Integrating Type Operators into the FreeST Programming Language

Paula Lopes, Diana Costa, and Vasco T. Vasconcelos

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

Abstract. Context-free session types emerged from the need to expand session type theory to non-regular protocols. Their inclusion in type systems enhances the expressiveness and adaptability of communication protocols in programming languages, yet pose a significant challenge for type equivalence algorithms. In this work, we study System $F_{\omega}^{\mu *;}$, the higher-order polymorphic lambda calculus equipped with equirecursive and context-free session types, as well as its seamless integration into FreeST, a functional programming language governed by context-free session types. We propose a bisimulation-based type equivalence for System $F_{\omega}^{\mu *;}$ and ensure decidability by reducing the problem to the bisimilarity of simple grammars. This approach offers a practical and efficient solution for type equivalence checking, supporting the robust implementation of advanced type systems in programming languages.

Keywords: Higher-order Kinds · Context-Free Session Types · Session Polymorphism · Type Equivalence · Simple Grammar.

1 Introduction

In the field of programming languages, the drive for more advanced and expressive type systems never stops. This journey has led us from the foundational System F[12] to the more intricate System F_{ω}^{μ}[4]. Integrating these advanced systems into real-world programming languages, however, comes with its own set of challenges, especially when it comes to type equivalence algorithms.

FreeST [1], a functional programming language based on system $F^{\mu;}$, is regulated by context-free session types. Context-free session types, unlike regular session types, are not restricted to tail recursion, allowing the description of more sophisticated communication protocols. Almeida et al. [2] developed the current type equivalence algorithm for FreeST, based on the translation of types into a simple grammar. The next step is to extend the language to a higher-order setting where type equivalence is still decidable.

In this work, we study the System $F_{\omega}^{\mu *;}$, the higher-order polymorphic lambda calculus equipped with equirecursive and context-free session types and its incorporation into a programming language. We propose a bisimulation-based type equivalence method for System $F_{\omega}^{\mu *;}$, aiming to provide a robust solution that not only handles the theoretical complexities but is also practical for implementation. We seek to bridge the gap between advanced type theory and practical

compiler design, ensuring that the powerful capabilities of context-free session types can be effectively utilized without compromising on performance or reliability.

Outline. The rest of the paper is organised as follows: Section 2 summarises the motivation behind our research; Section 3 introduces System $F_{\omega}^{\mu_*;}$ and type equivalence; Section 4 presents the challenges encountered during this research; Section 5 validates our work and Section 6 wraps up the paper.

2 Motivation

In the development of modern type systems, combining advanced features such as equirecursion, higher-order polymorphism, and higher-order context-free session types presents unique challenges and opportunities. The primary motivation for our research is to integrate these elements into a cohesive type system that can be practically incorporated into programming languages. Therefore, we are interested in practical algorithms for type equivalence checking to be incorporated into compilers.

Beyond first-order context-free session types. Types are categorised by kinds κ. For example, the recursive type $\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Leaf}\colon \mathsf{Skip}, \mathsf{Node}\colon \alpha;\ ?\mathsf{Int};\ \alpha\};\mathsf{Close}$ describes a protocol for safely streaming integer trees on a channel, where the recursion variable α is of kind session S. If the continuation of the channel is Leaf, then no communication occurs but the channel is still open for further composition whereas, if the continuation of the channel is Node, then we have a left subtree, followed by an integer and a right subtree. When the whole tree is received, the channel is closed. We want to go further, namely, we are interested in abstracting the type that is received on the tree channel, by writing $\lambda\beta\colon \mathrm{T}.\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Leaf}\colon \mathsf{Skip}, \mathsf{Node}\colon \alpha;\ ?\beta;\ \alpha\};\mathsf{Close}$, where T is the kind of functional types. Abstractions introduce higher-order kinds which lead to the introduction of type operators into our language.

Duality as an external macro (or not). Duality is the relationship between two session types that allows them to correctly engage in a protocol. For any session type describing one side of the communication (let us say the client), there is a dual session type that describes the other side (the server). For example, the session type $!\mathsf{Int}$ describes sending an integer and its dual session type is to receive an integer, $?\mathsf{Int}$. Typically, duality is offered as a built in constructor [8]. However, we believe duality should be internal, by means of a type operator.

