Information and Computation 289 (2022) 104948

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

Polymorphic lambda calculus with context-free session types

Bernardo Almeida [a,*], Andreia Mordido [a], Peter Thiemann [b], Vasco T. Vasconcelos [a]

[a] LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
[b] Faculty of Engineering, University of Freiburg, Germany

ARTICLE INFO

Article history:
Received 11 June 2021
Received in revised form 30 July 2022
Accepted 31 July 2022
Available online 5 August 2022

Keywords:
Polymorphism
Functional programming
Session types
Context-free types

ABSTRACT

Session types provide a typing discipline for structured communication on bidirectional channels. Context-free session types overcome the restriction to tail recursive protocols characteristic of conventional session types. This extension enables the serialization and deserialization of tree structures in a fully type-safe manner.

We present the theory underlying the language FreeST 2, which features context-free session types in an extension of System F with linear types and a kinding system to distinguish message types, session types, and channel types. The system presents metatheoretical challenges which we address: contractivity in the presence of polymorphism, a non-trivial equational theory on types, and decidability of type equivalence. We also establish standard results on typing preservation, progress, and a characterization of erroneous processes.

1. Introduction

Session types were discovered by Honda as a means to describe structured process interaction on typed communication channels [38,39,57]. Session types allow expressing elaborate protocols on communication channels, sharply contrasting with languages such as Concurrent ML [53] and Go [33,35] that force channels to carry objects of a common type during the whole lifetime of the channel.

Session types provide detailed protocol specifications by describing message exchanges and choice points: if T is a type and U and V session types, then $!T.U$ and $?T.U$ are session types that describe channels where a message of type T is sent or received, and where U describes the ensuing protocol. Choices are usually present in labelled form, so that, in the particular case of a binary choice, $\oplus\{l\colon U, m\colon V\}$ and $\&\{l\colon U, m\colon V\}$ describe channels selecting or offering labels l and m and continuing as U or V according to the choice taken. Type end denotes a channel on which no further interaction is possible. It terminates all session types. Starting from Honda et al. [39], most works on session types incorporate recursive types for unbounded behaviour: type $\mu a.T$ introduces a recursive type with a bound type variable a that can be used to refer to the whole μ-type. For example, type

$$\mu a.\, \oplus\{\mathsf{Push}\colon !\mathsf{Int}.a, \mathsf{Pop}\colon ?\mathsf{Int}.a, \mathsf{Done}\colon \mathsf{end}\}$$

* Corresponding author.
E-mail addresses: bpdalmeida@fc.ul.pt (B. Almeida), afmordido@fc.ul.pt (A. Mordido), thiemann@informatik.uni-freiburg.de (P. Thiemann), vmvasconcelos@fc.ul.pt (V.T. Vasconcelos).

https://doi.org/10.1016/j.ic.2022.104948

describes a channel providing for three operations Push, Pop, and Done. Clients that exercise option Push must then send an integer value, after which the protocol goes back to the beginning. Similarly, the processes that select option Pop must then be ready to receive an integer value before going back to the beginning. Finally, by exercising option Done the client signals protocol completion.

At the other end of the channel a server awaits. Its type is *dual* to that of clients, namely $\mu b.\&\{\mathsf{Push}: ?\mathsf{Int}.b, \mathsf{Pop}: !\mathsf{Int}.b, \mathsf{Done}: \mathsf{end}\}$. The server offers the three options and behaves according to the option selected by the client. The duality $\oplus/\&$ and $!/?$ guarantees that communication does not go wrong: if one partner selects, the other offers, if one side writes, the other reads, and conversely.

Session types are well suited to document communication protocols and there is a whole range of variants and extensions that make them amenable to describe realistic situations, including those featuring multiple partners [40], accounting for object-oriented programming [28], web programming [42] or exceptional behaviour [25]. Despite these advances, session types remain limited in their support for compositionality. Protocols U and V can only be combined under (internal or external) choice, where either U or V are used, but not both. Input and output do not qualify as protocol composition operators for they merely append a simple communication (input or output) at the head of a type. A sequential composition operator on session types, as in $U; V$, greatly increases the flexibility in protocol composition, opening perspectives for describing far richer protocols.

When implementing the client side of the stack protocol just introduced, the programmer is faced with the classical difficulty of what action to take when confronted with a Pop operation on an empty stack. Session types equipped with sequential composition can accurately describe a stack protocol where the Pop operation is not available on an empty stack. Rather than writing a single, long doubly recursive type, we factor the type in two which we introduce by means of two mutually recursive equations. Inspired in Padovani [47], the protocol can be written in FreeST [2] as follows:

```
type EStack = ⊕{Push: !Int ; Stack ; EStack, Done: Skip}
type Stack  = ⊕{Push: !Int ; Stack ; Stack,  Pop:  ?Int}
```

EStack describes an empty stack on which Push and Done operations are available. A Push must be followed by a non-empty stack followed by an empty stack again. According to the type, stack sessions can only terminate when the stack is empty. A non-empty stack offers Push and Pop operations. The former must be followed by two non-empty stacks, the first accounts for the element just pushed, the second for the state the stack was at prior to Push. The Pop operation returns the value at the top of the stack.

The syntactic changes with respect to conventional session types are mild: input and output become types of their own, $!T$ and $?T$, irrespective of the continuation, if any. Prefixes give way to sequential composition. In the process, we rename type end to Skip for the new type behaves differently from its conventional counterpart. Choices remain as they were. Then, what once was !Int.end is now simply !Int and what one once wrote as !Int.?Bool.end can now be written as !Int; ?Bool.

Session types arose in the scope of process calculi, the π-calculus in particular [45]. They however fit nicely with strongly typed functional languages, as shown by Gay et al. [32,62,66], and this is the trend we follow in this paper. Suppose that we are to program a client that must interact with a stack via a channel end c of type EStack. The code can be written in FreeST as follows,

```
let c = select Push c in let     c  = send  5 c in
let c = select Pop  c in let (x, c) = receive c in
let c = select Pop  c in let (y, c) = receive c in
```

where **select** chooses a branch on a given channel and returns a channel where to continue the interaction, **send** sends a value on a given channel and again returns the channel where to continue the interaction, and **receive** returns a pair composed of the value read from the channel and the continuation channel. We thus see that all session operations return a continuation channel; we identify them all by the same identifier c for simplicity. The code does not contain any syntax for unfolding the recursive session type. It is customary for session type systems to use equi-recursion which equates a recursive type with its unfolding. Running the above code through our interpreter [2] on a context that assigns type EStack to identifier c, we get

```
stack.fst:3:11: error:
    Branch Pop not present in external choice type EStack
```

Type EStack (or Stack for that matter) cannot be written with conventional session types. Almeida et al. [4] translate context-free session types to grammars for the purpose of deciding type equivalence. In contrast, the language described by traditional session types is regular. More precisely, taking infinite executions into account, each traditional session type can be related to the union of a regular language and a ω-regular language that describe the finite and infinite sequences of communication actions admitted by the type.

We now look at the code for the server side. We need two mutually recursive functions, one to handle the empty stack, the other for the non empty case. The code closely follows the types:

```
eStack c = match c with
  { Push c → let (x, c) = receive c in eStack (stack x c)
```

```
  , Done  c → c                                                    3
  }                                                                4
stack x c = match c with                                           5
  { Push c → let (y, c) = receive c in stack x (stack y c)         6
  , Pop  c → send x c                                              7
  }                                                                8
```

The **match** expression branches according to the label selected by the client. Next we focus on the types for the two functions. They both expect and return channels. Function eStack consumes an EStack channel to completion, so that its type must be EStack → **Skip** when called for the first time. Similarly, the type of function stack is **Int** → Stack → **Skip** when called for the first time. But then the type of the call (stack x c) in line 2 is **Skip** and thus its value cannot be used as a parameter to function eStack that expects an EStack. Consequently programming in FreeST requires support for polymorphism, and in particular for *polymorphic recursion*. Both functions must be polymorphic in the continuation, so that they may accommodate the top-level and the recursive calls equally. The types for the functions are as follows.

```
eStack : ∀a:SL . EStack;a → a
 stack : ∀a:SL . Int → Stack;a → a
```

The polymorphic types above abstract over a *linear session* type. FreeST distinguishes three base kinds: functional (T), session (S) and message (M) in order to control type formation in the presence of polymorphism. For example, only S types can be used in sequential composition, and only M types can be used in message exchanges (**send** or **receive**). To obtain a kind, we complement base kinds with their multiplicity: L for linear and U for unrestricted. For example, **Skip** : SU and U;V : SL if both U and V are session types. Kinding comes equipped with a partial order that allows both M and S types to be viewed as T, and U as L, so that **Skip** : TL if needed. We often omit the top kind, that is TL, in examples.

Contributions

- We introduce context-free session types that extend the expressiveness of regular session types, allowing type-safe implementation of algorithms beyond the reach of regular sessions.
- We propose a first-order kinding system that distinguishes messages from sessions from functional types as well as linear from unrestricted types, and a notion of contractiveness for recursive types that takes into account types equivalent to **Skip** and ensures that substitution is a total function.
- We formalise the core of FreeST as a functional language $F^{\mu;}$, a polymorphic linear language with equi-recursive types, records and variants, term-level type abstraction and application, channel and thread creation, and session communication primitives governed by context-free session types; we show soundness, absence of runtime errors of the language as well as progress for single-threaded programs.
- We show that type equivalence is decidable in the presence of context-free sessions; we present a bidirectional type checking algorithm and prove it correct with respect to the declarative system.

All these ideas are embodied in a freely available interpreter [2]. This paper polishes and expands Thiemann and Vasconcelos [58]. The two main novelties are impredicative (System F) polymorphism instead of predicative (Damas-Milner [19]) and algorithmic type checking. The original paper includes an embedding of a functional language for regular session types [32]; the embedding is still valid in the language of this paper even if we decided not to discuss it.

Outline The rest of the paper is organised as follows. Section 2 discusses the overall design of FreeST and explains the requirements to the metatheory with examples. Section 3 introduces the type language and the notions of type equivalence and duality. Section 4 introduces the process language and proves type preservation, absence of runtime errors and progress. Section 5 shows that type equivalence is decidable. Section 6 presents algorithmic type checking and proves its correctness with respect to the declarative system. Section 7 discusses related work and section 8 concludes the paper and points to future work.

2. $F^{\mu;}$ in action

Our language for polymorphic context-free session types is called $F^{\mu;}$. It is based on F^{μ}, System F with equi-recursive types. On top of this system we add multi-threading and communication based on context-free session types. Extending polymorphism to session types requires an appropriate kind structure. We freely use FreeST syntax in our examples, deferring the introduction of the formal syntax of $F^{\mu;}$ to section 3.

2.1. Polymorphic session types

Many session type systems model their communication operations with type-indexed families of constants. For instance, Gay and Vasconcelos [32, Fig. 9] specify informal *typing schemas for constants*, e.g. send: $T \to !T.S \to S$ where S may be instantiated with any session type and T with any arbitrary type. This design enables the use of the send operation at

different types, but it is somewhat restrictive. Formally, we would have to make the indices explicit as in $\mathsf{send}_{S,T}$, but more importantly this approach does not allow us to abstracting over operations that send (or receive) data. For example, suppose we want to write a function that sends all data encrypted:

```
sendEncrypted : ∀a b . (a → Int) → a → (!Int;b) → b
sendEncrypted encrypt x c = send (encrypt x) c
```

It takes an encryption function that encodes a value of type a into an integer, a value of type a, and a channel on which we can send an integer. It returns the continuation of the channel. This function cannot be written with a polymorphic type in most previous session-type systems (exceptions include lightweight functional session types [42] and nested session types [22,23]). In $F^{\mu;}$, we define the sending and receiving operations as polymorphic constants. In a first approximation, we might consider the following (incomplete) types:

$$\mathsf{send} : \forall a.\, a \to \forall b.\, !a; b \to b$$

$$\mathsf{receive} : \forall a.\forall b.\, ?a; b \to a \times b$$

The send operation takes a value of type a, a channel of type $!a; b$ (send a and then continue as b), and returns a channel of type b. Similarly, receive takes a channel of type $?a; b$ (receive a and continue as b) and returns a pair of the received value of type a and the remaining channel of type b. This typing suffers from several shortcomings. First, the type b must be a session type in both cases. Instantiating b with a record, a variant or a function type would result in an ill formed type. Second, to keep typing decidable, our system restricts the type a to any type that does *not* contain a session. Third, channel references must be handled in a linear fashion, which is also not reflected in the proposed signature, yet.

