System $F^{\mu;}_{\omega}$ with Context-free Session Types[⋆]

Diogo Poças(✉), Diana Costa, and Andreia Mordido, and Vasco T. Vasconcelos

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal
{dmpocas,dfdcosta,afmordido,vmvasconcelos}@ciencias.ulisboa.pt

Abstract. We study increasingly expressive type systems, from F^{μ}—an extension of the polymorphic lambda calculus with equirecursive types—to $F^{\mu;}_{\omega}$—the higher-order polymorphic lambda calculus with equirecursive types and context-free session types. Type equivalence is given by a standard bisimulation defined over a novel labelled transition system for types. Our system subsumes the contractive fragment of F^{μ}_{ω} as studied in the literature. Decidability results for type equivalence of the various type languages are obtained from the translation of types into objects of an appropriate computational model: finite-state automata, simple grammars and deterministic pushdown automata. We show that type equivalence is decidable for a significant fragment of the type language. We further propose a message-passing, concurrent functional language equipped with the expressive type language and show that it enjoys preservation and absence of runtime errors for typable processes.

Keywords: System F, Higher-order kinds, Context-free session types

1 Introduction

Extensions of the λ-calculus to include increasingly sophisticated type structures have been extensively studied and have led to systems whose importance is widely recognized: System F [60], System F^{μ} [30], System F_{ω} [36], System F^{μ}_{ω} [14]. Ideally, we would like to combine a *wishlist* of type structures and get a super-powerful system with vast expressiveness. However, the expressiveness of types is naturally limited by the universe where they are supposed to live: programming languages. Expressive type systems pose challenges to compilers that other (less expressive) types do not even reveal; one such example is type equivalence checking.

System F can be enriched with different type constructors for specifying communication protocols. We analyse the impact of combinations of such constructors on the type equivalence problem. In order to do so, we extend System F with session types [42,43,67]. Session types provide for detailed protocol specifications in the form of types. Traditional recursive session types are limited to tail

⋆ Support for this research was provided by the Fundação para a Ciência e a Tecnologia through project SafeSessions, ref. PTDC/CCI-COM/6453/2020, and by the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020. A full version is available on arXiv [20].

recursion, thus failing to capture all protocols whose traces cannot be characterized by regular languages. Context-free session types overcome this limitation by extending types with a notion of sequential composition, $T;U$ [2,68]. The set of types together with the ; binary operation constitutes a monoid, for which a new type, Skip, acts as the neutral element and End acts as an absorbing element.

The regular recursive type $\mu\,\alpha\colon \textsc{s}.\,\&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon ?\mathsf{Int};\alpha\}$ describes an integer *stream* as seen from the point of view of the consumer. It offers a choice between Done—after which the channel must be closed (as witnessed by type End)—and More—after which an integer value must be received, followed by the rest of the stream. Types are categorised by *kinds*, so that we know that the recursion variable α is of kind session—denoted by $\textsc{s}$—and, thus, can be used with semicolon. Instead, we might want to write a type with a more *context-free* flavour. The type $\mu\,\alpha\colon \textsc{s}.\,\&\{\mathsf{Leaf} : \mathsf{Skip}, \mathsf{Node} : \alpha; ?\mathsf{Int}; \alpha\};\mathsf{End}$ describes a protocol for the type-safe streaming of integer *trees* on channels. The continuation to the Leaf option is Skip, where no communication occurs but the channel is still open for further composition. The continuation to the Node choice receives a left subtree, an integer at the root and a right subtree. In either case, once the whole tree is received, the channel must be closed, as witnessed by the final End. Beyond first-order context-free session types (where only basic types are exchanged) [2,68] we may be interested in higher-order session types capable of exchanging values of complex types [19]. A goal of this paper is the integration of higher-order context-free session types into system F_ω^μ. We want to be able to abstract the type that is received on a tree channel, which is now possible by writing $\lambda\alpha\colon \textsc{t}.\mu\,\beta\colon \textsc{s}.\,\&\{\mathsf{Leaf} : \mathsf{Skip}, \mathsf{Node} : \beta; ?\alpha; \beta\};\mathsf{End}$, where $\textsc{t}$ is the kind of functional types.

A form of abstraction over session types with general recursion was proposed by Das et al. [24,25] via (nested) parametric polymorphism. In the notation of Das et al., we can write a type equation for abstracting the type being received on a stream channel $\mathsf{Stream}\langle\alpha\rangle \doteq \&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon ?\alpha; \mathsf{Stream}\langle\alpha\rangle\}$. Using abstractions, we can write Stream as a function of its parameter α, $\mathsf{Stream} \doteq \lambda\alpha\colon \textsc{t}.\&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon ?\alpha; \mathsf{Stream}\,\alpha\}$; alternatively, we can use the μ-operator to rewrite the Stream type as $\lambda\alpha\colon \textsc{t}.(\mu\,\beta\colon \textsc{s}.\,\&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon ?\alpha.\beta\})$. Das et al. proved that parametrized type definitions over regular session types are strictly more expressive than context-free session types. To some extent, this analogy guides our approach: if adding abstraction (via parametric polymorphism) to regular types leads to nested types, what exactly does it mean to add abstraction (via a type-level λ-operator) to context-free types? Throughout this paper we analyse several increments to System F^μ that culminate in adding λ-abstraction to context-free session types.

One of our focuses is necessarily the analysis of the type equivalence problem. The uncertainty about the decidability of this problem over recursive parametric types goes back to the 1970s [16,63]. Although the type equivalence problem for parametric (nested) session types and context-free session types is decidable, that for the combination of abstractions over context-free types may no longer be. In fact, this analysis constitutes an interesting journey towards a better understanding of the role of higher-order polymorphic recursion in presence of sequential composition, as well as the gains (and losses) resulting from combining abstraction with arbitrary (rather than tail) recursion.

Ultimately, decidability is not a sufficiently valuable measure regarding a type system's *practicality*. We look for type systems that may be incorporated into compilers. For that reason, we are interested in algorithms for type equivalence checking. Equivalence in F^{μ}_{ω} alone is already at least as hard as equivalence of deterministic pushdown automata. If we restrict recursion to the monomorphic case (requiring recursion variables to denote proper types, that is of kind S or T, collectively denoted by $*$) we lower the complexity of type equivalence to that of equivalence for finite-state automata. The extension with context-free session types is slightly more complex. In order to obtain "good" algorithms, we restrict the recursion to the monomorphic case, arriving at classes $F^{\mu *}_{\omega}, F^{\mu *;}_{\omega}$. Now the type equality problem for $F^{\mu *;}_{\omega}$ translates to the equivalence problem for simple grammars, which is still decidable [4,33]. Since $F^{\mu *;}_{\omega}$ subsumes $F^{\mu *}_{\omega}$, our proof of the decidability of type equivalence serves as an alternative to that of Cai et al. [14] (restricted to contractive types).

Higher-order polymorphism allows for the definition of type operators and the internalisation of various (session-type) constructs that would otherwise be offered as built-in constructors. In this way, we are able to internalise basic session-type constructors such as sequential composition ; and the Dual type operator (which reverses the direction of communication between parties). Duality is often treated as an external macro. Gay et al. [34] explore different ways of handling the dual operator, all in a monomorphic setting. In the presence of polymorphism the dual operator cannot be fully eliminated without introducing co-variables. Internalisation offers a much cleaner solution.

Due to the presence of sequential composition, regular trees are *not* a powerful enough model for representing types (`type TreeC a` in Section 2 is an example). The main technical challenge when combining System F^{μ}_{ω} and context-free session types is making sure that the resulting model can still be represented by simple grammars, so that type equivalence may be decided by a practical algorithm. The difficulties arise with renaming bound variables. For infinite types, both renaming with fresh variables and using de Bruijn indices may create an infinite number of distinct variables, which makes the construction of a simple grammar simply impossible. For example, take the type $\lambda\alpha\colon \mathrm{T}.\mu\gamma\colon \mathrm{T}.\lambda\beta\colon \mathrm{T}.\alpha \to \gamma$, which stands for the infinite type $\lambda\alpha\colon \mathrm{T}.\lambda\beta\colon \mathrm{T}.\alpha \to \lambda\beta\colon \mathrm{T}.\alpha \to \lambda\beta\colon \mathrm{T}\ldots$ Renaming this type using a fresh variable at each step would result in a type of the form $\lambda v_1\colon \mathrm{T}.\lambda v_2\colon \mathrm{T}.v_1 \to \lambda v_3\colon \mathrm{T}.v_1 \to \lambda v_4\colon \mathrm{T}\ldots$, requiring infinitely many variables. Similarly, de Bruijn indices [27] yield a type of the form $\lambda_{\mathrm{T}}\lambda_{\mathrm{T}}1 \to \lambda_{\mathrm{T}}2 \to \lambda_{\mathrm{T}}3 \to \ldots$ that requires an infinite number of natural indices. We thus introduce *minimal renaming* that uses the least amount of variable names as possible (cf. Gauthier and Pottier [30]). This ensures that only finitely many terminal symbols are necessary, allowing for translating types into simple grammars.

Type languages live in term languages and we propose a term language to consume $F^{\mu;}_{\omega}$ types. Based on Almeida et al. [2], we introduce a message-passing concurrent programming language. Type checking is decidable if type equivalence is, and it is, in particular, for $F_\omega^{\mu *;}$.

The main contributions of this paper are as follows.

- The integration of (higher-order) context-free session types into system F_ω^μ, dubbed $F_\omega^{\mu;}$.
- A semantic definition of type equivalence via a labelled transition system.
- The identification of a suitable fragment of System $F_\omega^{\mu;}$ for which type equivalence is reduced to the bisimilarity of simple grammars.
- A proof that type equivalence on the full System $F_\omega^{\mu;}$ is at least as hard as bisimilarity of deterministic pushdown automata.
- The first internalisation of the Dual type operator in a type language.
- A term language to consume $F_\omega^{\mu;}$ types and an accompanying metatheory.

The type system presented in the paper combines three constructions: sequential composition of session types, higher-order kinds via type-level abstraction and application, and higher-order recursion. Prior to our work there is the system by Almeida et al. [4] which incorporates sequential composition and (first-order) recursion, but no higher-order kinds. There is also the system by Cai et al. [14] which incorporates higher-order kinds and higher-order recursion, but no sequential composition. Our system is the first to incorporate all three constructions. Although some of the results are incremental and generalize results from the literature, the main technical challenge is understanding the border past which they do not hold anymore. For example, "just" including higher-order kinds into the system by Almeida et al. does not work, since we need to pay close attention to variable names, making sure that type equivalence is invariant with respect to alpha-conversion (renaming of bound variables). This called for a novel notion of renaming, inspired by Gauthier and Pottier [30]. Similarly, "just" including sequential composition into the system of Cai et al. does not work, since finite-state automata (or regular trees) are not enough to capture the expressive power of the new type system, *even* when restricted to first-order recursion. This required us to look at the more expressive framework of simple grammars, and introduce a translation from types to words of a simple grammar.

The rest of the paper is organised as follows. The next section motivates the type language and introduces the term language with an example. Section 3 introduces System $F_\omega^{\mu;}$, Section 4 discusses type equivalence and Section 5 shows that type equivalence is decidable for a fragment of the type language. Section 6 presents the term language and its metatheory. Section 7 discusses related work and Section 8 concludes the paper with pointers for future work. Proofs for the main results can be found in a technical report on arXiv [20].

2 Motivation

Our goal is to study type systems that combine equirecursion, higher-order polymorphism, and higher-order context-free session types, while incorporating these in programming languages.

$$(\!|\,|\!) ::= \{\} \mid \langle\rangle \qquad \sharp ::= ? \mid ! \qquad \odot ::= \& \mid \oplus \qquad * ::= \mathrm{T} \mid \mathrm{S}$$

$$\begin{array}{llll}
T & ::= T \to T \mid (\!|\overline{l_i\colon T_i}|\!) \mid \forall\alpha\colon\kappa.T \mid \mu\,\alpha\colon\kappa.T \mid \alpha & (F^{\mu}) & \kappa = \mathrm{T} \\
T & ::= (F^{\mu}) \mid \sharp T.T \mid \odot\{\overline{l_i\colon T_i}\} \mid \mathsf{End} & (F^{\mu\cdot}) & \kappa = * \\
T & ::= (F^{\mu}) \mid \sharp T \mid \odot\{\overline{l_i\colon T_i}\} \mid \mathsf{End} \mid T;T \mid \mathsf{Skip} & (F^{\mu;}) & \kappa = * \\
T & ::= (F^{M}) \mid \lambda\alpha\colon\kappa.T \mid T\,T & (F^{M}_{\omega}), M ::= \mu, \mu\cdot, \mu; & \\
 & & & \kappa = * \mid \kappa \Rightarrow \kappa
\end{array}$$

Fig. 1: Six F-systems.

