Higher-order Context-free Session Types in System F

Diana Costa[a], Andreia Mordido[a], Diogo Poças[a], Vasco T. Vasconcelos[a]

[a]*LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal*

Abstract

We present an extension of System F with higher-order context-free session types. The mixture of functional types and session types has proven to be a challenge for type equivalence formulation: whereas (finite) functional type equivalence is inductive and often presented as a system of rules, (infinite) session type equivalence is coinductive and usually presented as a bisimulation. We propose a unifying approach that handles the equivalence of functional and higher-order context-free session types together in the form of a system of rules to be read coinductively. Decidability of type equivalence is obtained via a reduction to bisimulation for simple grammars, for which practical algorithms are known. To bridge the gap between types and simple grammars, we introduce a language of type terms without bindings for polymorphic variables and use a notion of canonical renaming to translate types to terms.

Keywords: System F, higher-order types, context-free session types, type equivalence, bisimulation

1. Introduction

Session types describe the behaviour of structured communication [25, 26, 39]. The interface of a function `sendInt` can be expressed by type $\mathsf{int} \to !\mathsf{int}.b \to b$ asserting that the function is given a value of type int and a channel endpoint of type $!\mathsf{int}.b$ and returns a channel endpoint 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 $\mathsf{IntList}$ such that $\mathsf{IntList} \doteq \oplus\{\mathsf{Cons}: !\mathsf{int}.\mathsf{IntList}, \mathsf{Nil}: \mathsf{end}\}$, 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 [40]. In the context-free world, clients can send integer trees on channels in a type-safe way, without requiring the exchange of additional channels: $\mathsf{IntTree} \doteq \oplus\{\mathsf{Node}: \mathsf{IntTree}; !\mathsf{int}; \mathsf{IntTree}, \mathsf{Leaf}: \mathsf{skip}\}$. Context-free session types introduce 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 [40], but a practical type equivalence algorithm was only provided a few years later by Almeida et al. [4].

So far, 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

Email addresses: `dfdcosta@ciencias.ulisboa.pt` (Diana Costa), `afmordido@ciencias.ulisboa.pt` (Andreia Mordido), `dmpocas@ciencias.ulisboa.pt` (Diogo Poças), `vmvasconcelos@ciencias.ulisboa.pt` (Vasco T. Vasconcelos)

compared for equality) are exchanged in messages [1, 3, 30, 31, 40]. 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: $\mathsf{InputTree} \doteq \oplus\{\mathsf{Node}: \mathsf{InputTree}; !(?\mathsf{int}); \mathsf{InputTree}, \mathsf{Leaf}: \mathsf{skip}\}$.

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 $\mathsf{skip}; !\mathsf{int}$; if unit, then we get a bogus type $\mathsf{unit}; !\mathsf{int}$. In order to distinguish functional types from session types in the presence of polymorphic types we use 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, $\forall\alpha:\kappa.T$. 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.

Polymorphism introduces *type variables* ($\alpha, \beta, \ldots$). To combine polymorphism with recursive types, we use *parameterized type identifiers* ($X(\overline{\alpha})$, $Y(\overline{\beta})$, $\ldots$) defined by equations. For example, consider a function that first receives a type. Name it α_1. Then it receives a second type (β_1), a value of type α_1, a third type (α_2) and another value this time of type β_1, and so forth. The initial part of the infinite type can be written as

$$T \triangleq \forall\alpha_1.\forall\beta_1.\alpha_1 \to \forall\alpha_2.\beta_1 \to \forall\beta_2.\alpha_2 \to \forall\alpha_3.\beta_2 \to \cdots$$

Using equations, the type above can be written in finite format as

$$\forall\alpha.X(\alpha) \quad \text{where} \quad X(\alpha) \doteq \forall\beta.\alpha \to X(\beta) \tag{1}$$

In the equation above, the 'delay' between receiving a type and receiving a value of that type can be expressed by changing the bound variable α in $X(\alpha)$ by β in $X(\beta)$. For another example, consider the infinite type

$$U \triangleq \forall\alpha_1.\forall\beta_1.\alpha_1 \to \forall\beta_2.\alpha_1 \to \forall\beta_3.\alpha_1 \to \forall\beta_4.\alpha_1 \to \cdots$$

For a function witnessing this type, the values received are always of type α_1, even though the function keeps receiving types $\beta_1, \beta_2, \ldots$ Using equations, we can promptly identify a finite representation for U:

$$\forall\alpha.Y(\alpha) \quad \text{where} \quad Y(\alpha) \doteq \forall\beta.\alpha \to Y(\alpha) \tag{2}$$

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 language of this paper in the FreeST compiler [2]. For example, the simple grammar associated with type InputTree is as follows ($\bot$ denotes a symbol without productions).

$$X \to \oplus\mathsf{Node}\,X\,X_1\,X \qquad X \to \oplus\mathsf{Leaf} \qquad X_1 \to !_d\,X_2\bot \qquad X_1 \to !_c$$
$$X_2 \to ?_d\,X_3\bot \qquad X_2 \to ?_c \qquad X_3 \to \mathsf{int} \qquad \bot \not\to$$

Type constructors are converted into terminal symbols ($\oplus\mathsf{Node}$, $\oplus\mathsf{Leaf}$, $!_d$, $!_c$, $?_d$, $?_c$, int) through a translation that we present in section 6. This translation explores the commonalities between session types and pushdown automata, capitalizing on the decidability of bisimilarity on simple grammars.

This work explores the type equivalence problem for higher-order context-free session types. Our main contributions can be summarized as follows.

- We provide a new formulation of context-free session types allowing a clean integration of higher-order session types with functional types (including impredicative polymorphism).
- We provide a syntactic, rule-based definition for type equivalence, $T \simeq U$.
- In order to algorithmically enforce the α axiom scheme [12] (equivalence does not depend on the name of bound variables), we define a universe of terms and provide a canonical renaming operation $T \mapsto N(T)$ that converts a type to a term, explicitly renaming bound variables.

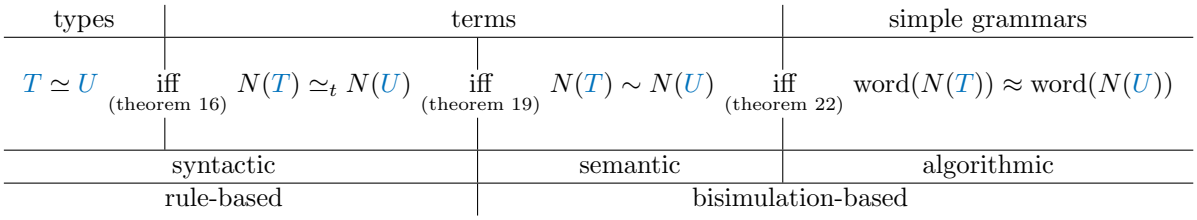


Figure 1: Comparison between the four notions of equivalence studied in the paper.

- We define for terms a syntactic (rule-based, $T \simeq_t U$) and a semantic (bisimulation-based, $T \sim U$) notion of equivalence.
- We define a procedure $T \mapsto \text{word}(T)$ that converts a term into a simple grammar, for which there are known algorithms for deciding bisimilarity.
- We show that all four notions of equivalence coincide (see fig. 1).
- We provide 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.

The outline of the rest of the paper is as follows. In section 2 we present the syntax of types, together with the rules of type formation. In section 3 we provide a syntactic, rule-based definition, of type equivalence. In section 4 we present the syntax of terms and a syntactic, rule-based definition, of term equivalence. We also provide a notion of canonical renaming, converting types to terms, which preserves and reflects syntactic equivalence. In section 5, we provide a semantic, bisimulation-based definition, of term equivalence, which we show sound and complete with respect to the syntactic formulation. In section 6 we present a translation from types to simple grammars that preserves and reflects semantic equivalence. In section 7 we put all these ingredients together to obtain a practical algorithm for deciding type equivalence. In sections 8 and 9 we briefly discuss possible extensions to our work and alternative type representations. Finally, in sections 10 and 11 we discuss related work and provide concluding remarks.

An early version of this work appeared in PLACES [11]. The main changes for the current version are the explicit use of variable names instead of De Bruijn indices together with parameterized type identifiers, a new approach for converting types to grammars based on the notions of terms and canonical renaming, and additional sections 8 and 9 discussing the obtained results, extensions and alternative approaches.

2. Polymorphic Higher-order Context-free Session Types

This section introduces an extension of System F [21, 36] with higher-order context-free session types. We rely on an infinite *ordered* set $\mathbb{A}$ of variable names—denoted by α and β—to describe polymorphic variables; a set $\mathbb{L}$ of labels—denoted by k and ℓ—to specify labelled choices, records and variant types; and a set $\mathbb{X}$ of type identifiers—denoted by X and Y—to provide for type definitions.

Pretypes. The syntax of pretypes is in fig. 2. We use *pretypes*, rather than *types*, to distinguish syntactic objects that have not been validated for good formation. We distinguish between functional pretypes—denoted by symbol T—and session pretypes—denoted by S, and use symbol κ to denote either T or S. The first pretypes in the figure are functional: unit, functions $T \to V$, records $\{\ell\colon T_\ell\}_{\ell\in L}$, variants $\langle \ell\colon T_\ell\rangle_{\ell\in L}$, universal $\forall\alpha\colon\kappa\,.\,T$ and existential $\exists\alpha\colon\kappa\,.\,T$ types. The next pretypes are sessions: skip, 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 sequential composition $T;U$. The last pretypes in the figure may denote either functional or session types: they are polymorphic variables α and identifiers $X(\overline{\alpha})$ parameterised on a sequence of type variables $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$. When the sequence of parameters is empty, we abbreviate $X()$ to X.

Identifiers are defined by equations of the form $X(\overline{\alpha}) \doteq T$. The type formation system to be introduced shortly guarantees that the free variables of T are in set $\{\overline{\alpha}\}$. A signature is a finite,

$$
\begin{array}{rll}
\sharp & ::= ? \mid ! & \text{Polarity (input, output)} \\
\odot & ::= \& \mid \oplus & \text{View (external, internal)} \\
(\!|\cdot|\!) & ::= \{\cdot\} \mid \langle\cdot\rangle & \text{Record or variant} \\
\exists\forall & ::= \forall \mid \exists & \text{Quantifier} \\
\kappa & ::= \mathrm{T} \mid \mathrm{S} & \text{Kind (functional, session)} \\
T & ::= \mathsf{unit} \mid T \to T \mid (\!|\ell\colon T_\ell|\!)_{\ell\in L} \mid \exists\forall\alpha\colon\kappa\,.\,T \mid & \text{Pretype} \\
 & \quad \mathsf{skip} \mid \sharp T \mid \odot\{\ell\colon T_\ell\}_{\ell\in L} \mid T;U \mid \alpha \mid X(\overline{\alpha}) &
\end{array}
$$

Figure 2: The syntax of pretypes.

Is-terminated (*inductive*) $\boxed{T\checkmark}$

$$
\mathsf{skip}\checkmark \quad (\checkmark\text{-Skip}) \qquad \frac{T\checkmark \quad U\checkmark}{T;U\checkmark}\ (\checkmark\text{-Seq}) \qquad \frac{X(\overline{\alpha}) \doteq T \quad T\checkmark}{X(\overline{\beta})\checkmark}\ (\checkmark\text{-Id})
$$

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

$$
\frac{T = \mathsf{unit}, T \to U, (\!|\ell\colon T_\ell|\!)_{\ell\in L}, \exists\forall\alpha\colon\kappa\,.\,T, \mathsf{skip}, \sharp T, \odot\{\ell\colon T_\ell\}_{\ell\in L}, \alpha}{T\ \mathrm{contr}}\ (\text{C-Axiom})
$$

$$
\frac{T\checkmark \quad U\ \mathrm{contr}}{T;U\ \mathrm{contr}}\ (\text{C-Seq1}) \qquad \frac{T\not\checkmark \quad T\ \mathrm{contr}}{T;U\ \mathrm{contr}}\ (\text{C-Seq2}) \qquad \frac{X(\overline{\alpha}) \doteq T \quad T\ \mathrm{contr}}{X(\overline{\beta})\ \mathrm{contr}}\ (\text{C-Id})
$$

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

$$
\Delta \vdash \mathsf{unit} : \mathrm{T}\ (\text{K-Unit}) \qquad \frac{\Delta \vdash T : \kappa \quad \Delta \vdash V : \kappa'}{\Delta \vdash T \to V : \mathrm{T}}\ (\text{K-Arrow}) \qquad \frac{\Delta \vdash T_\ell : \kappa_\ell \quad (\forall \ell \in L)}{\Delta \vdash (\!|\ell\colon T_\ell|\!)_{\ell\in L} : \mathrm{T}}\ (\text{K-Rcd}) \qquad \frac{\Delta, \alpha\colon\kappa \vdash T : \kappa'}{\Delta \vdash \exists\forall\alpha\colon\kappa\,.\,T : \mathrm{T}}\ (\text{K-Quant})
$$

$$
\Delta \vdash \mathsf{skip} : \mathrm{S}\ (\text{K-Skip}) \qquad \frac{\Delta \vdash T : \kappa}{\Delta \vdash \sharp T : \mathrm{S}}\ (\text{K-Msg}) \qquad \frac{\Delta \vdash T_\ell : \mathrm{S} \quad (\forall \ell \in L)}{\Delta \vdash \odot\{\ell\colon T_\ell\}_{\ell\in L} : \mathrm{S}}\ (\text{K-Choice}) \qquad \frac{\Delta \vdash T : \mathrm{S} \quad \Delta \vdash U : \mathrm{S}}{\Delta \vdash T;U : \mathrm{S}}\ (\text{K-Seq}) \qquad \frac{\alpha\colon\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ (\text{K-Var})
$$

$$
\frac{X(\overline{\alpha}) \doteq T \quad T\ \mathrm{contr} \quad \overline{\alpha}\colon\overline{\kappa} \vdash T : \kappa}{\Delta, \overline{\beta}\colon\overline{\kappa} \vdash X(\overline{\beta}) : \kappa}\ (\text{K-Id})
$$

Figure 3: Type formation.

possibly empty collection Σ of equations $X_1(\overline{\alpha_1}) \doteq T_1, \ldots, X_m(\overline{\alpha_m}) \doteq T_m$ where identifiers X_i are pairwise distinct.

Renaming and the variable convention. The language of pretypes includes two binders for variables: variable α occurs bound in types $\forall\alpha\colon\kappa\,.\,T$ and $\exists\alpha\colon\kappa\,.\,T$. We follow the *variable convention* whereby types that differ only in the names of the bound variables are interchangeable in all contexts [32]. Based on the convention, (capture avoiding) variable renaming is defined as usual. We denote by $[\beta/\alpha]T$ the result of replacing the free occurrences of variable α by variable β in type T. For the purposes of this paper, we need not concern ourselves with more general substitutions, such as $[U/\alpha]T$.

Types. Not all pretypes are of interest to us. The semicolon operator can only be applied to session types: a pretype of the form $\mathsf{unit};\mathsf{unit}$ cannot be considered a proper type. Type identifiers are

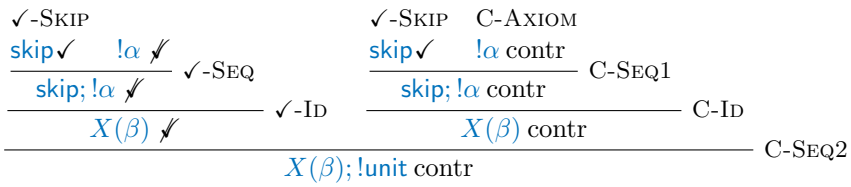


Figure 4: Example of a successful derivation of $X(\beta); !\mathsf{unit}\ \mathrm{contr}$ under signature $X(\alpha) \doteq \mathsf{skip}; !\alpha$.

supposed to exhibit a type constructor after a finite number of equation expansions; a pretype X under signature $X \doteq Y, Y \doteq X$ is not a proper type. What constitutes a proper type (henceforth, simply a type) is defined by the type formation rules in fig. 3. Type formation relies on contractivity, which in turn relies on the is-terminated predicate. We discuss these in turn.

The *is-terminated* predicate on pretypes, $T\checkmark$, materializes the intuition about a (session) pretype that prescribes no operation (lemma 9 testifies this intuition); it is *inductively* defined and comprises only sequential composition of *terminated* pretypes and identifiers defined by means of *terminated* pretypes. *Contractivity* ensures that a pretype eventually rewrites to a pretype constructor, eschewing pretypes such as $X(\beta)$ with $X(\alpha) \doteq \mathsf{skip}; Y(\alpha)$ and $Y(\alpha) \doteq X(\alpha); Y(\alpha)$. The is-terminated and the contractivity predicates are inductively defined, hence we are looking for finite derivations. The result below states that the is-terminated predicate is decidable, which justifies the inclusion of its negation ($T\not\checkmark$) in rule C-Seq2 [4].

Lemma 1 (Decidability of termination)**.** *We can efficiently decide if $T\checkmark$ for a given pretype T.*

Proof. To implement the is-terminated predicate, we may simply traverse the abstract syntax trees of the given type T and the right-hand sides of equations $X_i(\overline{\alpha_i}) \doteq T_i$, starting at the root node of T. To ensure that we do not get stuck in a loop, we keep track of the type identifiers visited throughout this traversal. In practice, a step of this traversal looks at the current node, corresponding to some type U, and proceeds as follows:

1. If $U = \mathsf{skip}$, then return `true`.
2. If $U = U_1; U_2$, then return $U_1\checkmark$ `and` $U_2\checkmark$.
3. If $U = X_i(\overline{\beta})$ for a not yet visited type identifier X_i, then mark X_i as visited and return $T_i\checkmark$, where $X_i(\overline{\alpha}) \doteq T_i$.
4. If $U = X_i(\overline{\beta})$ for an already visited type identifier X_i, then return `false`.
5. If U is not of the form skip, $U_1; U_2$ or $X(\overline{\beta})$, then return `false`.

The procedure visits each node of the abstract syntax trees at most once, and hence it terminates in polynomial time on the size of T and Σ. □

Figures 4 and 5 give examples of successful and unsuccessful derivations of contractivity. We abuse notation and include a derivation for predicate $T\not\checkmark$ to represent the case where the derivation for $T\checkmark$ would fail.

Kinds for polymorphic variables are kept in a kinding context Δ containing bindings of the form $\alpha : \kappa$. Type formation is given by judgement $\Delta \vdash T : \kappa$, stating that type T has kind κ under context Δ. Unlike the is-terminated and contractivity predicates, the rules for type formation must be interpreted coinductively. The rules are mostly straightforward; we describe a few. Function, record and polymorphic types are of kind T irrespective of its constituents. Rule K-Quant moves the binding $\alpha : \kappa$ from the type to the context. Notation $\Delta, \alpha : \kappa$ requires α not to be in Δ; we use the variable convention to rename the type in case α happens to be in Δ. K-Id checks the formation of the body T of equation $X(\overline{\alpha}) \doteq T$ under context $\overline{\alpha} : \overline{\kappa}$, thus forcing all free variables in T to be in $\{\overline{\alpha}\}$.

By inspection of the rules in fig. 3, we realise that, for each given pretype, there is at most one rule that applies. Hence, type formation derivations are unique and, in particular, any pretype has at most one kind, in which case we call the pretype a type: we say that T *is a type* when there

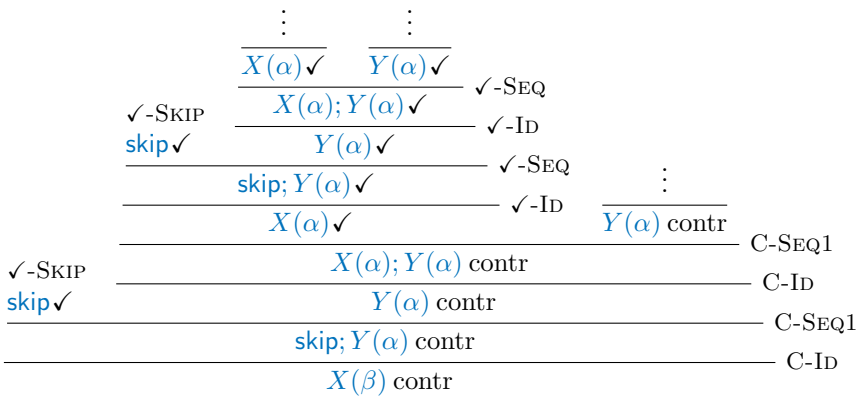


Figure 5: Example of an unsuccessful derivation of $X(\beta)$ contr under signature $X(\alpha) \doteq \mathsf{skip}; Y(\alpha)$, $Y(\alpha) \doteq X(\alpha); Y(\alpha)$. The vertical dots denote infinite derivations, which are not allowed in the inductive setting.

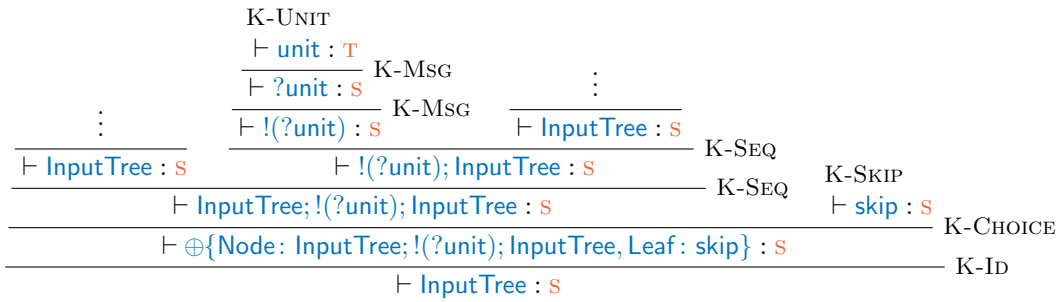


Figure 6: Example of a successful derivation of judgement $\vdash \mathsf{InputTree} : \mathrm{S}$ under signature $\mathsf{InputTree} \doteq \oplus\{\mathsf{Node}: \mathsf{InputTree}; !(?\mathsf{int}); \mathsf{InputTree}, \mathsf{Leaf}: \mathsf{skip}\}$. For simplicity, we do not include the contractivity judgements. Vertical dots denote infinite derivations, which are allowed in the coinductive setting.

are Δ and κ such that $\Delta \vdash T : \kappa$. When context Δ is empty, we simply write $\vdash T : \kappa$. We denote by $\mathbb{T}$ the language of all types. Figure 6 illustrates the rules of type formation by showing that $\mathsf{InputTree}$ (introduced in section 1) is a (session) type.