3 System $F_{\omega}^{\mu_*;}$

In programming languages, terms are categorized by types, which in turn may be categorized by kinds. In our system, a *kind* κ is either a base kind $*$–which is either a session or a functional kind, S or T, respectively–or a higher-order

T ::= Type

ι	type constructor	$\lambda\alpha\colon\kappa.T$	type-level abstraction
α	type variable	$T\,T$	type-level application

ι ::= Type constructor

$\to$	$* \Rightarrow * \Rightarrow \mathrm{T}$	arrow	$(\!\vert\overline{l_i}\vert\!)$	$\overline{* \Rightarrow \mathrm{T}}$	record/variant
μ_κ	$(\kappa \Rightarrow \kappa) \Rightarrow \kappa$	recursive type	$\sharp$	$* \Rightarrow \mathrm{S}$	input/output
;	$\mathrm{S} \Rightarrow \mathrm{S} \Rightarrow \mathrm{S}$	seq.composition	$\odot\{\overline{l_i}\}$	$\overline{\mathrm{S} \Rightarrow \mathrm{S}}$	internal/external choice
$\exists\forall_\kappa$	$(\kappa \Rightarrow *) \Rightarrow \mathrm{S}$	exists/forall	Dual	$\mathrm{S} \Rightarrow \mathrm{S}$	dual operator
Wait	S	wait	Close	S	close
Skip	S	skip			

$(\!|\,|\!) ::= \{\} \mid \langle\rangle \qquad \sharp ::= ? \mid ! \qquad \odot ::= \oplus \mid \& \qquad \exists\forall_\kappa ::= \exists_\kappa \mid \forall_\kappa$

Fig. 1: The syntax of types and constructors.

kind $\kappa \Rightarrow \kappa'$. A *proper type* refers to a type that classifies a value, such as an integer or a boolean, thus has a base kind. In contrast, *type operators* act upon types to create more complex types, and are associated with higher-order kinds. For example, the type operator $\to$ takes two types, T and U, and constructs a new type, a function type $\to TU$, which we sometimes write in infix notation as $T \to U$. This is the core of our work. Our goal is to expand the programming language FreeST, currently limited to types of kind $*$, to higher-order types.

A type is either a type constructor ι, a type variable α, an abstraction $\lambda\alpha\colon\kappa.T$ or an application $T\,T$. A detailed list of type constructors is in Figure 1. Note that the last three constructors are actually type constants, while the rest are type constructors. Observe also that $\mu\,\alpha\colon\kappa.\,T$ is syntactic sugar for $\mu_\kappa(\lambda\alpha\colon\kappa.T)$ and likewise $\exists\forall\,\alpha\colon\kappa.\,T$ is an abbreviation for $\forall_\kappa(\lambda\alpha\colon\kappa.T)$ or $\exists_\kappa(\lambda\alpha\colon\kappa.T)$.

Before we can define well-formed types, we must define the normal form of a type; we do so by defining a system of reduction rules in Figure 2. This system is an adaptation of that proposed by Costa et al. [11] that is confluent, by adding the proviso that $T \neq T_1; T_2$ to rules R-Seq2 and R-DCtx. *Confluence* states that, if there are two distinct reduction paths for a given type, $T \to U$ and $T \to V$, then both paths will eventually converge into the same final reduced type W. Our variant features a single reduction path, thus confluence immediately follows. With respect to Costa et al. [11], we promoted polymorphism to kind S, which entails the introduction of the existential type $\exists_\kappa$, dual of $\forall_\kappa$, and the corresponding reduction rules. We also introduce reductions under abstractions with rule R-Abs.

A type is in *normal form*, denoted $T\,\mathsf{nf}$, if it has been completely reduced, *i.e.*, no further reductions are possible. In other words, $T\,\mathsf{nf}$ iff $T \not\to$. Then we can

Type reduction $\boxed{T \longrightarrow T}$

$$\text{R-Seq1}\quad \mathsf{Skip}; T \longrightarrow T$$

$$\text{R-Seq2}\quad \frac{T \longrightarrow V \qquad T \neq T_1; T_2}{T; U \longrightarrow V; U}$$

$$\text{R-Assoc}\quad (T; U); V \longrightarrow T; (U; V)$$

$$\text{R-}\mu\quad \mu_k\, T \longrightarrow T\,(\mu_k\, T)$$

$$\text{R-}\beta\quad (\lambda\alpha\colon \kappa.T)\, U \longrightarrow T[U/\alpha]$$

$$\text{R-Abs}\quad \frac{T \longrightarrow U}{\lambda\alpha\colon \kappa.T \longrightarrow \lambda\alpha\colon \kappa.U}$$

$$\text{R-TApplL}\quad \frac{T \longrightarrow U}{TV \longrightarrow UV}$$

$$\text{R-D;}\quad \mathsf{Dual}\,(T; U) \longrightarrow \mathsf{Dual}\,T; \mathsf{Dual}\,U$$