Our solution is to classify types with suitable kinds. We identify three base kinds, together with their linearity variants: $\mathbf{M}^{\mathsf{un}}$ and $\mathbf{M}^{\mathsf{lin}}$ stand for types that can be exchanged on channels; $\mathbf{S}^{\mathsf{un}}$ and $\mathbf{S}^{\mathsf{lin}}$ refer to session types; $\mathbf{T}^{\mathsf{un}}$ and $\mathbf{T}^{\mathsf{lin}}$ stand for arbitrary types.

Adding kinds to the types of send and receive we get:

$$\mathsf{send} : \forall a : \mathbf{M}^{\mathsf{lin}}.\, a \to_{\mathsf{un}} \forall b : \mathbf{S}^{\mathsf{lin}}.\, !a; b \to_{\mathsf{lin}} b$$

$$\mathsf{receive} : \forall a : \mathbf{M}^{\mathsf{lin}}.\, \forall b : \mathbf{S}^{\mathsf{lin}}.\, ?a; b \to_{\mathsf{un}} a \times b$$

What types are classified as $\mathbf{M}$ is limited by type equivalence alone. For regular session types, $\mathbf{M}$ coincides with $\mathbf{T}$. For context free session types we restrict $\mathbf{M}$ to base types $()_{\mathsf{un}}$ and $()_{\mathsf{lin}}$ but this could be easily extended to other base types, records and variants, for example.

We annotate the last function arrow in the type of send with lin to cater for the possibility that the first argument $x : a$ is in fact linear. In that case, the closure obtained by partial application send x has type $!a; b \to_{\mathsf{lin}} b$ to indicate that it must be used exactly once (i.e., linearly). The remaining function arrows are unrestricted as indicated by the subscript $\to_{\mathsf{un}}$.

There is no equally pleasing way to enable abstraction over the operations to offer and accept branching in a session type. Hence, these operations are still hardwired as expressions with parametric typing rules.

We conclude with the implementation of encrypted sending and receiving in FreeST. Syntactically, ∀ introduces a universal type, ML stands for $\mathbf{M}^{\mathsf{lin}}$, SL for $\mathbf{S}^{\mathsf{lin}}$, and we use the Haskell notation for pair types.

```
sendEncrypted :
    ∀a:ML . ∀b:SL . (a → Int) → a → (!Int; b) → b
sendEncrypted encrypt x c = send (encrypt x) c

recvEncrypted :
    ∀a:ML . ∀b:SL . (Int → a) → (?Int; b) → (a, b)
recvEncrypted decrypt c =
    let (i, c) = receive c in
    (decrypt i, c)
```

As an example of the use of these abstractions, we implement the addition server with encryption. It takes an encoding function, a decoding function, and an encoded channel to produce **Skip**, i.e., a depleted channel. As customary in System F, we must provide explicit type arguments (in square brackets) to the polymorphic operations for encrypted sending and receiving. The typechecker has special support to infer type arguments for send and receive. We omit the corresponding client implementation, which is straightforward.

```
server : (Int→Int) → (Int→Int) → (?Int;?Int;!Int) → Skip
server enc dec c =
    let (x, c) = recvEncrypted [Int, ?Int;!Int] dec c in
    let (y, c) = recvEncrypted [Int, !Int] dec c in
    sendEncrypted [Int, Skip] enc (x + y) c
```

2.2. Parameterization over sub-protocols

As a more advanced example of polymorphism we consider parameterization over fragments of a protocol. This facility is not available in the standard, tail-recursive formulations of session types. It relies crucially on the availability of the composition operator ";" that forms the sequential composition of two protocols. Using composition we can write parametric wrapper functions to implement protocols such as !Auth; &{Ok: s; ?Acct, Denied: **Skip**}, where the parameter s is the session type of the actual payload protocol of a client. For example, the client wrapper might send an authentication token of type Auth. If the authentication is accepted, then the actual payload protocol s proceeds and accounting information of type Acct is attached. Otherwise, execution of the payload protocol is skipped.

An implementation of the client wrapper would have to parameterise over the client function implementing the payload protocol. The proper type for this wrapper involves higher-rank polymorphism to parameterise over the type of the continuation. In this example, we hardcode the password for authentication as a constant in the code.

```
password : Auth
password = ...

clientWrapper :
    ∀a b:MU .
    ∀s cont:SL .
    (∀d:SL . a → s;d → (b, d)) →
    a → b →
    !Auth; &{ Ok: s;?Acct, Denied: Skip}; cont →
    (b, cont)
clientWrapper proto init def c =
    let c = send password c in
    match c with {
        Ok c → let (ret, c) = proto [?Int; cont] init c in
               let (acc, c) = receive c in
               (ret, c),
        Denied c → (def, c)
    }
```

The first parameter is a function that processes the payload protocol. It is polymorphic in its continuation parameter d so that it could be invoked in different contexts. It takes an initial parameter of type a, transforms a channel of type s;d, and returns a result of type b paired with the continuation d. Then there are parameters for the initial parameter of the payload and to provide a default value of type b. Finally, the channel with the augmented protocol is transformed into a pair of the result of the payload and the continuation channel.

The client wrapper can be used with multiple instantiations for the protocol. As an example, we consider a recursive protocol to send a list of integer items.

```
type ListC:SL = ⊕{Item: !Int; ListC, Stop: Skip}
```

We write a client for this protocol where the initial value and the output are integers. The infix operator & : ∀a:TL . ∀b:TL . a → (a → b) → b represents reverse function application; we use it to hide the continuation channel in the code.

```
clientPayload : ∀d:SL . Int → ListC;d → (Int, d)
clientPayload init c =
    if init == 0 then
        (0, select Stop c)
    else
        select Item c &
        send init &
        clientPayload [d] (init − 1)
```

Putting everything together, we choose arbitrary initialisation and default parameters (111 and 999) to obtain

```
clientFull :
    !Int;&{Ok: ListC;?Int, Denied: Skip} → (Int, Skip)
clientFull =
    clientWrapper [Int,Int,ListC,Skip] clientPayload 111 999
```

It is crucial that clientPayload is polymorphic in the continuation.

2.3. Tree-structured transmission

The next example fully exploits the freedom gained with context-free session types. We develop code for the type-safe serialisation and deserialisation of a binary tree. Such trees may be defined by the following algebraic datatype definition.

```
data Tree = Leaf | Node Tree Int Tree
```

To transmit such a tree faithfully requires a context-free structure as embodied by the following protocol.

```
type TreeChannel:SL = ⊕{
  Leaf: Skip,
  Node: TreeChannel ; !Int ; TreeChannel
 }
```

A TreeChannel either selects a Leaf and then nothing, or it selects a Node, sends the left subtree, followed by the root and then the right subtree. While the tail-recursive occurrence of TreeChannel (that for the right subtree) is acceptable in a regular session type, the first occurrence of TreeChannel (that for the left subtree) is not. In FreeST it is straightforward to implement a function to write a Tree on a TreeChannel.

```
write : ∀a:SL . Tree → TreeChannel;a → a                    1
write t c =                                                 2
  case t of {                                               3
    Leaf →                                                  4
      select Leaf c,                                        5
    Node t1 x t2 →                                          6
      select Node c &                                       7
      write [TreeChannel;a] t1 &                            8
      send x &                                              9
      write [a] t2                                          10
  }                                                         11
```

Both recursive invocations of write take the type of their continuation session as a parameter. The continuation of the first write consists of the TreeChannel generated by the second write and the enclosing continuation a. The second write occurs in tail position and its continuation corresponds with the enclosing continuation a.

It is similarly straightforward to read a Tree from a session of type **dualof** TreeChannel. Instead of reifying a tree, we show a function that computes the sum of all values in a tree directly from the serialised format on the channel.

```
treeSum : ∀a:SL . dualof TreeChannel; a → (Int, a)
treeSum c =
  match c with {
    Leaf c →
      (0, c),
    Node c →
      let (sl, c) = treeSum [dualof TreeChannel;a] c in
      let (x, c) = receive c in
      let (sr, c) = treeSum [a] c in
      (sl + x + sr, c)
  }
```

The examples of write and treeSum demonstrate the requirement for polymorphism when handling proper context-free sessions. The recursive invocations of write and treeSum require different instantiations of the session continuation TreeChannel ;a (**dualof** TreeChannel;a, respectively) and a, which are evidence for polymorphic recursion.

A closer look at the typing of write reveals further requirements on the type structure of $F^{\mu;}$, which features non-trivial identities beyond the ones imposed by equi-recursion (see Lemma 5). The initial type of c in line 2 is TreeChannel;a, which is a recursive type followed by a. Applying **select** Leaf to c (in line 5) requires unrolling the recursive type in TreeChannel; a to ⊕{Leaf: **Skip**, Node: TreeChannel;!**Int**;TreeChannel}; a, but the top-level type is still a sequence. Hence, we must allow choice to distribute over sequence as in c : ⊕{Leaf: **Skip**;a, Node: TreeChannel;!**Int**;TreeChannel;a}. With this argument type, the inferred type of **select** Leaf c is **Skip**;a, which is still different from the expected type a. As **Skip** has no effect, we consider it a left and right unit of the sequencing operator so that **Skip**;a is equivalent to a, which finally fits the expected type. Typing the Node branch of the code yields no further insights. With the same approach one can easily write type-safe code for serialising and deserialising JSON data.

3. Types

This section introduces the notion of types, type equivalence and session type duality.

3.1. Types and type formation

We rely on a couple of base sets: *type variables*, denoted by a, b, c, and *labels*, denoted by k, ℓ. The syntax of kinds and types is in Fig. 1. We identify three basic kinds, together with their multiplicity variants.

$$
\begin{array}{rrl}
\text{Multiplicity} & m & ::= \mathsf{un} \mid \mathsf{lin} \\
\text{Basic kinds} & \upsilon & ::= \mathbf{M} \mid \mathbf{S} \mid \mathbf{T} \\
\text{Kinds} & \kappa & ::= \upsilon^m \\
\text{Polarity} & \sharp & ::= ! \mid ? \\
\text{View} & \star & ::= \oplus \mid \& \\
\text{Types} & T & ::= \mathsf{Skip} \mid \sharp T \mid \star\{\ell\colon T_\ell\}_{\ell\in L} \mid T;T \\
 & & \quad ()_m \mid T \to_m T \mid \{\ell\colon T_\ell\}_{\ell\in L} \mid \langle \ell\colon T_\ell\rangle_{\ell\in L} \\
 & & \quad \forall a\colon \kappa\,.\,T \mid \mu a\colon \kappa.T \mid a \\
\text{Kinding contexts} & \Delta & ::= \varepsilon \mid \Delta, a\colon \kappa
\end{array}
$$

Fig. 1. Syntax of kinds, type and kinding contexts.

Type T is terminated $\boxed{\vdash T\ \checkmark}$

$$
\textsc{Te-Skip}\ \frac{}{\vdash \mathsf{Skip}\ \checkmark} \qquad \textsc{Te-Seq}\ \frac{\vdash T\ \checkmark \qquad \vdash U\ \checkmark}{\vdash T;U\ \checkmark} \qquad \textsc{Te-Rec}\ \frac{\vdash T\ \checkmark}{\vdash \mu a\colon \mathbf{s}^m.T\ \checkmark}
$$

Fig. 2. The is-terminated predicate.

$\mathbf{M}^{\mathsf{un}}, \mathbf{M}^{\mathsf{lin}}$	for types that can be exchanged on channels
$\mathbf{S}^{\mathsf{un}}, \mathbf{S}^{\mathsf{lin}}$	for session types
$\mathbf{T}^{\mathsf{un}}, \mathbf{T}^{\mathsf{lin}}$	for arbitrary types

Multiplicities describe the number of times a value can be used: exactly once for lin and zero or more times for un.

Session types include the terminated type Skip, output messages $!T$ and input messages $?T$, internal labelled choices $\oplus\{\ell\colon T_\ell\}_{\ell\in L}$ and external choices $\&\{\ell\colon T_\ell\}_{\ell\in L}$, and the sequential composition $T;U$.