The is-terminated predicate, contractivity and type formation are preserved by renaming.

Lemma 2 (Renaming)**.**

1. $T\checkmark$ *iff* $[\overline{\beta}/\overline{\alpha}]T\checkmark$.
2. T contr *iff* $[\overline{\beta}/\overline{\alpha}]T$ contr.
3. *If* $\Delta, \overline{\alpha}: \overline{\kappa} \vdash T : \kappa$, *then* $\Delta, \overline{\beta}: \overline{\kappa} \vdash [\overline{\beta}/\overline{\alpha}]T : \kappa$.

Proof. Items 1 and 2: A straightforward induction on the structure of T in both directions. Item 3: A coinductive proof. Consider the relation

$$\mathcal{R} = \{(\Delta, \overline{\beta}: \overline{\kappa}, [\overline{\beta}/\overline{\alpha}]T, \kappa) \quad : \quad \Delta, \overline{\alpha}: \overline{\kappa} \vdash T : \kappa\}.$$

We show that $\mathcal{R}$ is backward closed for the rules defining type formation, so that $\mathcal{R} \subseteq \vdash$. Therefore, $\Delta, \overline{\alpha}: \overline{\kappa} \vdash T : \kappa$ implies $(\Delta, \overline{\beta}: \overline{\kappa}, [\overline{\beta}/\overline{\alpha}]T, \kappa) \in \mathcal{R}$, which in turn implies $\Delta, \overline{\beta}: \overline{\kappa} \vdash [\overline{\beta}/\overline{\alpha}]T : \kappa$. Take $(\Delta, \overline{\beta}: \overline{\kappa}, [\overline{\beta}/\overline{\alpha}]T, \kappa) \in \mathcal{R}$, meaning that $\Delta, \overline{\alpha}: \overline{\kappa} \vdash T : \kappa$. We proceed by a case analysis on the last rule used to derive $\Delta, \overline{\alpha}: \overline{\kappa} \vdash T : \kappa$. We sketch only a few relevant cases.

(**Case** K-Unit): then $T = \mathsf{unit}$ and $\kappa = \mathrm{T}$. Clearly, also $[\overline{\beta}/\overline{\alpha}]T = \mathsf{unit}$, and we can apply the same rule K-Unit to $(\Delta, \overline{\beta}: \overline{\kappa}, [\overline{\beta}/\overline{\alpha}]T, \kappa)$.

(**Case** K-Var): then $T = \gamma$ for some $\gamma: \kappa \in \Delta, \overline{\alpha}: \overline{\kappa}$. If $\gamma: \kappa \in \Delta$, then $[\overline{\beta}/\overline{\alpha}]T = \gamma$, and we can apply the same rule K-Var to $(\Delta, \overline{\beta}: \overline{\kappa}, [\overline{\beta}/\overline{\alpha}]T, \kappa)$. Otherwise, γ equals some $\alpha_i \in \{\overline{\alpha}\}$ and κ equals the corresponding κ_i. Then $[\overline{\beta}/\overline{\alpha}]T = \beta_i \in \{\overline{\beta}\}$, and we can apply the same rule K-Var to $(\Delta, \overline{\beta}: \overline{\kappa}, [\overline{\beta}/\overline{\alpha}]T, \kappa)$.

(**Case** K-Quant, $\forall$): then $T = \forall\gamma\colon\kappa'.U$ and $\kappa = \mathrm{T}$. By renaming the bound variable γ if necessary, we can assume that γ does not appear in Δ, $\overline{\alpha}$ or $\overline{\beta}$. Applying rule K-Quant to $\Delta, \overline{\alpha}\colon\overline{\kappa} \vdash T : \kappa$ yields $\Delta, \gamma\colon\kappa', \overline{\alpha}\colon\overline{\kappa} \vdash U : \kappa''$. Notice that $[\overline{\beta}/\overline{\alpha}]T = \forall\gamma\colon\kappa'.[\overline{\beta}/\overline{\alpha}]U$. We can apply rule K-Quant to $(\Delta, \overline{\beta}\colon\overline{\kappa}, [\overline{\beta}/\overline{\alpha}]T, \kappa)$, obtaining the triple $(\Delta, \gamma\colon\kappa', \overline{\beta}\colon\overline{\kappa}, [\overline{\beta}/\overline{\alpha}]U, \kappa'')$. This triple is in $\mathcal{R}$ since $\Delta, \gamma\colon\kappa', \overline{\alpha}\colon\overline{\kappa} \vdash U : \kappa''$.

(**Case** K-Id): then $T = X(\overline{\gamma})$ for some $X(\overline{\alpha'}) \doteq U$. Split $\overline{\gamma}\colon\overline{\kappa'}$ into $\overline{\gamma^1}\colon\overline{\kappa^1}, \overline{\gamma^2}\colon\overline{\kappa^2}$, with $\overline{\gamma^1}\colon\overline{\kappa^1} \subseteq \Delta$ and $\overline{\gamma^2}\colon\overline{\kappa^2} \subseteq \overline{\alpha}\colon\overline{\kappa}$. Let $\overline{\alpha^2}$ be the variables in $\overline{\alpha}$ corresponding to $\overline{\gamma^2}$ and $\overline{\beta^2}$ be the variables in $\overline{\beta}$ corresponding to $\overline{\alpha^2}$. Applying rule K-Id to $\Delta, \overline{\alpha}\colon\overline{\kappa} \vdash T : \kappa$ yields U contr and $\overline{\alpha'}\colon\overline{\kappa'} \vdash U : \kappa$. Notice that $[\overline{\beta}/\overline{\alpha}]T = X([\overline{\beta}/\overline{\alpha}]\overline{\gamma}) = X(\overline{\gamma^1}, \overline{\beta^2})$. We can apply rule K-Id to $(\Delta, \overline{\beta}\colon\overline{\kappa}, [\overline{\beta}/\overline{\alpha}]T, \kappa)$, since $\overline{\gamma^1}\colon\overline{\kappa^1} \subseteq \Delta$, $\overline{\beta^2}\colon\overline{\kappa^2} \subseteq \overline{\beta}\colon\overline{\kappa}$ and U contr. The resulting triple is $(\overline{\alpha'}\colon\overline{\kappa'}, U, \kappa)$, which is in $\mathcal{R}$ since $\overline{\alpha'}\colon\overline{\kappa'} \vdash U : \kappa$ (empty substitution).

□

Lemma 3 (Weakening)**.** *If* $\Delta \vdash T : \kappa$*, then* $\Delta, \alpha\colon\kappa' \vdash T : \kappa$*.*

Proof. A coinductive proof. Consider the relation

$$\mathcal{R} = \{(\Delta, \alpha\colon\kappa', T, \kappa) \quad : \quad \Delta \vdash T : \kappa\}.$$

Similarly as in the proof of lemma 2 (item 3), we can show that $\mathcal{R}$ is backward closed for the rules defining type formation, so that $\mathcal{R} \subseteq \vdash$. Therefore, $\Delta \vdash T : \kappa$ implies $(\Delta, \alpha\colon\kappa', T, \kappa) \in \mathcal{R}$, which in turn implies $\Delta, \alpha\colon\kappa' \vdash T : \kappa$. □

Unravelling Pretypes. To conclude this section, we take a step back from types to define a fundamental operation on pretypes, namely the *unravel* operation (similar to the so-called unfold operation). We start with the one-step unravel function, $\mathrm{unravel}_1(T)$. To determine whether T unravels in one step to some pretype U, we proceed by a case analysis on the structure of T.

$$\begin{aligned}
\mathrm{unravel}_1(X(\overline{\beta})) &= [\overline{\beta}/\overline{\alpha}]T && \text{when } X(\overline{\alpha}) \doteq T \in \Sigma \\
\mathrm{unravel}_1(\mathsf{skip}; T) &= T \\
\mathrm{unravel}_1(\odot\{\ell\colon T_\ell\}_{\ell\in L}; U) &= \odot\{\ell\colon T_\ell; U\}_{\ell\in L} \\
\mathrm{unravel}_1((T; U); V) &= T; (U; V) \\
\mathrm{unravel}_1(X(\overline{\beta}); U) &= [\overline{\beta}/\overline{\alpha}]T; U && \text{when } X(\overline{\alpha}) \doteq T \in \Sigma
\end{aligned}$$

Each of unit, $T \to U$, $(\!|\ell\colon T_\ell|\!)_{\ell\in L}$, $\exists\forall\alpha\colon\kappa\,.\,T$, skip, $\sharp T$, $\odot\{\ell\colon T_\ell\}_{\ell\in L}$, α, $\sharp T; U$, $\alpha; T$ is an *unravelled* pretype, and thus it is not in the domain of the $\mathrm{unravel}_1$ function.

One-step unravelling expands the definitions for type identifiers (applying the appropriate variable renaming), removes trailing skip types (given that skip is neutral for sequential composition), moves the continuation pretype U of a choice within each branch (making use of distributivity of sequential composition over choice), and flattens sequential composition into list-like pretypes (taking advantage of the associativity of sequential composition). Notice how $\mathrm{unravel}_1$ is defined by a case analysis on the outermost constructor of the pretype. If the outermost constructor is a sequential composition $(T; U)$, a second case analysis is performed on the outermost constructor of T.

Now, given pretype T, consider the sequence $T_0, T_1, \ldots$ where $T_0 = T$ and each T_{i+1} is the one-step unravel of T_i. We say that $\mathrm{unravel}(T)$ is defined if this sequence is finite, reaching T_n for which $\mathrm{unravel}_1$ is not defined. In this case $\mathrm{unravel}(T) = T_n$. If the sequence is infinite, then $\mathrm{unravel}(T)$ is undefined. We conclude this section with some immediate properties relating unraveling, termination, contractivity and type formation.

Proposition 4. *For pretypes* T*,* U*, let* $\mathrm{unravel}(T) = \mathrm{unravel}(U)$ *mean that either both sides are defined and equal, or that both sides are undefined. We have:*

1. $\mathrm{unravel}(X(\overline{\beta})) = \mathrm{unravel}([\overline{\beta}/\overline{\alpha}]T)$ *when* $X(\overline{\alpha}) \doteq T \in \Sigma$*;*
2. $\mathrm{unravel}(\mathsf{skip}; T) = \mathrm{unravel}(T)$*;*
3. $\mathrm{unravel}(\odot\{\ell\colon T_\ell\}_{\ell\in L}; U) = \odot\{\ell\colon T_\ell; U\}_{\ell\in L}$*;*

4. $\mathrm{unravel}((T;U);V) = \mathrm{unravel}(T;(U;V))$;

5. $\mathrm{unravel}(X(\overline{\beta});U) = \mathrm{unravel}([\overline{\beta}/\overline{\alpha}]T;U)$ *when* $X(\overline{\alpha}) \doteq T \in \Sigma$;

6. $\mathrm{unravel}(T) = T$ *for any other pretype* T.

Proposition 5. *Let* T *be a pretype.*

1. *If* $T = \mathrm{unravel}(T)$ *then* T *is one of* unit, $U \to V$, $(\!|\ell\colon T_\ell|\!)_{\ell\in L}$, $\forall\alpha\colon\kappa\,.\,U$, skip, $\sharp U$, $\odot\{\ell\colon T_\ell\}_{\ell\in L}$, α, $\sharp U;V$, $\alpha;U$.

2. *Otherwise,* T *is one of* $X(\overline{\beta})$, $\mathsf{skip};U$, $\odot\{\ell\colon T_\ell\}_{\ell\in L};U$, $(U;V);W$, $X(\overline{\beta});U$.

The following result bridges the rule-based and the unravel-based notions of contractivity. Both notions have been used interchangeably [40], but to the best of our knowledge, they have never been proven to be equivalent.

Lemma 6 (Contractivity and unravelling)**.** *If* T *is a pretype, then* $T\,\mathrm{contr}$ *iff* $\mathrm{unravel}(T)$ *is defined.*

Proof. In Appendix A. □

Lemma 7 (Preservation for unravelling)**.**

1. *Let* T *be a pretype and* $U = \mathrm{unravel}_1(T)$.

   (a) $T\checkmark$ *iff* $U\checkmark$.

   (b) $T\,\mathrm{contr}$ *iff* $U\,\mathrm{contr}$.

   (c) *If* $\Delta \vdash T : \kappa$, *then* $\Delta \vdash U : \kappa$.

2. *Let* T *be a pretype and* $U = \mathrm{unravel}(T)$.

   (a) $T\checkmark$ *iff* $U\checkmark$.

   (b) $T\,\mathrm{contr}$ *iff* $U\,\mathrm{contr}$.

   (c) *If* $\Delta \vdash T : \kappa$, *then* $\Delta \vdash U : \kappa$.

3. *If* $\Delta \vdash T : \kappa$, *then* $T\,\mathrm{contr}$ *and* $\mathrm{unravel}(T)$ *is defined.*

Proof. For item 1, the proof follows by a case analysis on T. By definition of one-step unraveling, T is one of $X(\overline{\beta})$, $\mathsf{skip};V$, $\odot\{\ell\colon T_\ell\}_{\ell\in L};V$, $X(\overline{\beta});V$ or $(M;V);W$.

(**1a. Case** $T = X(\overline{\beta})$): let $X(\overline{\alpha}) \doteq W$ be the equation for X, so that $U = [\overline{\beta}/\overline{\alpha}]W$. Any derivation for $T\checkmark$ must end with rule ✓-Id, implying that $T\checkmark$ iff $W\checkmark$. Lemma 2 (renaming) yields $[\overline{\beta}/\overline{\alpha}]W\checkmark$ iff $W\checkmark$, as desired.

(**1a. Case** $T = \mathsf{skip};V$): any derivation for $T\checkmark$ must end with rule ✓-Seq, implying that $T\checkmark$ iff $\mathsf{skip}\checkmark$ and $V\checkmark$. Since $\mathsf{skip}\checkmark$, we get that $T\checkmark$ iff $U\checkmark$.

(**1a. Case** $T = \odot\{\ell\colon T_\ell\}_{\ell\in L};V$): in this case $U = \odot\{\ell\colon T_\ell;V\}_{\ell\in L}$ and it is clear that $T\not\checkmark$, $U\not\checkmark$.

(**1a. Case** $T = X(\overline{\beta});V$): let $X(\overline{\alpha}) \doteq W$ be the equation for X, so that $U = [\overline{\beta}/\overline{\alpha}]W;V$. Any derivation for $T\checkmark$ must end with rule ✓-Seq, implying that $T\checkmark$ iff $X(\overline{\beta})\checkmark$ and $V\checkmark$. Any derivation for $X(\overline{\beta})\checkmark$ must end with rule ✓-Id, implying that $X(\overline{\beta})\checkmark$ iff $W\checkmark$. Any derivation for $U\checkmark$ must end with rule ✓-Seq, implying that $U\checkmark$ iff $[\overline{\beta}/\overline{\alpha}]W\checkmark$ and $V\checkmark$. Lemma 2 (renaming) yields $[\overline{\beta}/\overline{\alpha}]W\checkmark$ iff $W\checkmark$. Putting all these together, we conclude that $T\checkmark$ iff $U\checkmark$.

(**1a. Case** $T = (M;V);W$): then we have $U = M;(V;W)$. Any derivation for $T\checkmark$ must end with two applications of rule ✓-Seq, implying that $T\checkmark$ iff $M\checkmark$, $V\checkmark$ and $W\checkmark$. Any derivation for $U\checkmark$ must end with two applications of rule ✓-Seq, implying that $U\checkmark$ iff $M\checkmark$, $V\checkmark$ and $W\checkmark$. Therefore, $T\checkmark$ iff $U\checkmark$.

(**1b.** and **1c.**): Similar to **1a.** (**1c. Case** $T = X(\overline{\beta})$) deserves a note. If $X(\overline{\alpha}) \doteq V$ then, starting from $\Delta', \overline{\beta}\colon\overline{\kappa} \vdash X(\overline{\beta}) : \mathrm{S}$, we know that $\overline{\alpha}\colon\overline{\kappa} \vdash V : \mathrm{S}$. Then lemma 2 (renaming) yields $\overline{\beta}\colon\overline{\kappa} \vdash [\overline{\beta}/\overline{\alpha}]V : \mathrm{S}$ and lemma 3 (weakening) yields $\Delta', \overline{\beta}\colon\overline{\kappa} \vdash [\overline{\beta}/\overline{\alpha}]V : \mathrm{S}$, from which we conclude the result by rule K-Id.

Item 2 follows from item 1 since $\mathrm{unravel}(T)$ is reached in a finite number of steps when defined.

For item 3, we know by lemma 6 that T contr iff unravel(T) is defined. Hence, it suffices to show that $\Delta \vdash T : \kappa$ implies T contr. For any $n \in \mathbb{N}$, let

$$\mathbb{T}_n = \{T \colon \Delta \vdash T : \kappa \text{ for some } \Delta, \kappa \text{ and the abstract syntax tree for } T \text{ has at most } n \text{ nodes}\}$$

be the set of types whose abstract syntax trees have at most n nodes. Notice that the set of all types is the union of all $\mathbb{T}_n$. We prove, by induction on n, that $T \in \mathbb{T}_n$ implies T contr, giving the desired result.

The base case (with $n = 0$) is trivial, as $\mathbb{T}_0 = \emptyset$. For the induction step, suppose $n > 0$ and $T \in \mathbb{T}_n$. We perform a case analysis on the last rule of the derivation for $\Delta \vdash T : \kappa$.

(**Case** K-Unit): then $T = \mathsf{unit}$ and, by C-Axiom, we have T contr. Cases K-Arrow, K-Rcd, K-Skip, K-Msg, K-Choice, K-Var, K-Quant are all similar.

(**Case** K-Id): then $T = X(\overline{\beta})$ with a corresponding equation $X(\overline{\alpha}) \doteq T'$. Given that rule K-Id was applied, we know that T' contr. Therefore, by rule C-Id, we get that $X(\overline{\beta})$ contr.

(**Case** K-Seq): then $T = U;V$ and $\Delta \vdash U : \mathrm{S}$, $\Delta \vdash V : \mathrm{S}$. Moreover, each of the abstract syntax trees for U and V has fewer than n nodes. This implies that $U, V \in \mathbb{T}_{n-1}$. By induction hypothesis, U contr and V contr. Finally, either $U\checkmark$, in which case T contr by rule C-Seq1, or $U \not\checkmark$, in which case T contr by rule C-Seq2. □

The converse of items 1c and 2c does not hold. A counter-example is the pretype $\mathsf{skip};\mathsf{unit}$ which is not a type (one cannot find Δ and κ such that $\Delta \vdash \mathsf{skip};\mathsf{unit} : \kappa$). However, it unravels to unit, which is a type of kind T under the empty context.

3. Type Equivalence

From this point on, our definitions and results assume that we are given types T, U (not just pretypes). This section introduces the notion of type equivalence.

Type Equivalence. The rules for type equivalence are in fig. 7. The novelty lies in sequential composition. Intuitively, sequential composition has a monoidal structure where skip is the left and right neutral element:

$$(T;U);V \simeq T;(U;V)$$
$$\mathsf{skip};T \simeq T;\mathsf{skip} \simeq T$$

In addition, sequential composition right distributes over 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-Var, are the congruence rules for all type constructors. Note that, because we rely on the variable convention (section 2), rule E-Quant relies on implicit renaming when present with types that differ on bound variables.

Rules E-IdL and E-IdR deal with type identifiers and associated equations equirecursively. Without surprise, we resort to the equation associated with the type identifier applying the necessary substitution.

The remaining rules are 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 ($V \simeq T;U$). We incorporate the monoidal properties of sequential composition in the proposed rules: rules E-SkipSeqL and E-SkipSeqR cope with the neutral element of sequential composition, rules E-ChoiceSeqL and E-ChoiceSeqR incorporate the distribution of sequential composition with respect to choices, and rules E-SeqSeqL and E-SeqSeqR materialize the associative property. With other type constructors, sequential composition is handled component-wise. Intuitively, given that we are working under a coinductive setting, we have rules that 'move' the sequential composition operator 'down' the syntax tree (or, to put it in another way, that 'move' type constructors that actually produce something 'up' the syntax tree). This is why for types of the form $T;U$ we look at the structure of T to decide which rule to apply next.

The two main results in this section establish type agreement and that $\simeq$ is an equivalence relation (theorems 10 and 11). Before that, we offer some preliminary observations about the structure of type equivalence derivations. We can organise the type equivalence rules in three groups.