Extensions of System F. Figure 1 motivates the construction by proposing six different type languages, culminating with $F^{\mu;}_{\omega}$. The initial system, F^{μ}, includes well-known basic type operators [57]: functions $T \to U$, records $\{\overline{l_i\colon T_i}\}$ and variants $\langle\overline{l_i\colon T_i}\rangle$. Type Unit is short for $\{\}$, the empty record; we can imagine that Unit stands in place of an arbitrary scalar type such as Int and Bool. We also include variable names α, type quantification $\forall\alpha\colon\kappa.T$ and recursion $\mu\,\alpha\colon\kappa.T$. To control type formation, all variable bindings must be kinded with some kind κ, even if for the initial system, F^{μ}, we only use the functional kind T.

We then build on F^{μ} by considering (regular, tail recursive) session types; we represent the resulting system by $F^{\mu\cdot}$. For example $?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End}$ is a type for a channel endpoint that receives an integer, sends a boolean, and terminates. At this point we introduce a kind S of session types to restrict the ways in which we can combine session and functional types together. For example, a well-formed type $?T.U$ is of kind S and requires U to be also of kind S (whereas T can be of kind $*$, that is S or T). An example of an infinite session type is $\mu\,\alpha\colon\mathrm{S}.\,!\mathsf{Int}.\alpha$ that endlessly outputs integer values. For a more elaborate example consider the type $\mathsf{IntStream} = \mu\,\alpha\colon\mathrm{S}.\&\{\mathsf{Done}\colon\mathsf{End}, \mathsf{More}\colon ?\mathsf{Int}.\alpha\}$ that specifies a channel endpoint for receiving a (finite or infinite) stream of integer values. Communication ends after choice Done is selected.

The next step of our construction takes us to context-free session types; the resulting system is denoted by $F^{\mu;}$. We introduce a new construct for sequential composition $T;U$, and a new type Skip, acting as the neutral element of sequential composition [68]. The message constructors are now unary ($?T$ and $!T$) rather than binary. In System $F^{\mu;}$ we distinguish between the traditional End type and the Skip type. These types have different behaviours: End terminates a channel, while Skip allows for further communication. Type equality is more subtle for context-free session types, because of the monoidal semantics of sequential composition. It is derivable from the following axioms:

$$\begin{array}{rl}
\mathsf{Skip};T \sim T & \text{Neutral element} \\
\mathsf{End};T \sim \mathsf{End} & \text{Absorbing element} \\
(T;U);V \sim T;(U;V) & \text{Associativity} \\
\odot\{\overline{l_i\colon T_i}\};U \sim \odot\{\overline{l_i\colon T_i;U}\} & \text{Distributivity}
\end{array} \tag{1}$$

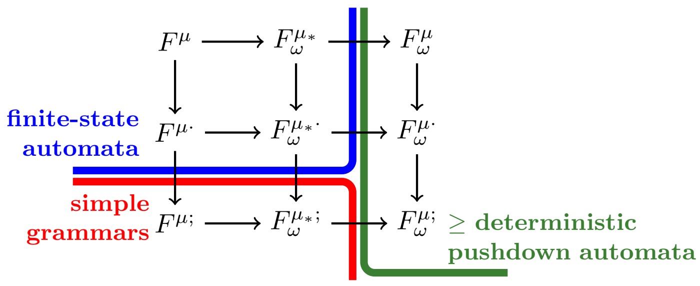

Fig. 2: Relation between the main classes of types in this paper (arrows denote strict inclusions).

Although the syntax of $F^{\mu\cdot}$ is not formally included in the syntax of $F^{\mu;}$, we can embed recursive session types into context-free session types by mapping $\sharp T.U$ into $\sharp T;U$. It is well-known that context-free session types allow for higher computational expressivity: while F^{μ} and $F^{\mu\cdot}$ can be represented via finite-state automata, $F^{\mu;}$ can only be represented with simple grammars [4,33].

To finalise our construction, we include type abstraction $\lambda\alpha\colon\kappa.T$ and type application $T\ U$. Again, type abstraction binds a variable which must be kinded. Kinds can now be of higher-order $\kappa \Rightarrow \kappa'$. For each of the three systems F^{μ}, $F^{\mu\cdot}$, $F^{\mu;}$ we arrive at a higher-order version, respectively F^{μ}_{ω}, $F^{\mu\cdot}_{\omega}$, $F^{\mu;}_{\omega}$ (all of which we represent as F^{M}_{ω}). In System $F^{\mu\cdot}_{\omega}$, for example, we can specify channels for receiving (finite or infinite) sequences of values of arbitrary (but fixed) types,

$$\mathsf{Stream} = \lambda\alpha\colon \textsc{t}.(\mu\,\beta\colon \textsc{s}.\,\&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon\, ?\alpha.\beta\})$$

where α can be instantiated with the desired type; in particular, $\mathsf{Stream\ Int}$ would be equivalent to the aforementioned $\mathsf{IntStream}$.

It turns out that the expressive power of general higher-order systems F^{M}_{ω} is too large for practical purposes. Even the simplest case F^{μ}_{ω} is at least as expressive as deterministic pushdown automata (or equivalently, first-order grammars), for which known equivalence algorithms are notoriously impractical. By impractical we mean that, although there exists a proof of decidability (due to Sénizergues [61], later improved by Stirling and Jancar [46,65]), the underlying algorithm is rather complex. To the best of our knowledge, there is no practical implementation of an algorithm to decide the equivalence of deterministic pushdown automata. This is essentially due to polymorphic recursion, which can be encoded by a higher-order μ-operator (we provide an example at the end of Section 5). Therefore, it makes sense to restrict the kind κ of the recursion operator $\mu\,\alpha\colon\kappa.\,T$. We use the notation μ_* to mean the subclass of types written using only $*$-kinded recursion, i.e., $\mu\,\alpha\colon\textsc{t}.\,T$ or $\mu\,\alpha\colon\textsc{s}.\,T$.

Figure 2 summarizes the main relations between the classes of types in our paper. Firstly, we obtain a lattice where the expressive power increases as we travel down (from functional to session to context-free session types) and right (from simple polymorphism to higher-order polymorphism with monomorphic recursion to arbitrary recursion). Four of the classes can be represented using finite-state automata (up to $F^{\mu *\cdot}_{\omega}$). By including sequential composition ($F^{\mu;}$ and $F^{\mu *;}_{\omega}$) we are still able to represent types using simple grammars. Once we allow for arbitrary recursion, the expressiveness of our model requires the computational power of deterministic pushdown automata.

Programming with $F^{\mu;}_{\omega}$. We now turn our attention to the term language, a message passing, concurrent functional language, equipped with context-free session types. Start with a stream of values of type `a`. Such a stream, when seen from the side of the reader, offers two choices: `Done` and `More`. In the former case the interaction is over; in the latter the reader reads a value of type `a`, as in `?a`, and recurses. This is the stream type we have seen before only that, rather than closing the channel endpoint (with type `End`), it terminates with type `Skip`, so that it may be sequentially composed with other types. In this informal introduction to the term language we omit the kinds of type variables.

```
type Stream a = &{Done: Skip, More: ?a ; Stream a}
```

A fold channel, as seen from the side of the folder, is a type of the following form. We assume that application binds tighter than semicolon, that is, type `Stream a ; !b ; End` is interpreted as `(Stream a) ; !b ; End`.

```
type Fold a b = ?(b → a → b) ; ?b ; Stream a ; !b ; End
```

Consumers of this type first receive the folding function, then the starting element, then the elements to fold in the form of a stream, and finally output the result of the fold. The type terminates with `End` for we do not expect type `Fold` to be further composed. Compare `Fold` with the type for a conventional functional left fold: `(b → a → b) → b → List a → b`.

We now develop a function that consumes a `Fold` channel. Syntax `x ▷ f` is for the inverse function application with low priority, that is `x ▷ f ▷ g = g (f x)`. Recall that `Unit` is an alternative notation for the empty record type, `{}`.

```
foldServer : ∀a.∀b. Fold a b → Unit
foldServer c = let (f, c) = receive c in
               let (e, c) = receive c in foldS f e c

foldS : ∀a.∀b. (b → a → b) → b → Stream a;!b;End → Unit
foldS f e c = match c with
  { Done c → c ▷ send e ▷ close
  , More c → let (x, c) = receive c in foldS f (f e x) c
  }
```

Function `foldServer` consumes the initial part of the channel and passes the rest of the channel to the recursive function `foldS` that consumes the whole stream while accumulating the fold value. In the end, when branch `Done` is selected, the fold value is written on the channel and the channel closed. In general, the channel operators—`receive`, `send`, `select`—return the same channel in the form of a new identifier. It is customary to reuse the identifier name—`c` in the example, as in `let (f, c)= receive c`—since it denotes the same channel. Syntax `c ▷ ...`

hides the continuation channel. The case for the external choice—`match`—also returns the continuation (in each branch) so that interaction on the channel endpoint may proceed.

We may now write different clients for the `foldServer`. Examples include a client that generates a stream from a pair of integer values (denoting an interval); another that generates the stream from a list of values; and yet another that generates the stream from a binary tree. We propose a further client. Consider the type of a channel that exchanges trees in a serialized format [68]. Its polymorphic version, as seen from the point of view of the reader, is as follows:

```
type TreeChannel a = TreeC a ; End
type TreeC a = &{Leaf: Skip, Node: TreeC a;?a;TreeC a}
```

We transform trees as we read from tree channels into streams. Function `flatten` receives a tree channel and a stream channel (as seen from the point of view of the writer, hence the `Dual`) and returns the unused part of the latter.

```
flatten : ∀a.∀c. TreeChannel a → (Dual Stream a);c → c
```

We are now in a position to write a client that checks whether all values in a tree channel are positive.

```
allPositive : TreeChannel Int → Dual (Fold Int Bool) → Bool
allPositive t c =
  let c = send (λx:Bool.λy:Int. x && y > 0) c in
  let c = send True c in
  let c = flatten [Int] [?Bool;End] t c in
  let (x, c) = receive c in
  close c; x
```

The client sends a function and the starting value on the fold channel. Then, it flattens the given tree `t`, receives the folded value and closes the channel. Syntax `flatten [Int] [?Bool;End]` is for term-level type application. We mean to flatten a tree of `Int` values on a stream channel whose continuation is of type `?Bool;End`. The continuation channel is bound to `c` so that we may further receive the fold value and thereupon close the channel. Syntax `e1;e2` is for sequential composition and abbreviates `let {} = e1 in e2` given that `{}`, the `Unit` value, is linear and hence must be consumed.

Finally, a simple application creates a new `TreeC` channel, passing one end to a thread that produces a tree channel. Function `new` creates a channel and returns its two ends. It then creates a `Fold` channel, distributes one end to a thread `foldServer` and the other to function `allPositive`. The `fork` primitive receives a suspended computation (a thunk, of the form `λx:Unit.e`) and creates a new thread that runs in parallel with that from where the `fork` was issued.

```
system : Bool
system = let (tr, tw) = new [TreeC Int] () in
  fork (λ_:Unit. produce tw);
  let (fr, fw) = new [Fold Int Bool] () in
  fork (λ_:Unit. foldServer fr);
  allPositive tr fw
```

$*$	$::=$		Kind of proper types
		S	session
		T	functional
κ	$::=$		Kind
		$*$	kind of proper types
		$\kappa \Rightarrow \kappa$	kind of type operators
T	$::=$		Type
		ι	type constant
		α	type variable
		$\lambda\alpha\colon \kappa.T$	type-level abstraction
		$T\,T$	type-level application

Fig. 3: The syntax of types.

ι $::=$		Type constant
$\rightarrow$	$* \Rightarrow * \Rightarrow \mathrm{T}$	arrow
$(\!\vert \overline{l_i} \vert\!)$	$\overline{* \Rightarrow} \mathrm{T}$	record, variant
μ_κ	$(\kappa \Rightarrow \kappa) \Rightarrow \kappa$	recursive type
$\forall_\kappa$	$(\kappa \Rightarrow *) \Rightarrow \mathrm{T}$	universal type
Skip	S	skip
End	S	end
$\sharp$	$* \Rightarrow \mathrm{S}$	input, output
$;$	$\mathrm{S} \Rightarrow \mathrm{S} \Rightarrow \mathrm{S}$	seq. composition
$\odot\{\overline{l_i}\}$	$\overline{\mathrm{S} \Rightarrow} \mathrm{S}$	choice operators
Dual	$\mathrm{S} \Rightarrow \mathrm{S}$	dual operator

Fig. 4: Type constants and kinds.

