We remove every production of the form $X \to \varepsilon$ from our grammar and erase each such X from the right-hand side of every production where it appears. Next, suppose we have a production $X \to Y\gamma$ where γ is a (possibly empty) word of nonterminal symbols. We remove this production and, for each production $Y \to \sigma$, include a production $X \to \sigma\gamma$. We continue in this fashion until all productions are in Greibach normal form.

Proposition 4. *For any type T described by a signature Σ, the construction outlined in Section 5 terminates yielding a simple grammar.*

The above construction introduces a nonterminal symbol $\perp$ without productions. Intuitively, $\perp$ is used to separate the two descendants of a send/receive operation. A type $!T;U$ must have a data transition $!_d$ to T and a continuation transition $!_c$ to U. It must have two different transitions, since we want to distinguish $!T;U$ from $!(T;U)$. For example, the type $!\mathrm{skip};!\mathrm{skip}$ sends two (empty) channels in sequence, whereas $!(\mathrm{skip};!\mathrm{skip})$ sends a channel which in turns sends an empty channel. Moreover, when transitioning to the data T of a sequential composition $!T;U$, we want to make sure that we follow the grammar corresponding only to T. The following example provides some more insight. Suppose we have types T, U given by equations

$$T \doteq !V;W \qquad U \doteq !(V;V);W \qquad V \doteq \oplus\{\mathrm{go: skip}\} \qquad W \doteq \oplus\{\mathrm{go}: W\}$$

Notice that $T \not\simeq U$, as the type being sent in T offers a choice only once, whereas the type being sent in U offers that choice twice. Following the construction above, we arrive at the grammar with productions $X_T \xrightarrow{!_d} X_V \perp X_W, X_T \xrightarrow{!_c} X_W, X_U \xrightarrow{!_d} X_V X_V \perp X_W, X_U \xrightarrow{!_c} X_W, X_V \xrightarrow{\oplus\mathrm{go}} \varepsilon, X_W \xrightarrow{\oplus\mathrm{go}} X_W$. Now we can check that $X_T \not\approx X_U$, as

$$X_T \xrightarrow{!_d} X_V \perp X_W \xrightarrow{\oplus\mathrm{go}} \perp X_W \not\rightarrow \qquad \text{but} \qquad X_U \xrightarrow{!_d} X_V X_V \perp X_W \xrightarrow{\oplus\mathrm{go}} X_V \perp X_W \xrightarrow{\oplus\mathrm{go}} \perp X_W.$$

Suppose instead that we did not have the nonterminal $\perp$. In this case we would have productions $X_T \xrightarrow{!_d} X_V X_W$ and $X_U \xrightarrow{!_d} X_V X_V X_W$ instead. Because W is an infinitely repeating type, we would undesirably conclude that $X_T \approx X_U$; in particular, we would have the infinite sequences of transitions

$$X_T \xrightarrow{!_d} X_V X_W \xrightarrow{\oplus\mathrm{go}} X_W \xrightarrow{\oplus\mathrm{go}} X_W \xrightarrow{\oplus\mathrm{go}} \cdots \quad \text{and} \quad X_U \xrightarrow{!_d} X_V X_V X_W \xrightarrow{\oplus\mathrm{go}} X_V X_W \xrightarrow{\oplus\mathrm{go}} X_W \xrightarrow{\oplus\mathrm{go}} \cdots.$$

Theorem 5 (Soundness and completeness for grammars)**.** *Let T, U be types and $(\mathscr{T}_T, \mathscr{N}_T, X_T, \mathscr{R}_T)$, $(\mathscr{T}_U, \mathscr{N}_U, X_U, \mathscr{R}_U)$ the corresponding simple grammars obtained by the construction outlined in Section 5. Then $T \simeq U$ iff $X_T \approx X_U$.*

6 An algorithm to decide type equivalence

We are now in a position to describe the algorithm to decide type equivalence. Our algorithm builds on the construction outlined in Section 5, as well as on a procedure for deciding bisimulation of simple grammars. Almeida et al. [4] describe one such algorithm, which incidentally equips the FreeST programming language [2]. See Section 7 for alternative algorithms for checking the bisimilarity of simple grammars.

Input: Two types T, U built on a common signature Σ.