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

$$\textsc{E-Unit}\ \ \mathsf{unit} \simeq \mathsf{unit} \qquad \textsc{E-Arrow}\ \ \frac{T \simeq U \quad V \simeq W}{T \to V \simeq U \to W} \qquad \textsc{E-Rcd}\ \ \frac{T_\ell \simeq U_\ell \quad (\forall \ell \in L)}{(\!|\ell \colon T_\ell|\!)_{\ell \in L} \simeq (\!|\ell \colon U_\ell|\!)_{\ell \in L}} \qquad \textsc{E-Quant}\ \ \frac{T \simeq U}{\exists \forall \alpha \colon \kappa \,.\, T \simeq \exists \forall \alpha \colon \kappa \,.\, U}$$

$$\textsc{E-Skip}\ \ \mathsf{skip} \simeq \mathsf{skip} \qquad \textsc{E-Msg}\ \ \frac{T \simeq U}{\sharp T \simeq \sharp U} \qquad \textsc{E-Choice}\ \ \frac{T_\ell \simeq U_\ell \quad (\forall \ell \in L)}{\odot\{\ell \colon T_\ell\}_{\ell \in L} \simeq \odot\{\ell \colon U_\ell\}_{\ell \in L}} \qquad \textsc{E-Var}\ \ \alpha \simeq \alpha$$

$$\textsc{E-IdL}\ \ \frac{X(\overline{\alpha}) \doteq T \quad T \text{ contr} \quad [\overline{\beta}/\overline{\alpha}]T \simeq U}{X(\overline{\beta}) \simeq U} \qquad \textsc{E-IdR}\ \ \frac{X(\overline{\alpha}) \doteq U \quad U \text{ contr} \quad T \simeq [\overline{\beta}/\overline{\alpha}]U}{T \simeq X(\overline{\beta})} \qquad \textsc{E-SkipSeqL}\ \ \frac{T \simeq U}{\mathsf{skip}; T \simeq U}$$

$$\textsc{E-SkipSeqR}\ \ \frac{T \simeq U}{T \simeq \mathsf{skip}; U} \qquad \textsc{E-MsgSeq1L}\ \ \frac{T \simeq U \quad V\checkmark}{\sharp T; V \simeq \sharp U} \qquad \textsc{E-MsgSeq1R}\ \ \frac{T \simeq U \quad V\checkmark}{\sharp T \simeq \sharp U; V} \qquad \textsc{E-MsgSeq2}\ \ \frac{T \simeq U \quad V \simeq W}{\sharp T; V \simeq \sharp U; W} \qquad \textsc{E-ChoiceSeqL}\ \ \frac{\odot\{\ell \colon T_\ell; U\}_{\ell \in L} \simeq V}{\odot\{\ell \colon T_\ell\}_{\ell \in L}; U \simeq V}$$

$$\textsc{E-ChoiceSeqR}\ \ \frac{U \simeq \odot\{\ell \colon T_\ell; V\}_{\ell \in L}}{U \simeq \odot\{\ell \colon T_\ell\}_{\ell \in L}; V} \qquad \textsc{E-SeqSeqL}\ \ \frac{T; (U; V) \simeq W}{(T; U); V \simeq W} \qquad \textsc{E-SeqSeqR}\ \ \frac{T \simeq U; (V; W)}{T \simeq (U; V); W} \qquad \textsc{E-VarSeq1L}\ \ \frac{T\checkmark}{\alpha; T \simeq \alpha} \qquad \textsc{E-VarSeq1R}\ \ \frac{T\checkmark}{\alpha \simeq \alpha; T}$$

$$\textsc{E-VarSeq2}\ \ \frac{T \simeq U}{\alpha; T \simeq \alpha; U} \qquad \textsc{E-IdSeqL}\ \ \frac{X(\overline{\alpha}) \doteq T \quad T \text{ contr} \quad ([\overline{\beta}/\overline{\alpha}]T); V \simeq U}{X(\overline{\beta}); V \simeq U}$$

$$\textsc{E-IdSeqR}\ \ \frac{X(\overline{\alpha}) \doteq U \quad U \text{ contr} \quad T \simeq ([\overline{\beta}/\overline{\alpha}]U); V}{T \simeq X(\overline{\beta}); V}$$

Figure 7: Type equivalence.

Progressing: E-Unit, E-Arrow, E-Rcd, E-Quant, E-Skip, E-Msg, E-Choice, E-Var, E-MsgSeq1L, E-MsgSeq1R, E-MsgSeq2, E-VarSeq1L, E-VarSeq1R and E-VarSeq2. These rules consume the types on both sides of the relation. In other words, if we apply one of these rules starting from judgement $T \simeq U$, we end up with judgements $T' \simeq U'$ where T', U' are proper subterms of T, U respectively. Moreover, these rules can be applied when $T = \text{unravel}(T)$ and $U = \text{unravel}(U)$.

Right-preserving: E-IdL, E-SkipSeqL, E-ChoiceSeqL, E-SeqSeqL and E-IdSeqL. These rules replace type T on the left-hand side of the relation by $\text{unravel}_1(T)$. The type on the right-hand side, however, remains the same. The rules can be applied when $\text{unravel}_1(T)$ is defined, hence when $T \neq \text{unravel}(T)$.

Left-preserving: E-IdR, E-SkipSeqR, E-ChoiceSeqR, E-SeqSeqR and E-IdSeqR. These rules replace type U on the right-hand side of the relation by $\text{unravel}_1(U)$.

Furthermore, by inspection of the rules, we can observe the following.

- If we can apply a progressing rule to judgement $T \simeq U$, then this is the only rule that can be applied.
- If we can apply a left-preserving rule for judgement $T \simeq U$, then this is the only left-preserving rule that can be applied (but we can possibly also apply a right-preserving rule).
- If we can apply a right-preserving rule for judgement $T \simeq U$, then this is the only right-preserving rule that can be applied (but we can possibly also apply a left-preserving rule).

- If we can apply a left-preserving rule as well as a right-preserving rule for judgement $T \simeq U$, then we can apply them one after the other (in any order); and moreover, any successful derivation for $T \simeq U$ must eventually apply both rules.

To illustrate this, we give the following example of two derivations for the same judgement that differ in the order in which the left-preserving or right-preserving rules are applied.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{skip} \simeq \mathsf{skip}}\,\text{E-Skip}}{\mathsf{skip} \simeq \mathsf{skip};\mathsf{skip}}\,\text{E-SkipSeqR}}{\oplus\{\mathsf{Go}\colon \mathsf{skip}\} \simeq \oplus\{\mathsf{Go}\colon \mathsf{skip};\mathsf{skip}\}}\,\text{E-Choice}}{\oplus\{\mathsf{Go}\colon \mathsf{skip}\} \simeq \oplus\{\mathsf{Go}\colon \mathsf{skip}\};\mathsf{skip}}\,\text{E-ChoiceSeqR}}{\mathsf{skip};\oplus\{\mathsf{Go}\colon \mathsf{skip}\} \simeq \oplus\{\mathsf{Go}\colon \mathsf{skip}\};\mathsf{skip}}\,\text{E-SkipSeqL}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{skip} \simeq \mathsf{skip}}\,\text{E-Skip}}{\mathsf{skip} \simeq \mathsf{skip};\mathsf{skip}}\,\text{E-SkipSeqR}}{\oplus\{\mathsf{Go}\colon \mathsf{skip}\} \simeq \oplus\{\mathsf{Go}\colon \mathsf{skip};\mathsf{skip}\}}\,\text{E-Choice}}{\mathsf{skip};\oplus\{\mathsf{Go}\colon \mathsf{skip}\} \simeq \oplus\{\mathsf{Go}\colon \mathsf{skip};\mathsf{skip}\}}\,\text{E-SkipSeqL}}{\mathsf{skip};\oplus\{\mathsf{Go}\colon \mathsf{skip}\} \simeq \oplus\{\mathsf{Go}\colon \mathsf{skip}\};\mathsf{skip}}\,\text{E-ChoiceSeqR}$$

The final observation is that, due to contractivity, a derivation cannot have an infinite sequence of consecutive preserving rules. Both left-preserving and right-preserving rules can only be applied as long as one of the sides has a one-step unravelling. Once we reach a judgement $T \simeq U$ where $T = \mathrm{unravel}(T)$ (i.e., $\mathrm{unravel}_1(T)$ is not defined), we cannot apply further right-preserving rules until we consume T (with a progressing rule), and similarly, if $U = \mathrm{unravel}(U)$, we cannot apply further left-preserving rules until we consume U (with a progressing rule). Since types are contractive (lemma 7), we must eventually (after finitely many steps) finish unravelling, and thus any derivation tree must eventually continue with a progressing rule after finitely many preserving rules.

Results. From the above discussion we can derive the following immediate results.

Lemma 8 (Unravel and equivalence)**.** *Let T, U be types.*

1. *Let $T' = \mathrm{unravel}_1(T)$. Then, $T \simeq U$ iff $T' \simeq U$.*
2. *Let $T' = \mathrm{unravel}(T)$. Then, $T \simeq U$ iff $T' \simeq U$.*

Proof. Item 1 follows from the preceding discussion. Item 2 follows from item 1 since $\mathrm{unravel}(T)$ is reached in a finite number of steps when defined. □

Lemma 9. *Let T be a type. Then $T\checkmark$ iff $T \simeq \mathsf{skip}$.*

Proof. First consider the case that $T = \mathrm{unravel}(T)$. By inspection of the possibilities for T (prescribed by proposition 5) as well as the rules for termination and type equivalence (figs. 3 and 7), it is immediate that $T\checkmark$ iff $T \simeq \mathsf{skip}$ iff $T = \mathsf{skip}$.

Now suppose that $T \neq \mathrm{unravel}(T)$. Let $T' = \mathrm{unravel}(T)$, so that $T' = \mathrm{unravel}(T')$. Applying lemmas 7 and 8 as well as the previous case, we conclude that $T\checkmark$ iff $T'\checkmark$ iff $T' \simeq \mathsf{skip}$ iff $T \simeq \mathsf{skip}$. □

Theorem 10 (Agreement for type equivalence)**.** *If $\Delta \vdash T : \kappa$, $\Delta \vdash U : \kappa'$ and $T \simeq U$, then $\kappa = \kappa'$.*

Proof. By a case analysis on the last rule in derivation of $T \simeq U$.

The first case is that in which $T = \mathrm{unravel}(T)$ and $U = \mathrm{unravel}(U)$, i.e., T and U are a direct application of a type constructor. In these situations our derivation ends with a progressing rule and it should be immediate that $\kappa = \kappa'$.

(**Subcase** E-Unit): this means that T is unit and U is unit. The only way to derive $\Delta \vdash T : \kappa$ and $\Delta \vdash U : \kappa'$ is via rule K-Unit, which imply $\kappa = \kappa' = \mathrm{T}$. Cases E-Skip and E-Var are analogous.

Term rules $\boxed{T \simeq T}$

$$T ::= \cdots \mid \cancel{\forall \alpha\colon \kappa.T} \mid \alpha\, \exists\!\!\forall_\kappa\, T$$

C-Axiom'
$$\alpha\, \exists\!\!\forall_\kappa\, T \text{ contr}$$

E-Quant'
$$\frac{T \simeq_t U}{\alpha\, \exists\!\!\forall_\kappa\, T \simeq_t \alpha\, \exists\!\!\forall_\kappa\, U}$$

Figure 8: Syntax and rules for terms. The syntax for terms replaces that in fig. 2. Rule C-Axiom' replaces rule C-Axiom in fig. 3. Rule E-Quant' replaces rule E-Quant in fig. 7.

(**Subcase** E-Arrow): this means that T is $T' \to T''$ and U is $U' \to U''$. The only way to derive $\Delta \vdash T : \kappa$ and $\Delta \vdash U : \kappa'$ is via rule K-Arrow, which imply $\kappa = \kappa' = \mathrm{T}$. Case E-Quant is analogous.

(**Subcase** E-Rcd): this means that T is $(\!|\ell\colon T_\ell|\!)_{\ell \in L}$ and U is $(\!|\ell\colon U_\ell|\!)_{\ell \in L}$. The only way to derive $\Delta \vdash T : \kappa$ and $\Delta \vdash U : \kappa'$ is via rule K-Rcd, which imply $\kappa = \kappa' = \mathrm{T}$. Case E-Choice is analogous.

(**Subcase** E-Msg): this means that T is $\sharp T'$ and U is $\sharp U'$. The only way to derive $\Delta \vdash T : \kappa$ and $\Delta \vdash U : \kappa'$ is via rule K-Msg, which imply $\kappa = \kappa' = \mathrm{S}$. Cases E-MsgSeq1L, E-MsgSeq1R, E-MsgSeq2 are analogous.

Next, we consider the case in which $T \neq \mathrm{unravel}(T)$. Without loss of generality, the derivation for $T \simeq U$ ends with a sequence of right-preserving rules that rewrite T until reaching $T' = \mathrm{unravel}(T)$. This sequence is necessarily finite due to contractivity. By lemmas 7 and 8, we have $\Delta \vdash T' : \kappa$ for the same kind κ, and $T' \simeq U$. If $U = \mathrm{unravel}(U)$, then we arrive at the first case, thus concluding that $\kappa = \kappa'$. If $U \neq \mathrm{unravel}(U)$, then by a similar reasoning we can assume that the derivation continues with a sequence of left-preserving rules that rewrite U until reaching $U' = \mathrm{unravel}(U)$ with $\Delta \vdash U' : \kappa'$ and $T' \simeq U'$. We once more reach the first case, concluding that $\kappa = \kappa'$. □

Theorem 11 (Equivalence relation)**.** *Relation $\simeq$ is an equivalence on types.*

Proof. In Appendix B. □

4. Terms and Canonical Renaming

So far we relied on the variable convention to implicitly rename bound variables whenever necessary. For example, the convention plays a crucial rule in proving that types $\forall \alpha\colon \mathrm{T}.\alpha$ and $\forall \beta\colon \mathrm{T}.\beta$ are equivalent. In this section we introduce an operation that explicitly renames bound variables with 'canonical' names. This operation can be regarded as an algorithmic mechanism to enforce the variable convention (section 2). Mathematically, such a renaming operation could be performed 'on-the-fly' as we compare two quantified types. For practical purposes, we show that in order to determine equivalence between two types we can actually perform this canonical renaming once at the start, so that equivalence can be handled syntactically from that point on (i.e., without need of further renamings).

For that, we define a new universe of objects, which we call *terms*. The syntax of terms is similar to that of types, except that instead of $\forall \alpha\colon \kappa.T$ we have the *infix* construct $\alpha\, \forall_\kappa\, T$ (and similarly for $\exists$). In this term, $\forall_\kappa$ is a non-binding construct, treated like the other constructs (such as $\to$, for example); this means that α is *not* bound in $\alpha\, \forall_\kappa\, T$. Moreover, term identifiers are not parameterized; for ease of notation we write a term identifier as Y instead of $Y()$.

Similarly to types, we can define on terms the notions of termination, contractivity, unraveling and equivalence. For our purposes we do not need to define kinding on terms. The main differences are of course on the rules involving quantifiers (summarized in fig. 8).

- For termination, the rules are the same.
- For contractivity, rule C-Axiom is replaced by C-Axiom' stating that $\alpha\, \exists\!\!\forall_\kappa\, T$ contr.

- For one-step unraveling and (multi-step) unraveling, the definitions are the same. This means that $\alpha\,\exists\forall_\kappa\,T$ is not in the domain of the unravel$_1$ function, and that unravel$(\alpha\,\exists\forall_\kappa\,T) = \alpha\,\exists\forall_\kappa\,T$.
- For term equivalence, rule E-Quant is replaced by E-Quant'. This means that for a term $\alpha\,\exists\forall_\kappa\,T$, the actual choice of α matters, since the variable is not bound. We use the notation $T \simeq_t U$ to denote term equivalence, distinguishing it from type equivalence.

Given the huge similarities between types and terms, almost all of the results proved in the previous sections also hold for terms, with nearly identical proofs. The only exceptions are the results involving kinding, since we do not care about defining this notion for terms.

In order to introduce the notion of canonical renaming, which converts a type into a term, assume there are sufficiently many fresh variables $\alpha_1, \alpha_2, \ldots$ (in order), all of which precede the variables occuring (free or bound) in T and in Σ. The intuition behind canonical renaming is to rename type $\forall\alpha\colon\kappa.T$ into term $\alpha_i\,\forall_\kappa\,T'$ where α_i is chosen in a 'canonical' way (i.e., the first choice possible) and T' is the renaming of $[\alpha_i/\alpha]T$. To make this notion precise, we first define the set of *genuine variables* of a type T.

Definition 1 (Genuine variables)**.** Given a type T, we define the set $\mathrm{gv}(T)$ of genuine variables of T coinductively. gv is the smallest function (from types to sets of variables) that is closed with respect to the following rules.

1. $\mathrm{gv}(\alpha) = \{\alpha\}$.
2. $\mathrm{gv}(\exists\forall\alpha\colon\kappa\,.\,T) = \mathrm{gv}(T) \setminus \{\alpha\}$.
3. $\mathrm{gv}(X(\overline{\beta})) = \mathrm{gv}([\overline{\beta}/\overline{\alpha}]T)$, if $X(\overline{\alpha}) \doteq T$.
4. $\mathrm{gv}(T \to U) = \mathrm{gv}(T) \cup \mathrm{gv}(U)$ (and similarly for all other type constructors).

We write first(T) to mean the first variable *not* in $\mathrm{gv}(T)$, i.e., the first variable that is not in the set of genuine variables of T.

The genuine variables are the subset of the free variables that actually appear in the possibly infinite expansion of a type T. For example, for the equation $Y(\alpha_1, \alpha_2) \doteq \alpha_2 \to Y(\alpha_1, \alpha_2)$, we have that $\mathrm{gv}(Y(\alpha, \beta)) = \{\beta\}$ whereas $\mathrm{fv}(Y(\alpha, \beta)) = \{\alpha, \beta\}$. Although α is a free variable in $Y(\alpha, \beta)$, it is not genuine since it does not appear in its expansion. For another example, for the equation $Z(\alpha_1, \alpha_2) \doteq \mathsf{skip}$, we have that $\mathrm{gv}(\forall\alpha\colon\kappa.Z(\alpha, \beta)) = \emptyset$ whereas $\mathrm{fv}(\forall\alpha\colon\kappa.Z(\alpha, \beta)) = \{\beta\}$.

The main difference between the definitions of free and genuine variables is that $\mathrm{fv}(X(\overline{\beta}))$ can be computed in one step (it equals $\{\overline{\beta}\}$), whereas $\mathrm{gv}(X(\overline{\beta}))$ requires us to proceed with item 3 of definition 1. This means that we need to take extra care to argue that the computation of $\mathrm{gv}(T)$ actually terminates.

Proposition 12. *We can compute* $\mathrm{gv}(T)$ *for any given type* T *in time* $\mathcal{O}(n^2)$*, where* $n = \mathrm{size}(T, \Sigma)$.

Proof. By construction, $\mathrm{gv}(T) \subseteq \mathrm{fv}(T)$, which can be computed in linear time. Hence, it suffices to show that $\alpha \in \mathrm{gv}(T)$ can be decided efficiently, for a given T and each $\alpha \in \mathrm{fv}(T)$. The following is an inductive definition of the predicate $\beta \stackrel{?}{\in} \mathrm{gv}(T)$:

1. $\beta \in \mathrm{gv}(\beta)$.
2. $\beta \in \mathrm{gv}(\exists\forall\alpha\colon\kappa\,.\,T)$ if $\beta \neq \alpha$ and $\beta \in \mathrm{gv}(T)$.
3. $\beta \in \mathrm{gv}(X(\overline{\beta}))$ if $\beta = \beta_i$ for some i such that $\alpha_i \in \mathrm{gv}(T)$, where $X(\overline{\alpha}) \doteq T$.
4. $\beta \in \mathrm{gv}(T \to U)$ if $\beta \in \mathrm{gv}(T)$ or $\beta \in \mathrm{gv}(U)$ (and similarly for most other type constructors).

For item 2, the variable convention enables us to assume $\beta \neq \alpha$, since the bound variable α may be renamed as needed. Note that in using item 3, we change from variable $\beta \in \mathrm{gv}(X(\overline{\beta}))$ to variable $\alpha_i \in \mathrm{gv}(T)$. This yields the same definition of $\mathrm{gv}(T)$, since $\mathrm{gv}(T) \subseteq \mathrm{fv}(T) \subseteq \{\overline{\alpha}\}$ implies that $\beta \in \mathrm{gv}([\overline{\beta}/\overline{\alpha}]T)$ iff $\beta = \beta_i$ for some i such that $\alpha_i \in \mathrm{gv}(T)$.

We can use the above definition to compute $\mathrm{gv}(T)$, keeping track of each predicate $\alpha' \overset{?}{\in} T'$ visited in order to detect loops. For every such predicate, either: $\alpha' \in \mathrm{fv}(T)$ and T' is a subterm of T; or $\alpha' \in \{\overline{\alpha}\}$ and T' is a subterm of T_j for some $X_j(\overline{\alpha}) \doteq T_j \in \Sigma$. There are at most linearly many such α' and linearly many such T', and hence, there are $\mathcal{O}(n^2)$ such predicates $\alpha' \overset{?}{\in} T'$, where $n = \mathrm{size}(T, \Sigma)$. Thus, the computation of $\mathrm{gv}(T)$ only needs to consider $\mathcal{O}(n^2)$ distinct predicates. □

Definition 2 (Canonical renaming)**.** Given a type T, we define the canonical renaming $N(T)$ according to the following rules.

1. $N(\alpha) = \alpha$.
2. $N(\exists\forall\alpha\colon \kappa\,.\,T) = \alpha_i\,\exists\forall_\kappa\,N([\alpha_i/\alpha]T)$, where $\alpha_i = \mathrm{first}(\exists\forall\alpha\colon \kappa\,.\,T)$.
3. $N(X(\overline{\beta})) = Y_{\overline{\beta}}$, if $X(\overline{\alpha}) \doteq T$; here $Y_{\overline{\beta}}$ is a (new) term identifier with equation $Y_{\overline{\beta}} \doteq N([\overline{\beta}/\overline{\alpha}]T)$.
4. $N(T \to U) = N(T) \to N(U)$ (and similarly for most other type constructors).

