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 follow a bisimulation-based approach to type equivalence, translating types into simple grammars and verifying the bisimilarity of these grammars, a problem that is decidable.

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[9] 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.

FreeST [1], a concurrent 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, thus allowing the description of more sophisticated communication protocols. FreeST's current type equivalence algorithm, developed by Almeida et al. [2] decides the equivalence of context-free session types by reducing the problem to the bisimilarity of simple grammars. 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 follow a bisimulation-based type equivalence, proposed by Poças et al. [8], which supports the robust implementation of advanced type systems in programming languages. 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.

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.

Context-free session types and beyond. Context-free session types break free from tail recursion by introducing the sequential composition operator ; and the type Skip—neutral element of sequential composition—which represents a protocol with no actions. The recursive type $\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Leaf}\colon \mathsf{Skip}, \mathsf{Node}\colon \alpha;\ ?\mathsf{Int};\ \alpha\};\mathsf{End}_?$ describes a protocol for safely streaming integer trees on a channel. The channel presents an external choice & with two labels: 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. It is also important to distinguish type $\mathsf{End}_?$ from Skip—the former represents the closure of a channel, where no further communication is possible, while the latter allows continuing the communication. We want to move beyond context-free session types, namely, we are interested in abstracting the type that is received on the tree channel, by writing $\lambda\beta\colon \mathrm{T}.\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Leaf}\colon \mathsf{Skip}, \mathsf{Node}\colon \alpha;\ ?\beta;\ \alpha\};\mathsf{Wait}$. 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 !Int describes sending an integer and its dual session type is to receive an integer, ?Int. Typically, duality is offered as a built in constructor [6]. However, we believe duality should be internal, by means of a type operator, since in a polymorphic setting duality cannot be eliminated without the introduction of co-variables.

Outline. The rest of the paper is organised as follows: Section 2 introduces System $F^{\mu_{*}^{;}}_{\omega}$ and type equivalence; Section 3 presents the challenges encountered during this research and prove the decidability of type formation; Section 4 explains the decidability of type equivalence; Section 5 describes our implementation process and validates our work and Section 6 wraps up the paper.

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
$\exists\forall_\kappa$	$(\kappa \Rightarrow *) \Rightarrow \mathrm{T}$	exists/forall	Dual	$\mathrm{S} \Rightarrow \mathrm{S}$	dual operator
$\mathsf{End}_\sharp$	S	wait/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.

2 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 kind $\kappa \Rightarrow \kappa'$. A *proper type* refers to a type that 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 kinds.

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 types and constructors is in Fig. 1. Observe that $\mu\,\alpha\colon\kappa.T$ is syntactic sugar for $\mu_\kappa(\lambda\alpha\colon\kappa.T)$ and similarly for $\exists\forall\,\alpha\colon\kappa.T$.

Not all types are of interest—for example $\mathsf{Dual}\,\forall_{\mathrm{S}}\lambda\alpha\colon\mathrm{S}.\alpha$ since the dual operator makes no sense being applied to a functional type. Before we can discuss type formation, we must define the weak head normal form of a type; we do so by defining a system of reduction rules in Fig. 2. This system is that of Poças et al. [8] made 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 reductions 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.

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

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

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

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

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

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

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

$$\text{R-TAppL}\quad \frac{T \to U}{TV \to UV}$$

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

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

$$\text{R-DWait}\quad \mathsf{Dual\ End}_? \to \mathsf{End}_!$$

$$\text{R-DClose}\quad \mathsf{Dual\ End}_! \to \mathsf{End}_?$$

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

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

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

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

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

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

Fig. 2. Type reduction.

Then we can say that type T *normalises* to type U, written $T \Downarrow U$, if U whnf 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. Note that not all types normalise, i.e., some have an infinite sequence of reductions, such as $T = (\lambda x.xx)(\lambda x.xx) \longrightarrow (\lambda x.xx)[(\lambda x.xx)/x] = T \longrightarrow \cdots$–which is stuck in a loop of reductions to itself via the application of rule R-β–and $U = \mu_{\mathsf{S}}(\lambda\alpha\colon \mathsf{S}.\alpha; \mathsf{Skip}) \to (\lambda\alpha\colon \mathsf{S}.\alpha; \mathsf{Skip})U \to U; \mathsf{Skip} \to_2 U; \mathsf{Skip}; \mathsf{Skip} \to \cdots$–which successively applies rules R-μ, R-β in combination with R-Seq2, resulting in the unending addition of a trailing Skip.