$$\text{R-DSkip}\quad \mathsf{Dual\ Skip} \longrightarrow \mathsf{Skip}$$

$$\text{R-DWait}\quad \mathsf{Dual\ Wait} \longrightarrow \mathsf{Close}$$

$$\text{R-DClose}\quad \mathsf{Dual\ Close} \longrightarrow \mathsf{Wait}$$

$$\text{R-D?}\quad \mathsf{Dual}\,(?\,T) \longrightarrow !\,T$$

$$\text{R-D!}\quad \mathsf{Dual}\,(!\,T) \longrightarrow ?\,T$$

$$\text{R-D\&}\quad \mathsf{Dual}\,(\&\{\overline{l_i : T_i}\}) \longrightarrow \oplus\{\overline{l_i : \mathsf{Dual}(T_i)}\}$$

$$\text{R-D}\oplus\quad \mathsf{Dual}\,(\oplus\{\overline{l_i : T_i}\}) \longrightarrow \&\{\overline{l_i : \mathsf{Dual}(T_i)}\}$$

$$\text{R-DCtx}\quad \frac{T \longrightarrow U \qquad T \neq T_1; T_2}{\mathsf{Dual}\,T \longrightarrow \mathsf{Dual}\,U}$$

$$\text{R-DDVar}\quad \mathsf{Dual}\,(\mathsf{Dual}\,(\alpha\ T_1 \ldots T_m)) \longrightarrow \alpha\ T_1 \ldots T_m$$

$$\text{R-D}\exists\quad \mathsf{Dual}\,(\exists_\kappa T) \longrightarrow \forall_\kappa T$$

$$\text{R-D}\forall\quad \mathsf{Dual}\,(\forall_\kappa T) \longrightarrow \exists_\kappa T$$

Fig. 2: Type reduction.

say that type T *normalises* to type U, written $T \Downarrow U$, if U nf and U is reached from T in a finite number of reduction steps. The predicate T norm means that $T \Downarrow U$ for some U.

Lastly, in order to define a well-formed type, we introduce the concept of *pre-kinding*. We denote this as $\Delta \vdash_{\mathsf{pre}} T : \kappa$, that is, T is pre-kinded with kind κ under the kinding context Δ, a map from type variables to kinds. The rules for pre-kinding can be found in Figure 3. Note that in rule PK-TAbs, $\Delta + \alpha\colon \kappa$ represents updating the pre-kind of the type variable α to a new pre-kind κ in the context Δ if $\alpha\colon \kappa' \in \Delta$ for some κ', or storing the pre-kind of α in the context Δ if otherwise. Combined with normalisation, pre-kinding enables us to decide type formation. In section 4 we explain why this step is relevant to our solution.

Type formation is defined by the judgment $\Delta \vdash T : \kappa$, where Δ is the kinding context, storing the kinds of type variables as in $\alpha\colon \kappa$. Expanding the work by Costa et al. [11], kinding rules now depend on the pre-kinding system. A type T is *well-formed* if T is pre-kinded, $\Delta \vdash_{\mathsf{pre}} T : \kappa$, and T normalises, T norm. We must consider a restriction in the kind of the recursive type μ_κ to $\kappa = *$ so that type formation is decidable.

Pre-kinding $\boxed{\Delta \vdash_{\mathsf{pre}} T : \kappa}$

$$\textsc{PK-Const}\ \Delta \vdash_{\mathsf{pre}} \iota : \kappa_\iota \qquad \textsc{PK-Var}\ \frac{\alpha : \kappa \in \Delta}{\Delta \vdash_{\mathsf{pre}} \alpha : \kappa} \qquad \textsc{PK-TAbs}\ \frac{\Delta + \alpha : \kappa \vdash_{\mathsf{pre}} T : \kappa'}{\Delta \vdash_{\mathsf{pre}} \lambda\alpha : \kappa.T : \kappa \Rightarrow \kappa'} \qquad \textsc{PK-TApp}\ \frac{\Delta \vdash_{\mathsf{pre}} T : \kappa \Rightarrow \kappa' \quad \Delta \vdash_{\mathsf{pre}} U : \kappa}{\Delta \vdash_{\mathsf{pre}} T\,U : \kappa'}$$

Fig. 3: Pre-kinding.