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

$$\mathrm{rename}_S(\iota) = \iota \quad \mathrm{rename}_S(\alpha) = \alpha \quad \mathrm{rename}_S(T\,U) = \mathrm{rename}_{S\cup \mathrm{fv}(U)}(T)\,\mathrm{rename}_S(U)$$

$$\mathrm{rename}_S(\lambda\alpha\colon \kappa.T) = \lambda\upsilon\colon \kappa.\mathrm{rename}_S(T[\upsilon/\alpha]) \quad \text{where } \upsilon = \mathrm{first}_S(\lambda\alpha\colon \kappa.T)$$

Fig. 5: Type renaming.

3 Kinds and Types

This section introduces in detail System $F^{\mu;}_{\omega}$, an extension of System F^{μ}_{ω} incorporating higher-order context-free session types. The syntax of types is presented in Fig. 3. A type is either a constant ι (as in Fig. 4), a type variable α, an abstraction $\lambda\alpha\colon \kappa.T$ or an application $T\,U$. Besides incorporating the standard session type constructors as constants, system $F^{\mu;}_{\omega}$ also includes Dual as a constant for a type operator mapping a session type to its dual. Note also that $\forall\alpha\colon \kappa.\,T$ is syntactic sugar for $\forall_\kappa(\lambda\alpha\colon \kappa.T)$. Analogously, $\mu\,\alpha\colon \kappa.\,T$ abbreviates $\mu_\kappa(\lambda\alpha\colon \kappa.T)$. This simplifies our analysis as lambda abstraction becomes the only binding operator.

A distinction between session and functional types is made resorting to kinds S and T, respectively. These are the kinds of proper types, $*$; we use the symbol κ to represent either the kind of a proper type or that of a type operator, of the form $\kappa \Rightarrow \kappa'$. A kinding context Δ stores kinds for type variables using bindings of the form $\alpha\colon \kappa$. Notation $\Delta + \alpha\colon \kappa$ denotes the update of kinding context Δ, defined as $(\Delta, \alpha\colon \kappa) + \alpha\colon \kappa' = \Delta, \alpha\colon \kappa'$ and $\Delta + \alpha\colon \kappa = \Delta, \alpha\colon \kappa$ when $\alpha \notin \Delta$.

To define type formation, we require a few notions. Firstly comes the notion of *renaming*, adapted from Gauthier and Pottier [30] and presented in Fig. 5. Renaming essentially replaces a type T by a minimal alpha-conversion of T. By alpha-conversion we mean that $\mathrm{rename}_S(T)$ renames bound variables in T. By "minimal" we mean that each bound variable is renamed to its lowest possible value. We assume at our disposal a countable well-ordered set of type variables $\{\upsilon_1, \ldots, \upsilon_n, \ldots\}$. In $\mathrm{rename}_S(T)$, parameter S is a set containing type variables unavailable for renaming; in the outset of the renaming process S is the empty set, since all variables are available. In that case the subscript S is often omitted. The case for lambda abstraction renames the bound variable by the smallest variable not in the set $S \cup \mathrm{fv}(\lambda\alpha\colon \kappa.T)$, which we denote by $\mathrm{first}_S(\lambda\alpha\colon \kappa.T)$.

Renaming is what allows us to check whether type abstractions $\lambda\alpha\colon \kappa.T$, $\lambda\beta\colon \kappa.U$ are equivalent. For the types to be equivalent, both bound variables α and β ought to be renamed to the same variable υ_j. In summary, renaming provides a syntax-guided approach to the equivalence of lambda-abstractions, where the names of bound variables should not matter. Our notion of type equivalence preserves alpha-conversions up to renaming: if T and U only differ on bound variables, then $\mathrm{rename}(T) = \mathrm{rename}(U)$ and in particular $\mathrm{rename}(T) \sim \mathrm{rename}(U)$. We will come back to this point after we define type equivalence in Section 4.

We can easily see that renaming uses the minimum amount of variable names possible; for example, $\mathrm{rename}(\lambda\alpha\colon \mathrm{T}.\lambda\beta\colon \mathrm{S}.\beta) = \lambda\upsilon_1\colon \mathrm{T}.\lambda\upsilon_1\colon \mathrm{S}.\upsilon_1$. Notice how both bound variables α and β are renamed to υ_1, the first variable available for replacement. Also, renaming blatantly violates the Barendregt's variable convention [9] used in so many works; for example $\mathrm{rename}(\upsilon_1\,(\lambda\alpha\colon T.\alpha)) = \upsilon_1\,(\lambda\upsilon_1\colon T.\upsilon_1)$, where variable υ_1 is both free and bound in the resulting type. Even if renaming violates the variable convention, substitution can still be performed without resorting to the "on-the-fly" renaming of Curry and Feys [21,40]. When $\upsilon_1 \neq \upsilon_2$, we have that

$$(\lambda\upsilon_1\colon \kappa.\lambda\upsilon_2\colon \kappa'.U)\,T \quad \text{reduces to} \quad \mathrm{rename}((\lambda\upsilon_2\colon \kappa'.U)[T/\upsilon_1]).$$

Then, we have $(\lambda\upsilon_2\colon \kappa'.U)[T/\upsilon_1] = \lambda\upsilon_2\colon \kappa'.(U[T/\upsilon_1])$ since the renaming rule for application guarantees that $\upsilon_2 \notin \mathrm{fv}(T)$. Otherwise if $\upsilon_1 = \upsilon_2$, we have $(\lambda\upsilon_1\colon \kappa'.U)[T/\upsilon_1] = \lambda\upsilon_1\colon \kappa'.U$. This justifies the inclusion of set S in the renaming process. From now on, we assume that all types have gone through the renaming process.

Next comes the notion of *type reduction* (Fig. 6). Apart from beta reduction (rule R-β), the definition provides for sequential composition, for unfolding recursive types and for reducing $\mathsf{Dual}\,T$ types. Note that renaming is further invoked in rule R-β for beta reduction does not preserve renaming: consider the renamed type $(\lambda\upsilon_1\colon \mathrm{T}.\lambda\upsilon_2\colon \mathrm{T}.\upsilon_1 \to \upsilon_2)\,\mathsf{Unit}$. The type resulting from the substitution $(\lambda\upsilon_2\colon \mathrm{T}.\upsilon_1 \to \upsilon_2)[\mathsf{Unit}/\upsilon_1]$ is $\lambda\upsilon_2\colon \mathrm{T}.\mathsf{Unit} \to \upsilon_2$ which is not renamed and, therefore, not equivalent to $\lambda\upsilon_1\colon \mathrm{T}.\mathsf{Unit} \to \upsilon_1$ according to our rules in Section 4. Thanks to our modified rule R-β, we preserve renaming under reductions: if $T = \mathrm{rename}(T)$ and $T \longrightarrow U$ then $U = \mathrm{rename}(U)$.

We also need the notion of *weak head normal form* borrowed from the lambda calculus [9,10]. We say that a type T is in weak head normal form, $T\,\mathsf{whnf}$, if it is irreducible, i.e., $T \not\longrightarrow$. Although this is a negative definition, in the technical report we provide an equivalent, rule-based characterisation of weak head normal

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

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

R-Seq2 $\dfrac{T \longrightarrow V}{T; U \longrightarrow V; U}$

R-Assoc $(T; U); V \longrightarrow T; (U; V)$

R-μ $\mu_k\, T \longrightarrow T\,(\mu_k\, T)$

R-β $(\lambda\alpha\colon \kappa.T)\, U \longrightarrow \mathrm{rename}(T[U/\alpha])$

R-TAppL $\dfrac{T \longrightarrow U}{TV \longrightarrow UV}$

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

R-DSkip $\mathsf{Dual}\,\mathsf{Skip} \longrightarrow \mathsf{Skip}$

R-DEnd $\mathsf{Dual}\,\mathsf{End} \longrightarrow \mathsf{End}$

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

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

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

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

R-DCtx $\dfrac{T \longrightarrow U}{\mathsf{Dual}\, T \longrightarrow \mathsf{Dual}\, U}$

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

Fig. 6: Type reduction.

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

K-Const $\Delta \vdash \iota : \kappa_\iota$

K-Var $\dfrac{\alpha\colon \kappa \in \Delta}{\Delta \vdash \alpha : \kappa}$

K-TAbs $\dfrac{\Delta + \alpha\colon \kappa \vdash T : \kappa'}{\Delta \vdash \lambda\alpha\colon \kappa.T : \kappa \Rightarrow \kappa'}$

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

Fig. 7: Type formation.

form types, which can be used in a compiler as well as in our proofs. We 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 (note that any term which is already whnf normalises to itself). We write T norm to denote that $T \Downarrow U$ for some U.

For example, suppose we want to normalise the type $\mu_{\mathrm{S}}\, T$, where T is the type $\lambda v_1\colon \mathrm{S}.\oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\alpha\}; \mathsf{Dual}\, v_1$. By computing all reductions from $\mu_{\mathrm{S}} T$, we obtain $\mu_{\mathrm{S}} T \longrightarrow T\,(\mu_{\mathrm{S}} T) \longrightarrow \oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\alpha\}; \mathsf{Dual}\,(\mu_{\mathrm{S}} T) \not\longrightarrow$ for which we conclude that $\mu_{\mathrm{S}}\, T \Downarrow \oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\alpha\}; \mathsf{Dual}\,(\mu_{\mathrm{S}} T)$. Similarly, we can reason that $\mu_{\mathrm{T}}\,(\lambda v_1\colon \mathrm{T}.v_1)$, $\mu_{\mathrm{S}}\,(\lambda v_1\colon \mathrm{S}.\mathsf{Skip}; v_1)$ and $\mu_{\mathrm{S}}\,(\lambda v_1\colon \mathrm{S}.\mathsf{Dual}\, v_1)$ are all examples of non-normalising expressions.

Equipped with normalisation, we can introduce *type formation*, which we do via the rules in Fig. 7. Rule K-Const introduces constants as types whose kinds match those of Fig. 4. Rule K-Var reads the kind of a type variable from context Δ. An abstraction $\lambda\alpha\colon \kappa.T$ is a well-formed type with kind $\kappa \Rightarrow \kappa'$ if T is well formed in context Δ updated with entry $\alpha\colon \kappa$ (rule K-TAbs). The update is necessary since we are dealing with renamed types and the same type variable may appear with different kinds in nested abstractions.

It is not until we reach rule K-TApp that we find a proviso about the normalisation of a type. This is standard and analogous to a condition on contractivity. The goal is to eliminate types that reduce indefinitely without reaching a whnf.

Theorem 1. *Let $\Delta \vdash T : \kappa$.*

Preservation. *If $T \longrightarrow U$, then $\Delta \vdash U : \kappa$.*
Confluence. *If $T \longrightarrow U$ and $T \longrightarrow V$, then $U \longrightarrow^* W$ and $V \longrightarrow^* W$.*
Weak normalisation. *$T \Downarrow U$ for some U. Furthermore, if $T \Downarrow V$, then $U = V$.*

We finally arrive at the main decidability result in this section. In its proof, we make use of the fact that recursion is restricted to kind $*$ to limit the possible subexpressions of the form $\mu_* U$ that might appear in the normalisation of T.

Theorem 2 (Decidability of type formation). *$\Delta \vdash T : \kappa$ is decidable for types in $F^{\mu_*;}_{\omega}$.*

4 Type equivalence

This section introduces type bisimulation as our notion of type equivalence. We define a labelled transition system (LTS) on the space of all types and write $T \xrightarrow{a} U$ to denote that T has a transition by label a to U. The grammar for labels and the LTS rules are in Fig. 8.

If T is not in weak head normal form, then we must normalise it to some type U, so that T has the same transitions as U (rule L-Red). Otherwise if $T\ \mathsf{whnf}$, then the transitions of T can be immediately derived by looking at the corresponding rule for T as follows. If T is a variable, use rule L-Var1 (with $m = 0$). If T is a constant (other than Skip), use rule L-Const. Note that if T is a lone Skip, then it has no transitions. If T is an abstraction, use rule L-Abs.

If T is an application, then we need to look inside the head. We write T as $T_0\ T_1 \ldots T_m$ with $m \geq 1$ where T_0 is not an application, and look at T_0. If T_0 is a variable, use rules L-Var1 and L-Var2. If T_0 is one of the constants $\rightarrow$, $\forall_\kappa$, $\odot\{\overline{l_i}\}$ or $(\!|\overline{l_i}|\!)$, use rule L-ConstApp. Note that T_0 is neither an abstraction nor μ_κ, since T is in weak head normal form. If T_0 is $\sharp$, we use rules L-Msg1 and L-Msg2. If T_0 is Dual, then the only way for T to be well-formed and in weak head normal form is if $m = 1$ and T_1 is α or $\alpha\ U_1 \ldots U_m$, in which case we use rules L-DualVar1 and L-DualVar2.