Finally, we introduce *well-formed types*. We use $\Delta \vdash T : \kappa$ to denote that T is a *well-formed type* with kind κ under the kinding context Δ, a map from type variables to kinds. The kinds of constants can be found in Fig. 1. A variable α has kind κ if $\alpha\colon \kappa \in \Delta$. An abstraction $\lambda\alpha\colon \kappa.T$ has kind $\kappa \Rightarrow \kappa'$ if T is well-formed. Note that $\Delta + \alpha\colon \kappa$ in rule K-TAbs represents updating the kind of the type variable α to a new kind κ in the context Δ if $\alpha\colon \kappa' \in \Delta$ for some κ', or storing the kind of α in the context Δ if otherwise. Finally, rule K-TApp states that an application $T\,U$ is well-formed if T and U are types and $T\,U$ normalises, that is, $T\,U$ norm. In Section 3 we prove decidability of type formation, imposing a restriction to kind $*$ for recursive types, also adopted by Poças et al. in [**?**].

Type equivalence. Type equivalence allows us to check whether two types, even if syntactically different, correspond to the same protocol. It is expected that two types that are alpha-congruent are equivalent, like for example $\lambda\alpha\colon \kappa.\alpha$ and

Type formation $\boxed{\Delta \vdash T : \kappa}$

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

Fig. 3. Type formation.

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. 4. Pre-kinding.

$\lambda\beta : \kappa.\beta$. However, the task of checking if two types are equivalent may involve substitutions on-the-fly as one crosses along the types. We will avoid this by performing a renaming operation once on both types, right at the beginning of the type equivalence checking process. We follow the notion of renaming in [8].

3 Deciding Type Formation

The rules for type formation in Fig. 3 involve determining if an application type TU normalises. Poças et al. [8] propose a two-step solution to this problem. The first stage is the introduction of 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. The rules for pre-kinding are in Fig. 4. They differ from the rules for type formation in that, in rule PK-TApp, there is no verification of the normalisation of TU. Pre-kinding excludes some (but not all) types that do not normalise, as is the case of $(\lambda\alpha : \kappa.\alpha\alpha)(\lambda\alpha : \kappa.\alpha\alpha)$ in Fig. 5.

For a type which is pre-kinded, termination of $T\ \mathsf{norm}$ is guaranteed. Some recursive types are problematic for normalisation, as the application of reduction might not decrease their size. For example, the type $\mu_{\mathsf{S}}\ (\lambda\alpha : \mathsf{S}.\alpha; \mathsf{Skip})$ is pre-kinded but successive reduction steps—via R-μ and R-β—keep adding $;\mathsf{Skip}$ to the tail of the type so we must conclude that it does not normalise. When dealing with normalisation, we separate the treatment of recursive types from the remaining types. In particular, we divide the reduction rules in 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, $\rightarrow = \longrightarrow_{\beta;D} \cup \longrightarrow_\mu$. We may now lift this notion to normalisation, denoted by $T \Downarrow_{\beta;D} U$ and $T \Downarrow_\mu U$ respectively.

In order to check if a type T is well-formed, we first determine if $\vdash_{\mathsf{pre}} T : \kappa$ for some κ. If T fails to be pre-kinded, it is not kinded either. Otherwise, we check whether $\vdash T : \kappa$, which involves determining if the application types

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\kappa = \kappa'' \Rightarrow \kappa'}\,\text{PK-Var}}{\alpha : \kappa \vdash_{\mathsf{pre}} \alpha : \kappa'' \Rightarrow \kappa'} \quad \dfrac{\dfrac{}{\bot \quad (\kappa \neq \kappa'')}\,\text{PK-Var}}{\alpha : \kappa \vdash_{\mathsf{pre}} \alpha : \kappa''}}{\alpha : \kappa \vdash_{\mathsf{pre}} \alpha\alpha : \kappa'}\,\text{PK-TApp}}{\vdash_{\mathsf{pre}} \lambda\alpha\colon \kappa.\alpha\alpha : \kappa \Rightarrow \kappa'}\,\text{PK-TAbs} \qquad \dfrac{\vdots}{\vdash_{\mathsf{pre}} \lambda\alpha\colon \kappa.\alpha\alpha : \kappa}}{\vdash_{\mathsf{pre}} (\lambda\alpha\colon \kappa.\alpha\alpha)(\lambda\alpha\colon \kappa.\alpha\alpha) : \kappa'}\,\text{PK-TApp}$$