Again, we first provide a few examples. Consider the type $T = \forall\alpha\colon\kappa.\forall\beta\colon\kappa.\mathsf{skip}$. When computing $N(T)$, we rename α to α_1, obtaining $\alpha_1\,\forall_\kappa\,N(\forall\beta\colon\kappa.\mathsf{skip})$. In the next step, we rename β to the same α_1, obtaining $\alpha_1\,\forall_\kappa\,\alpha_1\,\forall_\kappa\,\mathsf{skip}$. Now consider the type $U = \forall\alpha\colon\kappa.\forall\beta\colon\kappa.X(\alpha,\beta)$, with equation $X(\alpha_1,\alpha_2) \doteq \mathsf{skip}$. Notice that $T \simeq U$, and accordingly, computing $N(U)$ yields $\alpha_1\,\forall_\kappa\,\alpha_1\,\forall_\kappa\,Y_{\alpha_1,\alpha_1}$ with a new term identifier $Y_{\alpha_1,\alpha_1} \doteq \mathsf{skip}$, so that $N(T) \simeq_t N(U)$. Notably, during the computation of $N(U)$ we reach $\alpha_1\,\forall_\kappa\,N(\forall\beta\colon\kappa.Y(\alpha_1,\beta))$. At this moment we correctly infer that $\mathrm{gv}(\forall\beta\colon\kappa.Y(\alpha_1,\beta)) = \emptyset$, so that we can rename β by the same α_1. If we were to instead consider the set $\mathrm{fv}(\forall\beta\colon\kappa.Y(\alpha_1,\beta)) = \{\alpha_1\}$, we would erroneously rename β by a different α_2, obtaining an inequivalent term. This example justifies our decision of defining $\mathrm{first}(T)$ based on the genuine variables $\mathrm{gv}(T)$ rather than the free variables $\mathrm{fv}(T)$.

We need to argue that the coinductive definition of canonical renaming actually yields a computable procedure. This amounts to proving that only finitely many term identifiers are created during the construction of $N(T)$.

Proposition 13. *We can compute $N(T)$ for any given type T in time $2^{\mathcal{O}(n \log n)}$, where $n = \mathrm{size}(T, \Sigma)$.*

Proof. Recall our assumption that there are sufficiently many fresh variables $\alpha_1, \alpha_2, \ldots$ which all precede the variables occurring (free or bound) in T and Σ. As such, each of the substitutions occuring when applying item 2 uses a variable α_i that does not occur at all in the original type. Therefore, all of these substitutions are 'blind' in the sense that they do not require further bound variable changes. This also entails that the substitutions never change the binding variables themselves, in other words $[\alpha_i/\alpha](\exists\forall\beta\colon\kappa\,.\,U)$ becomes $\exists\forall\beta\colon\kappa\,.\,U'$ with the same β. Hence we only need to care about substitutions of the form $[\alpha_i/\alpha]$ where α_i is one of the fresh variables.

Let K be an upper bound on the number of variables occuring (free or bound) in T and Σ. Trivially, K is linear in the input size $n = \mathrm{size}(T, \Sigma)$. Without loss assume that the variables in T and Σ are taken from a set $\{\beta_1, \ldots, \beta_K\}$. Notice that, in item 2 of definition 2, the index of the substituent chosen is at most K, since $\exists\forall\alpha\colon\kappa\,.\,T$ has at most $K-1$ free variables. Hence, by the above paragraph, all of the term identifiers produced by the construction of $N(T)$ are of the form $Y_{\gamma_1,\ldots,\gamma_k}$, where $k \leq K$ and each γ_i is one of $\beta_1, \ldots, \beta_K, \alpha_1, \ldots, \alpha_K$. Therefore, a straightforward algorithm exists for computing $N(T)$: follow the rules in definition 2, creating new term identifiers as needed, while keeping track of the created identifiers. Since we only need to create finitely many new nonterminals and definitions, we conclude that our construction terminates. The number of new identifiers created is bounded by $n \times (2K)^K = n \times 2^{\mathcal{O}(K \log K)}$. Since $K = \mathcal{O}(n)$, the total complexity is

$$\mathrm{poly}(n) \times 2^{\mathcal{O}(K \log K)} = n^{\mathcal{O}(1)} 2^{\mathcal{O}(n \log n)} = 2^{\mathcal{O}(n \log n)}.$$

□

Let us consider the type $X(\alpha, \beta)$ given by equation

$$X(\alpha,\beta) \doteq \alpha \to \beta \to (\forall\alpha\colon\kappa.X(\alpha,\beta)) \to (\forall\beta\colon\kappa.X(\alpha,\beta))$$

We sketch the steps required to produce a renaming of $X(\alpha, \beta)$. We begin by creating a term identifier $Y_{\alpha,\beta}$ and its definition

$$\begin{aligned}
Y_{\alpha,\beta} &\doteq N(\alpha \to \beta \to (\forall \alpha\colon \kappa.X(\alpha,\beta)) \to (\forall \beta\colon \kappa.X(\alpha,\beta))) \\
&= \alpha \to \beta \to N(\forall \alpha\colon \kappa.X(\alpha,\beta)) \to N(\forall \beta\colon \kappa.X(\alpha,\beta)) && \text{(items 1 and 4)} \\
&= \alpha \to \beta \to (\alpha_1 \forall_\kappa N(X(\alpha_1,\beta))) \to (\alpha_1 \forall_\kappa N(X(\alpha,\alpha_1)) && \text{(item 2)} \\
&= \alpha \to \beta \to (\alpha_1 \forall_\kappa Y_{\alpha_1,\beta}) \to (\alpha_1 \forall_\kappa Y_{\alpha,\alpha_1}) && \text{(item 3)}
\end{aligned}$$

Notice how on the third line we use the same α_1 to replace both α and β, as this is the first non-genuine variable in either branch. Next we need to compute $Y_{\alpha_1,\beta}$ and Y_{α,α_1}.

$$\begin{aligned}
Y_{\alpha_1,\beta} &\doteq N(X(\alpha_1,\beta)) \\
&= N(\alpha_1 \to \beta \to (\forall \alpha\colon \kappa.X(\alpha,\beta)) \to (\forall \beta\colon \kappa.X(\alpha_1,\beta))) \\
&= \alpha_1 \to \beta \to N(\forall \alpha\colon \kappa.X(\alpha,\beta)) \to N(\forall \beta\colon \kappa.X(\alpha_1,\beta)) \\
&= \alpha_1 \to \beta \to (\alpha_1 \forall_\kappa N(X(\alpha_1,\beta))) \to (\alpha_2 \forall_\kappa N(X(\alpha_1,\alpha_2))) \\
&= \alpha_1 \to \beta \to (\alpha_1 \forall_\kappa Y_{\alpha_1,\beta}) \to (\alpha_2 \forall_\kappa Y_{\alpha_1,\alpha_2}) \\
Y_{\alpha,\alpha_1} &\doteq N(X(\alpha,\alpha_1)) \\
&= N(\alpha \to \alpha_1 \to (\forall \alpha\colon \kappa.X(\alpha,\alpha_1)) \to (\forall \beta\colon \kappa.X(\alpha,\beta))) \\
&= \alpha \to \alpha_1 \to N(\forall \alpha\colon \kappa.X(\alpha,\alpha_1)) \to N(\forall \beta\colon \kappa.X(\alpha,\beta)) \\
&= \alpha \to \alpha_1 \to (\alpha_2 \forall_\kappa N(X(\alpha_2,\alpha_1))) \to (\alpha_1 \forall_\kappa N(X(\alpha,\alpha_1))) \\
&= \alpha \to \alpha_1 \to (\alpha_2 \forall_\kappa Y_{\alpha_2,\alpha_1}) \to (\alpha_1 \forall_\kappa Y_{\alpha,\alpha_1})
\end{aligned}$$

Notice that term identifier $Y_{\alpha_1,\beta}$ reappears in its definition, so we do not need to continue computing it. Also notice that in the definition of $N(X(\alpha_1,\beta))$, the second binding $(\forall \beta\colon \kappa.X(\alpha_1,\beta))$ becomes $(\alpha_2 \forall_\kappa N(X(\alpha_1,\alpha_2)))$ since α_2 is the first non-genuine variable in $X(\alpha_1,\beta)$. Something analogous happens with Y_{α,α_1}.

We still have to compute Y_{α_1,α_2} and Y_{α_2,α_1}.

$$\begin{aligned}
Y_{\alpha_1,\alpha_2} &\doteq N(X(\alpha_1,\alpha_2)) \\
&= N(\alpha_1 \to \alpha_2 \to (\forall \alpha\colon \kappa.X(\alpha,\alpha_2)) \to (\forall \beta\colon \kappa.X(\alpha_1,\beta))) \\
&= \alpha_1 \to \alpha_2 \to N(\forall \alpha\colon \kappa.X(\alpha,\alpha_2)) \to N(\forall \beta\colon \kappa.X(\alpha_1,\beta)) \\
&= \alpha_1 \to \alpha_2 \to (\alpha_1 \forall_\kappa N(X(\alpha_1,\alpha_2))) \to (\alpha_2 \forall_\kappa N(X(\alpha_1,\alpha_2))) \\
&= \alpha_1 \to \alpha_2 \to (\alpha_1 \forall_\kappa Y_{\alpha_1,\alpha_2}) \to (\alpha_2 \forall_\kappa Y_{\alpha_1,\alpha_2}) \\
Y_{\alpha_2,\alpha_1} &\doteq N(X(\alpha_2,\alpha_1)) \\
&= N(\alpha_2 \to \alpha_1 \to (\forall \alpha\colon \kappa.X(\alpha,\alpha_1)) \to (\forall \beta\colon \kappa.X(\alpha_2,\beta))) \\
&= \alpha_2 \to \alpha_1 \to N(\forall \alpha\colon \kappa.X(\alpha,\alpha_1)) \to N(\forall \beta\colon \kappa.X(\alpha_2,\beta)) \\
&= \alpha_2 \to \alpha_1 \to (\alpha_2 \forall_\kappa N(X(\alpha_2,\alpha_1))) \to (\alpha_1 \forall_\kappa N(X(\alpha_2,\alpha_1))) \\
&= \alpha_2 \to \alpha_1 \to (\alpha_2 \forall_\kappa Y_{\alpha_2,\alpha_1}) \to (\alpha_1 \forall_\kappa Y_{\alpha_2,\alpha_1})
\end{aligned}$$

Notice that our construction requires a total of 5 new term identifiers and equations in order to define $N(X(\alpha,\beta))$.

We conclude this section with a theorem stating that canonical renaming preserves and reflects equivalence (theorem 16). Our theorem requires a few auxiliary results.

Lemma 14. *Let T, U be types. If $T \simeq U$ then $\mathrm{gv}(T) = \mathrm{gv}(U)$.*

Proof. In Appendix C. □

Lemma 15. *Let T, U be types and α, α' be variables.*

1. *If $T \simeq U$ then $[\alpha'/\alpha]T \simeq [\alpha'/\alpha]U$.*
2. *If $N([\alpha'/\alpha]T) \simeq_t N([\alpha'/\alpha]U)$ and $\alpha' \notin \mathrm{gv}(T) \cup \mathrm{gv}(U)$, then $N(T) \simeq_t N(U)$.*

Proof. (sketch). Item 1: a coinductive proof. Consider the relation

$$\mathcal{R} = \{([\alpha'/\alpha]T, [\alpha'/\alpha]U) \,:\, T \simeq U\}.$$

We can show that $\mathcal{R}$ is backward closed for the rules defining type equivalence, so that $\mathcal{R} \subseteq \simeq$. Therefore, $T \simeq U$ implies $([\alpha'/\alpha]T, [\alpha'/\alpha]U) \in \mathcal{R}$, which in turn implies $[\alpha'/\alpha]T \simeq [\alpha'/\alpha]U$. We omit the details.

Item 2: a coinductive proof. Consider the relation

$$\mathcal{R}' = \{(N(T), N(U)) \,:\, T, U \text{ are types, } N([\alpha'/\alpha]T) \simeq_t N([\alpha'/\alpha]U) \text{ and } \alpha' \notin \mathrm{gv}(T) \cup \mathrm{gv}(U)\}.$$

We can show that $\mathcal{R}'$ is backward closed for the rules defining term equivalence, so that $\mathcal{R}' \subseteq \simeq_t$. Therefore, $N([\alpha'/\alpha]T) \simeq_t N([\alpha'/\alpha]U)$ and $\alpha' \notin \mathrm{gv}(T) \cup \mathrm{gv}(U)$ implies $(N(T), N(U)) \in \mathcal{R}'$, which in turn implies $T \simeq_t U$. Given $(T, U) \in \mathcal{R}'$, so that $N([\alpha'/\alpha]T) \simeq_t N([\alpha'/\alpha]U)$ and $\alpha' \notin \mathrm{gv}(T) \cup \mathrm{gv}(U)$, we proceed by a case analysis on the last rule on the derivation of $N([\alpha'/\alpha]T) \simeq_t N([\alpha'/\alpha]U)$. We only show the one case in which the proviso $\alpha' \notin \mathrm{gv}(T) \cup \mathrm{gv}(U)$ is used, omitting the others.

(**Case** E-Var): then $N([\alpha'/\alpha]T) = \beta$ and $N([\alpha'/\alpha]U) = \beta$, for some variable β. Since $\alpha' \notin \mathrm{gv}(T) \cup \mathrm{gv}(U)$, it follows that T, U are both variables different from α'. Then, $N([\alpha'/\alpha]T) = N([\alpha'/\alpha]U)$ implies $[\alpha'/\alpha]T = [\alpha'/\alpha]U$, which implies $T = U$, which implies $N(T) = N(U)$. We can thus apply rule E-Var to the pair $(N(T), N(U))$. □

Theorem 16 (Fundamental theorem of canonical renaming)**.** *Let T, U be types. Then $T \simeq U$ iff $N(T) \simeq_t N(U)$.*

Proof. In the forward direction, consider the relation

$$\mathcal{R} = \{(N(T), N(U)) \,:\, T, U \text{ are types and } T \simeq U\}.$$

We show that $\mathcal{R}$ is closed under the rules defining $\simeq_t$, so that $\mathcal{R} \subseteq \simeq_t$ and thus $T \simeq U$ implies $N(T) \simeq_t N(U)$. Let $(N(T), N(U)) \in \mathcal{R}$ be given, so that $T \simeq U$. We proceed by a case analysis on the last rule of the derivation of $T \simeq U$. We only show three cases, as the others are similar.

- Suppose a derivation of $T \simeq U$ ends with rule E-Arrow. Then $T = T_1 \to T_2$ and $U = U_1 \to U_2$ for some types T_1, T_2, U_1, U_2 such that $T_1 \simeq U_1$ and $T_2 \simeq U_2$. Therefore, both pairs $(N(T_1), N(U_1)), (N(T_2), N(U_2)) \in \mathcal{R}$. By the definition of renaming, $N(T) = N(T_1) \to N(T_2)$ and $N(U) = N(U_1) \to N(U_2)$. Applying rule E-Arrow, we arrive at pairs $(N(T_1), N(U_1)), (N(T_2), N(U_2))$, which are in $\mathcal{R}$ by the previous observation.
- Suppose a derivation of $T \simeq U$ ends with rule E-IdL. Then $T = X(\overline{\beta})$ for some type identifier X with definition $X(\overline{\alpha}) \doteq T'$ for some T'. Moreover, T' contr and $[\overline{\beta}/\overline{\alpha}]T' \simeq U$, so that $(N([\overline{\beta}/\overline{\alpha}]T'), N(U)) \in \mathcal{R}$. By the definition of renaming, $N(T) = Y_{\overline{\beta}}$ where $Y_{\overline{\beta}}$ has definition $Y_{\overline{\beta}} \doteq N([\overline{\beta}/\overline{\alpha}]T')$. Therefore, we can apply rule E-IdL, arriving at pair $(N([\overline{\beta}/\overline{\alpha}]T'), N(U))$, which is in $\mathcal{R}$ by the previous observation.
- Suppose a derivation of $T \simeq U$ uses rule E-Quant. Then (possibly after some use of the variable convention) $T = \exists\forall\alpha\colon\kappa\,.\,T'$ and $U = \exists\forall\alpha\colon\kappa\,.\,U'$ for some α, T', U' such that $T' \simeq U'$. By lemma 14, $\mathrm{gv}(T) = \mathrm{gv}(U)$, so that $\mathrm{first}(T) = \mathrm{first}(U)$. Thus, the same variable is chosen in the canonical renamings: $N(T) = \alpha_i \exists\forall_\kappa N([\alpha_i/\alpha]T')$ and $N(U) = \alpha_i \exists\forall_\kappa N([\alpha_i/\alpha]U')$. By item 1 of lemma 15, $[\alpha_i/\alpha]T' \simeq [\alpha_i/\alpha]U'$, so that $(N([\alpha_i/\alpha]T'), N([\alpha_i/\alpha]U')) \in \mathcal{R}$. Applying rule E-Quant to pair $(N(T), N(U))$, we arrive at pair $(N([\alpha_i/\alpha]T'), N([\alpha_i/\alpha]U'))$, which is in $\mathcal{R}$ by the previous observation.

In the reverse direction, consider the relation

$$\mathcal{R} = \{(T, U) \,:\, T, U \text{ are types and } N(T) \simeq_t N(U)\}.$$

We show that $\mathcal{R}$ is closed under the rules defining $\simeq$, so that $\mathcal{R} \subseteq \simeq$ and thus $N(T) \simeq_t N(U)$ implies $T \simeq U$. Let $(T, U) \in \mathcal{R}$ be given, so that $N(T) \simeq_t N(U)$. We proceed by a case analysis on the derivation of $N(T) \simeq_t N(U)$. We only show three cases, as the others are similar.

- Suppose a derivation of $N(T) \simeq_t N(U)$ uses rule E-Arrow. Then $T = T_1 \to T_2$ and $U = U_1 \to U_2$ for some types T_1, T_2, U_1, U_2 such that $N(T) = N(T_1) \to N(T_2)$ and $N(U) = N(U_1) \to N(U_2)$. In particular we get $N(T_1) \simeq_t N(U_1)$ and $N(T_2) \simeq_t N(U_2)$, so that both pairs $(T_1, U_1), (T_2, U_2) \in \mathcal{R}$. These are precisely the pairs we arrive at when applying rule E-Arrow from the pair (T, U).

- Suppose a derivation of $N(T) \simeq_t N(U)$ uses rule E-IdL. Then $T = X(\overline{\beta})$ for some type identifier X with definition $X(\overline{\alpha}) \doteq T'$ for some T'. Moreover, $N(T) = Y_{\overline{\beta}}$ where $Y_{\overline{\beta}}$ has definition $Y_{\overline{\beta}} \doteq N([\overline{\beta}/\overline{\alpha}]T')$. Since rule E-IdL was used, we conclude that $N([\overline{\beta}/\overline{\alpha}]T') \simeq_t N(U)$, so that $([\overline{\beta}/\overline{\alpha}]T', U) \in \mathcal{R}$. This is precisely the pair we arrive at when applying rule E-IdL from the pair (T, U).

- Suppose a derivation of $N(T) \simeq_t N(U)$ uses rule E-Quant'. Then (possibly after some use of the variable convention) $T = \exists\forall\alpha\colon\kappa\,.\,T'$ and $U = \exists\forall\alpha\colon\kappa\,.\,U'$ for some α, T' and U'. Since rule E-Quant' was used, the renaming must have used the same variable for the bound variable change, that is, $\alpha_i = \mathrm{first}(\exists\forall\alpha\colon\kappa\,.\,T') = \mathrm{first}(\exists\forall\alpha\colon\kappa\,.\,U')$. Moreover, $N(T) = \alpha_i\, \exists\forall_\kappa\, N([\alpha_i/\alpha]T')$ and $N(U) = \alpha_i\, \exists\forall_\kappa\, N([\alpha_i/\alpha]U')$, and thus $N([\alpha_i/\alpha]T') \simeq_t N([\alpha_i/\alpha]U')$. By item 2 of lemma 15, given that $\alpha_i \notin \mathrm{gv}(T') \cup \mathrm{gv}(U')$, we have that $N(T') \simeq_t N(U')$, and thus $(T', U') \in \mathcal{R}$. This is precisely the pair we arrive at when applying rule E-Quant from the pair (T, U). □

5. Semantic Term Equivalence

Following Gay and Hole [19], we build on a term 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, 40] (recast as terms) and introduce labelled transitions for functional and higher-order terms. The definition of the labelled transition system (LTS) is in fig. 9.

We start with functional terms. Term unit transitions to skip via label unit. Function terms induce two transitions: one via label $\to_d$ to the domain of the function, the other via $\to_r$ to the range. Records and variants step to each component k via labels $\{\}_k$ and $\langle\rangle_k$, respectively. Polymorphic terms transition via the respective label, $\forall_{\alpha\colon\kappa}$ or $\exists_{\alpha\colon\kappa}$, to its body.

For session type constructors, choices follow the original proposal [40] and step via $\oplus_k$ and $\&_k$ to the continuation term, for each label k in the choice. However, message exchanges for higher-order terms now feature two distinct transitions: one to the terms exchanged in the message (via label $!_d$, d for data) and the other to the continuation term (via label $!_c$, c for continuation). Term skip does not exhibit any transition. Variables α transition by label α to term skip (rule L-Var) and term identifiers inherit the transitions from the associated term (rule L-Id). Finally, sequential composition $T;U$ distinguishes cases for all term constructors in T (rules L-SkipSeq to L-IdSeq).

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

A bisimulation is defined in the usual way from the labelled transition system [37]. We say that a term relation $\mathcal{R}$ is a *bisimulation* if for all $(T, U) \in \mathcal{R}$ 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 $(T', U') \in \mathcal{R}$, and
2. for each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ and $(T', U') \in \mathcal{R}$.