Functional types include the unrestricted and linear units, $()_{\mathsf{un}}$ and $()_{\mathsf{lin}}$, unrestricted and linear functions, $T \to_{\mathsf{un}} U$ and $T \to_{\mathsf{lin}} U$, records $\{\ell\colon T_\ell\}_{\ell\in L}$, and variants $\langle \ell\colon T_\ell\rangle_{\ell\in L}$. The unit types are representative of all base types including boolean and integer used in examples. The linear multiplicity, lin, assigned to basic types enables to specify linear assets to be exchanged on channels (e.g., tokens). The multiplicity in the function type constrains the number of times a function can be used: exactly once for $\to_{\mathsf{lin}}$ case and zero or more times for $\to_{\mathsf{un}}$. A function capturing in its body a free linear value must itself be linear. Functions receiving linear values may still be reused (hence un) if they contain no free linear variables. Unrestricted functions are abbreviated in examples to $T \to U$. The pair type $T \times U$ used in examples abbreviates the record $\{\mathsf{fst}\colon T, \mathsf{snd}\colon U\}$. Recursive types $\mu a\colon \kappa.T$ are both session and functional. Type variable a denotes a type variable in a recursive type or a polymorphic variable in a universal type.

We say that a *change of bound variables* in T is the replacement of a part $\mu a\colon \kappa.U$ of T by $\mu b\colon \kappa.U'$ where b does not occur (at all) in U and where U' is obtained from U by replacing all free occurrences of a by b. We say that two types are α-equivalent if one can be obtained from the other by a series of changes of bound variables. This paper adopts the *α-identification convention* whereby processes, terms and types are always identified up to α-equivalence.

Terminated types Type termination in Fig. 2 applies to session types alone. Intuitively a type is terminated if it does not exhibit a communication action. Terminated types are composed of Skip, sequential composition and recursion. We say that type T is *terminated* when judgement $\vdash T\ \checkmark$ holds.

Contractive types Contractivity is adapted from the standard "no subterms of the form $\mu a\colon \kappa.\mu a_1\colon \kappa_1 \ldots \mu a_n\colon \kappa_n.a$" to take into account the nature of the semi-colon operator as the neutral for sequential composition. Following Harper [37], *contractivity on a given type variable* is captured by judgement $\mathcal{A} \vdash a\,.\,T$, read as "type T is contractive on type variable a over a set of polymorphic variables $\mathcal{A}$". The rules are in Fig. 3. The set of type variables $\mathcal{A}$ is expected to contain the polymorphic variables in scope at the time of the call to the contractivity judgement (see Fig. 5, rule K-Rec).

Contractivity is defined with respect to a given type variable. For example, $\mu b\colon \mathbf{s}^{\mathsf{lin}}.b;a$ is contractive on type variable a (under the empty set of variables) even if $b;a$ is not contractive on b. Central to the definition of contractivity is C-Var rule. The contractivity of $b;a$ on a depends on b not being a polymorphic variable: when b is a recursion variable we have $\vdash a\,.\,b;a$, otherwise, when b is a polymorphic variable, we have $b \nvdash a\,.\,b;a$. Given that polymorphic variables may be replaced by arbitrary types, if b turns out to be replaced by Skip, we are left with $\mathsf{Skip};a$ which, by rule C-Seq2, is not contractive on a, because $\emptyset \nvdash a\,.\,a$.

Type T is contractive on type variable a $\boxed{\mathcal{A} \vdash a \,.\, T}$

$$\text{C-Seq1}\ \frac{\vdash T \checkmark \qquad \mathcal{A} \vdash a \,.\, U}{\mathcal{A} \vdash a \,.\, (T; U)} \qquad \text{C-Seq2}\ \frac{\nvdash T \checkmark \qquad \mathcal{A} \vdash a \,.\, T}{\mathcal{A} \vdash a \,.\, (T; U)} \qquad \text{C-TAbs}\ \frac{\mathcal{A} \vdash a \,.\, T}{\mathcal{A} \vdash a \,.\, (\forall b \colon \kappa \,.\, T)}$$

$$\text{C-Rec}\ \frac{\mathcal{A} \vdash a \,.\, T}{\mathcal{A} \vdash a \,.\, (\mu b \colon \kappa \,.\, T)} \qquad \text{C-Var}\ \frac{b \notin \mathcal{A}, a}{\mathcal{A} \vdash a \,.\, b} \qquad \text{C-Other}\ \frac{T = \mathsf{Skip}, \sharp U, \star\{\ell \colon U_\ell\}, ()_m, U \to_m V, \{\ell \colon U_\ell\}, \langle \ell \colon U_\ell \rangle}{\mathcal{A} \vdash a \,.\, T}$$

Fig. 3. Contractivity.

Subkinding $\boxed{m <: m}$ $\boxed{\nu <: \nu}$ $\boxed{\kappa <: \kappa}$

$$\mathsf{un} <: \mathsf{lin} \qquad \mathbf{M} <: \mathbf{T} \quad \mathbf{S} <: \mathbf{T} \qquad \frac{\upsilon_1 <: \upsilon_2 \qquad m_1 <: m_2}{\upsilon_1^{m_1} <: \upsilon_2^{m_2}}$$

Fig. 4. Subkinding.

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

$$\text{K-Type}\ \frac{\mathrm{dom}(\Delta) \mid \Delta \vdash T : \kappa}{\Delta \vdash T : \kappa}$$

Type formation (inner) $\boxed{\mathcal{A} \mid \Delta \vdash T : \kappa}$

$$\text{K-Skip}\ \mathcal{A} \mid \Delta \vdash \mathsf{Skip} : \mathbf{S}^{\mathsf{un}} \qquad \text{K-Msg}\ \frac{\mathcal{A} \mid \Delta \vdash T : \mathbf{M}^{\mathsf{lin}}}{\mathcal{A} \mid \Delta \vdash \sharp T : \mathbf{S}^{\mathsf{lin}}} \qquad \text{K-Ch}\ \frac{\mathcal{A} \mid \Delta \vdash T_\ell : \mathbf{S}^{\mathsf{lin}} \quad (\forall \ell \in L)}{\mathcal{A} \mid \Delta \vdash \star\{\ell \colon T_\ell\}_{\ell \in L} : \mathbf{S}^{\mathsf{lin}}}$$

$$\text{K-Seq}\ \frac{\mathcal{A} \mid \Delta \vdash T : \mathbf{S}^m \qquad \mathcal{A} \mid \Delta \vdash U : \mathbf{S}^m}{\mathcal{A} \mid \Delta \vdash (T; U) : \mathbf{S}^m} \qquad \text{K-Unit}\ \mathcal{A} \mid \Delta \vdash ()_m : \mathbf{M}^m$$

$$\text{K-Arrow}\ \frac{\mathcal{A} \mid \Delta \vdash T : \kappa_1 \qquad \mathcal{A} \mid \Delta \vdash U : \kappa_2}{\mathcal{A} \mid \Delta \vdash T \to_m U : \mathbf{T}^m} \qquad \text{K-Record}\ \frac{\mathcal{A} \mid \Delta \vdash T_\ell : \mathbf{T}^m \quad (\forall \ell \in L)}{\mathcal{A} \mid \Delta \vdash \{\ell \colon T_\ell\}_{\ell \in L} : \mathbf{T}^m}$$

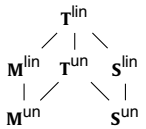
$$\text{K-Variant}\ \frac{\mathcal{A} \mid \Delta \vdash T_\ell : \mathbf{T}^m \quad (\forall \ell \in L)}{\mathcal{A} \mid \Delta \vdash \langle \ell \colon T_\ell \rangle_{\ell \in L} : \mathbf{T}^m} \qquad \text{K-TAbs}\ \frac{\mathcal{A}, a \mid \Delta, a \colon \kappa \vdash T : \mathbf{T}^m}{\mathcal{A} \mid \Delta \vdash \forall a \colon \kappa \,.\, T : \mathbf{T}^m}$$

$$\text{K-Rec}\ \frac{\mathcal{A} \vdash a \,.\, T \qquad \mathcal{A} \mid \Delta, a \colon \kappa \vdash T : \kappa}{\mathcal{A} \mid \Delta \vdash \mu a \colon \kappa \,.\, T : \kappa} \qquad \text{K-Var}\ \frac{a \colon \kappa \in \Delta}{\mathcal{A} \mid \Delta \vdash a : \kappa} \qquad \text{K-Sub}\ \frac{\mathcal{A} \mid \Delta \vdash T : \kappa \qquad \kappa <: \kappa'}{\mathcal{A} \mid \Delta \vdash T : \kappa'}$$

Fig. 5. Type formation.

The set of polymorphic type variables in the judgement allows for the contractive predicate to be closed under substitution. We note, however, that the same problem does not arise when a polymorphic type occurs in the body of a μ-type (e.g., $\mu a \colon \kappa \,.\, \forall b \colon \kappa' \,.\, T$). In this case, the corresponding polymorphic variable is not added to the context, as evidenced by the rule C-TAbs. Contractivity is a concept that concerns the behaviour of $\mathbf{S}$-kinded types and a polymorphic type is of kind $\mathbf{T}$ (see Fig. 5, rule K-TAbs). Would b be added to the context in the C-TAbsrule, type $\mu a \colon \mathbf{T}^{\mathsf{un}} \,.\, \forall b \colon \kappa \,.\, b$ would not be well-formed, without any apparent reason since the body of the μ-type has kind $\mathbf{T}$.

Subkinding Given the intended meaning for un and lin multiplicities it should be clear that an unrestricted type can be used where a linear type is sought. On a similar vein, a value that can be used in a message ($\mathbf{M}^{\mathsf{un}}$ or $\mathbf{M}^{\mathsf{lin}}$) can be used where an arbitrary type ($\mathbf{T}^{\mathsf{lin}}$) is required, and similarly for a value of a session type. This gives rise to a notion of subkinding $\kappa <: \kappa'$ governed by the rules in Fig. 4 and depicted by the partial order at the right.



Type formation Equipped with the notions of contractivity and subkinding we may address type formation. Type formation is captured by judgement $\Delta \vdash T : \kappa$ stating that type T has kind κ under kinding context Δ. The rules are in Fig. 5. The term language defined in section 4 may talk about types containing free polymorphic variables but not free recursion variables. The latter are collected in kinding context Δ. During type formation new type variables (both polymorphic and recursion) must be moved to the context. We have seen how important it is to be able to distinguish the two sorts of type variables. Therefore, judgement $\Delta \vdash T : \kappa$ for (external) type formation considers all variables in Δ to be polymorphic. The rules for inner type formation, judgement $\mathcal{A} \mid \Delta \vdash T : \kappa$, are then able to distinguish polymorphic from recursion variables: contrast rule K-TAbsagainst K-Rec. They both move the bind $a : \kappa$ to context Δ, but only the former rule moves a to $\mathcal{A}$. This setup allows us not to impose a syntactic distinction of the two sorts of type variables, important for programming languages.

The rules classify types into session types ($\mathbf{S}^{\mathsf{un}}, \mathbf{S}^{\mathsf{lin}}$), message types ($\mathbf{M}^{\mathsf{un}}, \mathbf{M}^{\mathsf{lin}}$) and arbitrary ($\mathbf{T}^{\mathsf{un}}, \mathbf{T}^{\mathsf{lin}}$) types. Rule K-Arrowgives kind $\mathbf{T}^m$ to an arrow type $T \to_m U$ regardless of the kinds for types T and U. This decouples the number of times a function can be used (governed by m) from the number of times the argument (and the result) can be used [44]. For simplicity, rule K-TAbsrestricts polymorphism to functional types. The only $\mathbf{S}^{\mathsf{un}}$ types are those made of Skip, such as Skip itself and Skip; Skip (that is, terminated types); all other session types are linear. Only base types can be transmitted in messages (represented by kind $\mathbf{M}^{\mathsf{lin}}$). Recursive types are required to be contractive (rule K-Recand Fig. 3).

Rule K-Subincorporates subkinding (Fig. 4) into type formation. Using K-Sub, any type is of kind $\mathbf{T}^{\mathsf{lin}}$ and any session type is of kind $\mathbf{S}^{\mathsf{lin}}$ (including Skip), and so, in a sense, all types may be viewed as linear. In the prose we often refer to linear types to mean those types whose only kind is υ^{lin}, that is, types that cannot be classified as unrestricted.

Continuing with the examples in the discussion of type termination and contractivity, a simple derivation concludes that $\vdash \mu a : \mathbf{S}^{\mathsf{lin}}.(!()_m; \mu b : \mathbf{S}^{\mathsf{lin}}.a; b) : \mathbf{S}^{\mathsf{lin}}$. On the other hand, a polymorphic type that repeatedly performs some protocol a, namely $\forall a : \mathbf{S}^{\mathsf{lin}} . \mu b : \mathbf{S}^{\mathsf{lin}}.a; b$ is not well formed, because $a \not\vdash b \,.\, a; b$. However the type of polymorphic streams as seen from the consumer side, $\forall a : \mathbf{M}^{\mathsf{un}} . \mu b : \mathbf{S}^{\mathsf{lin}}.?a; b$, is well formed.