Fig. 5. Example of an unsuccessful derivation $\vdash_{\mathsf{pre}} (\lambda\alpha\colon \kappa.\alpha\alpha)(\lambda\alpha\colon \kappa.\alpha\alpha) : \kappa'$.

within T normalise. The approach to determine if a type normalises seeks infinite reduction sequences. In the case of recursive types, such sequences would have a finite number of β-reductions between two μ-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 conclude that T normalises. Otherwise, since $\mu_* U$ is restricted to a base kind $*$, it must reduce by one of following cases.

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

We can easily notice that 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.

4 Deciding Type Equivalence

Following Poças et al. [8], the problem of checking whether two (renamed) types are equivalent is reduced to translating types into grammars and checking bisimilarity. A grammar in *Greibach normal form* [3] is a tuple $(\mathcal{T}, \mathcal{N}, \gamma, \mathcal{R})$, 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 production rules.

A production rule in $\mathcal{R}$ is written as $X \xrightarrow{a} \delta$. Grammars in GNF are *simple* when, for every non-terminal symbol X and every terminal symbol a, there is at most one production rule $X \xrightarrow{a} \delta$ [7].

The function word(T), described in Fig. 6, translates types to words of non-terminal symbols. If a type T is in weak head normal form, the construction of word(T) updates the set of productions of T, according to one of the cases found in word$'$. If T is not in weak head normal form and normalises to Skip, word(T) returns the empty word; otherwise, if there exists a type $U \neq \mathsf{Skip}$ such that T normalises to U, word(U) $= Z\delta$ and Y a fresh new terminal, then for each

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

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

$$\begin{aligned}
\text{word}'(\alpha\ T_1 \ldots T_m) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\alpha_0} \varepsilon, Y \xrightarrow{\alpha_j} \text{word}(T_j)\bot\} \\
\text{word}'(\mathsf{Skip}) &= \varepsilon \\
\text{word}'(\mathsf{End}_\sharp) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\mathsf{End}_\sharp} \bot\} \\
\text{word}'(\lambda\alpha\colon \kappa.T) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\lambda\alpha\colon\kappa} \text{word}(T)\} \\
\text{word}'(\iota) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\iota} \varepsilon\} \quad \textit{where } \iota \neq \mathsf{Skip}, \mathsf{End}_\sharp \\
\text{word}'(\iota\ T_1 \cdots T_m) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\iota_j} \text{word}(T_j)\} \quad \textit{where } \iota = \rightarrow, \odot\{\overline{l_i}\}, \langle\!|\overline{l_i}|\!\rangle \\
\text{word}'(\sharp T) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\sharp_1} \text{word}(T)\bot, Y \xrightarrow{\sharp_2} \varepsilon\} \\
\text{word}'(\exists\forall_\kappa T) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\exists\forall_\kappa} \text{word}(T)\bot\} \\
\text{word}'(;\ T) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{;1} \text{word}(T)\} \\
\text{word}'(T; U) &= \text{word}(T)\,\text{word}(U) \\
\text{word}'(\mathsf{Dual}\ (\alpha\ T_1 \ldots T_m)) &= Y, \mathcal{R} := \mathcal{R} \cup \{Y \xrightarrow{\mathsf{Dual}_1} \text{word}(\alpha\ T_1 \ldots T_m), Y \xrightarrow{\mathsf{Dual}_2} \varepsilon\}
\end{aligned}$$

Y is a fresh non-terminal symbol in all cases,
ϵ is the empty word,
and $\bot$ is a non-terminal symbol without productions.