Type renaming $\boxed{\mathrm{rename}_S(T)}$

$$\begin{aligned}
\mathrm{rename}_S(\iota) &= \iota \\
\mathrm{rename}_S(\alpha) &= \alpha \\
\mathrm{rename}_S(\lambda\alpha : \kappa.T) &= \lambda\upsilon : \kappa.\mathrm{rename}_S(T[\upsilon/\alpha]) \quad \text{where } \upsilon = \mathrm{first}(S \cup \mathrm{fv}(\lambda\alpha : \kappa.T)) \\
\mathrm{rename}_S(T\,U) &= \mathrm{rename}_{S \cup \mathrm{fv}(U)}(T)\,\mathrm{rename}_S(U)
\end{aligned}$$

Fig. 4: Type renaming.

Type equivalence. It is expected that two types that are alpha-congruent are equivalent, like for example $\lambda\alpha : \kappa.\alpha$ and $\lambda\beta : \kappa.\beta$. In a bisimulation-based approach to type equivalence, the task of checking if two types are equivalent may involve the performance of (variable) substitutions on-the-fly as one crosses along the types. We will avoid this by performing a renaming operation once, right at the beginning on both types.

Inspired by the renaming approach of Gauthier and Pottier [6], we introduce *minimal renaming*, which uses the least amount of variable names necessary. This renaming operation, defined in Figure 4 consists on replacing a type T by its minimal alpha-conversion. We assume that a countable ordered set of (fresh) type variables, $\{\upsilon_1, \ldots, \upsilon_n, \ldots\}$, is available. We rename bound variables in T by the smallest possible variable available, *i.e.*, the first which is not free on the type. Note that the S parameter in the definition corresponds to the set of unavailable type variables for renaming. Also, by $\mathrm{first}(S)$ we mean the smallest variable not in the set S. The notion of variable substitution, $T[\upsilon/\alpha]$, and free variables of a type, $\mathrm{fv}(T)$, are the standard ones from literature [10].

Moving on, we introduce our notion of type equivalence based on type bisimulation, along the lines of Gay and Hole [7]. This bisimulation is built from the labelled transition system [13] described in Figure 5.

A type relation $\mathcal{R}$ is a *bisimulation* if for all pairs of types $(T, U) \in \mathcal{R}$ and transition label a we have:

- For each T' with $T \xrightarrow{a} T'$, there is a U' such that $U \xrightarrow{a} U'$ and $(T', U') \in \mathcal{R}$;
- For each U' with $U \xrightarrow{a} U'$, there is a T' such that $T \xrightarrow{a} T'$ and $(T', U') \in \mathcal{R}$.

$a ::= \alpha_i \mid \iota_i \mid \lambda\alpha\colon\kappa \qquad (i \geq 0, \iota \neq \mathsf{Skip}) \qquad$ Transition labels

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

$$\text{L-Red}\ \frac{T \longrightarrow U \quad U \xrightarrow{a} V}{T \xrightarrow{a} V} \qquad \text{L-Var1}\ \frac{m \geq 0}{\alpha\ T_1 \ldots T_m \xrightarrow{\alpha_0} \mathsf{Skip}} \qquad \text{L-Var2}\ \frac{1 \leq j \leq m}{\alpha\ T_1 \ldots T_m \xrightarrow{\alpha_j} T_j} \qquad \text{L-Const}\ \frac{\iota \neq \mathsf{Skip}}{\iota \xrightarrow{\iota_0} \mathsf{Skip}}$$

$$\text{L-ConstApp}\ \frac{\iota = \rightarrow, \odot\{\overline{\ell_i}\}, (\!|\overline{\ell_i}|\!) \quad 1 \leq j \leq m}{\iota\ T_1 \ldots T_m \xrightarrow{\iota_j} T_j} \qquad \text{L-Abs}\ \lambda\alpha\colon\kappa.T \xrightarrow{\lambda\alpha\colon\kappa} T \qquad \text{L-Quant1}\ \forall_\kappa\ T \xrightarrow{\forall_{\kappa 1}} T$$

$$\text{L-Quant2}\ \forall_\kappa\ T \xrightarrow{\forall_{\kappa 2}} \mathsf{Skip} \qquad \text{L-Msg1}\ \sharp T \xrightarrow{\sharp_1} T \qquad \text{L-Msg2}\ \sharp T \xrightarrow{\sharp_2} \mathsf{Skip} \qquad \text{L-Seq1}\ ; T \xrightarrow{;1} T \qquad \text{L-VarSeq1}\ \frac{m \geq 0}{(\alpha\ T_1 \ldots T_m); U \xrightarrow{\alpha_0} U}$$