The reader may wonder why we explicitly annotate function types $T \to_m U$ with a multiplicity mark m, but not record or variant types. The key difference is that the linearity of a record or variant type is completely determined by that of its components (a record with a linear component must be linear). This is not the case with function types where the multiplicity of the type is independent of those of its components (an unrestricted function may accept and/or return linear values). Rather, the multiplicity of the function type depends on those of the values captured in the function closure whose types are not apparent in the type. Hence, annotating record/variant types is redundant (and may even lead to inconsistencies), whereas annotating function types is unavoidable.

3.2. Bindings and substitution

The binding occurrences for type variables a are types $\mu a : \kappa.T$ and $\forall a : \kappa \,.\, T$. The set $\mathrm{free}(T)$ of free type variables in type T is defined accordingly, and so is the *capture avoiding substitution* of a type variable a by a type T in a type U, denoted by $U[T/a]$. Type formation may be subject to weakening and strengthening on non free type variables. The following lemmas show that we can discard and insert variables into the context (under some assumptions), which turns out to be paramount to verify that substitution preserves the good properties of types (Lemma 3).

Lemma 1 *(Type strengthening). Let* $a \notin \mathrm{free}(T)$.

1. *If* $\mathcal{A}, a \vdash b \,.\, T$, *then* $\mathcal{A} \vdash b \,.\, T$.
2. *If* $\mathcal{A} \mid \Delta, a : \kappa' \vdash T : \kappa$, *then* $\mathcal{A} \mid \Delta \vdash T : \kappa$.
3. *If* $\mathcal{A}, a \mid \Delta, a : \kappa' \vdash T : \kappa$, *then* $\mathcal{A} \mid \Delta \vdash T : \kappa$.

Proof. By rule induction on the hypothesis. We sketch one illustrative case for item 3: rule K-Recwith $T = \mu b : \kappa.U$. Given the α-identification convention, we assume that $a \neq b$. The premises of rule K-Recare $\mathcal{A}, a \vdash b \,.\, U$ and $\mathcal{A}, a \mid \Delta, a : \kappa', b : \kappa \vdash U : \kappa$. Since $a \notin \mathrm{free}(\mu b : \kappa.U)$ then $a \notin \mathrm{free}(U)$ and so, by induction, we obtain $\mathcal{A} \mid \Delta, b : \kappa \vdash U : \kappa$. Item 1 gives $\mathcal{A} \vdash b \,.\, U$. The result follows from rule K-Rec. □

Lemma 2 *(Type weakening).*

1. *If* $\mathcal{A} \vdash b \,.\, T$, *then* $\mathcal{A}, a \vdash b \,.\, T$.
2. *If* $\mathcal{A} \mid \Delta \vdash T : \kappa$, *then* $\mathcal{A} \mid \Delta, a : \kappa \vdash T : \kappa$ *and* $\mathcal{A}, a \mid \Delta, a : \kappa \vdash T : \kappa$.

Proof. By straightforward rule induction on the first hypothesis. □

Lemma 3 shows that substitution preserves termination, contractivity and type formation, fundamental properties to guarantee the proper behaviour of contexts when subject to substitutions, as we elaborate in section 4.

Lemma 3 *(Type substitution). Suppose that* $\mathcal{A} \mid \Delta \vdash U : \kappa$.

Labels $\quad \lambda ::= \sharp T \mid \star\ell \mid a$

Labelled transition system $\quad \boxed{T \xrightarrow{\lambda} T}$

$$\text{L-Var}\ \frac{}{a \xrightarrow{a} \mathsf{Skip}} \qquad \text{L-Msg}\ \frac{}{\sharp T \xrightarrow{\sharp T} \mathsf{Skip}} \qquad \text{L-Ch}\ \frac{k \in L}{\star\{\ell\colon T_\ell\}_{\ell\in L} \xrightarrow{\star k} T_k}$$

$$\text{L-Seq1}\ \frac{T \xrightarrow{\lambda} T'}{T;U \xrightarrow{\lambda} T';U} \qquad \text{L-Seq2}\ \frac{\vdash T\ \checkmark \qquad U \xrightarrow{\lambda} U'}{T;U \xrightarrow{\lambda} U'} \qquad \text{L-Rec}\ \frac{T[\mu a\colon \kappa.T/a] \xrightarrow{\lambda} U}{\mu a\colon \kappa.T \xrightarrow{\lambda} U}$$

Fig. 6. Labelled transition system.

1. *If* $\mathcal{A}, a \mid \Delta, a\colon \kappa \vdash T : \kappa'$ *and* $\vdash T\ \checkmark$, *then* $\vdash T[U/a]\ \checkmark$.
2. *If* $\mathcal{A}, a \mid \Delta, a\colon \kappa, b\colon \kappa' \vdash T : \kappa''$ *and* $\mathcal{A}, a \vdash b\ .\ T$ *and* $b \notin \mathcal{A} \cup \mathsf{free}(U)$, *then* $\mathcal{A} \vdash b\ .\ T[U/a]$.
3. *If* $\mathcal{A}, a \mid \Delta, a\colon \kappa \vdash T : \kappa'$, *then* $\mathcal{A} \mid \Delta \vdash T[U/a] : \kappa'$.

Proof. Item 1. By rule induction on $\vdash T\ \checkmark$.

Item 2. By rule induction on $\mathcal{A}, a \vdash b\ .\ T$, using item 1. All cases follow by induction hypothesis except case for rule C-Var, where we have $T = c$ with $c \neq a, b$; the result follows from the hypothesis $\mathcal{A}, a \vdash b\ .\ c$.

Item 3. By rule induction on $\mathcal{A}, a \mid \Delta, a\colon \kappa \vdash T : \kappa'$. For the case K-Recwe have $T = \mu b\colon \kappa''.V$. The premises to the rule are $\mathcal{A}, a \vdash b\ .\ V$ and $\mathcal{A}, a \mid \Delta, a\colon \kappa, b\colon \kappa'' \vdash V : \kappa'$. Induction on the second premise gives $\mathcal{A} \mid \Delta, b\colon \kappa'' \vdash V[U/a] : \kappa'$. Item 2 gives $\mathcal{A} \vdash b\ .\ V[U/a]$. Rule K-Recgives $\mathcal{A} \mid \Delta \vdash \mu b\colon \kappa''.V[U/a] : \kappa'$. Conclude with the definition of substitution. For the case K-Varwith $T = b \neq a$, we have $a \notin \mathsf{free}(b)$. The result follows from hypothesis $\mathcal{A}, a \mid \Delta, a\colon \kappa \vdash b : \kappa'$ and strengthening (Lemma 1). For K-Varwith $T = a$ we have $k = k'$. The result follows from the hypothesis $\mathcal{A} \mid \Delta \vdash U : \kappa$. □

The following lemma is used in agreement for type equivalence (Lemma 6). It should hold for all kinds, but we only need it for non session types.

Lemma 4 *(Inversion of substitution). If* $\mathcal{A} \mid \Delta \vdash U[T/a] : \kappa$ *and* $a \notin \mathsf{free}(T)$ *and* $\kappa \neq \mathbf{s}^m$, *then* $\mathcal{A} \mid \Delta, a\colon \kappa \vdash U : \kappa$.

Proof. By rule induction on the hypothesis. We sketch a few cases.

Case K-Var with $U = b \neq a$. The premise is $b\colon \kappa \in \Delta$; rule K-Var gives $\mathcal{A} \mid \Delta \vdash b : \kappa$ and weakening (Lemma 2) yields the result.

Case K-Varwith $U = a$, conclude with rule K-Var.

Case K-TAbs. We have $U = \forall b\colon \kappa'\,.\,U'$ and $U[T/a] = \forall b\colon \kappa'\,.\,(U'[T/a])$ and $\kappa \neq \mathbf{s}^m$. The premise of the rule is $\mathcal{A}, b \mid \Delta, b\colon \kappa' \vdash U'[T/a] : \kappa$. Induction gives $\mathcal{A}, b \mid \Delta, b\colon \kappa', a\colon \kappa \vdash U' : \kappa$ and rule K-TAbs gives the result. □

3.3. Type equivalence

Type equivalence for context-free session types is based on bisimulation. In contrast, the same notion for regular types is a standard equi-recursive definition [49].

Session type bisimilarity The labelled transition system for session types is in Fig. 6. The labels we consider are $!T$ and $?T$ for output and input messages of type T, $\oplus\ell$ and $\&\ell$ for internal and external choices on label ℓ, and a for transition on a *polymorphic* variable. There are two rules for sequential composition $T;U$ depending on T being terminated or not.

A bisimulation is defined in the usual way from the labelled transition system [55]. We say that a type relation $\mathcal{R}$ is a *bisimulation* if for T and U whenever $T\mathcal{R}U$, for all λ we have:

- for each T' with $T \xrightarrow{\lambda} T'$, there is U' such that $U \xrightarrow{\lambda} U'$ and $T'\mathcal{R}U'$, and
- for each U' with $U \xrightarrow{\lambda} U'$, there is T' such that $T \xrightarrow{\lambda} T'$ and $T'\mathcal{R}U'$.

We say that two types are bisimilar, written $T \simeq_s U$, if there is a bisimulation $\mathcal{R}$ with $T\mathcal{R}U$. Below we identify some laws of session bisimilarity.

- Skip is left and right neutral for sequential composition; sequential composition is associative; choice right-distributes over sequential composition;
- μ-bindings for variables not free in the body can be eliminated; unfold preserves bisimilarity.

Type equivalence $\boxed{\Theta \mid \Delta \vdash T \simeq T : \kappa}$

$$\text{Q-ST}\ \frac{\Delta \vdash T : \mathbf{S}^m \qquad \Delta \vdash U : \mathbf{S}^m \qquad T \simeq_s U}{\Theta \mid \Delta \vdash T \simeq U : \mathbf{S}^m} \qquad \text{Q-Unit}\ \frac{}{\Theta \mid \Delta \vdash ()_m \simeq ()_m : \mathbf{M}^m}$$

$$\text{Q-Arrow}\ \frac{\Theta \mid \Delta \vdash T \simeq T' : \kappa_1 \qquad \Theta \mid \Delta \vdash U \simeq U' : \kappa_2}{\Theta \mid \Delta \vdash T \to_m U \simeq T' \to_m U' : \mathbf{T}^m} \qquad \text{Q-Var}\ \frac{\kappa \neq \mathbf{S}^m \quad a : \kappa \in \Delta}{\Theta \mid \Delta \vdash a \simeq a : \kappa}$$

$$\text{Q-Record}\ \frac{\Theta \mid \Delta \vdash T_\ell \simeq U_\ell : \mathbf{T}^m \qquad (\forall \ell \in L)}{\Theta \mid \Delta \vdash \{\ell : T_\ell\}_{\ell \in L} \simeq \{\ell : U_\ell\}_{\ell \in L} : \mathbf{T}^m} \qquad \text{Q-Variant}\ \frac{\Theta \mid \Delta \vdash T_\ell \simeq U_\ell : \mathbf{T}^m \qquad (\forall \ell \in L)}{\Theta \mid \Delta \vdash \langle \ell : T_\ell \rangle_{\ell \in L} \simeq \langle \ell : U_\ell \rangle_{\ell \in L} : \mathbf{T}^m}$$

$$\text{Q-TAbs}\ \frac{\Theta \mid \Delta, a : \kappa \vdash T \simeq U : \mathbf{T}^m}{\Theta \mid \Delta \vdash \forall a : \kappa \,.\, T \simeq \forall a : \kappa \,.\, U : \mathbf{T}^m} \qquad \text{Q-Fix}\ \frac{(\Delta, T, U, \kappa) \in \Theta \quad \Delta \vdash T : \kappa \quad \Delta \vdash U : \kappa}{\Theta \mid \Delta \vdash T \simeq U : \kappa}$$

$$\text{Q-RecL}\ \frac{\kappa \neq \mathbf{S}^m \qquad (\Delta', \mu a : \kappa.T, U, \kappa') \notin \Theta \qquad \mathrm{dom}(\Delta) \vdash a \,.\, T \qquad \Theta, (\Delta, \mu a : \kappa.T, U, \kappa) \mid \Delta \vdash T[\mu a : \kappa.T/a] \simeq U : \kappa}{\Theta \mid \Delta \vdash \mu a : \kappa.T \simeq U : \kappa}$$

$$\text{Q-RecR}\ \frac{T \text{ not } \mu \qquad \kappa \neq \mathbf{S}^m \qquad (\Delta', T, \mu a : \kappa.U, \kappa') \notin \Theta \qquad \mathrm{dom}(\Delta) \vdash a \,.\, U \qquad \Theta, (\Delta, T, \mu a : \kappa.U, \kappa) \mid \Delta \vdash T \simeq U[\mu a : \kappa.U/a] : \kappa}{\Theta \mid \Delta \vdash T \simeq \mu a : \kappa.U : \kappa}$$

Fig. 7. Type equivalence.