Fig. 6. Function word(T).

production of Z of the form $Z \xrightarrow{a} \gamma$, Y has a production of the form $Y \xrightarrow{a} \gamma\delta$. The application of the word function to a type T terminates producing a simple grammar. This is only possible because our well-formed types normalise, and all of its subterms normalise as well. Furthermore, we keep track of already visited types which enable reusing non-terminal symbols, which is crucial for dealing with recursive types.

We check whether two types are equivalent by translating the (renamed) types to a simple grammar, and then checking their bisimilarity, i.e., if word(T) $\approx$ word(U). The algorithm used to check bisimilarity of simple grammars is in [2].

Consider the type $T_0 = \lambda\beta\colon \mathrm{T}.\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Leaf}\colon \mathsf{Skip}, \mathsf{Node}\colon \alpha; ?\beta; \alpha\}; \mathsf{Wait}$ described in Section 1. We will demonstrate how the construction of word(T_0) terminates generating a simple grammar. Since T_0 is in weak head normal form, word(T_0) returns a fresh symbol, which we call X_0. We also add to the set of productions the production $X_0 \xrightarrow{\lambda\beta\colon \mathrm{T}}$ word(T_1), where T_1 is the type $\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Leaf}\colon \mathsf{Skip}, \mathsf{Node}\colon \alpha; ?\beta; \alpha\}; \mathsf{Wait}$.

Now T_1 is not in weak head normal form, so we must normalise it in order to obtain T_2 such that $T_1 \Downarrow T_2$. Then, word(T_1) returns a fresh non-terminal which we call X_1. To obtain the productions of T_1, we need to compute word(T_2), that returns a fresh symbol X_2. Since $T_2 = \&\{\mathsf{Leaf}\colon \mathsf{Skip}, \mathsf{Node}\colon T_1; ?\beta; T_1\}; \mathsf{Wait}$ is in weak head normal form, we need to first compute word(T_2) = word(T_3) word(Wait), where $T_3 = \&\{\mathsf{Leaf}\colon \mathsf{Skip}, \mathsf{Node}\colon T_1; ?\beta; T_1\}$. We have that word(Wait) = X_4 and $X_4 \xrightarrow{\mathsf{Wait}} \bot$ but we still need to compute word(T_3). This computation results in a fresh non-terminal X_3 with productions $X_3 \xrightarrow{\&_1}$ word(Skip) and $X_3 \xrightarrow{\&_2}$ word($T_1; ?\beta; T_1$). Therefore, we the transitions for X_2 are $X_2 \xrightarrow{\&_1} X_4$ and $X_2 \xrightarrow{\&_2} X_3 X_4$.

At last, we must compute word($T_1; ?\beta; T_1$), which is a fresh symbol X_5, because this type is not in weak head normal form. This type normalises to $T_2; ?\beta; T_1$, since $T_1 \Downarrow T_2$, therefore the productions of X_5 are the concatenation of word(T_2) word($?\beta$) word(T_1). At this point, we know that word(T_2) = X_2 and word(T_1) = X_1. Thus, we just need to compute word($?\beta$), which is a fresh symbol X_6 with productions $X_6 \xrightarrow{?_1}$ word(β)$\bot$ and $X_6 \xrightarrow{?_2} \varepsilon$. Finally, word($\beta$) is a fresh symbol X_7 with a production $X_7 \xrightarrow{\beta} \varepsilon$. This means that word($T_2; ?\beta; T_1$) = $X_2 X_6 X_1$, which we write as $X_5 \xrightarrow{\&_1} X_4 X_6 X_1$ and $X_5 \xrightarrow{\&_2} X_3 X_4 X_6 X_1$.

Putting everything together, we obtain the following simple grammar:

$$X_0 \xrightarrow{\lambda\beta:\mathrm{T}} X_1 \qquad X_1 \xrightarrow{\&_1} X_4 \quad X_1 \xrightarrow{\&_2} X_3 X_4 \quad X_2 \xrightarrow{\&_1} X_4$$
$$X_2 \xrightarrow{\&_2} X_3 X_4 \qquad X_3 \xrightarrow{\&_1} \varepsilon \quad X_3 \xrightarrow{\&_2} X_5 \quad X_4 \xrightarrow{\mathsf{Wait}} \bot$$
$$X_5 \xrightarrow{\&_1} X_4 X_6 X_1 \quad X_5 \xrightarrow{\&_2} X_3 X_4 X_6 X_1 \quad X_6 \xrightarrow{?_1} X_7 \bot \quad X_6 \xrightarrow{?_2} \varepsilon \quad X_7 \xrightarrow{\beta} \varepsilon$$

5 Implementation and Validation

Implementation consists on eight modules written in Haskell, as described in Table 1.