$$\text{L-VarSeq2}\ \frac{1 \leq j \leq m}{(\alpha\ T_1 \ldots T_m); U \xrightarrow{\alpha_j} T_j} \qquad \text{L-QuantSeq1}\ (\forall_\kappa\ T); U \xrightarrow{\forall_{\kappa 1}} T \qquad \text{L-QuantSeq2}\ (\forall_\kappa\ T); U \xrightarrow{\forall_{\kappa 2}} U \qquad \text{L-MsgSeq1}\ \sharp T; U \xrightarrow{\sharp_1} T$$

$$\text{L-MsgSeq2}\ \sharp T; U \xrightarrow{\sharp_2} U \qquad \text{L-ChoiceSeq}\ \frac{1 \leq j \leq m}{\odot\{\overline{\ell_i\colon T_i}\}; U \xrightarrow{\odot\{\overline{\ell_i}\}_j} T_j; U} \qquad \text{L-WaitSeq}\ \mathsf{Wait}; U \xrightarrow{\mathsf{Wait}} \mathsf{Skip} \qquad \text{L-CloseSeq}\ \mathsf{Close}; U \xrightarrow{\mathsf{Close}} \mathsf{Skip}$$

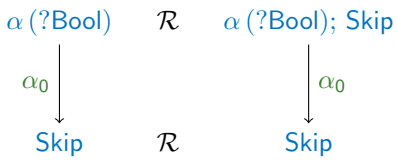
$$\text{L-DualVar1}\ \mathsf{Dual}\,(\alpha\ T_1 \ldots T_m) \xrightarrow{\mathsf{Dual}_1} \alpha\ T_1 \ldots T_m \qquad \text{L-DualVar2}\ \mathsf{Dual}\,(\alpha\ T_1 \ldots T_m) \xrightarrow{\mathsf{Dual}_2} \mathsf{Skip}$$

$$\text{L-DualSeq1}\ (\mathsf{Dual}\,(\alpha\ T_1 \ldots T_m)); U \xrightarrow{\mathsf{Dual}_1} \alpha\ T_1 \ldots T_m \qquad \text{L-DualSeq2}\ (\mathsf{Dual}\,(\alpha\ T_1 \ldots T_m)); U \xrightarrow{\mathsf{Dual}_2} U$$

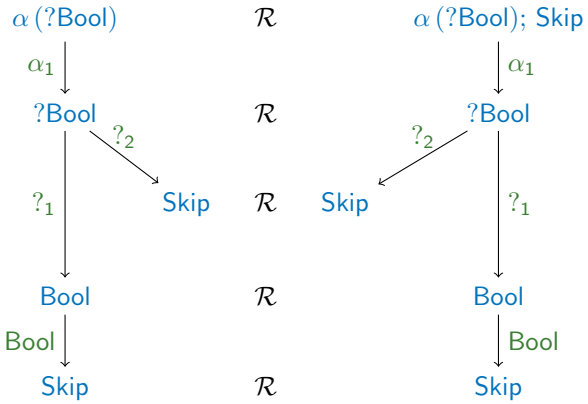
Fig. 5: Labelled transition system for types.

Two types T and U are *bisimilar*, written $T \sim U$, if there is a bisimulation $\mathcal{R}$ such that $(T, U) \in \mathcal{R}$. For example, consider type $T = \alpha\,(?\mathsf{Bool})$ and $U = \alpha\,(?\mathsf{Bool}); \mathsf{Skip}$. We show that $T \sim U$.

– $\alpha\,(?\mathsf{Bool}) \xrightarrow{\alpha_0} \mathsf{Skip}$ via rule L-Var1 and type $\alpha\,(?\mathsf{Bool}); \mathsf{Skip}$ shares the same transition to Skip through rule L-VarSeq1, $\alpha\,(?\mathsf{Bool}); \mathsf{Skip} \xrightarrow{\alpha_0} \mathsf{Skip}$. There are no transitions for type Skip.



– $\alpha\,(?\mathsf{Bool});\,\mathsf{Skip} \xrightarrow{\alpha_1} ?\mathsf{Bool}$ via rule L-VarSeq2 and type $\alpha\,(?\mathsf{Bool})$ shares the same transition to $?\mathsf{Bool}$ through rule L-Var2, $\alpha(?\mathsf{Bool}) \xrightarrow{\alpha_1} ?\mathsf{Bool}$. Not surprisingly, $?\mathsf{Bool}$ and $?\mathsf{Bool}$ will have the same transitions.



An equivalent approach is to translate types into grammars in Greibach normal form [3] and use a known algorithm to decide bisimilarity of grammars later.