Lemma 5 *(Properties of type bisimilarity).*

1. $\mathsf{Skip}; T \simeq_s T$; $\mathsf{Skip} \simeq_s T$
2. $(T_1; T_2); T_3 \simeq_s T_1; (T_2; T_3)$
3. $\star\{\ell : T_\ell\}_{\ell \in L}; T \simeq_s \star\{\ell : T_\ell; T\}_{\ell \in L}$
4. $\mu a : \kappa.T \simeq_s T$, *if* $a \notin \mathrm{free}(T)$
5. $\mu a : \kappa.T \simeq_s T[\mu a : \kappa.T/a]$

Proof. Each law is proved by exhibiting a suitable bisimulation. We show a couple of examples. For item 3 we use the relation $\mathcal{R}$ that contains the identity relation as well as all pairs of the form $(\star\{\ell : T_\ell\}_{\ell \in L}; T, \star\{\ell : T_\ell; T\}_{\ell \in L})$. For item 4, considering $\mathcal{R}$ to be a bisimulation that contains the identity relation, then the bisimulation we seek is $\mathcal{R} \cup \{(\mu a : \kappa.T, T)\}$. □

Type equivalence Fig. 7 introduces the type equivalence rules for arbitrary types. The rules use judgements of the form $\Theta \mid \Delta \vdash T \simeq U : \kappa$, where Θ is a list of kinded pairs of types (quadruples (Δ, T, U, κ) such that $\Delta \vdash T, U : \kappa$). Rule Q-STincorporates session type bisimilarity into type equivalence; in this case the two types are required to be session types. The rules for functional types (Q-Unit, Q-Arrow, Q-Record, Q-Variant, Q-TAbs and Q-Var) are the traditional congruence rules. Rule Q-Var is expected to be applied to polymorphic variables only, that is, neither session variables (hence the premise $\kappa \neq \mathbf{S}^m$) nor recursion variables (hence $a : \kappa \in \Delta$). Those for equi-recursion (Q-Fix, Q-RecL and Q-RecR) keep track of a set Θ containing the pairs of types visited so far, in which one of the components is recursive. This is essentially the Amadio-Cardelli system [5], brought to session types (in the context of subtyping) by Gay and Hole [29]. The algorithmic reading of the rules is known to be exponential [41,49]. Pierce proposes a polynomial variant that can be used in concrete implementations [49].

Lemma 6 *(Agreement for type equivalence). If* $\Theta \mid \Delta \vdash T \simeq U : \kappa$, *then* $\Delta \vdash T : \kappa$.

Proof. The proof is by rule induction on the hypothesis.

Case Q-ST. In this case $\kappa = \mathbf{S}^m$ and the premises state that $\Delta \vdash T : \mathbf{S}^m$.

Case Q-Unit. In this case, $T = ()_m$, $\kappa = \mathbf{M}^m$ and we have $\Delta \vdash T : \mathbf{M}^m$.

Case Q-Arrow, Q-Record, Q-Variant. For Q-Arrow, we know that T is of the form $T_1 \to_m T_2$, $\kappa = \mathbf{T}^m$ and the premises are $\Theta \mid \Delta \vdash T_1 \simeq U_1 : \kappa_1$ and $\Theta \mid \Delta \vdash T_2 \simeq U_2 : \kappa_2$, where $U = U_1 \to_m U_2$. By induction hypothesis we have $\Delta \vdash T_1 : \kappa_1$ and $\Delta \vdash T_2 : \kappa_2$. Applying K-Typeand K-Arrow, we conclude that $\Delta \vdash T_1 \to_m T_2 : \mathbf{T}^m$. The other cases are similar.

Case Q-TAbs. In this case T is of the form $\forall a : \kappa \,.\, T'$ and U is $\forall a : \kappa \,.\, U'$. The premise is $\Theta \mid \Delta, a : \kappa \vdash T' \simeq U' : \mathbf{T}^m$. By induction hypothesis, we have $\Delta, a : \kappa \vdash T' : \mathbf{T}^m$, i.e., $\mathrm{dom}(\Delta), a \mid \Delta, a : \kappa \vdash T' : \mathbf{T}^m$. Applying K-TAbswe conclude that $\mathrm{dom}(\Delta) \mid \Delta \vdash T : \mathbf{T}^m$, which means that $\Delta \vdash T : \mathbf{T}^m$.

Case Q-Var. Here, $T = U = a$ and $a : \kappa \in \Delta$. Using K-Varwe get $\Delta \vdash a : \kappa$.

Duality on polarities and views

$$\boxed{\overline{\sharp} = \sharp} \qquad \boxed{\overline{\star} = \star}$$

$$\overline{!} = ? \qquad \overline{?} = ! \qquad \overline{\oplus} = \& \qquad \overline{\&} = \oplus$$

The duality function on session types

$$\boxed{\overline{T} = T}$$

$$\overline{\mathsf{Skip}} = \mathsf{Skip} \qquad \overline{\sharp T} = \overline{\sharp}\, T \qquad \overline{\star\{\ell\colon T_\ell\}_{\ell\in L}} = \overline{\star}\{\ell\colon \overline{T}_\ell\}_{\ell\in L}$$

$$\overline{T; U} = \overline{T}; \overline{U} \qquad \overline{a} = a \qquad \overline{\mu a\colon \kappa.T} = \mu a\colon \kappa.\overline{T}$$

Fig. 8. The duality function on session types.

Case Q-Fix. The premises state that $\Delta \vdash T : \kappa$.

Case Q-RecL. In this case T is of the form $\mu a\colon \kappa.T'$. Premises include $\kappa \neq \mathbf{s}^m$ and $\mathrm{dom}\,(\Delta) \vdash a\,.\,T'$ and $\Theta, (\Delta, T, U, \kappa) \mid \Delta \vdash T'[T/a] \simeq U : \kappa$. By induction hypothesis, we know that $\Delta \vdash T'[T/a] : \kappa$. Applying K-Typegives $\mathrm{dom}\,\Delta \mid \Delta \vdash T'[T/a] : \kappa$. By Lemma 4, we have $\mathrm{dom}\,\Delta \mid \Delta, a\colon \kappa \vdash T' : \kappa$. Using K-Rec, we conclude that $\Delta \vdash T : \kappa$.

Case Q-RecR. Premises include $\Theta, (\Delta, T, U, \kappa) \mid \Delta \vdash T \simeq U'[U/a] : \kappa$, for $U = \mu a\colon \kappa.U'$. By induction hypothesis we conclude that $\Delta \vdash T : \kappa$. □

Lemma 7 *(Type equivalence). The relation $\Theta \mid \Delta \vdash T \simeq U : \kappa$ is reflexive, transitive, and symmetric.*

Proof. By rule induction. □

3.4. Session type duality

Duality is a notion central to session types. It allows "switching" the point of view from one end of a channel (say, the client side) to the other end (the server side). The rules for duality are in Fig. 8. The simplified formulation for recursion variables and recursive types is justified by the fact that the types we consider are first order (channels cannot be conveyed in messages), thus avoiding a complication known to arise in the presence of recursion [7,8,31]. The properties below state that duality is idempotent, preserves termination and contractivity, and that every session has a dual.

Lemma 8 *(Properties of duality).*

1. *If $\overline{T} = U$, then $\overline{U} = T$*
2. *If $\vdash T\ \checkmark$ and $\overline{T} = U$, then $\vdash U\ \checkmark$.*
3. *If $\mathcal{A} \vdash a\,.\,T$ and $\overline{T} = U$, then $\mathcal{A} \vdash a\,.\,U$.*
4. *If $\vdash T : \mathbf{s}^m$, then $\overline{T} = U$ and $\vdash U : \mathbf{s}^m$ for some U.*

Proof. By straightforward rule induction on the first hypothesis, in all cases. □

4. Expressions and processes, statics and dynamics

This section introduces the terms, typing and operational semantics of $F^{\mu;}$.

4.1. Expressions and processes

Fig. 9 introduces the syntax of values, expressions and processes. A value is either a variable (which may stand for a channel end), a constant, an expression abstraction, a type abstraction, a record whose fields are values, a variant injection of a value, a partial (type) application of the constants that denote the operations on channels. Notice that expression $\mathsf{fork}\,[T]$ is not a value because $\mathsf{fork}\,[T]\,e$ is evaluated before its body e becomes a value, see rule R-Forkin Fig. 15. The body of a type abstraction is restricted to a syntactic value as customary. This value restriction guarantees the sound interplay of call-by-value evaluation and polymorphism in the presence of effects like channel-based communication [70].

An *expression* e is a value, an application, a record, a record elimination by pattern matching (this is necessary because records may contain linear fields), a unit elimination, an injection in a variant type, a variant elimination, a type application, a channel creation, or matching on a tag received over a channel. Expression $\mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2$ used in examples abbreviates $\mathsf{let}\ \{\mathsf{fst}\colon x, \mathsf{snd}\colon y\} = e_1\ \mathsf{in}\ e_2$.

A *process* p is either an expression lifted to the process level, a parallel composition of two processes, or a channel restriction binding the two ends of a channel in a subsidiary process.

Context formation and context split Type formation (Fig. 5) is lifted pointwise to contexts in judgement $\Delta \vdash \Gamma : \kappa$ (see Fig. 10). Context split (also in Fig. 10) governs the distribution of linear bindings. If $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$, then the unrestricted bindings

$$
\begin{array}{rll}
\text{Constants} & c & ::= \ \mathsf{send} \mid \mathsf{receive} \mid \mathsf{fork} \mid ()_m \\
\text{Values} & v & ::= \ x \mid c \mid \lambda_m x\colon T.e \mid \Lambda a\colon \kappa.v \mid \{\ell{=}v_\ell\}_{\ell\in L} \mid \ell\, v \\
& & \quad \mid \mathsf{select}\,\ell \mid \mathsf{send}\,[T] \mid \mathsf{send}\,[T]\,v \mid \mathsf{send}\,[T]\,v\,[U] \\
& & \quad \mid \mathsf{receive}\,[T] \mid \mathsf{receive}\,[T]\,[U] \\
\text{Expressions} & e & ::= \ v \mid e\,e \mid \{\ell{=}e_\ell\}_{\ell\in L} \mid \mathsf{let}\,\{\ell{=}x_\ell\}_{\ell\in L} = e\,\mathsf{in}\,e \\
& & \quad \mid \mathsf{let}\,()_m = e\,\mathsf{in}\,e \mid \ell\,e \mid \mathsf{case}\,e\,\mathsf{of}\,\{\ell{\to}e_\ell\}_{\ell\in L} \\
& & \quad \mid e\,[T] \mid \mathsf{new}\,T \mid \mathsf{match}\,e\,\mathsf{with}\,\{\ell{\to}e_\ell\}_{\ell\in L} \\
\text{Processes} & p & ::= \ \langle e\rangle \mid p \mid p \mid (\nu xx)p \\
\text{Typing contexts} & \Gamma & ::= \ \varepsilon \mid \Gamma, x\colon T
\end{array}
$$

Fig. 9. The syntax of values, expressions, processes and typing contexts.

Context formation $\boxed{\Delta \vdash \Gamma : \kappa}$

$$\textsc{F-Empty}\ \frac{}{\Delta \vdash \varepsilon : \kappa} \qquad \textsc{F-Ext}\ \frac{\Delta \vdash \Gamma : \kappa \qquad \Delta \vdash T : \kappa}{\Delta \vdash \Gamma, x\colon T : \kappa}$$

Context split $\boxed{\Delta \vdash \Gamma = \Gamma \circ \Gamma}$

$$\textsc{S-Empty}\ \frac{}{\Delta \vdash \varepsilon = \varepsilon \circ \varepsilon} \qquad \textsc{S-Unr}\ \frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Delta \vdash T : \mathbf{T}^{\mathsf{un}}}{\Delta \vdash \Gamma, x\colon T = (\Gamma_1, x\colon T) \circ (\Gamma_2, x\colon T)}$$

$$\textsc{S-Left}\ \frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Delta \vdash T : \mathbf{T}^{\mathsf{lin}}}{\Delta \vdash \Gamma, x\colon T = (\Gamma_1, x\colon T) \circ \Gamma_2} \qquad \textsc{S-Right}\ \frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Delta \vdash T : \mathbf{T}^{\mathsf{lin}}}{\Delta \vdash \Gamma, x\colon T = \Gamma_1 \circ (\Gamma_2, x\colon T)}$$

Fig. 10. Context formation and context split.

of Γ are both in Γ_1 and Γ_2 whereas linear bindings are either in Γ_1 or Γ_2 but not in both (two rules). Splitting does not generate new bindings. For convenience, we write $\Gamma_1 \circ \Gamma_2$ in the conclusion of an inference rule instead of Γ with the additional premise $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$. We need a few technical lemmas about splittings.