Output: 'YES' if $T \simeq U$, 'NO' otherwise.

Algorithm:

1. Construct the simple grammar $(\mathscr{T}_T, \mathscr{N}_T, X_T, \mathscr{R}_T)$ corresponding to T.
2. Construct the simple grammar $(\mathscr{T}_U, \mathscr{N}_U, X_U, \mathscr{R}_U)$ corresponding to U.
3. Use a decision algorithm to decide whether $X_T \approx X_U$.

Theorem 6. *The type equivalence algorithm terminates. Its computational complexity is doubly exponential.*

Theorem 7. *The type equivalence algorithm is sound and complete with respect to $\simeq$.*

Corollary 8. *The type equivalence problem is decidable.*

7 Related work

Related work is varied; we focus on that related to non-regular session types, polymorphism and bisimulation checking algorithms.

Beyond regular session types   Since its original proposal in the 90s [22, 23, 34], the theory of session types has evolved substantially. The interest in non-regular protocols was already apparent [29, 30, 33] when Thiemann and Vasconcelos proposed context-free session types as a way to specify non-regular communication protocols [35]. Context-free session types were integrated in the FreeST programming language [3] as soon as an implementation for a type equivalence algorithm was developed [4]. More recently, the language was extended to System F [1]; here we follow the same strategy and promote context-free session types to higher-order types. An alternative implementation of context-free session type equivalence was proposed by Padovani [27] by resorting to explicit code annotations, thus greatly simplifying the decision problem. Despite the interest of types characterized by context-free languages, types that live beyond regular are not limited to context-free session types; Gay et al. [17] analyse different shades of session types that go beyond regular.

Polymorphic session types   There has been a myriad of attempts to integrate polymorphic types into session types: from bounded polymorphism [15], to parametric and bounded polymorphism without recursion [11] or with recursion but without polymorphism [10]; Wadler proposed the inclusion of explicit polymorphism [36], which was then considered with parametric polymorphism but without general recursion [8], and afterwards with recursion but without nested types [20]. Finally, Das et al. proposed parametric polymorphism with nested types [12, 13]. We propose an extension of System F types with higher-order context-free session types, taking advantage of polymorphic (functional) types, which is more closely related to polymorphic (first-order) context-free session types [1].

Semantic *vs* syntatic approaches to type equivalence definition   The syntactic method is the most common approach to type equivalence [28]. The first paper on sessions and recursion uses implicit equirecursive types, so that one may classify type equivalence as a semantic notion [23]. An explicit notion of type equivalence for session types (actually of subtyping) was proposed by Gay and Hole, making use of a bisimulation [16]. The same paper also presents a syntactic, rule based definition as a basis for an algorithm. A similar construction was used in building notions of equivalence for more complex session types [12, 35]. This paper introduces both a syntactic and a semantic approach for a fairly rich language of types.

Algorithms to decide the bisimilarity of simple grammars   To the best of our knowledge, the only running algorithm for checking the bisimilarity of simple grammars is that of Almeida et al. [4]. Quite close to simple grammars, but inspired by concurrent processes, one finds the basic process algebra (BPA) [5]. BPA processes were proven equivalent to grammars in Greibach normal form by Baeten et al. [6], so that decidability results and algorithms for BPA may be readily transposed to grammars in Greibach normal form (and hence to simple grammars). Baeten et al. presented a decidability result for normed BPA [6], which was then extended to the full BPA language by Christensen et al. [9]. An improved (elementary) algorithm was proposed by Burkart et al. [7], and the complexity of this algorithm was much later shown to be doubly exponential by Jančar [24]. For BPA processes, the bisimilarity problem is known to be EXPTIME-hard [25]; however, this does not exclude the possibility of a polynomial time algorithm for our model, since simple grammars are less expressive than grammars in Greibach normal form. For a different special case of normed BPA processes, Hirshfeld et al. presented a polynomial-time algorithm for deciding bisimilarity [21].

8   Conclusion

This paper promotes context-free session types to the higher-order setting: messages can now convey channels. We propose an extension of System F with higher-order context-free session types and present three approaches for defining type equivalence: a syntactic, rule-based version, a semantic version based on bisimulations and an algorithmic version by reduction to simple grammar bisimilarity. We show that the three formulations coincide. Algorithms exist for deciding the bisimilarity of simple grammars [4, 7], from which an algorithm for deciding type equivalence can be effectively constructed.