A grammar in Greibach normal form is a tuple of the form $(\mathcal{T}, \mathcal{N}, \gamma, \mathcal{R})$, where:

– $\mathcal{T}$ is a finite set of terminal symbols, a, b, c;
– $\mathcal{N}$ is a finite set of non-terminal symbols, X, Y, Z;
– $\gamma \in \mathcal{N}^*$ is the starting word;
– $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{T} \times \mathcal{N}^*$ is a finite set of productions.

A production rule in $\mathcal{R}$ is written as $X \xrightarrow{a} \delta$: the left side of the arrow must be a non-terminal, $X \in \mathcal{N}$, while the right side must be a word. Note that δ can be the empty word. Grammars in GNF are *simple* when for every non-terminal and terminal symbol there is at most one production $X \xrightarrow{a} \delta$ [9].

Decidability of equivalence. Deciding whether two types are bisimilar takes two steps: the first phase is based on function word(T), described in Figure 6, that translates types to words of non-terminal symbols. This function terminates producing a simple grammar. Then, we have to check if two grammars are bisimilar, that is if word(T) $\approx$ word(U). The algorithm used to check the bisimilarity of simple grammars is the one introduced by Almeida et al. [2].

$\boxed{\mathrm{word}(T)}$

$$\mathrm{word}(T) = \begin{cases} \mathrm{word}'(T) & T \ \mathsf{nf} \\ \varepsilon & T \Downarrow \mathsf{Skip} \\ Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{a} \gamma\delta \mid a, \gamma : Z \xrightarrow{a} \gamma\} & T \Downarrow U \neq \mathsf{Skip}, \mathrm{word}(U) = Z\delta \end{cases}$$

$$\begin{aligned}
\mathrm{word}'(\alpha\ T_1 \ldots T_m) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\alpha_0} \varepsilon, Y \xrightarrow{\alpha_j} \mathrm{word}(T_j)\bot\} \\
\mathrm{word}'(\mathsf{Skip}) &= \varepsilon \\
\mathrm{word}'(\mathsf{Wait}) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\mathsf{Wait}} \bot\} \\
\mathrm{word}'(\mathsf{Close}) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\mathsf{Close}} \bot\} \\
\mathrm{word}'(\lambda\alpha\colon\kappa.T) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\lambda\alpha:\kappa} \mathrm{word}(T)\} \\
\mathrm{word}'(\iota)^{*} &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\iota} \varepsilon\} \\
\mathrm{word}'(\iota\ T_1 \cdots T_m)^{**} &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\iota_j} \mathrm{word}(T_j)\} \\
\mathrm{word}'(\sharp T) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\sharp_1} \mathrm{word}(T)\bot, Y \xrightarrow{\sharp_2} \varepsilon\} \\
\mathrm{word}'(\exists\forall_\kappa T) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\exists\forall_{\kappa 1}} \mathrm{word}(T)\bot, Y \xrightarrow{\exists\forall_{\kappa 2}} \varepsilon\} \\
\mathrm{word}'(;\ T) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{;_1} \mathrm{word}(T)\} \\
\mathrm{word}'(T; U) &= \mathrm{word}(T)\,\mathrm{word}(U) \\
\mathrm{word}'(\mathsf{Dual}\ (\alpha\ T_1 \ldots T_m)) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\mathsf{Dual}_1} \mathrm{word}(\alpha\ T_1 \ldots T_m), Y \xrightarrow{\mathsf{Dual}_2} \varepsilon\}
\end{aligned}$$

* $\iota \neq \mathsf{Skip}, \mathsf{Wait}, \mathsf{Close}$
** $\iota = \rightarrow, \odot\{\overline{l_i}\}, \langle\!|\overline{l_i}|\!\rangle$
Y is a fresh non-terminal symbol
ϵ is the empty word
$\bot$ is a non-terminal symbol without productions

Fig. 6: Function word(T).

4 Challenges

In the original system, the rules for type formation instated that for an application type to be kinded, the type must normalise. So, types that reduce indefinitely are not considered well-formed.

$$\text{K-TApp}\quad \frac{\Delta \vdash T : \kappa \Rightarrow \kappa' \quad \Delta \vdash U : \kappa \quad T\,U\ \mathsf{norm}}{\Delta \vdash T\,U : \kappa'}$$

However, determining whether TU norm may not terminate. For example, the infamous type $(\lambda x.x\ x)(\lambda x.x\ x)$ never reaches a normal form. In order to address the challenge of non-terminating normalisation, we adopted a *pre-kinding*

approach. The type $(\lambda x.x\ x)(\lambda x.x\ x)$ for example, is not pre-kinded, so it is discarded right away.