Lemma 9 *(Agreement for context split). Let $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$. Then, $\Delta \vdash \Gamma : \kappa$ iff $\Delta \vdash \Gamma_1 : \kappa$ and $\Delta \vdash \Gamma_2 : \kappa$.*

Proof. By rule induction on the context split hypothesis. □

Given a well-formed context Γ, we distinguish the linear and unrestricted portions of Γ and denote them by $\mathcal{L}(\Gamma)$ and $\mathcal{U}(\Gamma)$, respectively [69].

Lemma 10 *(Properties of context split). Let $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$*

1. *$\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$*
2. *If $x\colon T \in \Gamma$ and $\Delta \vdash T : \mathbf{T}^{\mathsf{lin}}$, then either $x \in \mathrm{dom}(\Gamma_1)$ and $x \notin \mathrm{dom}(\Gamma_2)$, or $x \notin \mathrm{dom}(\Gamma_1)$ and $x \in \mathrm{dom}(\Gamma_2)$.*
3. *If $\Delta \vdash \Gamma_1 = \Gamma_{11} \circ \Gamma_{12}$, then there exists exactly one Γ' such that $\Delta \vdash \Gamma = \Gamma_{11} \circ \Gamma'$ and $\Delta \vdash \Gamma' = \Gamma_{12} \circ \Gamma_2$.*

Proof. Items 1 and 2. By rule induction on the context split hypothesis. Item 3. By rule induction on the context split for Γ_1. □

Lemma 11 *(Substitution and context split). If $\Delta, a : \kappa \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ and $\Delta \vdash U : \kappa$, then $\Gamma[U/a] = \Gamma_1[U/a] \circ \Gamma_2[U/a]$.*

Proof. By rule induction on the context split hypothesis. □

Expression and process formation Types for constants are in Fig. 11. Thanks to polymorphism, we can give types to most of the session operations. Previous work [32,58] includes specific typing rules for these primitives. Inspecting types (and kinds), we see that sending and receiving is restricted to $\mathbf{M}$-values. In both cases, the continuation is a linear session (of kind $\mathbf{S}^{\mathsf{lin}}$). The send and receive functions are unrestricted because they do not close over linear values. However, a partially

Types for constants $\boxed{\text{typeof}(c) = T}$

$$\text{typeof}(\text{send}) = \forall a\colon \mathbf{M}^{\text{lin}}.\, a \to_{\text{un}} \forall b\colon \mathbf{S}^{\text{lin}}.\, !a; b \to_{\text{lin}} b$$

$$\text{typeof}(\text{receive}) = \forall a\colon \mathbf{M}^{\text{lin}}.\, \forall b\colon \mathbf{S}^{\text{lin}}.\, ?a; b \to_{\text{un}} \{\text{fst}\colon a, \text{snd}\colon b\}$$

$$\text{typeof}(\text{fork}) = \forall a\colon \mathbf{T}^{\text{un}}.\, a \to_{\text{un}} ()_{\text{un}}$$

$$\text{typeof}(()_m) = ()_m$$

Fig. 11. Types for constants.

applied send must be linear for it captures a linear value (that of the value to be sent, of type $a\colon \mathbf{M}^{\text{lin}}$). The type of fork indicates that a new process may return any value of an unrestricted type, a value that is discarded by the operational semantics.

At a first glance, the use of linear polymorphic variables for the values exchanged in messages ($a\colon \mathbf{M}^{\text{lin}}$ in the types for send and receive) may seem restrictive, but the prescription of $\mathbf{M}^{\text{lin}}$ to a is just a convenience for checking type application. A send function, partially applied to an unrestricted value may still be used multiple times. By η-expanding the partial application, we obtain a thunk that can be used repeatedly. For example, function $\lambda_{\text{un}}\_\colon ()_{\text{un}}.\text{send}\,[\text{Int}]\ 5$ is a partially applied send of an unrestricted type: $()_{\text{un}} \to_{\text{un}} \forall b\colon \mathbf{S}^{\text{lin}}.\, !a; b \to_{\text{lin}} b$.

The remaining session operations retain their status as language primitives with specific typing rules. Expression new T creates a new channel of session type T and returns a pair of channel ends of types T and $\overline{T}$. Expression select k selects option k on a internal choice type. Expression match receives a label and branches on it.

Expression formation is defined by rules in Fig. 12. Constants are typed in T-Const according to Fig. 11 under the condition that the context is unrestricted (i.e., all linear values have been consumed). The type rule for variables, T-Var, insists that the unused part of the context must be unrestricted. In rule T-Abs, context formation under multiplicity m enforces that unrestricted abstractions do not capture linear values. Application T-App, kinded type abstraction T-TAbs, and type application T-TApp are all standard. Rule T-Apprule splits context in two, one part types the function, the other the argument. The introduction rule for records T-Record uses the notation $\circ\Gamma_\ell$ for iterated context split. The corresponding elimination rule T-RcdElim is standard. We cannot offer the dot notation to select single fields from records, as such projection might discard linear values. The elimination of $()_m$ is given by rule T-UnitElim. The rule T-Variant and T-Case are standard introduction and elimination rules for variants.

Process formation is defined in Fig. 13. Any expression of unrestricted type can be made into a process (P-Exp). Processes running in parallel split the resources among them (P-Par). Restriction introduces two channel ends of session type, which are dual to one another (P-New).

Reduction We consider processes up to structural congruence as defined in Fig. 14. The rules are standard [63]. Fig. 15 specifies the call-by-value, type-agnostic, reduction rules for expressions and processes. Expression reduction $e \to e$ is standard for call-by-value lambda calculus. Evaluation contexts E specify a left-to-right call-by-value reduction strategy. Thanks to the value restriction, we need not evaluate under type abstractions. Record components are evaluated from left to right as they appear in the program text.

Process reduction $p \to p$ includes standard rules for congruence (R-Cong), parallel execution (R-Par), and reduction under restriction (R-Bind). An expression process can perform an internal reduction step (R-Exp). Forking creates a new process from the argument (R-Fork). The operations on channels are reasonably standard for session type calculi. Creation of a new channel, R-New, ignores the type argument, wraps the process in a restriction, and returns the channel ends in a pair. Rule R-Com matches a send with a receive operation across threads. It transfers the value ignoring the type arguments. Rule R-Ch matches a select with a suitable match operation and dispatches according to the label k selected by the internal choice process.

4.2. Type safety

Before proving the typing preservation for congruence (Theorem 15), we need to establish some auxiliary results.

Lemma 12 *(Weakening).*

1. *If* $\Theta \mid \Delta \vdash T_1 \simeq T_2 : \kappa$, *then* $\Theta \mid \Delta, a : \kappa' \vdash T_1 \simeq T_2 : \kappa$.
2. *If* $\Delta \vdash \Gamma : \kappa$, *then* $\Delta, a : \kappa' \vdash \Gamma : \kappa$.
3. *If* $\Delta \vdash T : \mathbf{T}^{\text{lin}}$, *then* $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ *if and only if* $\Delta \vdash (\Gamma, x\colon T) = (\Gamma_1, x\colon T) \circ \Gamma_2$.
4. *If* $\Delta \mid \Gamma \vdash e : T$, *then* $\Delta, a : \kappa \mid \Gamma \vdash e : T$.
5. *If* $\Delta \mid \Gamma \vdash e : T$ *and* $\Delta \vdash U : \mathbf{T}^{\text{un}}$, *then* $\Delta \mid \Gamma, x : U \vdash e : T$.
6. *If* $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$, *then* $\Delta, a : \kappa \vdash \Gamma = \Gamma_1 \circ \Gamma_2$.
7. *If* $\Gamma \vdash p$ *and* $\Delta \vdash T : \mathbf{T}^{\text{un}}$, *then* $\Gamma, x\colon T \vdash p$.

Expression formation $\boxed{\Delta \mid \Gamma \vdash e : T}$

$$\textsc{T-Const}\ \frac{\Delta \vdash \Gamma : \mathbf{T}^{\mathsf{un}}}{\Delta \mid \Gamma \vdash c : \mathsf{typeof}(c)} \qquad \textsc{T-Var}\ \frac{\Delta \vdash \Gamma : \mathbf{T}^{\mathsf{un}} \quad \Delta \vdash T : \kappa}{\Delta \mid \Gamma, x\colon T \vdash x : T}$$

$$\textsc{T-Abs}\ \frac{\Delta \vdash \Gamma : \mathbf{T}^{m} \quad \Delta \mid \Gamma, x\colon T_1 \vdash e : T_2}{\Delta \mid \Gamma \vdash \lambda_m x\colon T_1.e : T_1 \to_m T_2} \qquad \textsc{T-App}\ \frac{\Delta \mid \Gamma_1 \vdash e_1 : T_1 \to_m T_2 \quad \Delta \mid \Gamma_2 \vdash e_2 : T_1}{\Delta \mid \Gamma_1 \circ \Gamma_2 \vdash e_1\, e_2 : T_2}$$

$$\textsc{T-TAbs}\ \frac{\Delta, a\colon \kappa \mid \Gamma \vdash v : T \quad a \notin \mathsf{free}(\Gamma)}{\Delta \mid \Gamma \vdash \Lambda a\colon \kappa.v : \forall a\colon \kappa\, .T} \qquad \textsc{T-TApp}\ \frac{\Delta \mid \Gamma \vdash e : \forall a\colon \kappa\, .U \quad \Delta \vdash T : \kappa}{\Delta \mid \Gamma \vdash e\,[T] : U[T/a]}$$

$$\textsc{T-Record}\ \frac{\Delta \mid \Gamma_\ell \vdash e_\ell : T_\ell \quad (\forall \ell \in L)}{\Delta \mid \circ\Gamma_\ell \vdash \{\ell{=}e_\ell\}_{\ell\in L} : \{\ell\colon T_\ell\}_{\ell\in L}}$$

$$\textsc{T-RcdElim}\ \frac{\Delta \mid \Gamma_1 \vdash e_1 : \{\ell\colon T_\ell\}_{\ell\in L} \quad \Delta \mid \Gamma_2, (x_\ell\colon T_\ell)_{\ell\in L} \vdash e_2 : U}{\Delta \mid \Gamma_1 \circ \Gamma_2 \vdash \mathsf{let}\,\{\ell{=}x_\ell\}_{\ell\in L} = e_1\,\mathsf{in}\,e_2 : U}$$

$$\textsc{T-UnitElim}\ \frac{\Delta \mid \Gamma_1 \vdash e_1 : ()_m \quad \Delta \mid \Gamma_2 \vdash e_2 : T}{\Delta \mid \Gamma_1 \circ \Gamma_2 \vdash \mathsf{let}\,()_m = e_1\,\mathsf{in}\,e_2 : T}$$

$$\textsc{T-Variant}\ \frac{\Delta \mid \Gamma \vdash e : T_k \quad \Delta \vdash T_\ell : \mathbf{T}^{\mathsf{lin}} \quad k \in L \quad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash k\,e : \langle \ell\colon T_\ell\rangle_{\ell\in L}}$$

$$\textsc{T-Case}\ \frac{\Delta \mid \Gamma_1 \vdash e : \langle \ell\colon T_\ell\rangle_{\ell\in L} \quad \Delta \mid \Gamma_2 \vdash e_\ell : T_\ell \to_{\mathsf{lin}} T \quad (\forall \ell \in L)}{\Delta \mid \Gamma_1 \circ \Gamma_2 \vdash \mathsf{case}\,e\,\mathsf{of}\,\{\ell{\to}e_\ell\}_{\ell\in L} : T}$$

$$\textsc{T-Sel}\ \frac{\Delta \vdash \Gamma : \mathbf{T}^{\mathsf{un}} \quad \Delta \vdash T_\ell : \mathbf{S}^{\mathsf{lin}} \quad k \in L \quad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash \mathsf{select}\,k : \oplus\{\ell\colon T_\ell\}_{\ell\in L} \to_m T_k}$$

$$\textsc{T-Match}\ \frac{\Delta \mid \Gamma_1 \vdash e : \&\{\ell\colon T_\ell\}_{\ell\in L} \quad \Delta \mid \Gamma_2 \vdash e_\ell : T_\ell \to_{\mathsf{lin}} T \quad (\forall \ell \in L)}{\Delta \mid \Gamma_1 \circ \Gamma_2 \vdash \mathsf{match}\,e\,\mathsf{with}\,\{\ell{\to}e_\ell\}_{\ell\in L} : T}$$

$$\textsc{T-New}\ \frac{\Delta \vdash \Gamma : \mathbf{T}^{\mathsf{un}} \quad \varepsilon \vdash T : \mathbf{S}^{\mathsf{lin}}}{\Delta \mid \Gamma \vdash \mathsf{new}\,T : \{\mathsf{fst}\colon T, \mathsf{snd}\colon \overline{T}\}} \qquad \textsc{T-Eq}\ \frac{\Delta \mid \Gamma \vdash e : T_1 \quad \varepsilon \mid \Delta \vdash T_1 \simeq T_2 : \kappa}{\Delta \mid \Gamma \vdash e : T_2}$$

Fig. 12. Expression formation.