Session types (kind S) are sometimes seen as special cases of functional types (kind T), meaning that a session type can be used in any context where a functional type is expected [1, 2, 3]. Such a notion can be captured by a subkind preorder generated by the S <: T inequality. Likewise, types such as arrows, records or variants can be tagged as linear or unrestricted (two alternative multiplicities), depending on their intended usage. We plan to investigate the incorporation of subkinding and multiplicities in the present system.

The polymorphic types we manipulate are functional, that is, type $\forall^\kappa T$ is of kind T. As such, type HTree with HTree $\doteq \oplus\{\text{Node}: \text{HTree}; \forall^\text{T}!0; \text{HTree}, \text{Leaf}: \text{skip}\}$, streaming a binary tree of heterogeneous values, is considered ill formed. This is a crucial assumption to guarantee that our translation yields a simple grammar, as opposed to a context-dependent grammar. We plan to analyse the implications of polymorphism over session types on the decidability of type equivalence.

Another challenge for future work is the incorporation of higher-order kinds. The present kinds, S and T, are kinds of proper types. One can also consider kinds for type families. For example, type dualof is a type constructor that, when given a session type, yields the dual session type.

References

[1] Bernardo Almeida, Andreia Mordido, Peter Thiemann & Vasco T. Vasconcelos (2021): *Polymorphic Context-free Session Types*. *CoRR* abs/2106.06658, doi:10.48550/arXiv.2106.06658.

[2] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2019): *FreeST, a Programming Language with Context-free Session Types*. http://rss.di.fc.ul.pt/tools/freest/.

[3] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2019): *FreeST: Context-free Session Types in a Functional Language*. In: *PLACES*, *EPTCS* 291, pp. 12–23, doi:10.4204/EPTCS.291.2.

[4] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2020): *Deciding the Bisimilarity of Context-Free Session Types*. In: *TACAS*, *LNCS* 12079, Springer, pp. 39–56, doi:10.1007/978-3-030-45237-7_3.

[5] Jos C. M. Baeten, Jan A. Bergstra & Jan Willem Klop (1987): *Decidability of Bisimulation Equivalence for Processes Generating Context-Free Languages*. In: *PARLE*, *LNCS* 259, Springer, pp. 94–111, doi:10.1007/3-540-17945-3_5.

[6] Jos C. M. Baeten, Jan A. Bergstra & Jan Willem Klop (1993): *Decidability of Bisimulation Equivalence for Processes Generating Context-Free Languages*. *J. ACM* 40(3), pp. 653–682, doi:10.1145/174130.174141.

[7] Olaf Burkart, Didier Caucal & Bernhard Steffen (1995): *An Elementary Bisimulation Decision Procedure for Arbitrary Context-Free Processes*. In: *MFCS*, *LNCS* 969, Springer, pp. 423–433, doi:10.1007/3-540-60246-1_148.

[8] Luís Caires, Jorge A. Pérez, Frank Pfenning & Bernardo Toninho (2013): *Behavioral Polymorphism and Parametricity in Session-Based Communication*. In: *ESOP*, *LNCS* 7792, Springer, pp. 330–349, doi:10.1007/978-3-642-37036-6_19.

[9] Søren Christensen, Hans Hüttel & Colin Stirling (1995): *Bisimulation Equivalence is Decidable for All Context-Free Processes*. *Inf. Comput.* 121(2), pp. 143–148, doi:10.1006/inco.1995.1129.

[10] Ornela Dardha (2014): *Recursive Session Types Revisited*. *EPTCS* 162, p. 27–34, doi:10.4204/eptcs.162.4.

[11] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2017): *Session types revisited*. *Inf. Comput.* 256, pp. 253–286, doi:10.1016/j.ic.2017.06.002.

[12] Ankush Das, Henry DeYoung, Andreia Mordido & Frank Pfenning (2021): *Nested Session Types*. In: *ESOP*, *LNCS* 12648, Springer, pp. 178–206, doi:10.1007/978-3-030-72019-3_7.