For a type which is pre-kinded, we can guarantee termination of T norm. In particular, we found a way to deal with other problematic types for normalisation, as are recursive types. Note that the reduction of recursive types might not decrease the size of the type. For example, the type $\mu_{\mathrm{S}}\ (\lambda\alpha\colon \mathrm{S}.\alpha)$ is pre-kinded but it keeps reducing to itself in two steps, so we must conclude that it does not normalise. When we deal with normalisation, we will separate the way we treat recursive types from the rest. In particular, we divide the reduction rules into two groups: $\longrightarrow_{\mu}$ refers to reductions that use the R-μ rule and $\longrightarrow_{\beta;D}$ refers to reductions that never invoke the R-μ rule. Thus, $\longrightarrow\ =\ \longrightarrow_{\beta;D} \cup \longrightarrow_{\mu}$. We can now extend this notion into normalisation $T \Downarrow_{\beta;D} U$ and $T \Downarrow_{\mu} U$ respectively.

Decidability of type formation. Take Δ, T, κ. First we determine if $\Delta \vdash_{\mathsf{pre}} T : \kappa$. If T fails to be pre-kinded, it is not kinded either. Otherwise, we check whether T normalises, specifically seeking infinite reduction sequences. In the presence of recursive types, such sequences would have between two μ-reductions a finite number of β-reductions. $T = T_0 \Downarrow_{\beta;D} T_0' \longrightarrow_{\mu} T_1 \Downarrow_{\beta;D} T_1' \longrightarrow_{\mu} T_2 \Downarrow_{\beta;D} T_2' \longrightarrow_{\mu} \cdots$. If T_i' does not reduce by any μ-reduction, we can determine that T normalises. Otherwise, since $\mu_*\ U$ is restricted to the base kind $*$, it must reduce by one of these:

$$
\begin{aligned}
&T_i' = \mu_*\ U \longrightarrow_{\mu} U\ (\mu_*\ U) && \text{(R-}\mu\text{)}\\
&T_i' = (\mu_*\ U); V \longrightarrow_{\mu} (U\ (\mu_*\ U)); V && \text{(R-Seq2)}\\
&T_i' = \mathsf{Dual}\ (\mu_*\ U) \longrightarrow_{\mu} \mathsf{Dual}\ (U\ (\mu_*\ U)) && \text{(R-DCtx)}\\
&T_i' = (\mathsf{Dual}\ (\mu_*\ U)); V \longrightarrow_{\mu} (\mathsf{Dual}\ (U\ (\mu_*\ U))); V && \text{(R-Seq2 + R-DCtx)}
\end{aligned}
$$

We can easily notice that the expression $\mu_*\ U$ reappears after the μ-reduction, indicating potential infinite sequences. We can detect these by tracking occurrences of $\mu_*\ U$ and halting if a repetition is found. The following pseudo-code illustrates the process for determining if T normalises:

```
normalises(visited, t) =
    if reducesByBSD(t) then
        normalises(visited, t') -- t → t'
    else if memberOf(t, visited) then
        Nothing -- found an infinite sequence
    else if reducesByMu(t) then
        normalises(visited', t') -- update visited set
            with t
    else t
```

5 Validation

Our validation process uses a suite of randomly generated types, leveraging the Quickcheck library[5] to ensure these types have specific properties. An arbitrary

type generator is defined using the *Arbitrary* typeclass, employing the *frequency* function to generate type operators with specific probabilities. Variables are selected from a predefined range, abstractions are created by generating a variable, a kind, and a sub-type, and applications are formed by recursively generating two sub-types. The *sized* function is used to control the size of the generated types, ensuring manageable recursion depth. For better statistics we ensure proper distribution of type constructors. We present next the list of properties tested with a *maxSuccess* of 200.000 tests:

Property	**Description**	**Number of Tests**
prop_rename	Validates node structure remains unchanged after renaming	Passed all
prop_tree_structure	Confirms renaming maintains tree structure	Passed all
prop_rename_idempotent	Checks that renaming twice is equivalent to once	Passed all
prop_reduction_preserves_renaming	If $T \longrightarrow U$ and T = rename(T) then U = rename(U)	Passed 23464 Discarded 200000
prop_renaming_preserves_alpha_congruence	Ensures alpha congruent types remain congruent after renaming	Passed 5858 Discarded 200000
prop_renaming_reflects_alpha_congruence	If T and U are alpha-congruent, then rename(T) = rename(U)	Passed 5858 Discarded 200000
prop_nf_does_not_reduce	Ensures that types in normal form do not reduce further	Passed 2374 Discarded 200000
prop_reduced_is_not_nf	If $T \longrightarrow U$ then $\mathsf{not}(T\ \mathsf{nf})$	Passed 2374 Discarded 200000
prop_preservation	If $\Delta \vdash T : \kappa$ and $T \longrightarrow U$, then $\Delta \vdash U : \kappa$	Passed 576 Discarded 200000

Table 1: Tested properties with Quickcheck.