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

We can easily see that the type for a function send that first receives the type of the message, then receives the value to be sent and only afterwards receives the type for the continuation, that is, $\forall\alpha\colon \mathrm{T}.\alpha \to \forall\beta\colon \mathrm{S}.!\alpha;\beta \to \beta$, is not equivalent to the type of a function send′ that starts by receiving the types for value to be exchanged and for the continuation channel, $\forall\alpha\colon \mathrm{T}.\forall\beta\colon \mathrm{S}.\alpha \to !\alpha;\beta \to \beta$. The corresponding terms are $\alpha_1\, \forall_{\mathrm{T}}\, \alpha_1 \to \alpha_2\, \forall_{\mathrm{S}}\, !\alpha_1;\alpha_2 \to \alpha_2$ and $\alpha_1\, \forall_{\mathrm{T}}\, \alpha_2\, \forall_{\mathrm{S}}\, \alpha_1 \to !\alpha_1;\alpha_2 \to \alpha_2$ (after canonical renaming). These terms are not equivalent because, even if both exhibit a transition by label $\forall_{\alpha_1\colon \mathrm{T}}$, the first term then exhibits transitions by labels $\to_d, \to_r$, whereas the second term

Transition labels

$$a \;::=\; \mathsf{unit} \mid \to_d \mid \to_r \mid (\!|)_\ell \mid \exists\forall_{\alpha:\kappa} \mid \sharp_d \mid \sharp_c \mid \odot_\ell \mid n$$

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

L-Unit
$\mathsf{unit} \xrightarrow{\mathsf{unit}} \mathsf{skip}$

L-Arrow1
$T \to U \xrightarrow{\to_d} T$

L-Arrow2
$T \to U \xrightarrow{\to_r} U$

L-Rcd
$$\frac{k \in L}{(\!|\ell : T_\ell|\!)_{\ell\in L} \xrightarrow{(\!|)_k} T_k}$$

L-Quant
$\alpha\, \exists\forall_\kappa\, T \xrightarrow{\exists\forall_{\alpha:\kappa}} T$

L-Msg1
$\sharp T \xrightarrow{\sharp_d} T$

L-Msg2
$\sharp T \xrightarrow{\sharp_c} \mathsf{skip}$

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

L-Var
$\alpha \xrightarrow{\alpha} \mathsf{skip}$

L-Id
$$\frac{Y \doteq T \quad T \xrightarrow{a} U}{Y \xrightarrow{a} U}$$

L-SkipSeq
$$\frac{T \xrightarrow{a} U}{\mathsf{skip}; T \xrightarrow{a} U}$$

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-VarSeq
$\alpha; U \xrightarrow{\alpha} U$

L-IdSeq
$$\frac{Y \doteq T \quad T; U \xrightarrow{a} V}{Y; U \xrightarrow{a} V}$$

Figure 9: Labelled transition system for terms.

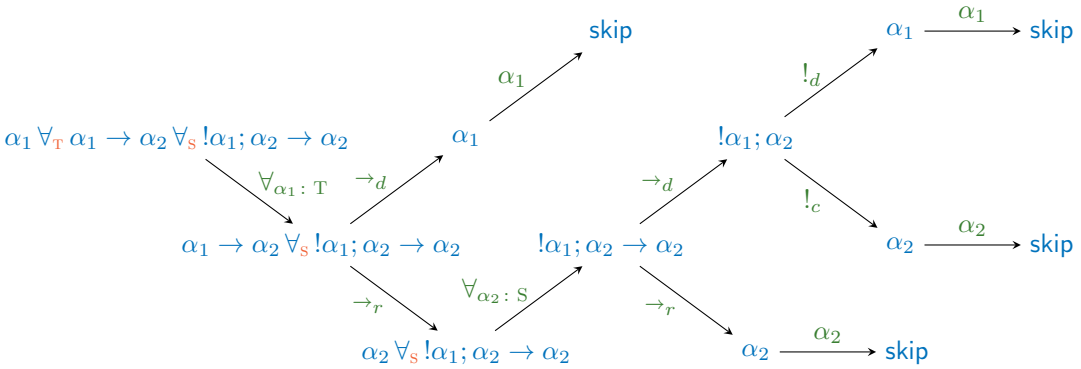


Figure 10: Fragment of the LTS generated by term $\alpha_1 \forall_{\mathrm{T}} \alpha_1 \to \alpha_2 \forall_{\mathrm{S}} !\alpha_1; \alpha_2 \to \alpha_2$.

then exhibits a transition by $\forall_{\alpha_2:\mathrm{S}}$. The LTS for the former term is in fig. 10; we leave to the reader drawing the LTS for the latter term.

Term bisimulation is deterministic (hence image-finite) and finitely branching. It features infinite transition sequences, as well as transition sequences that visit infinitely many different states [40].

Lemma 17.

1. *Let* T *be a term and* $T' = \mathrm{unravel}_1(T)$.
   (a) $T \xrightarrow{a} U$ *iff* $T' \xrightarrow{a} U$.
   (b) $T \sim U$ *iff* $T' \sim U$.
2. *Let* T *be a term and* $T' = \mathrm{unravel}(T)$.
   (a) $T \xrightarrow{a} U$ *iff* $T' \xrightarrow{a} U$.
   (b) $T \sim U$ *iff* $T' \sim U$.

Proof. Item 1a is immediate by inspection of the LTS rules. Item 1b follows from item 1a since T, T' have the same transitions. Item 2 follows from item 1 since unravel(T) is reached in a finite number of steps when defined. □

Terminated terms are bisimilar to skip (cf. lemma 9).

Lemma 18. *For any term T, $T{\checkmark}$ iff $T \sim \mathsf{skip}$.*

Proof. First consider the case that $T = \text{unravel}(T)$. By inspection of the possibilities of T (as per proposition 5), $T \sim \mathsf{skip}$ iff $T = \mathsf{skip}$ iff $T{\checkmark}$. Now suppose that $T \neq \text{unravel}(T) = T'$. Applying lemmas 7 and 17 as well as the previous case, we conclude that $T \sim \mathsf{skip}$ iff $T' \sim \mathsf{skip}$ iff $T'{\checkmark}$ iff $T{\checkmark}$. □

Theorem 19 (Soundness and completeness)**.** *Let T, U be terms. Then $T \simeq_t U$ iff $T \sim U$.*

Proof. In Appendix D. □

Given that bisimulation is an equivalence relation, theorem 19 together with theorem 16 give an alternative proof for theorem 11.

6. Bisimulation for Simple Grammars

A grammar is given by a tuple $(\mathcal{T}, \mathcal{N}, \gamma, \mathcal{P})$ where $\mathcal{T}$ is a set of terminal symbols, denoted by a, b, c, $\mathcal{N}$ is a set of nonterminal symbols, denoted by X, Y, Z, $\gamma \in \mathcal{N}^*$ is the starting word and $\mathcal{P} \subseteq \mathcal{N} \times (\mathcal{T} \cup \mathcal{N})^*$ 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 [22]. A grammar is in Greibach normal form if $\mathcal{P} \subseteq \mathcal{N} \times \mathcal{T} \times \mathcal{N}^*$, in other words, if every production is of the form $X \to a\gamma$. A grammar in Greibach normal form is said to be simple [29] 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 $\mathcal{N}^*$ 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 goal is to explain how to convert a term T into a simple grammar $(\mathcal{T}_T, \mathcal{N}_T, \text{word}(T), \mathcal{P}_T)$. This is done via a function word(T) that maps each term T into a word of nonterminal symbols. In the following, let $\bot$ denote a nonterminal symbol with no productions. word(T) is coinductively defined on T as follows.

- word(unit) $= X$ for a fresh nonterminal symbol X with a production $X \xrightarrow{\mathsf{unit}} \varepsilon$;
- word($T \to U$) $= X$ for a fresh symbol X with productions $X \xrightarrow{\to_d}$ word(T), $X \xrightarrow{\to_r}$ word(U);
- word($(\!|\ell\colon T_\ell|\!)_{\ell \in L}$) $= X$ for a fresh symbol X with productions $X \xrightarrow{(\!|\!|)_k}$ word(T_k) for each $k \in L$;
- word($\alpha\, \exists\forall_\kappa\, T$) $= X$ for a fresh nonterminal symbol X with a production $X \xrightarrow{\exists\forall_{\alpha:\kappa}}$ word(T);
- word(skip) $= \varepsilon$;
- word($\sharp T$) $= X$ for a fresh nonterminal symbol X with productions $X \xrightarrow{\sharp_d}$ word(T)$\bot$, $X \xrightarrow{\sharp_c} \varepsilon$;
- word($\odot\{\ell\colon T_\ell\}_{\ell \in L}$) $= X$ for a fresh X with productions $X \xrightarrow{\odot_k}$ word(T_k) for each $k \in L$;
- word(α) $= X$ for a fresh nonterminal symbol X with a production $X \xrightarrow{\alpha} \varepsilon$;
- word($T; U$) $=$ word(T) word(U);

- if $Y\checkmark$, then $\mathrm{word}(Y) = \varepsilon$;
- if $Y \not\checkmark$, then $\mathrm{word}(Y) = X$ for a fresh nonterminal symbol X. Let $Y \doteq T$ be the equation corresponding to the identifier Y and let $Z\delta = \mathrm{word}(T)$. Then X has productions $X \xrightarrow{a} \gamma\delta$ for each production $Z \xrightarrow{a} \gamma$.

In the above construction, we create fresh symbols each time we encounter a non-terminated term constructor or a non-terminated term identifier. In other words, $\mathcal{N}_T$ is the set containing $\bot$ as well as all nonterminals X created during the computation of $\mathrm{word}(T)$. Similarly, $\mathcal{T}_T$ and $\mathcal{R}_T$ are the sets of terminals and productions created in the process.

Recall that by lemma 18 $Y\checkmark$ iff $Y \sim \mathsf{skip}$; this is why we desire $\mathrm{word}(Y)$ and $\mathrm{word}(\mathsf{skip})$ to be bisimilar in such cases. For the last case in our construction to be well-defined, we require $\mathrm{word}(\mathrm{unravel}(Y))$ to be non-empty whenever $Y \not\checkmark$, which is a consequence of the following result.

Lemma 20. *$T\checkmark$ iff $\mathrm{word}(T) = \varepsilon$.*

Proof. By induction on the size of a (successful or unsuccesful) derivation for $T\checkmark$. We follow a case analysis on T.

(**Case** $T = \mathsf{skip}$): then $T\checkmark$ (rule ✓-Skip) and $\mathrm{word}(\mathsf{skip}) = \varepsilon$.

(**Case** $T = Y$): by construction, $\mathrm{word}(Y) = \varepsilon$ iff $Y\checkmark$.

(**Cases** $T = \mathsf{unit}, U \to V, (\!|\ell\colon U_\ell|\!)_{\ell\in L}, \alpha\,\exists\forall_\kappa\, U, \sharp U, \odot\{\ell\colon U_\ell\}_{\ell\in L}, \alpha$): then $T \not\checkmark$ and $\mathrm{word}(\mathsf{skip}) = X \neq \varepsilon$ for some nonterminal X.

(**Case** $T = U;V$): then $T\checkmark$ iff $U\checkmark$ and $V\checkmark$. We also have that $\mathrm{word}(T) = \mathrm{word}(U)\,\mathrm{word}(V)$, so $\mathrm{word}(T) = \varepsilon$ iff $\mathrm{word}(U) = \varepsilon$ and $\mathrm{word}(V) = \varepsilon$. By induction hypothesis, we know that $U\checkmark$ iff $\mathrm{word}(U) = \varepsilon$ and $V\checkmark$ iff $\mathrm{word}(V) = \varepsilon$. Therefore, $T\checkmark$ iff $\mathrm{word}(T) = \varepsilon$. □

We also need to argue that the construction of $\mathrm{word}(T)$ eventually terminates. In particular, we need to argue that only finitely many nonterminal symbols are created. To achieve this, we should only create a fresh symbol X for a term identifier Y the first time that we see it. On subsequent visits to Y (say, if it appears on the right-hand side of an equational definition) we reuse the same symbol X with the same productions. In practice, we can go one step beyond and keep track of all terms visited during the construction. We add a fresh nonterminal X to our grammar only if the term visited is syntactically different from all terms visited so far. This approach reduces the number of fresh nonterminals required. Finally, notice that the number of symbols in our grammar is bound by the number of (syntactically different) subterms of T and $\mathrm{unravel}(Y)$ for each term identifier Y in our signature, which is linear on the size of T and Σ. Therefore, the construction can be implemented in polynomial time.

The above construction introduces a nonterminal symbol $\bot$ without productions. Intuitively, $\bot$ is used to separate the two descendants of a send/receive operation. A term $!T;U$ 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 term $!\mathsf{skip};!\mathsf{skip}$ sends two (empty) channels in sequence, whereas $!(\mathsf{skip};!\mathsf{skip})$ 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

$$T \doteq !V;W \qquad U \doteq !(V;V);W \qquad V \doteq \oplus\{\mathsf{Go}\colon \mathsf{skip}\} \qquad W \doteq \oplus\{\mathsf{Go}\colon W\}$$

Notice that $T \not\approx U$, as the type being sent in T offers a choice only once, whereas the type being sent in U offers that choice twice. Further note that there is no polymorphism in this example, so $N(T) = T$ and $N(U) = U$, i.e., the terms and types are the same. Following the construction above, we arrive at the grammar with productions $X_T \xrightarrow{!_d} X_V \bot X_W$, $X_T \xrightarrow{!_c} X_W$, $X_U \xrightarrow{!_d} X_V X_V \bot X_W$, $X_U \xrightarrow{!_c} X_W$, $X_V \xrightarrow{\oplus_{\mathsf{Go}}} \varepsilon$, $X_W \xrightarrow{\oplus_{\mathsf{Go}}} X_W$. Now we can check that $X_T \not\approx X_U$, as

$$X_T \xrightarrow{!_d} X_V \bot X_W \xrightarrow{\oplus_{\mathsf{Go}}} \bot X_W \not\longrightarrow \quad \text{but} \quad X_U \xrightarrow{!_d} X_V X_V \bot X_W \xrightarrow{\oplus_{\mathsf{Go}}} X_V \bot X_W \xrightarrow{\oplus_{\mathsf{Go}}} \bot X_W$$

Suppose instead that we drop the nonterminal symbol $\bot$. 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 term, we

would undesirably conclude that $X_T \approx X_U$; in particular, we would have the infinite sequences of transitions

$$X_T \xrightarrow{!_d} X_V X_W \xrightarrow{\oplus_{\mathbb{G}_0}} X_W \xrightarrow{\oplus_{\mathbb{G}_0}} X_W \xrightarrow{\oplus_{\mathbb{G}_0}} \cdots \quad \text{and} \quad X_U \xrightarrow{!_d} X_V X_V X_W \xrightarrow{\oplus_{\mathbb{G}_0}} X_V X_W \xrightarrow{\oplus_{\mathbb{G}_0}} X_W \xrightarrow{\oplus_{\mathbb{G}_0}} \cdots.$$

The main goal of this section is to prove soundness and completeness for grammars (theorem 22). Grammar bisimulation $\gamma \approx \gamma'$ corresponds to bisimulation of the LTS associated to words γ, γ'. Theorem 19 shows that term equivalence $T \simeq U$ corresponds to bisimulation of the LTS associated to types T, U. The next technical lemma shows that the LTS of a term and the LTS of the corresponding word of nonterminals coincide.

Lemma 21. *Let T be a term and $(\mathcal{T}_T, \mathcal{N}_T, \text{word}(T), \mathcal{P}_T)$ the corresponding simple grammar. Let γ be a word such that $\text{word}(T) \approx \gamma$. Then*

- *If $T \xrightarrow{a} U$ for some U, then there exists γ' such that $\gamma \xrightarrow{a} \gamma'$ and $\text{word}(U) \approx \gamma'$.*
- *If $\gamma \xrightarrow{a} \gamma'$ for some γ', then there exists U such that $T \xrightarrow{a} U$ and $\text{word}(U) \approx \gamma'$.*

Proof. In Appendix E. □

Our next result shows that two terms T, U are bisimilar iff their corresponding words are bisimilar. Here (as well as in the algorithm of section 7 below) there is a fine detail concerning the conversion to simple grammars. For simplicity, and for the results that follow, we assume that the sets of nonterminals $\mathcal{N}_T$, $\mathcal{N}_U$ built during the construction of $\text{word}(T)$, $\text{word}(U)$ are disjoint. In this way we can generate both grammars separately (even potentially in parallel), there is no overlapping of nonterminal symbols, and we can trivially fuse the two simple grammars generated.

Alternatively (and, arguably, more efficiently), we could assume that T, U are defined over a common signature Σ, i.e., if a term identifier Y appears in both T and U, it is specified by the same equation $Y \doteq V$ in both T and U (this assumption might require a preprocessing stage in which we rename type identifiers to obtain a common signature). With this approach, we can generate a simple grammar sequentially, in two stages. In the first stage, construct the simple grammar $(\mathcal{T}_T, \mathcal{N}_T, \text{word}(T), \mathcal{P}_T)$ corresponding to T. In the second stage, construct $\text{word}(U)$ using the already constructed grammar as a starting point. In other words, only add fresh nonterminals whenever syntactically different subterms are encountered. The construction then yields a simple grammar $(\mathcal{T}_{T,U}, \mathcal{N}_{T,U}, \text{word}(U), \mathcal{P}_{T,U})$ which subsumes the simple grammar constructed in the first stage. Although we do not follow this incremental approach, we remark that it has been proposed in previous work [4] and it is the one being implemented in the FreeST language [2].

Theorem 22 (Soundness and completeness for grammars)**.** *Let T, U be terms and $(\mathcal{T}_T, \mathcal{N}_T, \text{word}(T), \mathcal{P}_T)$, $(\mathcal{T}_U, \mathcal{N}_U, \text{word}(T), \mathcal{P}_U)$ the corresponding simple grammars. Then $T \sim U$ iff $\text{word}(T) \approx \text{word}(U)$.*

Proof. For the left-to-right implication, let $\mathcal{N} = \mathcal{N}_T \cup \mathcal{N}_U$ and consider the relation on $\mathcal{N}^*$

$$\mathcal{R} = \{(\gamma, \delta) : \text{ there exist terms } V, W \text{ s.t. } V \sim W, \text{word}(V) \approx \gamma, \text{word}(W) \approx \delta\}.$$

Let us show that $\mathcal{R}$ is a bisimulation. Take $(\gamma, \delta) \in \mathcal{R}$ and V, W such that $V \sim W$, $\text{word}(V) \approx \gamma$, $\text{word}(W) \approx \delta$.

First suppose there exist γ' and a transition $\gamma \xrightarrow{a} \gamma'$. Since $\text{word}(V) \approx \gamma$ and by lemma 21, there exists a type V' such that $V \xrightarrow{a} V'$ and $\text{word}(V') \approx \gamma'$. Since $V \sim W$, there exists a matching type W' such that $W \xrightarrow{a} W'$ and $V' \sim W'$. Again since $\text{word}(W) \approx \delta$ and by lemma 21, there exists δ' such that $\delta \xrightarrow{a} \delta'$ and $\text{word}(W') \approx \delta'$. Putting all these together, we obtain $(\gamma', \delta') \in \mathcal{R}$ as desired.

Next suppose there exists δ' and a transition $\delta \xrightarrow{a} \delta'$. By the same reasoning we can conclude that there exist γ' and a matching transition $\gamma \xrightarrow{a} \gamma'$, and moreover $(\gamma', \delta') \in \mathcal{R}$.

This concludes that $\mathcal{R}$ is a bisimulation, so that $\mathcal{R} \subseteq \approx$. Now, if $T \sim U$ then trivially $(\text{word}(T), \text{word}(U)) \in \mathcal{R}$ as, by reflexivity, $\text{word}(T) \approx \text{word}(T)$ and $\text{word}(U) \approx \text{word}(U)$. Therefore $\text{word}(T) \approx \text{word}(U)$.

For the right-to-left implication, consider the relation on pairs of types

$$\mathcal{R}' = \{(V, W) : \text{word}(V) \approx \text{word}(W)\}.$$

A similar argument as before shows that $\mathcal{R}'$ is a bisimulation, so that $\mathcal{R}' \subseteq \sim$. Now, if $\text{word}(T) \approx \text{word}(U)$ then $(T, U) \in \mathcal{R}'$ and therefore $T \sim U$. □

7. 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 canonical renaming defined in section 4, the conversion to simple grammars outlined in section 6, 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 10 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 terms $N(T)$ and $N(U)$.
2. Construct the simple grammar $(\mathcal{T}_T, \mathcal{N}_T, \text{word}(N(T)), \mathcal{P}_T)$ corresponding to $N(T)$.
3. Construct the simple grammar $(\mathcal{T}_U, \mathcal{N}_U, \text{word}(N(U)), \mathcal{P}_U)$ corresponding to $N(U)$.
4. Use a decision algorithm to decide whether $\text{word}(N(T)) \approx \text{word}(N(U))$.

In the above algorithm, we assume for simplicity that the simple grammars in steps 2 and 3 are constructed separately (possibly even in parallel) and produce disjoint sets of nonterminal symbols. See also the discussion immediately preceding theorem 22.

Theorem 23. *The type equivalence algorithm terminates. Its computational complexity is triple exponential.*

Proof. Let $n = \max(\text{size}(T, \Sigma_T), \text{size}(U, \Sigma_U))$. Step 1 of the algorithm terminates in time $\ell = 2^{\mathcal{O}(n \log n)}$ due to proposition 13. Steps 2 and 3 terminate in time polynomial in ℓ due to the discussion in section 6. Step 4 has the worst complexity bound, since it is double exponential ($2^{2^{\ell^{\mathcal{O}(1)}}}$ [27]) on the size of the grammars obtained in Steps 1 and 2, which in turn is exponential on the input size. Therefore, the entire algorithm has triple exponential running time, $2^{2^{2^{\mathcal{O}(n \log n)}}}$. □

Theorem 24. *The type equivalence algorithm is sound and complete with respect to* $\simeq$.