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

$$\textsc{P-Exp}\ \frac{\varepsilon \mid \Gamma \vdash e : T \quad \vdash T : \mathbf{T}^{\mathsf{un}}}{\Gamma \vdash \langle e\rangle} \qquad \textsc{P-Par}\ \frac{\Gamma_1 \vdash p_1 \quad \Gamma_2 \vdash p_2}{\Gamma_1 \circ \Gamma_2 \vdash p_1 \mid p_2}$$

$$\textsc{P-New}\ \frac{\Gamma, x\colon T, y\colon \overline{T} \vdash p \quad \vdash T : \mathbf{S}^{\mathsf{lin}}}{\Gamma \vdash (\nu xy)p}$$

Fig. 13. Process formation.

Structural congruence $\boxed{p \equiv q}$

$$p \equiv \langle()_{\mathsf{un}}\rangle \mid p \qquad p \mid q \equiv q \mid p \qquad (p \mid q) \mid r \equiv p \mid (q \mid r)$$

$$((\nu xy)p) \mid q \equiv (\nu xy)(p \mid q) \qquad (\nu xy)\langle()_{\mathsf{un}}\rangle \equiv \langle()_{\mathsf{un}}\rangle \qquad (\nu xy)p \equiv (\nu yx)p$$

$$(\nu wx)(\nu yz)p \equiv (\nu yz)(\nu wx)p$$

Rules for congruence and equivalence omitted.

Fig. 14. Structural congruence.

Evaluation contexts

$$E ::= [] \mid Ee \mid vE \mid \ell\, E \mid E\,[T] \mid \mathsf{let}\,\{\ell{=}x_\ell\}_{\ell\in L} = E\,\mathsf{in}\,e$$
$$\mid \mathsf{let}\,()_m = E\,\mathsf{in}\,e \mid \{\ell_1 = v_1, \ldots, \ell_i = E, \ldots, \ell_n = e_n\}$$
$$\mid \mathsf{case}\,E\,\mathsf{of}\,\{\ell{\to}e_\ell\}_{\ell\in L} \mid \mathsf{match}\,E\,\mathsf{with}\,\{\ell{\to}e_\ell\}_{\ell\in L}$$

Expression reduction $\boxed{e \to e}$

E-App
$$(\lambda_m x\colon \_.e)v \to e[v/x]$$

E-RcdElim
$$\mathsf{let}\,\{\ell{=}x_\ell\}_{\ell\in L} = \{\ell{=}v_\ell\}_{\ell\in L}\,\mathsf{in}\,e \to e[v_\ell/x_\ell]_{\ell\in L}$$

E-UnitElim
$$\mathsf{let}\,()_m = ()_m\,\mathsf{in}\,e \to e$$

E-TApp
$$(\Lambda a\colon \_.v)\,[T] \to v[T/a]$$

E-Case
$$\frac{k \in L}{\mathsf{case}\,k\,v\,\mathsf{of}\,\{\ell{\to}e_\ell\}_{\ell\in L} \to e_k v}$$

E-Ctx
$$\frac{e \to e'}{E[e] \to E[e']}$$

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

R-Exp
$$\frac{e \to e'}{\langle e\rangle \to \langle e'\rangle}$$

R-Fork
$$\langle E[\mathsf{fork}\,[\_]\,e]\rangle \to \langle E[()_{\mathsf{un}}]\rangle \mid \langle e\rangle$$

R-New
$$\langle E[\mathsf{new}\,\_]\rangle \to (\nu xy)\langle E[(x, y)]\rangle$$

R-Com
$$(\nu xy)(\langle E_1[\mathsf{send}[\_]v[\_]x]\rangle \mid \langle E_2[\mathsf{receive}[\_][\_]y]\rangle) \to (\nu xy)(\langle E_1[x]\rangle \mid \langle E_2[(v, y)]\rangle)$$

R-Ch
$$\frac{k \in L}{(\nu xy)(\langle E_1[\mathsf{select}\,k\,x]\rangle \mid \langle E_2[\mathsf{match}\,y\,\mathsf{with}\,\{\ell{\to}e_\ell\}_{\ell\in L}]\rangle) \to (\nu xy)(\langle E_1[x]\rangle \mid \langle E_2[e_k\,y]\rangle)}$$

R-Par
$$\frac{p \to p'}{p \mid q \to p' \mid q}$$

R-Bind
$$\frac{p \to p'}{(\nu xy)p \to (\nu xy)p'}$$

R-Cong
$$\frac{p \equiv q \qquad q \to q'}{p \to q'}$$

Fig. 15. Reduction.

Proof. Item 3. Immediate from rule S-Left. All the remaining items follow by rule induction on the hypothesis. In the case of item 4, rules T-Const, T-Var, T-Abs, T-New, T-Sel, and T-Equse item 2. □

Lemma 13 *(Strengthening).*

1. *If* $\Delta \mid \Gamma, x\colon T \vdash e : U$ *and* $x \notin \mathrm{free}(T)$, *then* $\Delta \vdash T : \mathbf{T}^{\mathsf{un}}$ *and* $\Delta \mid \Gamma \vdash e : U$.
2. *If* $\Gamma, x\colon T \vdash p$ *and* $x \notin \mathrm{free}(p)$, *then* $\vdash T : \mathbf{T}^{\mathsf{un}}$ *and* $\Gamma \vdash p$.

Proof. By rule induction on the first hypothesis. □

Lemma 14 *(Agreement for process formation).*

1. *If* $\Delta \mid \Gamma \vdash e : T$, *then* $\Delta \vdash \Gamma : \kappa$ *and* $\Delta \vdash T : \kappa'$.
2. *If* $\Gamma \vdash p$, *then* $\vdash \Gamma : \kappa$.

Proof. Item 1. By rule induction on the hypothesis, using strengthening (Lemma 13) in the rule for type abstraction, substitution (Lemma 11) in the rule for type substitution, agreement for type equivalence (Lemma 6) in the corresponding rule, and agreement for context split (Lemma 9) in the rules involving context split.

Item 2. By rule induction on the hypothesis, using item 1 on P-Expand the agreement for context split (Lemma 9) on the rule for parallel composition. □

Theorem 15 *(Congruence preserves typing). If* $p \equiv q$ *and* $\Gamma \vdash p$, *then* $\Gamma \vdash q$.

Proof. The proof is by analysis of the derivation for each member of each axiom in Fig. 14. We use the properties of context split (Lemma 10), weakening (Lemma 12), strengthening (Lemma 13), and check both directions of each axiom.

We detail the case for scope restriction. In the forward direction of the axiom we have to show that if $\Gamma \vdash (\nu xy)p \mid q$ then $\Gamma \vdash (\nu xy)(p \mid q)$. We start by building the only derivation for $\Gamma \vdash (\nu xy)p \mid q$ and obtain: $\vdash \Gamma = \Gamma_1 \circ \Gamma_2$ and $\Gamma_1, x\colon T, y\colon \overline{T} \vdash p$ and $\vdash T : \mathbf{s}^{\mathsf{lin}}$ and $\Gamma_2 \vdash q$. To build the derivation for the conclusion, we distinguish two subcases based on the linearity of T. If T is linear, then we have $\Gamma_1, x\colon T, y\colon \overline{T} \circ \Gamma_2 = (\Gamma_1 \circ \Gamma_2), x\colon T, y\colon \overline{T}$. If T is unrestricted, we use weakening and obtain $\Gamma_2, x\colon T, y\colon \overline{T} \vdash q$ and thus, $(\Gamma_1 \circ \Gamma_2), x\colon T, y\colon \overline{T} = (\Gamma_1, x\colon T, y\colon \overline{T}) \circ (\Gamma_2, x\colon T, y\colon \overline{T})$. Conclude with P-Parand P-New.

In the reverse direction we show that if $\Gamma \vdash (\nu xy)(p \mid q)$ then $\Gamma \vdash (\nu xy)p \mid q$. We start by building the only derivation for $\Gamma \vdash (\nu xy)(p \mid q)$ and obtain $\Gamma, x\colon T, y\colon \overline{T} = \Gamma_1 \circ \Gamma_2$ and $\Gamma_1 \vdash p$, $\Gamma_2 \vdash q$ and $\vdash T : \mathbf{s}^{\mathsf{lin}}$. Again, we distinguish two subcases. If T is linear, the properties of the context split (Lemma 10) state that T is either in Γ_1 or in Γ_2, but not in both. Given that $x \notin \mathrm{free}(q)$, strengthening gives us that $x\colon T \notin \Gamma_2$, $x\colon T \in \Gamma_1$, and thus $\Gamma_1 = \Gamma_1', x\colon T, y\colon \overline{T}$. If T is unrestricted, then $\Gamma, x\colon T, y\colon \overline{T} = (\Gamma_1, x\colon T, y\colon \overline{T}) \circ (\Gamma_2, x\colon T, y\colon \overline{T})$. Applying strengthening on $\Gamma_2, x\colon T, y\colon \overline{T} \vdash q$ yields $\Gamma_2 \vdash q$. In both cases, conclude with P-Newand P-Par. □

Lemma 16 *(Normalised type derivation). Suppose that $\Delta \mid \Gamma \vdash e : T$ is derivable. Then there is a derivation of the same judgment that ends with exactly one use of the rule T-Eq.*

Proof. By induction on the number of uses of T-Eq that conclude the derivation of $\Delta \mid \Gamma \vdash e : T$. If the derivation does not end with T-Eq, then we may add one because type equivalence is reflexive. If the derivation ends with more than one application of T-Eq, then we may combine them to one rule application because type equivalence is transitive. □

Lemma 17 *(Inversion for expression formation). Let $\Delta \mid \Gamma \vdash e : T$.*