[13] Ankush Das, Henry DeYoung, Andreia Mordido & Frank Pfenning (2021): *Subtyping on Nested Polymorphic Session Types*. *CoRR* abs/2103.15193, doi:10.48550/arXiv.2103.15193.

[14] Nicolaas Govert De Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. In: Indagationes Mathematicae, 75, Elsevier, pp. 381–392, doi:10.1016/1385-7258(72)90034-0.

[15] Simon J. Gay (2008): *Bounded polymorphism in session types*. MSCS 18(5), pp. 895–930, doi:10.1017/S0960129508006944.

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

[17] Simon J. Gay, Diogo Poças & Vasco T. Vasconcelos (2022): *The Different Shades of Infinite Session Types*. CoRR abs/2201.08275, doi:10.48550/arXiv.2201.08275.

[18] Jean-Yves Girard (1971): *Une extension de L'interpretation de Gödel a L'analyse, et son application a L'elimination des coupures dans L'analyse et la theorie des types*. In: Studies in Logic and the Foundations of Mathematics, 63, Elsevier, pp. 63–92, doi:10.1016/S0049-237X(08)70843-7.

[19] Sheila A. Greibach (1965): *A New Normal-Form Theorem for Context-Free Phrase Structure Grammars*. J. ACM 12(1), pp. 42—-52, doi:10.1145/321250.321254.

[20] Dennis Edward Griffith (2016): *Polarized substructural session types*. Ph.D. thesis, University of Illinois at Urbana-Champaign, doi:10.2172/1562827.

[21] Yoram Hirshfeld, Mark Jerrum & Faron Moller (1996): *A Polynomial Algorithm for Deciding Bisimilarity of Normed Context-Free Processes*. Theor. Comput. Sci. 158(1&2), pp. 143–159, doi:10.1016/0304-3975(95)00064-X.

[22] Kohei Honda (1993): *Types for Dyadic Interaction*. In: CONCUR, LNCS 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[23] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: ESOP, LNCS 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[24] Petr Jančar (2012): *Bisimilarity on Basic Process Algebra is in 2-ExpTime (an explicit proof)*. Log. Methods Comput. Sci. 9(1), doi:10.2168/LMCS-9(1:10)2013.

[25] Stefan Kiefer (2013): *BPA bisimilarity is EXPTIME-hard*. Inf. Process. Lett. 113(4), pp. 101–106, doi:10.1016/j.ipl.2012.12.004.

[26] A. J. Korenjak & John E. Hopcroft (1966): *Simple Deterministic Languages*. In: SWAT, IEEE Computer Society, pp. 36–46, doi:10.1109/SWAT.1966.22.

[27] Luca Padovani (2019): *Context-Free Session Type Inference*. ACM Trans. Program. Lang. Syst. 41(2), pp. 9:1–9:37, doi:10.1145/3229062.

[28] Benjamin C. Pierce (2002): *Types and programming languages*. MIT Press.

[29] Franz Puntigam (1999): *Non-regular Process Types*. In: Euro-Par, LNCS 1685, Springer, pp. 1334–1343, doi:10.1007/3-540-48311-X_189.

[30] António Ravara & Vasco Thudichum Vasconcelos (1997): *Behavioural Types for a Calculus of Concurrent Objects*. In: Euro-Par, LNCS 1300, Springer, pp. 554–561, doi:10.1007/BFb0002782.

[31] John C. Reynolds (1974): *Towards a theory of type structure*. In: Programming Symposium, LNCS 19, Springer, pp. 408–423, doi:10.1007/3-540-06859-7_148.

[32] Davide Sangiorgi (2014): *An Introduction to Bisimulation and Coinduction*. Cambridge University Press.

[33] Mario Südholt (2005): *A Model of Components with Non-regular Protocols*. In: SC, LNCS 3628, Springer, pp. 99–113, doi:10.1007/11550679_8.

[34] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: PARLE, LNCS 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[35] Peter Thiemann & Vasco T. Vasconcelos (2016): *Context-free session types*. In: ICFP, ACM, pp. 462–475, doi:10.1145/2951913.2951926.

[36] Philip Wadler (2012): *Propositions as sessions*. In: *ICFP*, ACM, pp. 273–286, doi:10.1145/2364527. 2364568.