If T_0 is $;$, we require an additional case analysis on T_1. If $m = 1$, use rule L-Seq1. Otherwise $m = 2$ due to kinding. If T_1 is a variable, use rule L-VarSeq1 (with $m = 0$). If T_1 is a constant, then it must be of kind s. T_1 cannot be Skip, because T is in weak normal form, so it must be End, in which case we use rule L-EndSeq (End is an absorbing element, so $\mathsf{End}; U$ simply makes a transition to Skip without executing U). If T_1 is End. Note that T_1 cannot be an abstraction due to kinding.

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

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

L-Red
$$\frac{T \longrightarrow U \quad U \xrightarrow{a} V}{T \xrightarrow{a} V}$$

L-Var1
$$\frac{m \geq 0}{\alpha\ T_1 \dots T_m \xrightarrow{\alpha_0} \mathsf{Skip}}$$

L-Var2
$$\frac{1 \leq j \leq m}{\alpha\ T_1 \dots T_m \xrightarrow{\alpha_j} T_j}$$

L-Const
$$\frac{\iota \neq \mathsf{Skip}}{\iota \xrightarrow{\iota_0} \mathsf{Skip}}$$

L-ConstApp
$$\frac{\iota = \rightarrow, \forall_\kappa, \odot\{\overline{\ell_i}\}, \llparenthesis\overline{\ell_i}\rrparenthesis \quad 1 \leq j \leq m}{\iota\ T_1 \dots T_m \xrightarrow{\iota_j} T_j}$$

L-Abs
$$\lambda\alpha\colon \kappa.T \xrightarrow{\lambda\alpha\colon \kappa} T$$

L-Msg1
$$\sharp T \xrightarrow{\sharp_1} T$$

L-Msg2
$$\sharp T \xrightarrow{\sharp_2} \mathsf{Skip}$$

L-Seq1
$$;\ T \xrightarrow{;_1} T$$

L-VarSeq1
$$\frac{m \geq 0}{(\alpha\ T_1 \dots T_m); U \xrightarrow{\alpha_0} U}$$

L-VarSeq2
$$\frac{1 \leq j \leq m}{(\alpha\ T_1 \dots T_m); U \xrightarrow{\alpha_j} T_j}$$

L-MsgSeq1
$$\sharp T; U \xrightarrow{\sharp_1} T$$

L-MsgSeq2
$$\sharp T; U \xrightarrow{\sharp_2} U$$

L-ChoiceSeq
$$\frac{1 \leq j \leq m}{\odot\{\overline{\ell_i\colon T_i}\}; U \xrightarrow{\odot\{\overline{\ell_i}\}_j} T_j; U}$$

L-EndSeq
$$\mathsf{End}; U \xrightarrow{\mathsf{End}} \mathsf{Skip}$$

L-DualVar1
$$\mathsf{Dual}\,(\alpha\ T_1 \dots T_m) \xrightarrow{\mathsf{Dual}_1} \alpha\ T_1 \dots T_m$$

L-DualVar2
$$\mathsf{Dual}\,(\alpha\ T_1 \dots T_m) \xrightarrow{\mathsf{Dual}_2} \mathsf{Skip}$$

L-DualSeq1
$$(\mathsf{Dual}\,(\alpha\ T_1 \dots T_m)); U \xrightarrow{\mathsf{Dual}_1} \alpha\ T_1 \dots T_m$$

L-DualSeq2
$$(\mathsf{Dual}\,(\alpha\ T_1 \dots T_m)); U \xrightarrow{\mathsf{Dual}_2} U$$

Fig. 8: Labelled transition system for types.

If T_1 is an application, then again we write T_1 as $U_0\ U_1 \dots U_n$ with $n \geq 1$ where the head U_0 is not an application, and look at U_0. If U_0 is a variable, use rules L-VarSeq1 and L-VarSeq2. If U_0 is a constant, it must be one of $;$, μ_κ, $\sharp$, $\odot\{\overline{l_i}\}$ or Dual due to kinding. If U_0 is $\sharp$, use rules L-MsgSeq1 and L-MsgSeq2. If U_0 is $\odot\{\overline{l_i}\}$, use rule L-ChoiceSeq. If U_0 is Dual, the only way for T to be well-formed and in weak head normal form is if $n = 1$ and U_1 is α or $\alpha\ V_1 \dots V_\ell$, in which case we use rules L-DualSeq1 and L-DualSeq2. Note that U_0 cannot be $;$, μ_κ or an abstraction, since T is in weak normal form.

Let us clarify our LTS rules with an example. Consider the following type $\lambda\upsilon_1\colon \mathrm{T}.\mu\,\upsilon_2\colon \mathrm{S}.\oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\upsilon_1\}; \mathsf{Dual}\,\upsilon_2$ and call it T. T is a type abstraction (on type variable υ_1), of kind $\mathrm{T} \Rightarrow \mathrm{S}$. It specifies a channel alternating between: offer a choice and output a value of type υ_1; or select a choice and input a value of type υ_1. The polarity is swapped thanks to the application of constant Dual to the recursion variable υ_2. To construct the (fragment of the) LTS generated by this type, let us first desugar T into $\lambda\upsilon_1\colon \mathrm{T}.U$ where U is the

$$\begin{tikzcd}
& \mathsf{Skip} & \lambda v_1 \colon \mathrm{T}.U \arrow[r, "\lambda v_1 : \mathrm{T}"] & U \arrow[dr, Rightarrow] & & \mathsf{End}; \mathsf{Dual}\, U \arrow[r, "\mathsf{End}"] & \mathsf{Skip} \\
& v_1 \arrow[u, "v_1"] & \mathsf{Dual}\,(\mathsf{Dual}\, U) \arrow[rr, Rightarrow] & & \oplus\{\mathsf{Done} \colon \mathsf{End}, \mathsf{More} \colon !v_1\}; \mathsf{Dual}\, U \arrow[ur, "\oplus\{\mathsf{Done}, \mathsf{More}\}_1"'] \arrow[dr, "\oplus\{\mathsf{Done}, \mathsf{More}\}_2"] & & \\
& ?v_1; \mathsf{Dual}\,(\mathsf{Dual}\, U) \arrow[u, "?_1"] \arrow[ur, "?_2"] & & & & !v_1; \mathsf{Dual}\, U \arrow[dl, "!_2"] \arrow[dr, "!_1"] & \\
& \&\{\mathsf{Done} \colon \mathsf{End}, \mathsf{More} \colon ?v_1\}; \mathsf{Dual}\,(\mathsf{Dual}\, U) \arrow[u, "\&\{\mathsf{Done}, \mathsf{More}\}_2"] \arrow[d, "\&\{\mathsf{Done}, \mathsf{More}\}_1"'] & & & \mathsf{Dual}\, U \arrow[lll, Rightarrow] & & v_1 \arrow[d, "v_1"] \\
\mathsf{Skip} & \mathsf{End}; \mathsf{Dual}\,(\mathsf{Dual}\, U) \arrow[l, "\mathsf{End}"'] & & & & & \mathsf{Skip}
\end{tikzcd}$$

Fig. 9: The LTS for type $\lambda v_1 \colon \mathrm{T}.U$. Normalisation $T_1 \Downarrow T_2$ is represented as $T_1 \Rightarrow T_2$ and U is a shorthand for type $\mu_{\mathrm{S}}\,(\lambda v_2 \colon \mathrm{S}.\oplus\{\mathsf{Done} \colon \mathsf{End}, \mathsf{More} \colon !v_1\}; \mathsf{Dual}\, v_2)$.

type $\mu_{\mathrm{S}}\,(\lambda v_2 \colon \mathrm{S}.\oplus\{\mathsf{Done} \colon \mathsf{End}, \mathsf{More} \colon !v_1\}; \mathsf{Dual}\, v_2)$. Notice that U normalises to $\oplus\{\mathsf{Done} \colon \mathsf{End}, \mathsf{More} \colon !v_1\}; \mathsf{Dual}\, U$. The LTS for the example is sketched in Fig. 9. In this case, only finitely many types appear. However, more elaborate examples involving sequential composition or higher-order recursion may lead to an infinite graph of transitions.

Given the LTS rules, we can define, in the standard way, a notion of bisimulation. A binary relation R on types is called a *bisimulation* if, for every $(T, U) \in R$ and every transition label a:

1. if $T \xrightarrow{a} T'$, then there exists U' s.t. $U \xrightarrow{a} U'$ and $(T', U') \in R$;
2. if $U \xrightarrow{a} U'$, then there exists T' s.t. $T \xrightarrow{a} T'$ and $(T', U') \in R$.

We say that types T and U are bisimilar, written $T \sim U$, if there exists a bisimulation R such that $(T, U) \in R$.

Intuitively, a notion of type equivalence must preserve and reflect the syntax of type constructors: for example, a type $T \to U$ is equivalent to a type $T' \to U'$ iff T, T' are equivalent and U, U' are equivalent. Using the bisimulation technique, we achieve this by considering a labelled transition system on types: $T \to U$ has a transition labelled $\to_1$ to T and a transition labelled $\to_2$ to U. In this way, $T \to U$ can only be equivalent to another type which has two transitions with those same labels. For each of the type constructors ($\to$, $\forall_\kappa$, $!$, $?$, $\odot\{\overline{\ell_i}\}$, and so on) we have suitable transition rules. Moreover, a type sometimes needs to be reduced before a type constructor is found at the root of the syntax tree. If T normalizes to U, then we expect T and U to be bisimilar, which is achieved thanks to rule L-Red. This handles the various reductions: beta-reductions arising from lambda-abstraction and applications (e.g., $(\lambda\alpha \colon \kappa.T)\, U$ reduces to $\mathrm{rename}(T[U/\alpha])$), reductions arising from the monoidal structure of sequential composition (e.g., $\mathsf{Skip}; T$ reduces to T), reductions arising from the internalisation of duality as a type constructor (e.g., $\mathsf{Dual}\,(!T)$ reduces to $?T$) and reductions arising from the recursion (e.g., $\mu_\kappa\, T$ reduces to $T\,(\mu_\kappa\, T)$).

Our notion of type equivalence enjoys natural properties and behaves as expected with respect to the notions of reduction, normalisation and kinding from Section 3. We can derive rules for type equivalence, that could be used to define another coinductive notion of equivalence, via effective syntax-directed rules. We can show that type equivalence is preserved under renaming, reduction and normalisation. We can also show that the axioms for sequential composition in the introduction (1) are derivable from our notion of bisimulation. These additional results are presented in the technical report [20].

5 Decidability of type equivalence

This section presents results on decidability of type equivalence. Our approach consists in translating types to objects in some computational model. We look at finite-state automata (for types in F^{μ}, $F^{\mu *}_{\omega}$, $F^{\mu \cdot}$, and $F^{\mu * \cdot}_{\omega}$), simple grammars (for types in $F^{\mu ;}$ and $F^{\mu * ;}_{\omega}$) and deterministic pushdown automata (for types in F^{μ}_{ω}, $F^{\mu \cdot}_{\omega}$ and $F^{\mu ;}_{\omega}$).

We say that a *grammar in Greibach normal form* is a tuple $(\mathcal{T}, \mathcal{N}, \gamma, \mathcal{R})$ 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 $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{T} \times \mathcal{N}^*$ is a set of productions. A grammar is said to be *simple* if, for every nonterminal X and every terminal a, there is at most one production $(X, a, \delta) \in \mathcal{R}$ [51].

Greek letters γ and δ denote (possibly empty) words of nonterminal symbols. Productions are written as $X \xrightarrow{a} \delta$. We define a notion of bisimulation for grammars via a labelled transition system. The system comprises a set of states $\mathcal{N}^*$ corresponding to words of nonterminal symbols. For each production $X \xrightarrow{a} \gamma$ 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 (the definition is similar to that in Section 4).

For the moment we focus on the class $F^{\mu * ;}_{\omega}$ and we explain how to convert a type T into a simple grammar $(\mathcal{T}_T, \mathcal{N}_T, \mathrm{word}(T), \mathcal{R}_T)$. The conversion is based on a function $\mathrm{word}(T)$ that maps each type T into a word of nonterminal symbols, while introducing fresh nonterminals and productions. In our construction, following the approach by Costa et al. [19], we use a nonterminal symbol with no productions, denoted by $\perp$, in order to separate the two descendants of a send/receive operation such as $!T; U$. The sequence of nonterminal symbols $\mathrm{word}(T)$ is defined as follows. First consider the cases in which T whnf.