Proof. Let T, U be types and $\text{word}(N(T))$, $\text{word}(N(U))$ the corresponding starting nonterminals as described above. From theorem 16 we know that $T \simeq U$ iff $N(T) \simeq_t N(U)$. From theorem 19 we know that $N(T) \simeq_t N(U)$ iff $N(T) \sim N(U)$. From theorem 22 we know that $N(T) \sim N(U)$ iff $\text{word}(N(T)) \approx \text{word}(N(U))$. From the soundness and completeness of the Almeida et al. [4] algorithm, we know that Step 4 of our type equivalence algorithm correctly determines whether $\text{word}(N(T)) \approx \text{word}(N(U))$. Therefore, $T \simeq U$ iff the type equivalence algorithm return 'YES'. □

Corollary 25. *The type equivalence problem is decidable.*

Proof. Direct consequence of theorems 23 and 24. □

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

$$\textsc{K-Unit}\ \ \Delta \vdash \mathsf{unit} : \mathrm{T}^{\mathsf{un}} \qquad \textsc{K-Arrow}\ \ \frac{\Delta \vdash T : \mathrm{T}^{m} \quad \Delta \vdash V : \mathrm{T}^{m}}{\Delta \vdash T \to_{m} V : \mathrm{T}^{m}} \qquad \textsc{K-Rcd}\ \ \frac{\Delta \vdash T_\ell : \mathrm{T}^{m} \quad (\forall \ell \in L)}{\Delta \vdash (\!|\ell\colon T_\ell|\!)_{\ell \in L} : \mathrm{T}^{m}} \qquad \textsc{K-Quant}\ \ \frac{\Delta + \alpha\colon \kappa \vdash T : \mathrm{T}^{m}}{\Delta \vdash \forall \alpha\colon \kappa\,.\,T : \mathrm{T}^{m}}$$

$$\textsc{K-Skip}\ \ \Delta \vdash \mathsf{skip} : \mathrm{S}^{\mathsf{un}} \qquad \textsc{K-Msg}\ \ \frac{\Delta \vdash T : \mathrm{T}^{\mathsf{lin}}}{\Delta \vdash \sharp T : \mathrm{S}^{\mathsf{lin}}} \qquad \textsc{K-Choice}\ \ \frac{\Delta \vdash T_\ell : \mathrm{S}^{\mathsf{lin}} \quad (\forall \ell \in L)}{\Delta \vdash \odot\{\ell\colon T_\ell\}_{\ell \in L} : \mathrm{S}^{\mathsf{lin}}} \qquad \textsc{K-Seq}\ \ \frac{\Delta \vdash T : \mathrm{S}^{m} \quad \Delta \vdash U : \mathrm{S}^{m}}{\Delta \vdash T; U : \mathrm{S}^{m}}$$

$$\textsc{K-Var}\ \ \frac{\alpha\colon \kappa \in \Delta}{\Delta \vdash \alpha : \kappa} \qquad \textsc{K-Id}\ \ \frac{X(\overline{\alpha}) \doteq T \quad T\ \mathrm{contr} \quad \overline{\alpha} : \overline{\kappa} \vdash T : \kappa}{\Delta, \overline{\beta} : \overline{\kappa} \vdash X(\overline{\beta}) : \kappa} \qquad \textsc{K-Sub}\ \ \frac{\Delta \vdash T : \kappa \quad \kappa <\colon \kappa'}{\Delta \vdash T : \kappa'}$$

Figure 11: Type formation with subkinding and multiplicities (replaces the rules in fig. 3).

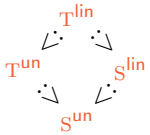
8. Subkinding and Type Multiplicities

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]. Likewise, types such as arrows, records or variants can be tagged as linear, lin, or unrestricted, un, depending on their intended usage (two alternative multiplicities). The intuitive meaning is that a linear resource can be used exactly once whereas an unrestricted resource can be used zero or more times.

In this section, we investigate how to incorporate subkinding and multiplicities in the present system. We focus on the changes, as all of the arguments and results seen throughout the paper generalize. First of all, there are now four different kinds, corresponding to the possible combinations between session / functional and linear / unrestricted. We use m to denote a generic multiplicity.

$$m \ ::=\ \mathsf{un} \mid \mathsf{lin} \qquad \kappa \ ::=\ \mathrm{S}^{m} \mid \mathrm{T}^{m}$$

We define a subkinding preorder generated by the inequalities $\mathrm{S} <: \mathrm{T}$ and $\mathsf{un} <: \mathsf{lin}$, i.e., described by the diamond on the right. Our next modification is on the syntax of arrows. In order to distinguish between unrestricted and linear functions, we replace the constructor $T \to U$ by two constructors $T \to_{\mathsf{lin}} U$, $T \to_{\mathsf{un}} U$. In the spirit of Almeida et al. [1], we explicitly annotate function types but not record or variant types. The rules for termination ($T\checkmark$) remain as before, and the only change in the rules for contractivity is to modify C-Axiom in accordance with the two new arrows.



$$\text{Replace}\quad \textsc{C-Axiom}\ \ T \to U\ \mathrm{contr} \quad \text{by} \quad \textsc{C-Axiom}\ \ T \to_{m} U\ \mathrm{contr}$$

The rules for type formation (fig. 3) require substantive changes. The modified rules are in fig. 11. We introduce *kind subsumption* as a new rule, K-Sub, which says that a type of a subkind can be used in any context where the superkind is expected. For the rules that allowed the kind of the antecedents to change (K-Arrow, K-Rcd, K-Quant, K-Msg) we now use the most general kind T. Types unit and skip are unrestricted, whereas messages and choices are linear; for the other cases, the multiplicity of a kind is the meet of its subterms. The only types of kind S^{un} are thus terminated ones, such as skip and skip; skip. Rule K-Sub allows us to "downgrade" a kind as necessary; fig. 12 illustrates a derivation.

Continuing with the rest of our construction, the rules for type equivalence remain essentially the same. The only change is for the equivalence of arrows.

$$\text{Replace}\quad \textsc{E-Arrow}\ \ \frac{T \simeq U \quad V \simeq W}{T \to V \simeq U \to W} \quad \text{by} \quad \textsc{E-Arrow}\ \ \frac{T \simeq U \quad V \simeq W}{T \to_{m} V \simeq U \to_{m} W}$$

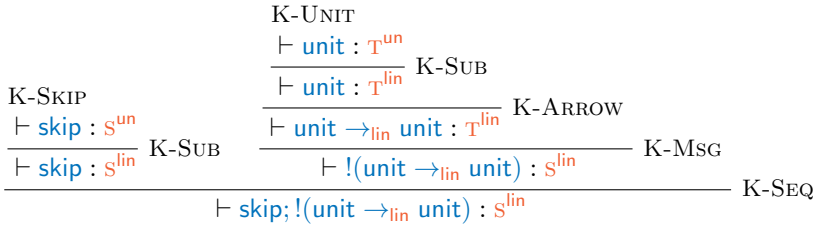


Figure 12: Example of a successful derivation of $\vdash \mathsf{skip}; !(\mathsf{unit} \to_{\mathsf{lin}} \mathsf{unit}) : \mathrm{S}^{\mathsf{lin}}$.

Similarly, the LTS associated with a term now needs to distinguish between linear and unrestricted functions. We have new transition labels $a ::= \cdots \mid \to_{md} \mid \to_{mr}$ and new transitions.

	L-Arrow1	L-Arrow2		L-Arrow1	L-Arrow2
Replace	$T \to U \xrightarrow{\to_d} T$	$T \to U \xrightarrow{\to_r} U$	by	$T \to_m U \xrightarrow{\to_{md}} T$	$T \to_m U \xrightarrow{\to_{mr}} U$

Finally, the grammar conversion can be adapted with new productions for functions.

- $\mathrm{word}(T \to_m U) = Y$ for a fresh Y with $Y \xrightarrow{\to_{md}} \mathrm{word}(T)$, $Y \xrightarrow{\to_{mr}} \mathrm{word}(U)$;

It should be clear that the above arguments and proofs have a straightforward extension to the setting of subkinding and multiplicities, with the changes described in this section. In particular, all four notions of equivalence (type-level, syntactic rule-based; term-level, syntactic rule-based; term-level, semantic bisimulation-based, and simple grammar-level, algorithmic-based) still coincide and a type equivalence algorithm can still be implemented by reduction to the bisimilarity of simple grammars.

9. Alternative Representations of Infinite Types

In this paper we study a fairly general of class of infinite types: those corresponding to equations with parameterized type identifiers. In this section, we look back at our syntax choices and investigate whether the techniques here developed can be applied to alternative representations of infinite types. We discuss μ-types and equations with nameless variables.

μ-types. Under the μ-notation, a type such as $\forall\alpha.\mu\, Y.(\alpha \to \forall\alpha.Y)$ expands to

$$\forall\alpha.(\alpha \to \forall\beta_1.(\alpha \to \forall\beta_2.(\alpha \to \forall\beta_3.\cdots))),$$

because the second quantifier, $\forall\alpha$, does not bind any free occurence of α, since α does not occur at all in type Y. This requires us to perform an on-the-fly change of bound variable during μ-unfolding. Without parameterizing type variables, the μ-notation restricts the universe of types; in particular, there is no way to represent the infinite type from section 1

$$T \triangleq \forall\alpha_1.\forall\beta_1.(\alpha_1 \to \forall\alpha_2.(\beta_1 \to \forall\beta_2.(\alpha_2 \to \forall\alpha_3.(\beta_2 \to \cdots)))).$$

In whichever way we try to split this type to introduce a μ, we will always separate some type variable from its binding position in a quantifier.

For μ-types with conventional (capture-avoiding) unfolding we can still use the tools present in this paper, provided we first perform a suitable renaming of the bound variables, as the example $\forall\alpha.\mu\, Y.(\alpha \to \forall\alpha.Y)$ suggests. Given a μ-type we start by renaming it in such a way that bound variables become distinct from free variables. In this case, our type becomes $\forall\alpha.\mu\, Y.(\alpha \to \forall\beta.Y)$. Afterwards, we can rewrite a type $\mu\, X.T$ as $X(\overline{\alpha})$, adding an equation $X(\overline{\alpha}) \doteq T$, where $\overline{\alpha}$ are the variables occurring free in T. In our example, we would obtain the equation $Y(\alpha) \doteq \alpha \to \forall\beta.Y(\alpha)$. In this way, we can convert a μ-type into a system of equations as those used in this paper. The conclusion is that we can still convert μ-types into simple grammars and use the algorithmic approach described here to decide type equivalence.

Equations with nameless variables. The main challenge with nameless variables is obtaining the expressivity of named variables, while still allowing the conversion of types into simple grammars. Let us look again at the infinite type U (from section 1) and its conversion to De Bruijn indices.

$$U \triangleq \forall\alpha_1.\forall\beta_1.\alpha_1 \to \forall\beta_2.\alpha_1 \to \forall\beta_3.\alpha_1 \to \forall\beta_4.\alpha_1 \to \cdots \quad \equiv \quad \forall.\forall.1 \to \forall.2 \to \forall.3 \to \forall.4 \to \cdots$$

The infinite type U can be represented in our model as $\forall\alpha.Y(\alpha)$ with $Y(\alpha) \doteq \forall\beta.\alpha \to Y(\alpha)$. However, its De Bruijn representation uses infinitely many indices, hence its grammar representation requires infinitely many terminals (one for each index variable $1, 2, 3, \ldots$). This means that simple grammars (which by definition must have finitely many symbols) are not expressive enough to represent the De Bruijn conversion of the infinite type U.

We sketch two approaches to address this problem. The simplest one is to just admit that nameless representations are less expressive. Under this approach, we could still apply the techniques in this paper: the resulting system would be less expressive, but could still be captured with simple grammars, and we would still be able to derive type equivalence algorithms.

The second approach would be to somehow extend the representation model beyond that of simple grammars. In this case we note that the infinite terminal symbols corresponding to indices are endowed with an additional structure given by the natural numbers. Perhaps one could devise a model combining simple grammars and a 'memory counter' that only increases, keeping track of the number of nested quantifiers as we traverse the type. So, for example, an equation $X \doteq 0 \to \forall X$ could be considered to expand into the type $0 \to \forall\ (1 \to \forall\ (2 \to \forall\ \cdots))$ as desired. It would be interesting to study exactly which computing model is obtained with this approach.

10. Related Work

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

Beyond Regular Session Types. Since its original proposal in the 90s [25, 26, 39], the theory of session types has evolved substantially. The interest in non-regular protocols was already apparent [34, 35, 38] when Thiemann and Vasconcelos proposed context-free session types as a way to specify non-regular communication protocols [40]. 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 [31] 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. [20] analyse different shades of session types that go beyond regular types.

Polymorphic Session Types. There has been a myriad of attempts to integrate polymorphic types into session types: from bounded polymorphism [18], to parametric and bounded polymorphism without recursion [14] or with recursion but without polymorphism [13]. Wadler proposed the inclusion of explicit polymorphism [41], which was then considered with parametric polymorphism but without general recursion [9], and afterwards with recursion but without nested types [23]. Finally, Das et al. proposed parametric polymorphism with nested types [15, 16]. We put forward an extension of System F 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]. The conference version of this paper [11] uses De Bruijn indices [17]. We have however found that such a representation of variables is somewhat restrictive, as explained in section 9. For this reason we decided to call variables by their names and avoid variable indices. We have also chosen to use equations instead of conventional μ-types, since our type identifiers may be parameterized by type variables.

Another challenge 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, Dual is a type constructor that, when given a session type, yields the dual session type. In recent work [33] we

studied the integration of higher-order context-free session types into an extension of the system F^{μ}_{ω} studied by Cai et al. [8]. In that work we were also able to internalise the Dual operator, which provides a cleaner approach for handling duality rather than treating it as an external macro.

Semantic vs Syntactic Approaches to Type Equivalence. The syntactic method is the most common approach to type equivalence [32]. The first paper on sessions and recursion uses implicit equirecursive types, so that one may classify type equivalence as a semantic notion [26]. An explicit notion of type equivalence for session types (actually of subtyping) was proposed by Gay and Hole, making use of a bisimulation [19]. 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 [15, 40]. 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. [10]. 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 [27]. For BPA processes, the bisimilarity problem is known to be EXPTIME-hard [28]; 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 [24].

11. 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.

The polymorphic types we manipulate are functional, that is, type $\forall\alpha\colon\kappa.T$ is of kind T. As such, type HTree with $\mathsf{HTree} \doteq \oplus\{\mathsf{Node}\colon \mathsf{HTree}; \forall\alpha\colon \mathrm{T}.!\alpha; \mathsf{HTree}, \mathsf{Leaf}\colon \mathsf{skip}\}$, 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.

Acknowledgements. Support for this research was provided by the Fundação para a Ciência e a Tecnologia through project SafeSessions, ref. PTDC/CCI-COM/6453/2020, by the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020, and by the COST Action CA20111. We also thank Luís Caires for his suggestions and comments on an early version of this paper.

References

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

[2] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2019): *FreeST, a Programming Language with Context-free Session Types.* `https://freest-lang.github.io/`.

[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] Yufei Cai, Paolo G. Giarrusso & Klaus Ostermann (2016): *System F-omega with equirecursive types for datatype-generic programming.* In: *POPL*, ACM, pp. 30–43, doi:10.1145/2837614.2837660.

[9] 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.

[10] 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.

[11] Diana Costa, Andreia Mordido, Diogo Poças & Vasco T. Vasconcelos (2022): *Higher-order Context-free Session Types in System F. EPTCS* 356, pp. 24–35, doi:10.4204/EPTCS.356.3.

[12] Haskell B. Curry, Robert Feys & William Craig (1959): *Combinatory Logic, Volume I. Philosophical Review* 68(4), pp. 548–550, doi:10.2307/2182503.

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

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

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

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

[30] Luca Padovani (2017): *Context-Free Session Type Inference.* In Hongseok Yang, editor: *ESOP*, *LNCS* 10201, Springer, pp. 804–830, doi:10.1007/978-3-662-54434-1_30.

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

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

[33] Diogo Poças, Diana Costa, Andreia Mordido & Vasco T. Vasconcelos (2023): *System F^{μ}_{ω} with Context-free Session Types.* In: *ESOP*, *LNCS* 13990, Springer, pp. 392–420, doi:10.1007/978-3-031-30044-8_15.

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

[35] 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.

[36] 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.

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

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

[39] 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.

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

[41] Philip Wadler (2012): *Propositions as sessions*. In: *ICFP*, ACM, pp. 273–286, doi:10.1145/2364527.2364568.

Appendix A. Proof of lemma 6

Whenever we write $\mathrm{unravel}(T) = \mathrm{unravel}(U)$, we mean that either both sides of the equation are defined and equal, or that both sides are undefined.

Lemma 26. *If* $T\checkmark$*, then* $\mathrm{unravel}(T) = \mathsf{skip}$ *and, for every* U*,* $\mathrm{unravel}(T;U) = \mathrm{unravel}(U)$.

Proof. By induction on the proof of $T\checkmark$.

(**Case** $T = \mathsf{skip}$): then trivially $\mathrm{unravel}(\mathsf{skip}) = \mathsf{skip}$ (no one-step unraveling is possible). Also $\mathrm{unravel}(\mathsf{skip};U) = \mathrm{unravel}(U)$ by proposition 4.

(**Case** $T = U;V$): then by ✓-Seq, both $U\checkmark$ and $V\checkmark$. By induction, $\mathrm{unravel}(U) = \mathsf{skip}$, $\mathrm{unravel}(V) = \mathsf{skip}$, and for every W, $\mathrm{unravel}(U;W) = \mathrm{unravel}(W)$ and $\mathrm{unravel}(V;W) = \mathrm{unravel}(W)$. In particular, $\mathrm{unravel}(U;V) = \mathrm{unravel}(V) = \mathsf{skip}$. Moreover, by proposition 4, $\mathrm{unravel}((U;V);W) = \mathrm{unravel}(U;(V;W)) = \mathrm{unravel}(V;W) = \mathrm{unravel}(W)$.

(**Case** $T = X(\overline{\beta})$): let $X(\overline{\alpha}) \doteq U$ be the equation defining X. Then by ✓-Id, $U\checkmark$. By lemma 2, $[\overline{\beta}/\overline{\alpha}]U\checkmark$. By induction, $\mathrm{unravel}([\overline{\beta}/\overline{\alpha}]U) = \mathsf{skip}$ and for every V, $\mathrm{unravel}([\overline{\beta}/\overline{\alpha}]U;V) = \mathrm{unravel}(V)$. Therefore by proposition 4, $\mathrm{unravel}(X(\overline{\beta})) = \mathrm{unravel}([\overline{\beta}/\overline{\alpha}]U) = \mathsf{skip}$ and $\mathrm{unravel}(X(\overline{\beta});V) = \mathrm{unravel}([\overline{\beta}/\overline{\alpha}]U;V) = \mathrm{unravel}(V)$. □

Lemma 6 (Contractivity and unravelling)**.** *If* T *is a pretype, then* T contr *iff* $\mathrm{unravel}(T)$ *is defined.*

Proof. Let us prove the direction from left to right, by induction on the proof of T contr.

(**Case** $T = \mathsf{unit}, U \to V, \langle\!|\ell\colon T_\ell|\!\rangle_{\ell\in L}, \mathsf{skip}, \sharp U, \odot\{\ell\colon T_\ell\}_{\ell\in L}, \alpha, \exists\alpha\colon\kappa\,.\,U$): in each case we immediately get that $\mathrm{unravel}(T)$ is defined and equal to T.

(**Case** $T = X(\overline{\beta})$): let $X(\overline{\alpha}) \doteq U$ be the equation defining X. Then by C-Id, U contr. By lemma 2, $[\overline{\beta}/\overline{\alpha}]U$ contr. By induction, $\mathrm{unravel}([\overline{\beta}/\overline{\alpha}]U)$ is defined. Finally, by proposition 4, $\mathrm{unravel}(X(\overline{\beta}))$ is defined and equal to $\mathrm{unravel}([\overline{\beta}/\overline{\alpha}]U)$.

(**Case** $T = U;V$ **with** $U\checkmark$): then by C-Seq1, V contr. By induction, $\mathrm{unravel}(V)$ is defined. Finally by lemma 26, $\mathrm{unravel}(U;V)$ is defined and equal to $\mathrm{unravel}(V)$.

(**Case** $T = U;V$ **with** $U\not\checkmark$): then by C-Seq2, U contr. In order to prove that $\mathrm{unravel}(U;V)$ is defined, we need to perform a second case analysis on U.

(**Sub-case** $U = \mathsf{unit}, W \to W', \langle\!|\ell\colon U_\ell|\!\rangle_{\ell\in L}, \sharp W, \alpha, \exists\alpha\colon\kappa\,.\,W$): in each case we immediately get that $\mathrm{unravel}(U;V)$ is defined and equal to $U;V$ (no one-step unraveling is possible).

(**Sub-case** $U = \mathsf{skip}$): not possible since $U\not\checkmark$.

(**Sub-case** $U = \odot\{\ell\colon U_\ell\}_{\ell\in L}$): we immediately get that $\mathrm{unravel}(U;V)$ is defined and equal to $\odot\{\ell\colon U_\ell;V\}_{\ell\in L}$.

(**Sub-case** $U = X(\overline{\beta})$): let $X(\overline{\alpha}) \doteq W$ be the equation defining X. Then by C-Id, W contr. Also by ✓-Id, $W\not\checkmark$. By lemma 2, we also have $[\overline{\beta}/\overline{\alpha}]W$ contr and $[\overline{\beta}/\overline{\alpha}]W\not\checkmark$. Thus $[\overline{\beta}/\overline{\alpha}]W;V$ contr, so that by induction $\mathrm{unravel}([\overline{\beta}/\overline{\alpha}]W;V)$ is defined. Finally, by proposition 4, $\mathrm{unravel}(X(\overline{\alpha});V)$ is defined and equal to $\mathrm{unravel}([\overline{\beta}/\overline{\alpha}]W;V)$.