Data was collected on a machine equipped with an Apple M3 Pro and 18GB of RAM, and tested with Haskell's version 9.6.3.

While randomly generated types facilitate a robust analysis, certain properties, such as the preservation property, prove challenging to test comprehensively. This difficulty arises from the simplicity of our generator and the inherent probability of randomly generated test cases preserving such properties. To achieve better results, more complex generators tailored to specific properties would be required, though such generators are often challenging to design and implement.

6 Conclusion

To summarise, we explored the integration of $F^{\mu *}_{\omega}$ with context-free session types into the functional programming language FreeST. Context-free session types enhance the expressiveness and adaptability of communication protocols in programming languages by extending beyond the limitations of regular session types. Our work addressed the significant challenges posed by type equivalence algorithms within this advanced type system. Our findings underscore the importance of handling recursive types separately to ensure termination of normalisation. By refining the reduction rules and employing a pre-kinding approach, we are able to decide type formation. We proposed a bisimulation-based type equivalence method for System $F^{\mu *;}_{\omega}$, demonstrating its practicality and efficiency in ensuring decidability. By reducing the problem to the bisimilarity of simple grammars, we provided a robust solution for type equivalence checking, facilitating the implementation of advanced type systems in real-world programming languages.

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 (https://doi.org/10.54499/UIDB/00408/2020) and ref. UIDP/00408/2020 (https://doi.org/10.54499/UIDP/00408/2020).

References

1. Almeida, B., Mordido, A., Thiemann, P., Vasconcelos, V.T.: Polymorphic lambda calculus with context-free session types. Inf. Comput. **289**(Part), 104948 (2022). https://doi.org/10.1016/J.IC.2022.104948
2. Almeida, B., Mordido, A., Vasconcelos, V.T.: Deciding the bisimilarity of context-free session types. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 39–56. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_3
3. Autebert, J., Boasson, L., Gabarró, J.: Context-free grammars in Greibach normal forms. Bull. EATCS **24**, 44–47 (1984)
4. Cai, Y., Giarrusso, P.G., Ostermann, K.: System F-omega with equirecursive types for datatype-generic programming. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 30–43. ACM (2016). https://doi.org/10.1145/2837614.2837660
5. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. In: Odersky, M., Wadler, P. (eds.) Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000. pp. 268–279. ACM (2000). https://doi.org/10.1145/351240.351266

6. Gauthier, N., Pottier, F.: Numbering matters: first-order canonical forms for second-order recursive types. In: Okasaki, C., Fisher, K. (eds.) Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004. pp. 150–161. ACM (2004). https://doi.org/10.1145/1016850.1016872
7. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2-3), 191–225 (2005). https://doi.org/10.1007/S00236-005-0177-Z
8. Gay, S.J., Thiemann, P., Vasconcelos, V.T.: Duality of session types: The final cut. In: Balzer, S., Padovani, L. (eds.) Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020. EPTCS, vol. 314, pp. 23–33 (2020). https://doi.org/10.4204/EPTCS.314.3
9. Korenjak, A.J., Hopcroft, J.E.: Simple deterministic languages. In: 7th Annual Symposium on Switching and Automata Theory, Berkeley, California, USA, October 23-25, 1966. pp. 36–46. IEEE Computer Society (1966). https://doi.org/10.1109/SWAT.1966.22
10. Pierce, B.: Types and programming languages. MIT Press ((2002))
11. Poças, D., Costa, D., Mordido, A., Vasconcelos, V.T.: System f^{μ} ømega with context-free session types. In: Wies, T. (ed.) Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13990, pp. 392–420. Springer (2023). https://doi.org/10.1007/978-3-031-30044-8_15
12. Reynolds, J.C.: Towards a theory of type structure. In: Robinet, B.J. (ed.) Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974. Lecture Notes in Computer Science, vol. 19, pp. 408–423. Springer (1974). https://doi.org/10.1007/3-540-06859-7_148
13. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press (2011)