– For any $m \geq 0$: $\mathrm{word}(\alpha\ T_1 \ldots T_m) = Y$ for Y a fresh nonterminal symbol with a production $Y \xrightarrow{\alpha_0} \varepsilon$ as well as $Y \xrightarrow{\alpha_j} \mathrm{word}(T_j)\perp$ for each $1 \leq j \leq m$.
– $\mathrm{word}(\mathsf{Skip}) = \varepsilon$.
– $\mathrm{word}(\mathsf{End}) = Y$ for Y a fresh symbol with a single production $Y \xrightarrow{\mathsf{End}} \perp$.
– for any $\iota \neq \mathsf{Skip}, \mathsf{End}$: $\mathrm{word}(\iota) = Y$ for Y a fresh nonterminal symbol with a single production $Y \xrightarrow{\iota} \varepsilon$.
– $\mathrm{word}(\lambda\alpha\colon \kappa.T) = Y$ for Y a fresh symbol with a production $Y \xrightarrow{\lambda\alpha\colon \kappa} \mathrm{word}(T)$.

- for any $m \geq 1$ and for ι one of $\rightarrow$, $\forall_\kappa$, $\odot\{\overline{l_i}\}$, $(\!|\overline{l_i}|\!)$: $\mathrm{word}(\iota\ T_1 \cdots T_m) = Y$ for a fresh nonterminal Y with a production $Y \xrightarrow{\iota_j} \mathrm{word}(T_j)$ for each $1 \leq j \leq m$.
- $\mathrm{word}(\sharp T) = Y$ for Y fresh with productions $Y \xrightarrow{\sharp_1} \mathrm{word}(T)\bot$ and $Y \xrightarrow{\sharp_2} \varepsilon$.
- $\mathrm{word}(;\ T) = Y$ for Y a fresh symbol with a production $Y \xrightarrow{;_1} \mathrm{word}(T)$.
- $\mathrm{word}(T;U) = \mathrm{word}(T)\,\mathrm{word}(U)$.
- $\mathrm{word}(\mathsf{Dual}\,(\alpha\ T_1 ... T_m)) = Y$ for Y a fresh symbol with productions $Y \xrightarrow{\mathsf{Dual}_1} \mathrm{word}(\alpha\ T_1 ... T_m)$ and $Y \xrightarrow{\mathsf{Dual}_2} \varepsilon$.

Finally, let us handle the cases where T is not in weak head normal form.

- If $T \Downarrow \mathsf{Skip}$, then $\mathrm{word}(T) = \varepsilon$.
- Otherwise if $T \Downarrow U \neq \mathsf{Skip}$, then $\mathrm{word}(T) = Y$ for Y a fresh nonterminal symbol. Let $Z\delta = \mathrm{word}(U)$. Then Y has a production $Y \xrightarrow{a} \gamma\delta$ for each production $Z \xrightarrow{a} \gamma$.

In the above construction, we create fresh symbols each time we encounter a weak head normal form other than Skip. In other words, $\mathcal{N}_T$ is the set containing $\bot$ and all nonterminals Y created during the computation of $\mathrm{word}(T)$. Another key insight is that the sequential composition of types is translated into a concatenation of words: $\mathrm{word}(T_1;T_2;\ldots;T_n) = \mathrm{word}(T_1)\,\mathrm{word}(T_2)\ldots\mathrm{word}(T_n)$. This allows our construction to terminate: even if the transitions lead to infinitely many types, they are split on the sequential composition operator, and so we only need to consider finitely many subexpressions.

For the last case in our construction to be well-defined, i.e., when $T \Downarrow U \neq \mathsf{Skip}$, we require $\mathrm{word}(U)$ to be non-empty. Indeed, if $U\,\mathsf{whnf}$, then we can observe (by inspecting all cases) that $\mathrm{word}(U) = \varepsilon$ iff $U = \mathsf{Skip}$. We also need to argue that the construction of $\mathrm{word}(T)$ eventually terminates. For this, we keep track of all types visited during the construction, and we only add a fresh nonterminal Y to our grammar if the type visited is syntactically different from all types visited so far. Therefore, we reuse the same symbol Y with the same productions each time we revisit a type. With all these observations, we get the following result.

Lemma 1. *Suppose that $T \in F^{\mu *;}_{\omega}$. Then the construction of* $\mathrm{word}(T)$ *terminates producing a simple grammar.*

We illustrate the above construction with the polymorphic tree exchanging example from Section 2,

```
type TreeC a = &{Leaf: Skip, Node: TreeC a; ?a ; TreeC a}
```

that is written in $F^{\mu *;}_{\omega}$ as $T_0 = \lambda v_1 : \mathtt{T}.\mu\, v_2 : \mathtt{S}.\, \&\{\mathsf{Leaf}: \mathsf{Skip}, \mathsf{Node}: v_2; ?v_1; v_2\}$. For ease of notation, in this example we write $\&_i$ as shorthand for $\&\{\mathsf{Leaf}, \mathsf{Node}\}_i$. Since T_0 is in weak head normal form, $\mathrm{word}(T_0)$ returns a fresh symbol, which we call X_0. We also have a production $X_0 \xrightarrow{\lambda v_1 : \mathtt{T}} \mathrm{word}(T_1)$, where T_1 is the type $\mu\, v_2 : \mathtt{S}.\, \&\{\mathsf{Leaf}: \mathsf{Skip}, \mathsf{Node}: v_2; ?v_1; v_2\}$. Since T_1 is not in whnf, we must normalise it, to get $T_2 = \&\{\mathsf{Leaf}: \mathsf{Skip}, \mathsf{Node}: T_1; ?v_1; T_1\}$. Therefore $\mathrm{word}(T_1)$ returns a fresh symbol, which we call X_1. To obtain the transitions of X_1, we must first compute $\mathsf{word}(T_2)$, which is a fresh symbol X_2 with transitions $X_2 \xrightarrow{\&_1} \mathsf{word}(\mathsf{Skip})$ and $X_2 \xrightarrow{\&_2} \mathsf{word}(T_1;?\upsilon_1;T_1)$. Thus we also get $X_1 \xrightarrow{\&_1} \mathsf{word}(\mathsf{Skip})$ and $X_1 \xrightarrow{\&_2} \mathsf{word}(T_1;?\upsilon_1;T_1)$.

We have $\mathsf{word}(\mathsf{Skip}) = \varepsilon$, but we still need to compute $\mathsf{word}(T_1;?\upsilon_1;T_1)$. This type normalises to $T_3 = T_2;?\upsilon_1;T_1$ since $T_1 \Downarrow T_2$. Thus $\mathsf{word}(T_1;?\upsilon_1;T_1)$ is a fresh symbol X_3. To obtain the productions of X_3 we must compute $\mathsf{word}(T_2;?\upsilon_1;T_1) = \mathsf{word}(T_2)\,\mathsf{word}(?\upsilon_1)\,\mathsf{word}(T_1)$. At this point we already have $\mathsf{word}(T_1) = X_1$ and $\mathsf{word}(T_2) = X_2$. We still need to compute $\mathsf{word}(?\upsilon_1)$, which is a fresh symbol X_4 with productions $X_4 \xrightarrow{?_1} \mathsf{word}(\upsilon_1)\bot$ and $X_4 \xrightarrow{?_2} \varepsilon$. In turn, $\mathsf{word}(\upsilon_1)$ is a fresh symbol X_5 with a production $X_5 \xrightarrow{\upsilon_1} \varepsilon$. Finally, we get $\mathsf{word}(T_2;?\upsilon_1;T_1) = X_2X_4X_1$, which means we can write the productions for X_3: $X_3 \xrightarrow{\&_1} X_4X_1$ and $X_3 \xrightarrow{\&_2} X_3X_4X_1$.

Putting all this together, we can finally obtain the simple grammar:

$$\begin{array}{ccccc} X_0 \xrightarrow{\lambda\upsilon_1:\mathsf{T}} X_1 & X_1 \xrightarrow{\&_1} \varepsilon & X_1 \xrightarrow{\&_2} X_3 & X_2 \xrightarrow{\&_1} \varepsilon & X_2 \xrightarrow{\&_2} X_3 \\ X_3 \xrightarrow{\&_1} X_4X_1 & X_3 \xrightarrow{\&_2} X_3X_4X_1 & X_4 \xrightarrow{?_1} X_5\bot & X_4 \xrightarrow{?_2} \varepsilon & X_5 \xrightarrow{\upsilon_1} \varepsilon \end{array}$$

Next, we argue that type equivalence (i.e., bisimilarity on types) corresponds to bisimilarity on the corresponding grammars. This is achieved by the following lemma, that asserts that the LTS of a type and the LTS of the corresponding word of nonterminals have exactly the same transitions.

Lemma 2 (Full abstraction). *Let $T \in F_\omega^{\mu *;}$ and $(\mathcal{T}_T, \mathcal{N}_T, \mathsf{word}(T), \mathcal{R}_T)$ the corresponding simple grammar. Suppose also that $\mathsf{word}(T) \approx \gamma$.*

1. *If $T \xrightarrow{a} U$ then there exists γ' such that $\gamma \xrightarrow{a} \gamma'$ and $\mathsf{word}(U) \approx \gamma'$.*
2. *If $\gamma \xrightarrow{a} \gamma'$ then there exists U such that $T \xrightarrow{a} U$ and $\mathsf{word}(U) \approx \gamma'$.*

As a consequence of the above result, we get soundness and completeness of the bisimilarity $\mathsf{word}(T) \approx \mathsf{word}(U)$ with respect to the bisimilarity $T \sim U$. Indeed by Lemma 2, any sequence of transitions starting from T can be matched by a sequence of transitions starting from $\mathsf{word}(T)$; and similarly for U. Thus $T \sim U$ iff $\mathsf{word}(T) \approx \mathsf{word}(U)$.

Theorem 3. *The type equivalence problem is decidable for types in $F_\omega^{\mu *;}$.*

For the remainder of this section, we look at the other classes of types in Fig. 2 and examine the computation models they correspond to. Since class $F^{\mu ;}$ is contained in $F_\omega^{\mu *;}$, we can express types without λ-abstractions with simple grammars as well. In this way we recover previous results in the literature [4,19].

Let us now look at the class $F_\omega^{\mu *\cdot}$. In this class we do not have Skip nor sequential composition and message operators are binary ($\sharp T.U$) rather than unary. Since we do not have sequential composition, there is no need to consider words of nonterminals, and instead it suffices to translate types into single symbols, i.e., states in an automaton. Moreover, since there is no recursion beyond μ_κ, only finitely many types can be reached from a given T. We can thus adapt our construction as follows for $F_\omega^{\mu *\cdot}$. In the definition of the LTS (Fig. 8):

- discard all rules involving sequential composition;
- discard rules L-Var1 for $m > 0$ and L-DualVar2 (they were only needed to distinguish types in sequential composition);
- discard case $\iota = \mathsf{End}$ in rule L-Const (so that End no longer has transitions);
- replace Skip with End on the right-hand side of rules L-Var1 with $m = 0$ and L-Const;
- discard rules L-Msg1 and L-Msg2 and treat $\iota = \sharp$ like the other constants in rule L-ConstApp.

Also replace the construction of word(T) into a construction of state(T), associating to each type T a state in a finite-state automata. For each transition $T \xrightarrow{a} U$ we have the corresponding transition state(T) $\xrightarrow{a}$ state(U). Notice that the resulting automata is deterministic since the original LTS is also deterministic (for each type T and label a, there is at most one transition $T \xrightarrow{a} U$). Since bisimilarity of deterministic finite-state automata can be decided in polynomial time [44], we get the following results.

Theorem 4.

1. *To each type T in $F^{\mu_*\cdot}_{\omega}$ we can associate a finite-state automata corresponding to the (fragment of the) LTS generated by T.*
2. *The type equivalence problem is polynomial-time decidable for types in $F^{\mu_*\cdot}_{\omega}$.*

Clearly, Theorem 4 applies to the subclasses of $F^{\mu_*\cdot}_{\omega}$: F^{μ}, $F^{\mu\cdot}$ and $F^{\mu_*}_{\omega}$. In this way we recover previous results in the literature [14,19,33].

Finally, we consider the classes F^{μ}_{ω}, $F^{\mu\cdot}_{\omega}$ and $F^{\mu;}_{\omega}$ involving arbitrarily-kinded recursion. We shall show that these classes are already powerful enough to simulate deterministic pushdown automata; hence, the type equivalence problem becomes impractical (i.e., no practical implementation of an algorithm is known). We only focus on the simplest case F^{μ}_{ω}, as the others two classes are even more expressive. Instead of looking at deterministic pushdown automata, we look at deterministic first-order grammars, which constitute an equivalent model of computation [46]. This choice simplifies our construction. We say that a *first-order grammar* is a tuple $(\mathcal{X}, \mathcal{T}, \mathcal{N}, E, \mathcal{R})$ where:

- $\mathcal{X}$ is a set of variables $\alpha, \beta, \ldots$; $\mathcal{T}$ is a set of terminal symbols $a, b, \ldots$; $\mathcal{N}$ is a set of nonterminal symbols $X, Y, \ldots$
- each nonterminal X has an arity $m = \text{arity}(X) \in \mathbb{N}$.
- the set $\mathcal{E}$ of expressions over $\mathcal{X}$, $\mathcal{N}$ is inductively defined by two rules: any variable α is an expression; if arity$(X) = m$ and $E_1, \ldots, E_m$ are expressions, then so is $X\ E_1 \ldots E_m$. Whenever $m = 0$, X is called a constant.
- E is an expression over $\mathcal{N}$, called the initial expression.
- $\mathcal{R}$ is a set of productions. Each production is a triple (X, a, E), written as $X\ \alpha_1 \ldots \alpha_m \xrightarrow{a} E$, where $m = \text{arity}(X)$ and the variables in E must be taken from $\alpha_1, \ldots, \alpha_m$.