(**Sub-case** $U = W;W'$ with $W\checkmark$): then by C-Seq1, W' contr. Also by ✓-Seq, $W'\not\checkmark$. Thus $W';V$ contr and also $W;(W';V)$ contr. By induction, $\mathrm{unravel}(W;(W';V))$ is defined. Finally, by proposition 4, $\mathrm{unravel}((W;W');V)$ is defined and equal to $\mathrm{unravel}(W;(W';V))$.

(**Sub-case** $U = W;W'$ with $W\not\checkmark$): then by C-Seq2, W contr. Thus $W;(W';V)$ contr. By induction, $\mathrm{unravel}(W;(W';V))$ is defined. Finally, by proposition 4, $\mathrm{unravel}((W;W');V)$ is defined and equal to $\mathrm{unravel}(W;(W';V))$.

The reverse direction from right to left is proven in a similar manner, by induction on the number of one-step unfoldings necessary to compute $\mathrm{unravel}(T)$.

(**Base case**): suppose $\mathrm{unravel}(T)$ is reached in zero steps, that is, $T = \mathrm{unravel}(T)$. By proposition 5, T is one of unit, $T \to U$, $\langle\!|\ell\colon T_\ell|\!\rangle_{\ell\in L}$, $\exists\alpha\colon\kappa\,.\,T'$, skip, $\sharp T'$, $\odot\{\ell\colon T_\ell\}_{\ell\in L}$, α, $\sharp T';T''$, $\alpha;T'$. In each of these cases, we easily derive that T contr using rule C-Axiom (for the last two cases, we also use rule C-Seq2).

(**Induction step**): suppose $\mathrm{unravel}(T)$ is reached in $n+1$ steps. In particular, $T \neq \mathrm{unravel}(T)$. By proposition 5, T is one of $X(\overline{\beta})$, $\mathsf{skip};T'$, $\odot\{\ell\colon T_\ell\}_{\ell\in L};U$, $X(\overline{\beta});U$ or $(U;V);W$.

(**Sub-case** $T = X(\overline{\beta})$): let $X(\overline{\alpha}) \doteq U$ be the equation defining X. Then $\mathrm{unravel}_1(X(\overline{\beta})) = [\overline{\beta}/\overline{\alpha}]U$, meaning that $\mathrm{unravel}([\overline{\beta}/\overline{\alpha}]U)$ is defined and reached in n steps. By induction hypothesis, $[\overline{\beta}/\overline{\alpha}]U$ contr. By lemma 2, U contr. By rule C-Id, X contr as desired. The remaining sub-cases are similar and we omit the proof. □

Appendix B. Proof of theorem 11

Theorem 11 (Equivalence relation)**.** *Relation $\simeq$ is an equivalence on types.*

Proof. We need to prove reflexivity, symmetry and transitivity.

Reflexivity. We present a coinductive proof that $T \simeq T$ for every type T. Consider the relation

$$\begin{aligned}\mathcal{R} = \{&(T,T) : \text{there exists } \Delta, \kappa \text{ s.t. } \Delta \vdash T : \kappa\}\\ \cup \{&(T,T') : \text{there exists } \Delta, \kappa \text{ s.t. } \Delta \vdash T : \kappa \text{ and } T' = \mathrm{unravel}_1(T)\}.\end{aligned}$$

We show that $\mathcal{R}$ is backward closed for the rules of type equivalence. This shows that $\mathcal{R} \subseteq \simeq$ and therefore $T \simeq T$ for every type T.

Let $(T,T) \in \mathcal{R}$. We begin by considering the cases in which T fits a type constructor, i.e., $T = \mathrm{unravel}(T)$. By proposition 5, we have the following case analysis for T.

(**Case** $T = \mathsf{unit}$): we apply axiom E-Unit to (T,T).

(**Case** $T = U \to V$): we apply rule E-Arrow to (T,T), arriving at goals (U,U) and (V,V). Moreover, the derivation of $\Delta \vdash T : \kappa$ must use rule K-Arrow, implying that $\Delta \vdash U : \kappa_1$ and $\Delta \vdash V : \kappa_2$, so that $(U,U), (V,V) \in \mathcal{R}$.

(**Case** $T = \{\ell\colon T_\ell\}_{\ell \in L}$): we apply rule E-Rcd, arriving at goals (T_k, T_k) for each $k \in L$. Moreover, the derivation of $\Delta \vdash T : \kappa$ must use rule K-Rcd, implying that $\Delta \vdash T_k : \kappa_k$ for each $k \in L$, so that $(T_k, T_k) \in \mathcal{R}$ for each $k \in L$. The cases with $\langle\rangle$, $\oplus$, $\&$ are similar.

(**Case** $T = \mathsf{skip}$): we apply axiom E-Skip to (T,T).

(**Case** $T = !U$): we apply rule E-Msg, arriving at goal (U,U). The derivation of $\Delta \vdash T : \kappa$ must use rule K-Msg, implying that $\Delta \vdash U : \kappa'$, so that $(U,U) \in \mathcal{R}$. The case with $?$ is similar.

(**Case** $T = \forall\alpha\colon \kappa.U$): we apply rule E-Quant, arriving at goal (U,U). The derivation of $\Delta \vdash T : \kappa$ must use rule K-Quant, implying that $\Delta, \alpha\colon \kappa \vdash U : \kappa'$, so that $(U,U) \in \mathcal{R}$. The case with $\exists\alpha\colon \kappa$ is similar.

(**Case** $T = \alpha$): we apply axiom E-Var to (T,T).

(**Case** $T = !U;V$): we apply rule E-MsgSeq2, arriving at goals (U,U) and (V,V). By similar considerations, we get that $(U,U), (V,V) \in \mathcal{R}$. The case with $?$ is similar.

(**Case** $T = \alpha;U$): we apply rule E-VarSeq2, arriving at goal (U,U). By similar considerations, we get that $(U,U) \in \mathcal{R}$.

Next, we consider the cases in which $T \neq \mathrm{unravel}(T)$. Again by proposition 5, they are as follows.

(**Case** $T = X(\overline{\beta})$): the derivation of $\Delta \vdash T : \kappa$ must use rule K-Id; therefore, there must exist an equation $X(\overline{\alpha}) \doteq U$ for identifier $X(\overline{\alpha})$. We apply rule E-IdR to (T,T), arriving at goal $(T, [\overline{\beta}/\overline{\alpha}]U)$. Since $[\overline{\beta}/\overline{\alpha}]U = \mathrm{unravel}_1(T)$, we obtain that $(T, [\overline{\beta}/\overline{\alpha}]U) \in \mathcal{R}$.

(**Case** $T = \mathsf{skip};T'$): we apply rule E-SkipSeqR, arriving at goal (T,T'). Since $T' = \mathrm{unravel}_1(T)$, we obtain that $(T,T') \in \mathcal{R}$.

(**Case** $T = \oplus\{\ell\colon T_\ell\}_{\ell\in L};U$): we apply rule E-ChoiceSeqR to (T,T), arriving at goal (T,T') where $T' = \oplus\{\ell\colon T_\ell;U\}_{\ell\in L}$. Since $T' = \mathrm{unravel}_1(T)$, we obtain that $(T,T') \in \mathcal{R}$. The case with $\&$ is similar.

(**Case** $T = X(\overline{\beta});V$): the derivation of $\Delta \vdash T : \kappa$ must use rules K-Seq and K-Id; therefore, there must exist an equation $X(\overline{\alpha}) \doteq U$ for identifier $X(\overline{\alpha})$. We apply rule E-IdSeqR to (T,T), arriving at goal (T,T') where $T' = ([\overline{\beta}/\overline{\alpha}]U);V$. Since $T' = \mathrm{unravel}_1(T)$, we obtain that $(T,T') \in \mathcal{R}$.

(**Case** $T = (U;V);W$): we apply rule E-SeqSeqR, arriving at goal (T,T') where $T' = U;(V;W)$. Since $T' = \mathrm{unravel}_1(T)$, we obtain that $(T,T') \in \mathcal{R}$.

Finally we need to consider the cases $(T,T') \in \mathcal{R}$ where $T \neq T'$. By definition, this means that $T' = \mathrm{unravel}_1(T)$, and thus $T \neq \mathrm{unravel}(T)$. By proposition 5, we have the following case analysis for T.

(**Case** $T = X(\overline{\beta})$): the derivation of $\Delta \vdash T : \kappa$ must use rule K-Id; therefore, there must exist an equation $X(\overline{\alpha}) \doteq U$ for identifier $X(\overline{\alpha})$. We apply rule E-IdL to (T,T'), arriving at goal $([\overline{\beta}/\overline{\alpha}]U, T')$. Since $[\overline{\beta}/\overline{\alpha}]U = \mathrm{unravel}_1(T) = T'$, we obtain that $(T',T') \in \mathcal{R}$.

(**Case** $T = \mathsf{skip};T'$): the derivation of $\Delta \vdash T : \kappa$ must use rule K-Seq, implying that $\Delta \vdash T' : \mathrm{S}$. We apply rule E-SkipSeqL to (T,T'), arriving at goal $(T',T') \in \mathcal{R}$.

(**Case** $T = \oplus\{\ell\colon T_\ell\}_{\ell\in L}; U$): the derivation of $\Delta \vdash T : \kappa$ must use rules K-Seq and K-Choice, implying that $\Delta \vdash T_k : \mathrm{S}$ for each $k \in L$ and $\Delta \vdash U : \mathrm{S}$. Again by rule K-Seq, we get that $\Delta \vdash T_k; U : \mathrm{S}$ for each $k \in L$ and thus (by rule K-Choice) $\Delta \vdash \oplus\{\ell\colon T_\ell; U\}_{\ell\in L} : \mathrm{S}$. Since $T' = \mathrm{unravel}_1(T)$, we know that $T' = \oplus\{\ell\colon T_\ell; U\}_{\ell\in L}$. We apply rule E-SkipSeqL to (T, T'), arriving at goal $(T', T') \in \mathcal{R}$. The case with $\&$ is similar.

(**Case** $T = X(\overline{\beta}); V$): the derivation of $\Delta \vdash T : \kappa$ must use rules K-Seq and K-Id; therefore, there must exist an equation $X(\overline{\alpha}) \doteq U$ for identifier $X(\overline{\alpha})$. Since $T' = \mathrm{unravel}_1(T)$, we know that $T' = ([\overline{\beta}/\overline{\alpha}]U); V$. We apply rule E-IdSeqL to (T, T'), arriving at goal $(T', T') \in \mathcal{R}$.

(**Case** $T = (U; V); W$): the derivation of $\Delta \vdash T : \kappa$ must use rule K-Seq, so that $\Delta \vdash U : \mathrm{S}$, $\Delta \vdash V : \mathrm{S}$, $\Delta \vdash W : \mathrm{S}$. Hence (by rule K-Seq) $\Delta \vdash U; (V; W) : \mathrm{S}$. Since $T' = \mathrm{unravel}_1(T)$, we know that $T' = U; (V; W)$. We apply rule E-SeqSeqL to (T, T'), arriving at goal $(T', T') \in \mathcal{R}$.

Symmetry. By inspection of the rules, we can see that every rule either is symmetric or has a symmetric counterpart. Therefore, we can transform a derivation for $T \simeq U$ into a derivation for $U \simeq T$ by swapping the left and right-hand sides of each equivalence judgement.

Transitivity. We present a coinductive proof that for all types T, U, V, if we have $T \simeq U$ and $U \simeq V$, then we also have $T \simeq V$. Consider the relation

$$\mathcal{R} = \{(T, V) : T, V \text{ are types and there exists a type } U \text{ s.t. } T \simeq U, U \simeq V\}.$$

We show that $\mathcal{R}$ is backward closed for the rules defining type equivalence. This shows that $\mathcal{R} \subseteq \simeq$, giving the desired property.

Firstly, we can assume that there are no left-preserving rules applicable to judgements $T \simeq U$. If there was such a rule, then the symmetric rule could be applied to judgement $U \simeq V$, and we would get a different U' for which $T \simeq U'$ and $U' \simeq V$. Without loss of generality, our derivation for $T \simeq U$ ends with a (finite) sequence of left-preserving rules reaching a type U' that can only be consumed. The symmetric sequence of right-preserving rules could then be applied in the derivation for $U \simeq V$ to reach the same type U'. Thus, we can just assume that our type U already is in a form that must be consumed.

Secondly, suppose a derivation for $T \simeq U$ starts with a right-preserving rule. After this rule we get the judgement $T' \simeq U$ for some type T' (note that U remains the same). But then, we can apply the corresponding rule to (T, V), arriving at goal (T', V), which is in $\mathcal{R}$ since $T' \simeq U$ and $U \simeq V$. In a similar manner, we can handle the case in which $U \simeq V$ starts with a left-preserving rule.

The remaining possibility is that both derivations for $T \simeq U$ and $U \simeq V$ end with a progressing rule. Here we need to split our analysis into several cases, depending on which rule is at the end of the derivation for $T \simeq U$.

(**Case** E-Skip): then $T = \mathsf{skip}$ and $U = \mathsf{skip}$. The only progressing rule that can be applied at $U \simeq V$ is also E-Skip, implying that $V = \mathsf{skip}$ as well. Therefore, we can apply axiom E-Skip to (T, V). The cases E-Unit and E-Var are handled similarly.

(**Case** E-Msg, $\sharp = !$): then $T = !T'$ and $U = !U'$ for some T', U'. Furthermore, we have $T' \simeq U'$. The two possible progressing rules for $U \simeq V$ are E-Msg and E-MsgSeq1R.

In the first case, we have $V = !V'$ for some V'. It follows that $U' \simeq V'$. We apply rule E-Msg to (T, V), arriving at goal $(T', V') \in \mathcal{R}$.

In the second case, we have $V = !V'; V''$ for some V' and V''. It follows that $U' \simeq V'$ and $V''\checkmark$. We apply rule E-MsgSeq1R to (T, V), arriving at goal $(T', V') \in \mathcal{R}$.

The case for $\sharp = ?$, and the cases E-MsgSeq1L, E-MsgSeq1R, E-MsgSeq2, E-VarSeq1L, E-VarSeq1R and E-VarSeq2 are handled similarly.

(**Case** E-Choice, $\odot = \oplus$): then $T = \oplus\{\ell\colon T_\ell\}_{\ell\in L}$ and $U = \oplus\{\ell\colon U_\ell\}_{\ell\in L}$ for some L, T_k, U_k, $k \in L$. Furthermore, we have $T_k \simeq U_k$ for all $k \in L$. The only progressing rule that can be applied to $U \simeq V$ is also E-Choice, implying that $V = \oplus\{\ell\colon T_\ell\}_{\ell\in L}$ for some L, V_k, $k \in L$. Furthermore, we have $U_k \simeq V_k$ for each $k \in L$. We apply rule E-Choice to (T, V), arriving at goals $(T_k, V_k) \in \mathcal{R}$ for each $k \in L$.

The case for $\odot = \&$, and the case E-Rcd are handled similarly.

(**Case** E-Arrow): then $T = T' \to T''$ and $U = U' \to U''$ for some T', T'', U', U''. Furthermore, we have $T' \simeq U'$ and $T'' \simeq U''$. The only progressing rule that can be applied to $U \simeq V$ is also

E-Arrow, implying that $V = V' \to V''$ for some V', V''. Furthermore, we have $U' \simeq V'$ and $U'' \simeq V''$. We apply rule E-Choice to (T, V), arriving at goals $(T', V'), (T'', V'') \in \mathcal{R}$.

(**Case** E-Quant, $\exists\forall^\kappa = \forall^\kappa$): then (renaming the bound variable if necessary) $T = \forall\alpha\colon\kappa.T'$ and $U = \forall\alpha\colon\kappa.U'$ for some α, T', U'. Furthermore, $T' \simeq U'$. The only progressing rule that can be applied to $U \simeq V$ is also E-Quant, implying that (renaming the bound variable if necessary) $V = \forall\alpha\colon\kappa.V'$ for some V'. Furthermore, $U' \simeq V'$. We apply rule E-Choice to (T, V), arriving at goal $(T', V') \in \mathcal{R}$.

The case for $\exists\forall^\kappa = \exists^\kappa$ is handled similarly. □

Appendix C. Proof of lemma 14

Proposition 27.

1. *If T is a type and $U = \mathrm{unravel}_1(T)$, then $\mathrm{gv}(T) = \mathrm{gv}(U)$.*
2. *If T is a type and $U = \mathrm{unravel}(T)$, then $\mathrm{gv}(T) = \mathrm{gv}(U)$.*

Proof. Item 1 follows from a case analysis on T and the inductive definition of gv. Item 2 follows from item 1 since $\mathrm{unravel}(T)$ is reached in a finite number of steps. □

Lemma 14. *Let T, U be types. If $T \simeq U$ then $\mathrm{gv}(T) = \mathrm{gv}(U)$.*

Proof. Let T, U be types such that $T \simeq U$. Without loss assume that $T = \mathrm{unravel}(T)$ and $U = \mathrm{unravel}(U)$; otherwise replace T or U by its unraveling T' or U'(lemma 8 ensures that $T' \simeq U'$, and proposition 27 ensures that $\mathrm{gv}(T) = \mathrm{gv}(T')$, $\mathrm{gv}(U) = \mathrm{gv}(U')$).

It is enough to show that $\beta \in \mathrm{gv}(T)$ implies $\beta \in \mathrm{gv}(U)$, as this will entail $\mathrm{gv}(T) \subseteq \mathrm{gv}(U)$ and a symmetric reasoning would yield $\mathrm{gv}(U) \subseteq \mathrm{gv}(T)$. We recall the inductive definition of the predicate $\beta \overset{?}{\in} \mathrm{gv}(T)$ presented in the proof of proposition 12:

1. $\beta \in \mathrm{gv}(\beta)$.
2. $\beta \in \mathrm{gv}(\exists\forall\alpha\colon\kappa\,.\,T)$ if $\beta \neq \alpha$ and $\beta \in \mathrm{gv}(T)$.
3. $\beta \in \mathrm{gv}(X(\overline{\beta}))$ if $\beta = \beta_i$ for some i such that $\alpha_i \in \mathrm{gv}(T)$, where $X(\overline{\alpha}) \doteq T$.
4. $\beta \in \mathrm{gv}(T \to U)$ if $\beta \in \mathrm{gv}(T)$ or $\beta \in \mathrm{gv}(U)$ (and similarly for all other type constructors).

To prove that $\beta \in \mathrm{gv}(T)$ implies $\beta \in \mathrm{gv}(U)$, we use induction in the number n of steps necessary to derive $\beta \in \mathrm{gv}(T)$. If $n = 0$, there is nothing to prove. Otherwise we consider four cases.

1. Suppose that $T = \beta$. Since $T \simeq U$ and $U = \mathrm{unravel}(U)$, either $U = \beta$ (rule E-Var) or $U = \beta; V$ with $V\checkmark$ (rule E-VarSeq1R). In either case, we have (by item 1, and possibly also item 4) that $\beta \in \mathrm{gv}(U)$.
2. Suppose that $T = \exists\forall\alpha\colon\kappa\,.\,T'$ and $\beta \neq \alpha$. Then we can derive $\beta \in \mathrm{gv}(T')$ in $n-1$ steps. Moreover, since $T \simeq U$ and $U = \mathrm{unravel}(U)$, we must have $U = \exists\forall\alpha\colon\kappa\,.\,U'$ with $T' \simeq U'$. By induction, $\beta \in \mathrm{gv}(U')$, so that by item 2 $\beta \in \mathrm{gv}(U)$.
3. The case that $T = X(\overline{\beta})$ cannot occur, due to our assumption that $T = \mathrm{unravel}(T)$.
4. Suppose that $T = T_1 \to T_2$. Then we can derive in $n-1$ steps either $\beta \in \mathrm{gv}(T_1)$ or $\beta \in \mathrm{gv}(T_2)$. Moreover, since $T \simeq U$ and $U = \mathrm{unravel}(U)$, we must have $U = U_1 \to U_2$ with $T_1 \simeq U_1$ and $T_2 \simeq U_2$. By induction, we must have $\beta \in \mathrm{gv}(U_1)$ or $\beta \in \mathrm{gv}(U_2)$, so that by item 4 $\beta \in \mathrm{gv}(U)$. (The other type constructors are handled similarly.)

□

Appendix D. Proof of theorem 19

Theorem 19 (Soundness and completeness)**.** *Let* T, U *be terms. Then* $T \simeq_t U$ *iff* $T \sim U$.

Proof. We start with the direction from left to right. Consider the relation

$$\mathcal{R} = \{(T, U) : T, U \text{ are terms and } T \simeq_t U\}.$$

We must show that $\mathcal{R}$ is a bisimulation, that is, for all $(T, U) \in \mathcal{R}$: for every transition a and type T', if $T \xrightarrow{a} T'$, then there exists a type U' such that $U \xrightarrow{a} U'$ and $(T', U') \in \mathcal{R}$; and for every transition a and type U', if $U \xrightarrow{a} U'$, then there exists a type T' such that $T \xrightarrow{a} T'$ and $(T', U') \in \mathcal{R}$. This shows that $\mathcal{R} \subseteq \sim$, and hence that $T \simeq U$ implies $T \sim U$.

The proof has two parts. First consider cases $(T, U) \in \mathcal{R}$ such that $\text{unravel}(T) = T$ and $\text{unravel}(U) = U$. We proceed by a case analysis for the last rule in the derivation of $T \simeq_t U$ (which must be a progressing rule).

(**Case** E-Skip): then $T = \mathsf{skip}$ and $U = \mathsf{skip}$. Since there is no transition a that can be applied to skip in the LTS, the bisimulation conditions trivially hold.

(**Case** E-Unit): then $T = \mathsf{unit}$ and $U = \mathsf{unit}$. The unique transition that can be applied to unit is $\mathsf{unit} \xrightarrow{\mathsf{unit}} \mathsf{skip}$ (L-Unit). We arrive at pair $(\mathsf{skip}, \mathsf{skip})$ which is obviously in $\mathcal{R}$.