The current FreeST compiler features an algorithm for checking the bisimilarity of simple grammars, which we use for testing. The testing process takes a suite of randomly generated types—a small subset of FreeST's types, based on the syntax presented in Fig. 1—leveraging the Quickcheck library [5] to ensure these types have specific properties. Formal proofs regarding decidability of type formation and equivalence can be found elsewhere [8].

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. The list of properties can be found in Table 2. A total of 200.000 tests were made for each property.

Module name	LoC	Description
Syntax	118	Defines the `Type` data constructor as well as the higher-order kind system, based on Fig. 1.
Substitution	50	Implements capture-avoiding substitution on types.
Normalisation	80	Reduces types to weak head normal form, i.e., until no further reduction is possible.
TypeFormation	58	Implements the type checking algorithm.
Rename	30	Renames bound variables in a type by the smallest possible variable available, i.e., the first which is not free in the type.
WeakHeadNormalForm	86	Checks whether a type is in weak head normal form.
Grammar	74	Defines the `Grammar` data constructor, based on the definition found in Section 4.
TypeToGrammar	179	Implements the function word, that converts types into simple grammars.

Table 1. Haskell modules.

Property	Tests passed	Tests discarded
If T whnf then $T \nrightarrow$	200.000	24.643
If $T \to U$ then not(T whnf)	200.000	1.671.940
If $\vdash T : \kappa$ and $T \to U$ then $\vdash U : \kappa$	90	2.000.000
If $\vdash T : \kappa$ and $T \to U$ then word(T) $\sim$ word(U)	90	2.000.000

Table 2. Properties tested 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 type-formation preservation property and bisimilarity of simple grammars, prove challenging to test comprehensively. The difficulty arises from the simplicity of our generator and the inherent low probability that randomly generated test cases. The probability of generating a type T that is both well-formed and reduces is quite small. Therefore, most of the tests cases do not satisfy the predicate, $\vdash T : \kappa$ and $T \to U$, and Quickcheck ends up discarding 2.000.000 tests for the last two properties. To achieve better results, more

complex generators, tailored to specific properties, would be required. Such generators are often challenging to design and implement.

6 Conclusion

To summarize, we investigated 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, surpassing the limitations of regular session types.

Our research tackled the significant challenges posed by type equivalence algorithms within this advanced type system. We emphasized the importance of handling recursive types separately to ensure the termination of normalisation. By refining reduction rules and employing a pre-kinding approach, type formation is decidable.

By reducing the problem to the bisimilarity of simple grammars, a robust solution for type equivalence checking is met, 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, and by the LASIGE Research Unit ref. UIDB/00408/2020 and 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: TACAS. LNCS, 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: POPL. 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: ICFP. pp. 268–279. ACM (2000). https://doi.org/10.1145/351240.351266
6. Gay, S.J., Thiemann, P., Vasconcelos, V.T.: Duality of session types: The final cut. In: PLACES. EPTCS, vol. 314, pp. 23–33 (2020). https://doi.org/10.4204/EPTCS.314.3
7. Korenjak, A.J., Hopcroft, J.E.: Simple deterministic languages. In: 7th Annual Symposium on Switching and Automata Theory. pp. 36–46. IEEE Computer Society (1966). https://doi.org/10.1109/SWAT.1966.22

8. Poças, D., Costa, D., Mordido, A., Vasconcelos, V.T.: System f^{μ} ømega with context-free session types. In: ESOP. LNCS, vol. 13990, pp. 392–420. Springer (2023). https://doi.org/10.1007/978-3-031-30044-8_15, https://doi.org/10.1007/978-3-031-30044-8_15
9. Reynolds, J.C.: Towards a theory of type structure. In: Programming Symposium, Proceedings Colloque sur la Programmation. LNCS, vol. 19, pp. 408–423. Springer (1974). https://doi.org/10.1007/3-540-06859-7_148