A first-order grammar is *deterministic* if, for every X and a, there is at most one production $(X, a, E) \in \mathcal{R}$.

Just as a simple grammar defines an LTS over words of nonterminals, a first-order grammar defines an LTS over the set $\mathcal{E}_0$ of closed expressions. For each production $X\ \alpha_1 \dots \alpha_m \xrightarrow{a} E$ we have the labelled transition $X\ E_1 \dots E_m \xrightarrow{a} E[E_1/\alpha_1, \dots, E_m/\alpha_m]$.

Let $\approx$ denote bisimilarity over closed expressions according to a first-order grammar. We now present a fully abstract (i.e., preserving bisimilarity) translation of a deterministic first-order grammar into a type in F^{μ}_{ω}. Each grammar variable α has a corresponding type variable α (of kind T). An expression $X\ E_1 \dots E_m$ is represented as a type application $X\ E_1 \dots E_m$. If X has arity m and the productions $X\ \alpha_1 \dots \alpha_m \xrightarrow{a_j} E_j$ for a range of j, then we write the equation specifying X as a record (since the first-order grammar is deterministic, all record labels are distinct, and thus the right-hand side on the equation specifying X is well-formed).

$$X \doteq \lambda\alpha_1 \colon \mathrm{T}.\dots\lambda\alpha_m \colon \mathrm{T}.\{a_1 \colon E_1, \dots, a_m \colon E_m\}$$

This gives rise to a system of equations $\{X_i \doteq T_i\}$, one for each nonterminal X_i, where the nonterminals may appear in the right-hand sides T_i. Finally, given an initial expression E, it is standard how to convert it into a μ-type using the system above.

Using the above translation, we are able to simulate a transition $E \xrightarrow{a_j} F$ of the first-order grammar as a transition $E \xrightarrow{\{\overline{a_i}\}_j} F$ on the corresponding types. Therefore, the translation is fully abstract and we get the following result.

Theorem 5. *Let E and F be closed expressions on a first-order grammar and E, F the corresponding types. Then $E \approx F$ iff $E \sim F$.*

Let us work on an example to better understand the above translation. Consider the language $L_3 = \{\ell^n a r^n a \mid n \geq 0\} \cup \{\ell^n b r^n b \mid n \geq 0\}$ over the alphabet $\{a, b, \ell, r\}$. L_3 is a typical example of a language that cannot be described with a simple grammar, but can be accepted by a deterministic pushdown automaton [51]. Consider the first-order grammar with nonterminals $X, R, A, B, \bot$, initial expression $X\ A\ B$, and productions

$$X\ \alpha\ \beta \xrightarrow{\ell} X\ (R\ \alpha)\ (R\ \beta) \qquad X\ \alpha\ \beta \xrightarrow{a} \alpha \qquad X\ \alpha\ \beta \xrightarrow{b} \beta$$
$$R\ \alpha \xrightarrow{r} \alpha \qquad A \xrightarrow{a} \bot \qquad B \xrightarrow{b} \bot$$

Note that $\bot$ is a constant without productions. It is easy to see that the traces of this first-order grammar correspond exactly to the words in L_3. By following the steps in the above translation, we arrive at the system of equations

$$X \doteq \lambda\alpha \colon \mathrm{T}.\lambda\beta \colon \mathrm{T}.\{\ell \colon X(R\alpha)(R\beta), a \colon \alpha, b \colon \beta\} \qquad R \doteq \lambda\alpha \colon \mathrm{T}.\{r \colon \alpha\}$$
$$A \doteq \{a \colon \bot\} \qquad B \doteq \{b \colon \bot\} \qquad \bot \doteq \{\}$$

Therefore, the initial expression $X\ A\ B$ becomes the type

$$(\mu\,\xi \colon \mathrm{T} \Rightarrow \mathrm{T} \Rightarrow \mathrm{T}.\,\lambda\alpha \colon \mathrm{T}.\lambda\beta \colon \mathrm{T}.\{\ell \colon \xi\{r \colon \alpha\}\{r \colon \beta\}, a \colon \alpha, b \colon \beta\})\{a \colon \{\}\}\{b \colon \{\}\},$$

whose transitions simulate the transitions of the first-order grammar.

$$\begin{array}{rcl}
v & ::= & c \mid x \mid \lambda x\colon T.t \mid \mathsf{rec}\, x\colon T.v \mid \Lambda\alpha\colon \kappa.v \mid \{\overline{l_i = v_i}\} \mid \langle l = v\rangle\, \mathsf{as}\, T \\
 & & \mathsf{receive}[T] \mid \mathsf{receive}[T][T] \mid \mathsf{send}[T] \mid \mathsf{send}[T]\, v \mid \mathsf{send}[T]\, v[T] \\
t & ::= & v \mid t\, t \mid t[T] \mid \{\overline{l_i = t_i}\} \mid \mathsf{let}\, \{\overline{l_i = x_i}\} = t\, \mathsf{in}\, t \\
 & & \langle l = t\rangle\, \mathsf{as}\, T \mid \mathsf{case}\, t\, \mathsf{of}\, t \mid \mathsf{match}\, t\, \mathsf{with}\, t \\
p & ::= & \langle t\rangle \mid p \mid p \mid (\nu x x)p
\end{array}$$

c ::=		Term constant
$\mathsf{receive}$	$\forall\alpha\colon \textsc{t}.\, \forall\beta\colon \textsc{s}.\, ?\alpha.\beta \to \alpha \otimes \beta$	receive on a channel
send	$\forall\alpha\colon \textsc{t}.\, \alpha \to \forall\beta\colon \textsc{s}.\, !\alpha.\beta \to \beta$	send on a channel
$\mathsf{select}\, l_j\, \mathsf{as} \oplus\{\overline{l_i\colon T_i}\}$	$\oplus\{\overline{l_i\colon T_i}\} \to T_j$	internal choice
close	$\mathsf{End} \to \mathsf{Unit}$	channel close
fork	$(\mathsf{Unit} \to \mathsf{Unit}) \to \mathsf{Unit}$	fork a new thread
new	$\forall\alpha\colon \textsc{s}.\, a \to \alpha \otimes \mathsf{Dual}\, \alpha$	channel creation

Fig. 10: Terms and types for term constants.

6 The term language and its metatheory

This section briefly introduces a concurrent functional language equipped with $F^{\mu_*;}_{\omega}$ types, together with its metatheory. The results mostly follow from those in the literature, although explicit recursion at the term level and the unrestricted bindings in typing contexts are somewhat new in session types. The complete set of rules is to be found in the technical report [20].

The syntax of terms and processes is defined by the grammar in Fig. 10. The same figure introduces types for the constants. The term language is essentially the polymorphic lambda calculus with support for session operators, formulated as in Almeida et al. and Cai et al. [2,14]. From System F it comprises terms and type abstractions, records and variants, including constructors and destructors in each case. The support for session operations and concurrency includes channel creation (new), the different channel operations (receive, send, match, select and close) and thread creation (fork). We program at the term level and use processes only for the runtime. Processes include terms as threads, parallel composition and channel creation, all inspired in the pi-calculus with double binders [73].

Process typing and an excerpt of term typing is in Fig. 11. A judgement of the form $\Delta \mid \Gamma \vdash t\colon T$ records the fact that term t has type T under contexts Δ (recording kinds for type variables) and Γ (recording types for term variables). The judgement for processes, $\Gamma \vdash p$, says that p is well-typed under context Γ. It simplifies that for terms, since processes feature no free type variables and are assigned no particular type. Once again, the rules are adapted from the two above cited works. The difference to Cai et al. is that we work in a linear setting and hence axioms (T-Const and T-Var) work on an empty context, and most of the other rules must split the context accordingly. Rule T-TAbs simplifies

Term typing $\boxed{\Delta \mid \Gamma \vdash t \colon T}$

T-Const
$$\frac{\Delta \vdash T_c : *}{\Delta \mid \cdot \vdash c \colon T_c}$$

T-Var
$$\Delta \mid x \colon T \vdash x \colon T$$

T-App
$$\frac{\Delta \mid \Gamma_1 \vdash t_1 \colon U \to T \qquad \Delta \mid \Gamma_2 \vdash t_2 \colon U}{\Delta \mid \Gamma_1, \Gamma_2 \vdash t_1\, t_2 \colon T}$$

T-Rec
$$\frac{\Delta \vdash T : * \qquad \Delta \mid \Gamma, x :^{\omega} T \to U \vdash v \colon T \to U}{\Delta \mid \Gamma \vdash \mathsf{rec}\, x \colon T \to U.v \colon T \to U}$$

T-TAbs
$$\frac{\Delta, \alpha \colon \kappa \mid \Gamma \vdash v \colon T\,\alpha}{\Delta \mid \Gamma \vdash (\Lambda\alpha \colon \kappa.v) \colon \forall_{\kappa}\, T}$$

T-Match
$$\frac{\Delta \mid \Gamma_1 \vdash t_1 \colon \&\{\overline{l_i : T_i}\} \qquad \Delta \mid \Gamma_2 \vdash t_2 \colon \{\overline{l_i \colon T_i \to T}\}}{\Delta \mid \Gamma_1, \Gamma_2 \vdash \mathsf{match}\, t_1\, \mathsf{with}\, t_2 \colon T}$$

T-Eq
$$\frac{\Delta \mid \Gamma \vdash t \colon U \qquad \Delta \vdash U : * \qquad U \sim T}{\Delta \mid \Gamma \vdash t \colon T}$$

T-Dereliction
$$\frac{\Delta \mid \Gamma, x \colon T \vdash t \colon U}{\Delta \mid \Gamma, x :^{\omega} T \vdash t \colon U}$$

T-Weakening
$$\frac{\Delta \mid \Gamma \vdash t \colon U}{\Delta \mid \Gamma, x :^{\omega} T \vdash t \colon U}$$

T-Contraction
$$\frac{\Delta \mid \Gamma, y :^{\omega} T, z :^{\omega} T \vdash t \colon U}{\Delta \mid \Gamma, x :^{\omega} T \vdash t[x/y][x/z] \colon U}$$

Process typing $\boxed{\Gamma \vdash p}$

$$\frac{\varepsilon \mid \Gamma \vdash t \colon \mathsf{Unit}}{\Gamma \vdash \langle t \rangle} \qquad \frac{\Gamma_1 \vdash p_1 \qquad \Gamma_2 \vdash p_2}{\Gamma_1, \Gamma_2 \vdash p_1 \mid p_2} \qquad \frac{\Gamma, x \colon T, y \colon \mathsf{Dual}\, T \vdash p}{\Gamma \vdash (\nu x y) p}$$

Fig. 11: Typing (excerpt).

that of Cai et al.; we can easily show that both rules are interchangeable. We support exponentials [37] for recursive functions, so that one may write functions that feature more than one recursive call (good for consuming binary trees, for example) and branches that do not use the recursive function (for code that is supposed to terminate). Towards this end, we add an unrestricted binding $x :^{\omega} T$ in term variable contexts, an explicit rule for rec (as opposed to making rec a constant as in Cai et al. [14]) and substructural rules for unrestricted bindings (T-Dereliction, T-Weakening and T-Contraction).

Thanks to the power of System F, most of the session and concurrency operators are expressed as constants. For example, receive receives a session type $!\alpha.\beta$ with α, the payload of the message, an arbitrary type and β, the continuation, a session type, and returns a pair of the value received and the continuation channel. As usual $\forall\alpha \colon \kappa.T$ abbreviates the type $\forall_{\kappa}\,(\lambda\alpha \colon \kappa.T)$. The exception is the external choice (T-Match) which can not be captured by a type (similarly to T-Case) and hence requires a dedicated typing rule.

Process reduction is in Fig. 12. Following Milner [55] we factor out processes by means of a structural congruence relation that accounts for the associative and commutative nature of parallel composition, scope extrusion and exchanging the order of channel bindings.We now address the metatheory of our language, starting with preservation for both terms and processes.