(**Case** E-Var): analogous.

(**Case** E-Arrow): then $T = T' \to T''$ and $U = U' \to U''$. The only transitions that can be applied to T are $T \xrightarrow{\to_d} T'$ and $T \xrightarrow{\to_r} T''$ (L-Arrow1, L-Arrow2). Similarly, the only transitions that can be applied to U are $U \xrightarrow{\to_d} U'$ and $U \xrightarrow{\to_r} U''$. Clearly, T', T'', U', U'' are terms and, because E-Arrow was used, $T' \simeq_t U'$ and $T'' \simeq_t U''$. Thus $(T', U'), (T'', U'') \in \mathcal{R}$.

(**Case** E-Rcd with $\langle\!|\,|\!\rangle = \{\}$): then $T = \{\ell \colon T_\ell\}_{\ell \in L}$ and $U = \{\ell \colon U_\ell\}_{\ell \in L}$. The only transitions that can be applied to T are $T \xrightarrow{\{\}_k} T_k$ for each $k \in L$ (L-Rcd). Similarly, the only transitions that can be applied to U are $U \xrightarrow{\{\}_k} U_k$ for each $k \in L$. Clearly, T_k, U_k are types for each $k \in L$ and, because E-Rcd was used, $T_k \simeq_t U_k$ for each $k \in L$. Thus $(T_k, U_k) \in \mathcal{R}$ for each $k \in L$.

(**Cases** E-Rcd with $\langle\!|\,|\!\rangle = \langle\rangle$, and E-Choice): analogous.

(**Case** E-Quant): analogous.

(**Case** E-Msg, with $\sharp = !$): then $T = !T'$ and $U = !U'$. The only transitions that can be applied to T are $T \xrightarrow{!_d} T'$ and $T \xrightarrow{!_c} \mathsf{skip}$ (L-Msg1, L-Msg2). Similarly, the only transitions that can be applied to U are $U \xrightarrow{!_d} U'$ and $U \xrightarrow{!_c} \mathsf{skip}$. Clearly, T', U' are types and, because E-Msg was used, $T' \simeq_t U'$. Thus $(T', U') \in \mathcal{R}$. We also have $(\mathsf{skip}, \mathsf{skip}) \in \mathcal{R}$.

(**Case** E-Msg, with $\sharp = ?$): analogous.

(**Case** E-MsgSeq1L, with $\sharp = !$): then $T = !T'; V$ and $U = !U'$. The only transitions that can be applied to T are $T \xrightarrow{!_d} T'$ and $T \xrightarrow{!_c} V$ (L-MsgSeq1, L-MsgSeq2). Similarly, the only transitions that can be applied to U are $U \xrightarrow{!_d} U'$ and $U \xrightarrow{!_c} \mathsf{skip}$. Clearly, T', U', V are types and, because E-MsgSeq1L was used, $T' \simeq_t U'$ and $V\checkmark$. Due to lemma 9 (restated for terms), $V \simeq_t \mathsf{skip}$. Thus $(T', U'), (V, \mathsf{skip}) \in \mathcal{R}$.

(**Cases** E-MsgSeq1L, with $\sharp = ?$, E-MsgSeq1R and E-MsgSeq2): analogous.

(**Cases** E-VarSeq1L, E-VarSeq1R, E-VarSeq2): analogous.

Now consider that $T \neq \text{unravel}(T)$. Since $T \simeq_t U$, by lemma 8 (restated for terms) it follows that $\text{unravel}(T) \simeq_t U$. If $U = \text{unravel}(U)$, then from the above case analysis, every transition $\text{unravel}(T) \xrightarrow{a} T'$ is matched by a transition $U \xrightarrow{a} U'$ with $(T', U') \in \mathcal{R}$ and vice-versa. Since, by lemma 17, T and $\text{unravel}(T)$ have the same transitions, i.e., $T \xrightarrow{a} T'$ iff $\text{unravel}(T) \xrightarrow{a} T'$, there is also a matching between transitions of T and U as desired. Otherwise if $U \neq \text{unravel}(U)$, it similarly follows by lemma 8 (restated for terms) that $\text{unravel}(T) \simeq_t \text{unravel}(U)$. By the previous case analysis there is a matching between transitions of $\text{unravel}(T)$ and $\text{unravel}(U)$. Since, by lemma 17, U and $\text{unravel}(U)$ have the same transitions, there is also a matching between transitions of T and U as desired. The case with $T = \text{unravel}(T), U \neq \text{unravel}(U)$ is analogous.

We now prove the direction from right to left. Consider the relation

$$\mathcal{S} = \{(T, U) : T, U \text{ are terms and } T \sim U\}.$$

We prove that the relation $\mathcal{S}$ is backward closed for the rules of term equivalence, that is, for all $(T, U) \in \mathcal{S}$, there is a term equivalence rule that can be applied to (T, U), and all the resulting judgements are also in $\mathcal{S}$. This shows that $\mathcal{S} \subseteq \simeq_t$, and hence that $T \sim U$ implies $T \simeq_t U$.

The proof has two parts. First consider cases $(T, U) \in \mathcal{S}$ such that $\text{unravel}(T) = T$ and $\text{unravel}(U) = U$. We proceed by a case analysis on the structure of T, as prescribed by proposition 5 (restated for terms).

(**Case** $T = \mathsf{unit}$): the unique transition that can be applied to T is $T \xrightarrow{\mathsf{unit}} \mathsf{skip}$. Since $T \sim U$ and $U = \text{unravel}(U)$, then $U = \mathsf{unit}$. Therefore, we can apply E-Unit.

(**Case** $T = \mathsf{skip}$): there are no transitions that can be applied to T. Since $T \sim U$ and $U = \text{unravel}(U)$, then $U = \mathsf{skip}$. Therefore, we can apply E-Skip.

(**Case** $T = \alpha$): the unique transition that can be applied to T is $T \xrightarrow{\alpha} \mathsf{skip}$. Since $T \sim U$ and $U = \text{unravel}(U)$, then either $U = \alpha$ or $U = \alpha; U'$, with $\mathsf{skip} \sim U'$. In the first case, we can apply E-Var. In the second case, since $\mathsf{skip} \sim U'$ implies $U'\checkmark$ by lemma 9 (restated for terms), we can apply E-VarSeq1R.

(**Case** $T = T' \to T''$): the only transitions that can be applied to T are $T \xrightarrow{\to_d} T'$ and $T \xrightarrow{\to_r} T''$. Since $T \sim U$ and $U = \text{unravel}(U)$, then $U = U' \to U''$. Moreover, $T' \sim U'$ and $T'' \sim U''$. We can apply E-Arrow arriving at $(T', U'), (T'', U'') \in \mathcal{S}$.

(**Case** $T = \{\ell \colon T_\ell\}_{\ell \in L}$): the only transitions that can be applied to T are $T \xrightarrow{\{\}_k} T_k$ for each $k \in L$. Since $T \sim U$ and $U = \text{unravel}(U)$, then $U = \{\ell \colon U_\ell\}_{\ell \in L}$. Moreover, $T_k \sim U_k$ for each $k \in L$. We can apply E-Rcd arriving at $(T_k, U_k) \in \mathcal{S}$ for each $k \in L$.

(**Cases** $T = \langle \ell \colon T_\ell \rangle_{\ell \in L}$, $T = \oplus\{\ell \colon T_\ell\}_{\ell \in L}$, $T = \&\{\ell \colon T_\ell\}_{\ell \in L}$, $T = \alpha \forall_\kappa T'$, $T = \alpha \exists_\kappa T'$): analogous.

(**Case** $T = !T'$): the only transitions that can be applied to T are $T \xrightarrow{!_d} T'$ and $T \xrightarrow{!_c} \mathsf{skip}$. Since $T \sim U$ and $U = \text{unravel}(U)$, then either $U = !U'$ or $U = !U'; V$, with $\mathsf{skip} \sim V$. In either case, $T' \sim U'$. In the first case, we can apply E-Msg, arriving at $(T', U') \in \mathcal{S}$. In the second case, since $\mathsf{skip} \sim V$ implies $V\checkmark$ by lemma 9 (restated for terms), we can apply E-MsgSeq1R, arriving at $(T', U') \in \mathcal{S}$.

(**Case** $T = ?T'$): analogous.

(**Case** $T = !T'; T''$): the only transitions that can be applied to T are $T \xrightarrow{!_d} T'$ and $T \xrightarrow{!_c} T''$. Since $T \sim U$ and $U = \text{unravel}(U)$, then either $U = !U'$ or $U = !U'; U''$. In the first case, $T' \sim U'$ and $T'' \sim \mathsf{skip}$, which implies that $T''\checkmark$ by lemma 9 (restated for terms); we can apply E-MsgSeq1L, arriving at $(T', U') \in \mathcal{S}$. In the second case, $T' \sim U'$ and $T'' \sim U''$; we can apply E-MsgSeq2, arriving at $(T', U'), (T'', U'') \in \mathcal{S}$.

(**Case** $T = ?T'; T''$, $T = \alpha; U$): analogous.

Now consider that $T \neq \text{unravel}(T)$. From $T \sim U$ it follows that $T' \sim U$, where T' is the one-step unfolding of T, due to the fact that T and T' have the same transitions (lemma 17). Then we can apply an appropriate right-preserving rule to (T, U), arriving at $(T', U) \in \mathcal{S}$. The case with $T = \text{unravel}(T), U \neq \text{unravel}(U)$ is analogous. □

Appendix E. Proof of lemma 21

Proposition 28. *Let T be a term and $(\mathcal{T}_T, \mathcal{N}_T, \text{word}(T), \mathcal{P}_T)$ the corresponding simple grammar. For every $\gamma, \gamma' \in \mathcal{N}_T^*$ we have that $\gamma \approx \gamma \bot \gamma'$.*

Proof. Immediate from the fact that $\bot$ has no productions. □

Lemma 21. *Let T be a term and $(\mathcal{T}_T, \mathcal{N}_T, \text{word}(T), \mathcal{P}_T)$ the corresponding simple grammar. Let γ be a word such that $\text{word}(T) \approx \gamma$. Then*

- *If $T \xrightarrow{a} U$ for some U, then there exists γ' such that $\gamma \xrightarrow{a} \gamma'$ and $\text{word}(U) \approx \gamma'$.*
- *If $\gamma \xrightarrow{a} \gamma'$ for some γ', then there exists U such that $T \xrightarrow{a} U$ and $\text{word}(U) \approx \gamma'$.*

Proof. Let us define the relation

$$\mathcal{R} = \{(T, \gamma) : \mathrm{word}(T) \approx \gamma\}.$$

We show that $\mathcal{R}$ is backward closed for the transition relation. This shows the desired property. We begin by considering the cases in which T fits a type constructor, i.e., $T = \mathrm{unravel}(T)$. We have the following case analysis for T.

(**Case** $T = \mathsf{unit}$): by rule L-Unit, the LTS at T has the unique transition $T \xrightarrow{\mathsf{unit}} \mathsf{skip}$. Similarly, $\mathrm{word}(T) = Y$ for some Y with the unique transition $Y \xrightarrow{\mathsf{unit}} \varepsilon$. Since $\mathrm{word}(T) \approx \gamma$, the LTS at γ has the unique transition $\gamma \xrightarrow{\mathsf{unit}} \gamma'$ for some γ' s.t. $\varepsilon \approx \gamma'$. Since $\mathrm{word}(\mathsf{skip}) = \varepsilon$, we conclude that $\mathrm{word}(\mathsf{skip}) \approx \gamma'$, so that $(\mathsf{skip}, \gamma') \in \mathcal{R}$. The case with α is similar.

(**Case** $T = U \to V$): by rule L-Arrow, the LTS at T has exactly two transitions $T \xrightarrow{\to_d} U$ and $T \xrightarrow{\to_r} V$. Similarly, $\mathrm{word}(T) = Y$ for some Y with exactly two transitions $Y \xrightarrow{\to_d} \mathrm{word}(U)$ and $Y \xrightarrow{\to_r} \mathrm{word}(V)$. Since $\mathrm{word}(T) \approx \gamma$, the LTS at γ has exactly two transitions $\gamma \xrightarrow{\to_d} \gamma'_1$ and $\gamma \xrightarrow{\to_r} \gamma'_2$ for some γ'_1, γ'_2 s.t. $\mathrm{word}(U) \approx \gamma'_1$ and $\mathrm{word}(V) \approx \gamma'_2$. Hence $(U, \gamma'_1), (V, \gamma'_2) \in \mathcal{R}$.

(**Case** $T = \{\ell \colon T_\ell\}_{\ell \in L}$): by rule L-Rcd, the LTS at T has exactly the transitions $T \xrightarrow{\{\}_k} T_k$ for each $k \in L$. Similarly, $\mathrm{word}(T) = Y$ for some Y with exactly the transitions $Y \xrightarrow{\{\}_k} \mathrm{word}(T_k)$ for each $k \in L$. Since $\mathrm{word}(T) \approx \gamma$, the LTS at γ has exactly the transitions $\gamma \xrightarrow{\{\}_k} \gamma_k$ for some γ_k s.t. $\mathrm{word}(T_k) \approx \gamma_k$ for each $k \in L$. Hence $(T_k, \gamma_k) \in \mathcal{R}$ for each $k \in L$. The cases with $\langle\rangle$, $\oplus$, $\&$ are similar.

(**Case** $T = \mathsf{skip}$): the LTS at T has no transitions. Similarly, $\mathrm{word}(T) = \varepsilon$ which has no transitions. Since $\mathrm{word}(T) \approx \gamma$, γ has no transitions either.

(**Case** $T = !U$): by rule L-Msg, the LTS at T has exactly two transitions $T \xrightarrow{!_d} U$ and $T \xrightarrow{!_c} \mathsf{skip}$. Similarly, $\mathrm{word}(T) = Y$ for some Y with exactly two transitions $Y \xrightarrow{!_d} \mathrm{word}(U)\bot$ and $Y \xrightarrow{!_c} \varepsilon$. Since $\mathrm{word}(T) \approx \gamma$, the LTS at γ has exactly two transitions $\gamma \xrightarrow{!_d} \gamma'_1$ and $\gamma \xrightarrow{!_c} \gamma'_2$ for some γ'_1, γ'_2 s.t. $\mathrm{word}(U)\bot \approx \gamma'_1$ and $\varepsilon \approx \gamma'_2$. By proposition 28, we have $\mathrm{word}(U) \approx \mathrm{word}(U)\bot \approx \gamma'_1$; and since $\mathrm{word}(\mathsf{skip}) = \varepsilon$, we have $\mathrm{word}(\mathsf{skip}) \approx \gamma'_2$ as well. Hence $(U, \gamma'_1), (\mathsf{skip}, \gamma'_2) \in \mathcal{R}$. The case with $?$ is similar.

(**Case** $T = \alpha\, \forall_\kappa\, U$): by rule L-Quant, the LTS at T has the unique transition $T \xrightarrow{\forall \alpha : \kappa} U$. Similarly, $\mathrm{word}(T) = Y$ for some Y with the unique transition $Y \xrightarrow{\forall \alpha : \kappa} \mathrm{word}(U)$. Since $\mathrm{word}(T) \approx \gamma$, the LTS at γ has the unique transition $\gamma \xrightarrow{\forall \alpha : \kappa} \gamma'$ for some γ' s.t. $\mathrm{word}(U) \approx \gamma'$. Hence $(U, \gamma') \in \mathcal{R}$. The case with $\exists_\kappa$ is similar.

(**Case** $T = !U; V$): by rule L-MsgSeq, the LTS at T has exactly two transitions $T \xrightarrow{!_d} U$ and $T \xrightarrow{!_c} V$. Similarly, $\mathrm{word}(T) = Y\,\mathrm{word}(V)$ for some Y with exactly two transitions $Y \xrightarrow{!_d} \mathrm{word}(U)\bot$ and $Y \xrightarrow{!_c} \varepsilon$, yielding the two transitions $Y\mathrm{word}(V) \xrightarrow{!_d} \mathrm{word}(U)\bot\mathrm{word}(V)$ and $Y\mathrm{word}(V) \xrightarrow{!_c} \mathrm{word}(V)$. Since $\mathrm{word}(T) \approx \gamma$, the LTS at γ has exactly two transitions $\gamma \xrightarrow{!_d} \gamma'_1$ and $\gamma \xrightarrow{!_c} \gamma'_2$ for some γ'_1, γ'_2 s.t. $X_U\bot\mathrm{word}(V) \approx \gamma'_1$ and $\mathrm{word}(V) \approx \gamma'_2$. By proposition 28, we have $\mathrm{word}(U) \approx \mathrm{word}(U)\bot\mathrm{word}(V) \approx \gamma'_1$. Hence $(U, \gamma'_1), (V, \gamma'_2) \in \mathcal{R}$. The case with $?$ is similar.

(**Case** $T = \oplus\{\ell \colon T_\ell\}_{\ell \in L}; U$): by rule L-ChoiceSeq, the LTS at T has exactly the transitions $T \xrightarrow{\oplus_k} T_k; U$ for each $k \in L$. Similarly, $\mathrm{word}(T) = Y\,\mathrm{word}(U)$ for some Y with exactly the transitions $Y \xrightarrow{\oplus_k} \mathrm{word}(T_k)$ for each $k \in L$, yielding the transitions $Y\mathrm{word}(U) \xrightarrow{\oplus_k} \mathrm{word}(T_k)\mathrm{word}(U)$ for each $k \in L$. Since $\mathrm{word}(T) \approx \gamma$, the LTS at γ has exactly the transitions $\gamma \xrightarrow{\oplus_k} \gamma_k$ for some γ_k, s.t. $\mathrm{word}(T_k)\mathrm{word}(U) \approx \gamma_k$ for each $k \in L$. Hence $\mathrm{word}(T_k; U) = \mathrm{word}(T_k)\mathrm{word}(U) \approx \gamma_k$ and thus $(T_k; U, \gamma_k) \in \mathcal{R}$ for each $k \in L$. The case with $\&$ is similar.

(**Case** $T = \alpha; U$): by rule L-IndexSeq, the LTS at T has the unique transition $T \xrightarrow{\alpha} U$. Similarly, $\mathrm{word}(T) = Y\,\mathrm{word}(U)$ for some Y with the unique transition $Y \xrightarrow{\alpha} \mathsf{skip}$, yielding the transition $Y\mathrm{word}(U) \xrightarrow{\alpha} \mathrm{word}(U)$. Since $\mathrm{word}(T) \approx \gamma$, the LTS at γ has the unique transition $\gamma \xrightarrow{\alpha} \gamma'$ for some γ' s.t. $\mathrm{word}(U) \approx \gamma'$. Hence $(U, \gamma') \in \mathcal{R}$.

Now we consider the cases in which $T \neq \mathrm{unravel}(T)$. By lemma 17, the LTS at T has a transition $T \xrightarrow{a} U$ iff the LTS at $\mathrm{unravel}(T)$ has a corresponding transition $\mathrm{unravel}(T) \xrightarrow{a} U$ (with the same

U). A case analysis also shows, on the side of grammars, that word(T) and word(T') have exactly the same transitions whenever T' is a one-step unfolding of T:

(**Case** $T = \mathsf{skip}; T'$): here word(T) = word(skip) word(T') = word(T'), so word(T) and word(T') have exactly the same transitions.

(**Case** $T = Y$): let $Y \doteq T'$ be the equation corresponding to the identifier Y. If $Y\checkmark$ then (by lemma 7) $T'\checkmark$ so that (by lemma 20) word(Y) = word(T') = ε, having exacly the same transitions. Now suppose that $Y \not\checkmark$. Let $Z\delta$ = word(unravel(Y)), so that word(Y) = X for some X having exactly the transitions $X \xrightarrow{a} \gamma\delta$ for each transition $Z \xrightarrow{a} \gamma$. In particular, X and $Z\delta$ have exactly the same transitions.

(**Case** $T = Y; U$): let $Y \doteq V$ be the equation corresponding to the identifier Y. If $Y\checkmark$ then (by lemma 7) $V\checkmark$ so that (by lemma 7) word($Y;U$) = word(Y) word(U) = word(U) = word(V) word(U) = word($V;U$); thus $Y;U$ and $V;U$ have exactly the same transitions. Now suppose that $Y \not\checkmark$. Let $Z\delta$ = word(unravel(Y)), so that word(Y) = X for some X having exactly the transitions $X \xrightarrow{a} \gamma\delta$ for each transition $Z \xrightarrow{a} \gamma$. In particular, word($Y;U$) = X word(U) has exactly the same transitions as $Z\delta$ word(U) = word(unravel(Y);U).

(**Case** $T = (U;V);W$): here word(T) = word($U;V$) word(W) = word(U) word(V) word(W) = word(U) word($V;W$) = word($U;(V;W)$), so that word(T) and word(T') have exactly the same transitions.

Given that word(T), word(T') have exactly the same transitions, and since one-step unfoldings eventually reach unravel(T), a straightforward inductive argument shows that word(T) and word(unravel(T)) have exactly the same transitions, and thus word(unravel(T)) $\approx$ word(T) $\approx \gamma$. Finally, we can handle the case $T \neq$ unravel(T). Suppose that $T \xrightarrow{a} U$; then unravel(T) $\xrightarrow{a}$ U; since word(unravel(T)) $\approx \gamma$, our previous case analysis yields $\gamma \xrightarrow{a} \gamma'$ for some γ' s.t. word(U) $\approx \gamma'$; concluding that $(U, \gamma') \in \mathcal{R}$. In the other direction, suppose that $\gamma \xrightarrow{a} \gamma'$; since word(unravel($T$)) $\approx \gamma$, our previous case analysis yields unravel(T) $\xrightarrow{a} U$ for some U s.t. word(U) $\approx \gamma'$; therefore $T \xrightarrow{a} U$; concluding that $(U, \gamma') \in \mathcal{R}$. $\square$