Process reduction $\boxed{p \to p}$

$$\frac{t_1 \to t_2}{\langle t_1\rangle \to \langle t_2\rangle} \qquad \langle E[\mathsf{fork}\, v]\rangle \to \langle E[\{\}]\rangle \mid \langle v\,\{\}\rangle \qquad \langle E[\mathsf{new}[T]]\rangle \to (\nu xy)\langle E[\{x,y\}]\rangle$$

$$(\nu xy)(\langle E_1[\mathsf{receive}[T][U]\, y]\rangle \mid \langle E_2[\mathsf{send}[V][W]\, v\, x]\rangle) \to (\nu xy)(\langle E_1[\{y,v\}]\rangle \mid \langle E_2[x]\rangle)$$

$$(\nu xy)(\langle E_1[\mathsf{match}\, y\, \mathsf{with}\, \{\overline{l_i = t_i}\}]\rangle \mid \langle E_2[(\mathsf{select}\, l_j\, \mathsf{as}\, T)\, x]\rangle) \to (\nu xy)\langle E_1[t_j\, y]\rangle \mid \langle E_2[x]\rangle$$

$$(\nu xy)(\langle E_1[\mathsf{close}\, y]\rangle \mid \langle E_2[\mathsf{close}\, x]\rangle) \to \langle E_1[\{\}]\rangle \mid \langle E_2[\{\}]\rangle \qquad \frac{p_1 \to p_2}{p_1 \mid q \to p_2 \mid q}$$

$$\frac{p_1 \to p_2}{(\nu xy)p_1 \to (\nu xy)p_2} \qquad \frac{p_1 \equiv p_2 \qquad p_2 \to p_3 \qquad p_3 \equiv p_4}{p_1 \to p_4}$$

Fig. 12: Process reduction.

Theorem 6 (Preservation).

1. *If $\Delta \mid \Gamma \vdash t : T$ and $t \to t'$, then $\Delta \mid \Gamma \vdash t' : T$.*
2. *If $\Gamma \vdash p$ and $p \equiv p'$, then $\Gamma \vdash p'$.*
3. *If $\Gamma \vdash p$ and $p \to p'$, then $\Gamma \vdash p'$.*

Progress for the term language is assured only when the typing context contains channel endpoints only. When Δ is understood from the context we write Γ^{S} to mean that Γ contains only types of kind S, that is $\Delta \vdash T : \mathsf{S}$ for all types T in Γ. Well typed terms are values, or else they may reduce or are ready to reduce at the process level. Reduction in the case of session operations—receive, send, match, select, close—is pending a matching counterpart.

Theorem 7 (Progress for the term language). *If $\Delta \mid \Gamma^{\mathsf{S}} \vdash t : T$, then t is a value, t reduces, or t is stuck in one of the following forms: $E[\mathsf{fork}\, v]$, $E[\mathsf{new}[T]]$, $E[\mathsf{receive}[T][U]\, v]$, $E[\mathsf{send}[U]\, T[v]\, x]$, $E[\mathsf{match}\, y\, \mathsf{with}\, \{\overline{l_i = t_i}\}]$, $E[(\mathsf{select}\, l_j\, \mathsf{as}\, T)\, x]$, or $E[\mathsf{close}\, x]$.*

In order to state our result on the absence of runtime errors we need a few notions on the structure of terms and processes; here we follow Almeida et al. [2]. The *subject* of an expression e, denoted by subj(e), is x in the following cases.

$$\mathsf{receive}[T][U]\, x \qquad \mathsf{send}[T]\, v[U]\, x \qquad \mathsf{match}\, x\, \mathsf{with}\, t \qquad (\mathsf{select}\, l_j\, \mathsf{as}\, T)\, x \qquad \mathsf{close}\, x$$

Two terms e_1 and e_2 *agree* on channel xy, notation $\mathrm{agree}^{xy}(e_1, e_2)$, in the following cases (symmetric forms omitted).

$$\mathrm{agree}^{xy}(\mathsf{receive}[T][U]\, x, \mathsf{send}[V]\, v[W]\, y) \qquad \mathrm{agree}^{xy}(\mathsf{close}\, x, \mathsf{close}\, y)$$
$$\mathrm{agree}^{xy}(\mathsf{match}\, x\, \mathsf{with}\, \{\overline{l_i = t_i}\}_{i\in I}, (\mathsf{select}\, l_j\, \mathsf{as}\, T)\, y) \quad j \in I$$

A closed process is a *runtime error* if it is structurally congruent to some process that contains a subexpression or subprocess of one of the following forms.

1. $v\,u$ where v is not a λ or a rec and $v \neq \mathsf{receive}[T][U]$, $\mathsf{send}[T]\,u$, $\mathsf{send}[T]\,w[U]$, $\mathsf{select}\,l_j\,\mathsf{as}\,T$, close, fork;
2. $v[T]$ where v in not a Λ and $v \neq \mathsf{receive}$, $\mathsf{receive}[U]$, send, $\mathsf{send}[U]$, new;
3. $\mathsf{let}\,\{\overline{l_i = x_i}\} = v\,\mathsf{in}\,t$ and v is not of the form $\{\overline{l_i = u_i}\}$;
4. $\mathsf{case}\,v\,\mathsf{of}\,t$ and $v \neq \langle l_j = u\rangle\,\mathsf{as}\,T$ or $t \neq \{\overline{l_i = u_i}\}_{i \in I}$ with $j \notin I$;
5. $\mathsf{receive}[T][U]\,v$ or $\mathsf{send}[T]\,u[U]\,v$ or $\mathsf{match}\,v\,\mathsf{with}\,t$ or $(\mathsf{select}\,l\,\mathsf{as}\,T)\,v$ or $\mathsf{close}\,v$ and v is not an endpoint x;
6. $\langle E_1[e_1]\rangle \mid \langle E_2[e_2]\rangle$ and $\mathrm{subj}(e_1) = \mathrm{subj}(e_2)$;
7. $(\nu xy)(\langle E_1[e_1]\rangle \mid \langle E_2[e_2]\rangle)$ and $\mathrm{subj}(e_1) = x$, $\mathrm{subj}(e_2) = y$, $\neg\,\mathrm{agree}^{xy}(e_1, e_2)$.

The four cases are standard to system F with records and variants. The support for session types and concurrency in the first two cases (term and type application) are derived from the types of values for such operators (Fig. 10). Item 5 addresses session operators applied to non endpoints. Item 6 is for two concurrent session operators on the same channel end. Finally, Item 7 is for mismatches on two session operations on two endpoints for the same channel.

Theorem 8 (Safety). *If $\Gamma^{\mathsf{s}} \vdash p$, then p is not a runtime error.*

An *algorithmic typing system* can be easily extracted from the declarative system for terms in Fig. 11 via a bidirectional type system, formulated along the lines of Almeida et al. [2].

7 Related Work

Equirecursion in system F. In first investigations on equirecursive types, the notion of type equivalence is often formulated in a coinductive fashion [5,11,18,29,38]. Two types are equivalent if they unroll into the same infinite tree. Whenever this unrolling is the only type-level computation, such trees are regular, enabling efficient decision procedures. Some authors have studied equirecursion together with other notions of type-level computation. Solomon considers parameterized type definitions, which correspond to higher-order kinds [63]. These implicitly correspond to λ-terms, since reduction occurs as types are allowed to call other types. Some authors consider equirecursion in system F_ω, with weaker or stronger notions of equality [1,12,14,41]. Regarding equirecursion in system F, the model of Cai et al. [14] is the closest to ours, and indeed our results up to $F_\omega^{\mu_*\cdot}$ can be seen as a generalisation of theirs. However, Cai et al. depart from the usual setting by allowing non-contractive types (which most authors forbid, including this work), requiring a sort of infinitary lambda calculus. Moreover, this work further extends additional equivalence properties by including session types with their distinctive semantics, such as sequential composition and duality.

Session type systems. Session types were introduced in the 90s by Honda et al. [42,43,67]. Equirecursion was the first approach used to construct infinite session types, which often allows type equality to be interpreted according to a coinductive notion of bisimulation [52]. In this vein, Keizer et al. [48] utilize coalgebras to represent session types. Since the inception of session types, there has been an interest in extending the theory to nonregular protocols [58,59,66]. Context-free session types emerged as a natural extension, as it still allowed for practical type equality algorithms [3,4,19,28,56,68]. Other approaches that go beyond regular session types include nested session types [24] as well as 1-counter, pushdown and 2-counter session types [33]. However, the more expressive notions are not amenable to practical type equivalence algorithms, just like the higher-order types present in our system F^{μ}_{ω}. Polymorphism in session types has also been a topic of interest, with or without recursion [15,22,23,31,39].

Dual type operator. This work is, to the best of our knowledge, the first that internalises duality as a type constructor. Other settings, such as the language Alms [72], consider duality for session types as a user-definable, not built in, type function. Our Dual is a type operator, not a type function. The difference is that a type function involves a type-level computation, which converges to a type written without dual. For example, in Alms we would have $\mathsf{dual}(!\mathsf{Int}.\mathsf{End}) = ?\mathsf{Int}.\mathsf{End}$ (as a type-level computation), both sides being the *same* type. In our setting, $\mathsf{Dual}\,(!\mathsf{Int}; \mathsf{End})$ is a type on its own, which happens to be equivalent to $?\mathsf{Int}; \mathsf{End}$. At the same time, our setting allows for types such as $\mathsf{Dual}\,\alpha$, or $(\mathsf{Dual}\,\alpha); T_1; T_2$, which do not reduce.

Type equivalence algorithms. Algorithms for deciding the equivalence of types must inherently be related to the computational power of the corresponding type system. This has been used implicitly or explicitly to obtain decidability results. As already explained, if equirecursion is the only type-level computation, types can be represented as finite-state automata (or equivalently, infinite regular trees). Although some exponential time algorithms were first proposed [32], it has been established that the problem can be solved in quadratic time [53], which is to be expected as it matches the corresponding problem of bisimulation of finite-state automata [44]; see also Pierce [57].

The next ‘simplest’ model of computation is that of simple grammars, which intuitively correspond to deterministic pushdown automata with a single state [33]. Almeida et al. [4] provided a practical algorithm for checking the bisimilarity of simple grammars. By dropping the determinism assumption, we arrive at Greibach normal form grammars, which are equivalent to basic process algebras [6,7]. Bisimilarity algorithms have been studied extensively in this setting [13,17,47,49]; presently it is known that the complexity of the problem lies between EXPTIME and 2-EXPTIME, which does not exclude the possibility of a polynomial time algorithm for the simpler model of simple grammars.

In this paper we present a reduction from first-order grammars to F^{μ}_{ω}-types, showing that the more expressive type systems (F^{μ}_{ω}, presented here and in Cai et al. [14], as well as its extensions) are at least as powerful as deterministic pushdown automata. As far as we know, the closest result to ours is by Solomon [63], which shows conversions between a universe of “context-free types” and deterministic context-free languages. The universe of types studied by Solomon is different from F^{μ}_{ω}. With some work we could prove that Solomon’s types can be embedded into F^{μ}_{ω}, which would entail our result as a corollary. However, it is easier and simpler to prove directly the reduction as we did.

The equivalence problem for deterministic pushdown automata was a notorious open problem for a long time, until Sénizergues showed it to be decidable [61,62]. Since his proof, many authors have tried to refine the result in an attempt to arrive at an implementable algorithm [46,64,65].

Concurrent term languages. The usefulness of a type system is directly related to its capability to be used in a programming language. Type systems such as the ones discussed in this work lend themselves quite readily to functional term languages [45]. For session types, existing term languages are either inspired in the pi calculus [26,73,69] or in the lambda calculus [35,54,70], or even the two [71]. The system presented in this paper is linear, meaning that resources must be used exactly once [50,74]. Some authors go beyond linearity by considering unrestricted type qualifiers [48,73] or manifest sharing [8].

8 Conclusion and future work

This paper introduces an extension of system F which includes equirecursion, lambda abstractions, and context-free session types. We present type equivalence algorithms, and a term language and its metatheory. Although we have defined a rather general system, it turns out that for practical purposes one must restrict recursion to μ_*, that is, to type-level monomorphic recursion. In any case, the main system $F_\omega^{\mu_*;}$ is a non-trivial extension of (the contractive fragment of) $F_\omega^{\mu_*}$ (studied by Cai et al. [14]) as well as $F^{\mu;}$ (studied by Almeida et al. [19]).

We have only considered polymorphic types of a functional nature: type $\forall\alpha\colon\kappa.T$ must always be of kind T. It is worth investigating polymorphism over session types, as it would allow further additional behaviour. For example, we could be interested in streaming values of heterogeneous nature, as in type $\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Done}\colon \mathsf{Skip}, \mathsf{More}\colon \forall\beta\colon \mathrm{T}.\,?\beta;\alpha\}$. It is however unclear whether this extension would still allow a translation into a simple grammar.

We proved that the type equivalence problem for systems F_ω^{μ}, $F_\omega^{\mu\cdot}$, $F_\omega^{\mu;}$ is at least as hard as a non-efficiently-decidable problem. We conjecture that these systems have the same power as deterministic pushdown automata (and hence, admit decidable type equivalence), but we do not have a construction to prove this result. In any case, our proof that the type equivalence problem is at least as hard as the bisimilarity of deterministic pushdown automata is enough to justify focus on the significant fragment with restricted recursion.

We study either full recursion (for theoretical results) or recursion limited to kind $*$ (for algorithmic results). It would be interesting to study in-between kinds of recursion; the next natural example is $\mu_{*\Rightarrow *}$. What model of computation would we arrive at if we consider types written with this recursion operator? We conjecture that types F_ω^{μ} and $F_\omega^{\mu\cdot}$, when restricted to recursion of kind $* \Rightarrow *$, would still be expressible as simple grammars, whereas such a restriction in the more powerful $F_\omega^{\mu;}$ would take us beyond this model, but perhaps without reaching the expressivity of deterministic pushdown automata.

References

1. Abel, A.: Type-based termination: a polymorphic lambda-calculus with sized higher-order types. Ph.D. thesis, Ludwig Maximilians University Munich (2007), https://d-nb.info/984765581
2. 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
3. Almeida, B., Mordido, A., Vasconcelos, V.T.: FreeST: Context-free session types in a functional language. In: PLACES. EPTCS, vol. 291, pp. 12–23 (2019). https://doi.org/10.4204/EPTCS.291.2
4. 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
5. Amadio, R.M., Cardelli, L.: Subtyping recursive types. In: POPL. pp. 104–118. ACM Press (1991). https://doi.org/10.1145/99583.99600
6. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Decidability of bisimulation equivalence for processes generating context-free languages. In: PARLE. LNCS, vol. 259, pp. 94–111. Springer (1987). https://doi.org/10.1007/3-540-17945-3_5
7. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Decidability of bisimulation equivalence for processes generating context-free languages. J. ACM **40**(3), 653–682 (1993). https://doi.org/10.1145/174130.174141
8. Balzer, S., Pfenning, F.: Manifest sharing with session types. Proc. ACM Program. Lang. **1**(ICFP), 37:1–37:29 (2017). https://doi.org/10.1145/3110281
9. Barendregt, H.P.: The lambda calculus - its syntax and semantics, Studies in logic and the foundations of mathematics, vol. 103. North-Holland (1985)
10. Barendregt, H.P.: The type free lambda calculus. In: Studies in Logic and the Foundations of Mathematics, vol. 90, pp. 1091–1132. Elsevier (1977)
11. Brandt, M., Henglein, F.: Coinductive axiomatization of recursive type equality and subtyping. Fundam. Informaticae **33**(4), 309–338 (1998). https://doi.org/10.3233/FI-1998-33401
12. Bruce, K.B., Cardelli, L., Pierce, B.C.: Comparing object encodings. In: TACS. LNCS, vol. 1281, pp. 415–438. Springer (1997). https://doi.org/10.1007/BFb0014561
13. Burkart, O., Caucal, D., Steffen, B.: An elementary bisimulation decision procedure for arbitrary context-free processes. In: MFCS. LNCS, vol. 969, pp. 423–433. Springer (1995). https://doi.org/10.1007/3-540-60246-1_148
14. 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
15. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: ESOP. LNCS, vol. 7792, pp. 330–349. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_19
16. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. **17**(4), 471–522 (1985). https://doi.org/10.1145/6041.6042
17. Christensen, S., Hüttel, H., Stirling, C.: Bisimulation equivalence is decidable for all context-free processes. Inf. Comput. **121**(2), 143–148 (1995). https://doi.org/10.1006/inco.1995.1129

18. Colazzo, D., Ghelli, G.: Subtyping recursive types in kernel Fun. In: LICS. pp. 137–146. IEEE Computer Society (1999). https://doi.org/10.1109/LICS.1999.782605
19. Costa, D., Mordido, A., Poças, D., Vasconcelos, V.T.: Higher-order context-free session types in system F. In: PLACES. EPTCS, vol. 356, pp. 24–35 (2022). https://doi.org/10.4204/EPTCS.356.3
20. Costa, D., Mordido, A., Poças, D., Vasconcelos, V.T.: System F^{μ}_{ω} with context-free session types. CoRR **abs/2301.08659** (2023), http://arxiv.org/abs/2301.08659
21. Curry, H.H., Feys, R., Craig, W. (eds.): Combinatory Logic, Volume I. North-Holland (1958)
22. Dardha, O.: Recursive session types revisited. In: BEAT. EPTCS, vol. 162, pp. 27–34 (2014). https://doi.org/10.4204/EPTCS.162.4
23. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017). https://doi.org/10.1016/j.ic.2017.06.002
24. Das, A., DeYoung, H., Mordido, A., Pfenning, F.: Nested session types. In: ESOP. LNCS, vol. 12648, pp. 178–206. Springer (2021). https://doi.org/10.1007/978-3-030-72019-3_7
25. Das, A., DeYoung, H., Mordido, A., Pfenning, F.: Nested session types. ACM Trans. Program. Lang. Syst. **44**(3), 19:1–19:45 (2022). https://doi.org/10.1145/3539656
26. Das, A., Pfenning, F.: Rast: A language for resource-aware session types. Log. Methods Comput. Sci. **18**(1) (2022). https://doi.org/10.46298/lmcs-18(1:9)2022
27. De Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In: Indagationes Mathematicae. vol. 75, pp. 381–392. Elsevier (1972). https://doi.org/10.1016/1385-7258(72)90034-0
28. The FreeST programming language. https://freest-lang.github.io/ (2019)
29. Gapeyev, V., Levin, M.Y., Pierce, B.C.: Recursive subtyping revealed: functional pearl. In: ICFP. pp. 221–231. ACM (2000). https://doi.org/10.1145/351240.351261
30. Gauthier, N., Pottier, F.: Numbering matters: first-order canonical forms for second-order recursive types. In: ICFP. pp. 150–161. ACM (2004). https://doi.org/10.1145/1016850.1016872
31. Gay, S.J.: Bounded polymorphism in session types. MSCS **18**(5), 895–930 (2008). https://doi.org/10.1017/S0960129508006944
32. 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
33. Gay, S.J., Poças, D., Vasconcelos, V.T.: The different shades of infinite session types. In: FoSSaCS. LNCS, vol. 13242, pp. 347–367. Springer (2022). https://doi.org/10.1007/978-3-030-99253-8_18
34. 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
35. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. **20**(1), 19–50 (2010). https://doi.org/10.1017/S0956796809990268
36. Girard, J.Y.: Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. thesis, Éditeur inconnu (1972)
37. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
38. Glew, N.: A theory of second-order trees. In: ESOP. LNCS, vol. 2305, pp. 147–161. Springer (2002). https://doi.org/10.1007/3-540-45927-8_11
39. Griffith, D.E.: Polarized substructural session types. Ph.D. thesis, University of Illinois at Urbana-Champaign (2016). https://doi.org/10.2172/1562827

40. Hindley, J.R., Seldin, J.P.: Introduction to Combinators and Lambda-Calculus. Cambridge University Press (1986)
41. Hinze, R.: Polytypic values possess polykinded types. Sci. Comput. Program. **43**(2-3), 129–159 (2002). https://doi.org/10.1016/S0167-6423(02)00025-4
42. Honda, K.: Types for dyadic interaction. In: CONCUR. LNCS, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35
43. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
44. Hopcroft, J.E., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Tech. rep., Cornell University (1971)
45. Im, H., Nakata, K., Park, S.: Contractive signatures with recursive types, type parameters, and abstract types. In: ICALP. LNCS, vol. 7966, pp. 299–311. Springer (2013). https://doi.org/10.1007/978-3-642-39212-2_28
46. Jančar, P.: Short decidability proof for DPDA language equivalence via 1st order grammar bisimilarity. CoRR **abs/1010.4760** (2010), http://arxiv.org/abs/1010.4760
47. Jančar, P.: Bisimilarity on basic process algebra is in 2-ExpTime (an explicit proof). Log. Methods Comput. Sci. **9**(1) (2012). https://doi.org/10.2168/LMCS-9(1:10)2013
48. Keizer, A.C., Basold, H., Pérez, J.A.: Session coalgebras: A coalgebraic view on session types and communication protocols. In: ESOP. LNCS, vol. 12648, pp. 375–403. Springer (2021). https://doi.org/10.1007/978-3-030-72019-3_14
49. Kiefer, S.: BPA bisimilarity is EXPTIME-hard. Inf. Process. Lett. **113**(4), 101–106 (2013). https://doi.org/10.1016/j.ipl.2012.12.004
50. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Trans. Program. Lang. Syst. **21**(5), 914–947 (1999). https://doi.org/10.1145/330249.330251
51. Korenjak, A.J., Hopcroft, J.E.: Simple deterministic languages. In: SWAT. pp. 36–46. IEEE Computer Society (1966). https://doi.org/10.1109/SWAT.1966.22
52. Kozen, D., Silva, A.: Practical coinduction. Math. Struct. Comput. Sci. **27**(7), 1132–1152 (2017). https://doi.org/10.1017/S0960129515000493
53. Lange, J., Yoshida, N.: Characteristic formulae for session types. In: TACAS. LNCS, vol. 9636, pp. 833–850. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_52
54. Lindley, S., Morris, J.G.: Talking bananas: structural recursion for session types. In: ICFP. pp. 434–447. ACM (2016). https://doi.org/10.1145/2951913.2951921
55. Milner, R.: Functions as processes. Math. Struct. Comput. Sci. **2**(2), 119–141 (1992). https://doi.org/10.1017/S0960129500001407
56. Padovani, L.: Context-free session type inference. ACM Trans. Program. Lang. Syst. **41**(2), 9:1–9:37 (2019). https://doi.org/10.1145/3229062
57. Pierce, B.C.: Types and programming languages. MIT Press (2002)
58. Puntigam, F.: Non-regular process types. In: Euro-Par. LNCS, vol. 1685, pp. 1334–1343. Springer (1999). https://doi.org/10.1007/3-540-48311-X_189
59. Ravara, A., Vasconcelos, V.T.: Behavioural types for a calculus of concurrent objects. In: Euro-Par. LNCS, vol. 1300, pp. 554–561. Springer (1997). https://doi.org/10.1007/BFb0002782
60. Reynolds, J.C.: Towards a theory of type structure. In: Programming Symposium. LNCS, vol. 19, pp. 408–423. Springer (1974). https://doi.org/10.1007/3-540-06859-7_148

61. Sénizergues, G.: The equivalence problem for deterministic pushdown automata is decidable. In: ICALP. LNCS, vol. 1256, pp. 671–681. Springer (1997). https://doi.org/10.1007/3-540-63165-8_221
62. Sénizergues, G.: L(A) = L(B)? decidability results from complete formal systems. In: ICALP. LNCS, vol. 2380, p. 37. Springer (2002). https://doi.org/10.1007/3-540-45465-9_4
63. Solomon, M.H.: Type definitions with parameters. In: POPL. pp. 31–38. ACM Press (1978). https://doi.org/10.1145/512760.512765
64. Stirling, C.: Decidability of DPDA equivalence. Theor. Comput. Sci. **255**(1-2), 1–31 (2001). https://doi.org/10.1016/S0304-3975(00)00389-3
65. Stirling, C.: Deciding DPDA equivalence is primitive recursive. In: ICALP. Lecture Notes in Computer Science, vol. 2380, pp. 821–832. Springer (2002). https://doi.org/10.1007/3-540-45465-9_70
66. Südholt, M.: A model of components with non-regular protocols. In: SC. LNCS, vol. 3628, pp. 99–113. Springer (2005). https://doi.org/10.1007/11550679_8
67. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE. LNCS, vol. 817, pp. 398–413. Springer (1994). https://doi.org/10.1007/3-540-58184-7_118
68. Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: ICFP. pp. 462–475. ACM (2016). https://doi.org/10.1145/2951913.2951926
69. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: PPDP. pp. 161–172. ACM (2011). https://doi.org/10.1145/2003476.2003499
70. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: ESOP. LNCS, vol. 7792, pp. 350–369. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_20
71. Toninho, B., Yoshida, N.: On polymorphic sessions and functions: A tale of two (fully abstract) encodings. ACM Trans. Program. Lang. Syst. **43**(2), 7:1–7:55 (2021). https://doi.org/10.1145/3457884
72. Tov, J.A.: Practical programming with substructural types. Ph.D. thesis, Northeastern University (2012)
73. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/j.ic.2012.05.002
74. Walker, D.: Advanced Topics in Types and Programming Languages, chap. Substructural Type Systems, pp. 3–44. The MIT Press (2005)