1. *If $e = x$, then $\Gamma = \Gamma', x\colon U$ and $\Delta \vdash \Gamma' : \mathbf{T}^{\mathsf{un}}$ and $\varepsilon \mid \Delta \vdash U \simeq T : \kappa$.*
2. *If $e = \lambda_m x\colon U.e$, then $\Delta \vdash \Gamma : \mathbf{T}^m$ and $\Delta \mid \Gamma, x\colon U \vdash e : V$ and $\varepsilon \mid \Delta \vdash U \to_m V \simeq T : \mathbf{T}^m$.*
3. *If $e = e_1\, e_2$, then $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ and $\Delta \mid \Gamma_1 \vdash e_1 : U \to_m V$ and $\Delta \mid \Gamma_2 \vdash e_2 : U$ and $\varepsilon \mid \Delta \vdash V \simeq T : \kappa$.*
4. *If $e = \Lambda a\colon \kappa.v$, then $\Delta, a\colon \kappa \mid \Gamma \vdash v : U$ and $\varepsilon \mid \Delta \vdash \forall a\colon \kappa\, .U \simeq T : \mathbf{T}^m$.*
5. *If $e = e\,[U]$, then $\Delta \mid \Gamma \vdash e : \forall a\colon \kappa\, .V$ and $\Delta \vdash U : \kappa$ and $\varepsilon \mid \Delta \vdash V[U/a] \simeq T : \kappa$.*
6. *If $e = \{\ell{=}e_\ell\}_{\ell \in L}$, then $\Delta \vdash \Gamma = \circ\Gamma_\ell$ and $\Delta \mid \Gamma_\ell \vdash e_\ell : T_\ell$ and $\varepsilon \mid \Delta \vdash \{\ell\colon T_\ell\}_{\ell \in L} \simeq T : \mathbf{T}^m$.*
7. *If $e = \mathsf{let}\,\{\ell{=}x_\ell\}_{\ell \in L} = e_1\,\mathsf{in}\,e_2$, then $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ and $\Delta \mid \Gamma_1 \vdash e_1 : \{\ell\colon T_\ell\}_{\ell \in L}$ and $\Delta \mid \Gamma_2, (x_\ell\colon T_\ell)_{\ell \in L} \vdash e_2 : U$ and $\varepsilon \mid \Delta \vdash U \simeq T : \kappa$.*
8. *If $e = \mathsf{let}\,()_m = e_1\,\mathsf{in}\,e_2$, then $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ and $\Delta \mid \Gamma_1 \vdash e_1 : ()_m$ and $\Delta \mid \Gamma_2 \vdash e_2 : U$ and $\varepsilon \mid \Delta \vdash U \simeq T : \kappa$.*
9. *If $e = k\,e$, then exist $\{T_\ell\}_{\ell \in L}$ such that $\Delta \mid \Gamma \vdash e : T_k$ for $k \in L$ and $\Delta \vdash T_\ell : \mathbf{T}^{\mathsf{lin}}$ and $\varepsilon \mid \Delta \vdash \langle \ell\colon T_\ell \rangle_{\ell \in L} \simeq T : \mathbf{T}^m$.*
10. *If $e = \mathsf{case}\,e\,\mathsf{of}\,\{\ell \to e_\ell\}_{\ell \in L}$, then $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ and $\Delta \mid \Gamma_1 \vdash e : \langle \ell\colon T_\ell \rangle_{\ell \in L}$ and $\Delta \mid \Gamma_2 \vdash e_\ell : T_\ell \to_{\mathsf{lin}} U$ and $\varepsilon \mid \Delta \vdash U \simeq T : \kappa$.*
11. *If $e = \mathsf{select}\,k$, then exist $\{T_\ell\}_{\ell \in L}$ such that $\Delta \vdash T_\ell : \mathbf{s}^{\mathsf{lin}}$ and $\varepsilon \mid \Delta \vdash \oplus\{\ell\colon T_\ell\}_{\ell \in L} \to_m T_k \simeq T : \mathbf{T}^m$ for $k \in L$ and $\Delta \vdash \Gamma : \mathbf{T}^{\mathsf{un}}$.*

Proof. All cases are similar; we detail item 3. By Lemma 16, there is a derivation that ends with a single use of T-Eq. Its premises read $\Delta \mid \Gamma \vdash e_1\, e_2 : V$ and $\varepsilon \mid \Delta \vdash V \simeq T : \kappa$. The premises to rule T-Appon $\Delta \mid \Gamma \vdash e_1\, e_2 : V$ include $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ and $\Delta \mid \Gamma_1 \vdash e_1 : U \to_m V$ and $\Delta \mid \Gamma_2 \vdash e_2 : U$ which proves the claim. □

The next lemma is used to establish type preservation for the communication reductions. Since such rules move values from one process to another, the values need to take a part of its typing context with them.

Lemma 18 *(Context typing). Let $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$. Then, $\Delta \mid \Gamma \vdash E[e] : T$ if and only if $\Delta \mid \Gamma_1 \vdash e : U$ and $\Delta \mid \Gamma_2, x\colon U \vdash E[x] : T$.*

Proof. The forward implication is by induction on the structure of evaluation context E. We sketch one case: Ee', in which case $(Ee')[e] = (E[e])e'$. Inversion of the typing relation (Lemma 17) yields (1) $\varepsilon \mid \Delta \vdash W \simeq T : \kappa$ and (2) $\Delta \mid \Gamma_1 \vdash E[e] : V \to_m W$ and (3) $\Delta \mid \Gamma_2 \vdash e' : V$. From (2) and induction we get (4) $\Delta \mid \Gamma_{11}, x\colon U \vdash E[x] : V \to_m W$ and (5) $\Delta \mid \Gamma_{12} \vdash e : U$ and (6) $\Delta \vdash \Gamma_1 = \Gamma_{11} \circ \Gamma_{12}$. From the hypothesis $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ and (6), properties of context split (Lemma 10) guarantee that $\Gamma_{11} \circ \Gamma_2$ is defined. Then, from (3), (4) and rule T-Appwe have (7) $\Delta \mid \Gamma_{11} \circ \Gamma_2, x\colon U \vdash (E[x])e' : W$. Given that $(E[x])e' = (Ee')[x]$, from (1), (7) and rule T-Eqwe have $\Delta \mid \Gamma_{11} \circ \Gamma_2, x\colon U \vdash (Ee')[x] : T$, which concludes the case together with item (5).

The reverse direction is a simple application of the appropriate typing rule. For example, for context Ee' we use rule T-App. □

Lemma 19 *(Substitution preserves equivalence).*
If $\Theta \mid \Delta, a : \kappa \vdash T_1 \simeq T_2 : \kappa'$ and $\Delta \vdash U : \kappa$, then $\Theta \mid \Delta \vdash T_1[U/a] \simeq T_2[U/a] : \kappa'$.

Proof. By rule induction on the first hypothesis. The key step is the case for rule Q-Var when the type variable considered is a. Here we need to appeal to reflexivity of type equivalence to establish $\Theta \mid \Delta \vdash U \simeq U : \kappa$ (Lemma 7). □

Lemma 20 *(Type substitution). If $\Delta, a\colon \kappa \mid \Gamma \vdash e : T$ and $\Delta \vdash U : \kappa$, then $\Delta \mid \Gamma[U/a] \vdash e[U/a] : T[U/a]$.*

Proof. The proof is by rule induction on the first hypothesis. We rely on Lemma 3 to handle premises on type and typing context formation. Lemma 11 handles the interaction with context splitting. Lemma 19 handles the case for T-Eq. We detail one case: rule T-Varwith $e = x$. The premises to the rule are $\Delta, a\colon \kappa \vdash \Gamma : \mathbf{T}^{\mathsf{un}}$ and $\Delta, a\colon \kappa \vdash T : \kappa'$ Noticing that $x[U/a] = x$, the result follows by applying type substitution (Lemma 3) to the second hypothesis and to the premises to the rule. □

Lemma 21 *(Value substitution). If* $\Delta \mid \Gamma_1, x : T \vdash e : U$ *and* $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ *and* $\Delta \mid \Gamma_2 \vdash v : T$*, then* $\Delta \mid \Gamma \vdash e[v/x] : U$.

Proof. The proof is by rule induction on the first hypothesis. In some cases (variables, application and abstraction), we have to distinguish whether T (or the type of the abstracted variable) is linear or not. Unrestricted weakening (Lemma 12) is required in the case for abstraction. □

Theorem 22 *(Typing preservation for expression reduction). If* $e_1 \to e_2$ *and* $\Delta \mid \Gamma \vdash e_1 : T$*, then* $\Delta \mid \Gamma \vdash e_2 : T$.

Proof. By rule induction on $e_1 \to e_2$.

Case E-TApp. We apply inversion (Lemma 17) to the typing judgement for the redex $\Delta \mid \Gamma \vdash (\Lambda a\colon \kappa . v)\,[T'] : T$ and obtain

$$\Delta \mid \Gamma \vdash (\Lambda a\colon \kappa . v) : \forall a\colon \kappa\,.\,U \tag{1}$$

$$\Delta \vdash T' : \kappa \tag{2}$$

$$\varepsilon \mid \Delta \vdash T \simeq U[T'/a] : \kappa \tag{3}$$

Inversion of equation (1) (Lemma 17) yields:

$$\Delta, a\colon \kappa \mid \Gamma \vdash v : U' \tag{4}$$

$$\varepsilon \mid \Delta \vdash \forall a\colon \kappa\,.\,U' \simeq \forall a\colon \kappa\,.\,U : \kappa' \tag{5}$$

By Q-TAbs

$$\varepsilon \mid \Delta, a : \kappa \vdash U' \simeq U : \kappa' \tag{6}$$

Applying rule T-Eqto equations (4) and (6) yields

$$\Delta, a\colon \kappa \mid \Gamma \vdash v : U \tag{7}$$

Applying the substitution lemma (Lemma 20) to equations (2) and (7) yields

$$\Delta \mid \Gamma \vdash v[T'/a] : U[T'/a] \tag{8}$$

noticing that $a \notin \mathrm{free}(\Gamma)$, hence $\Gamma[T'/a] = \Gamma$. Conclude by applying T-Equsing equation (3).

Case E-App. We apply inversion (Lemma 17) to the typing judgement for the redex $\Delta \mid \Gamma \vdash (\lambda_m x\colon U . e)\, v : T$ and obtain

$$\Delta \mid \Gamma \vdash \lambda_m x\colon U . e : U \to_m U' \tag{9}$$

$$\Delta \mid \Gamma \vdash v : U \tag{10}$$

$$\varepsilon \mid \Delta \vdash T \simeq U' : \kappa \tag{11}$$

By inversion (Lemma 17) applied to (9) we obtain

$$\Delta \mid \Gamma, x : U \vdash e : U'' \tag{12}$$

$$\varepsilon \mid \Delta \vdash U' \simeq U'' : \kappa \tag{13}$$

Applying substitution (Lemma 21) to (12) and (10) we have

$$\Delta \mid \Gamma \vdash e[v/x] : U'' \tag{14}$$

Using transitivity on (11) and (13), we apply T-Eq to obtain

$$\Delta \mid \Gamma \vdash e[v/x] : T \tag{15}$$

Case E-RcdElim. We apply inversion (Lemma 17) to the typing judgement for the redex $\Delta \mid \Gamma \vdash \mathsf{let}\,\{\ell{=}x_\ell\}_{\ell\in L} = \{\ell{=}v_\ell\}_{\ell\in L}\,\mathsf{in}\,e : T$ and obtain

$$\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \tag{16}$$
$$\Delta \mid \Gamma_1 \vdash \{\ell{=}v_\ell\}_{\ell\in L} : \{\ell : T_\ell\}_{\ell\in L} \tag{17}$$
$$\Delta \mid \Gamma_2, (x_\ell : T_\ell)_{\ell\in L} \vdash e : T' \tag{18}$$
$$\varepsilon \mid \Delta \vdash T \simeq T' : \kappa \tag{19}$$

Inversion (Lemma 17) and Q-Recordapplied to (18) yields

$$\Delta \vdash \Gamma_1 = \circ\Gamma_\ell \tag{20}$$
$$\Delta \mid \Gamma_\ell \vdash v_\ell : T'_\ell \tag{21}$$
$$\varepsilon \mid \Delta \vdash T_\ell \simeq T'_\ell : \kappa \tag{22}$$

By T-Eq we have $\Delta \mid \Gamma_\ell \vdash v_\ell : T_\ell$, for each ℓ, by substitution (Lemma 21) on (18) we have $\Delta \mid \Gamma_2 \circ \Gamma_1 \vdash e[v_\ell/x_\ell]_{\ell\in L} : T'$, and by T-Eq its type is also T.

Case E-Case. We apply inversion (Lemma 17) to the typing judgement for the redex

$$\Delta \mid \Gamma \vdash \mathsf{case}\, k\, v\, \mathsf{of}\, \{\ell \to e_\ell\}_{\ell\in L} : T \tag{23}$$

(where $k \in L$) and obtain some T' such that

$$\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \tag{24}$$
$$\Delta \mid \Gamma_1 \vdash k\, v : \langle \ell : T_\ell \rangle_{\ell\in L} \tag{25}$$
$$\Delta \mid \Gamma_2 \vdash e_\ell : T_\ell \to_m T' \tag{26}$$
$$\varepsilon \mid \Delta \vdash T \simeq T' : \kappa \tag{27}$$

By inversion (Lemma 17) on (24), exist $\langle \ell : T_\ell \rangle_{\ell\in L}$ such that

$$\Delta \mid \Gamma_1 \vdash v : T' \tag{28}$$
$$\varepsilon \mid \Delta \vdash T_k \simeq T' : \kappa \tag{29}$$

Putting things back together: By T-Eq, $\Delta \mid \Gamma_1 \vdash v : T_k$. Hence, we can apply e_k using T-App

$$\Delta \mid \Gamma \vdash e_k\, v : T' \tag{30}$$

and conclude with T-Eq to obtain type T.

Case E-Ctx. From context typing (Lemma 18) we know that $\Delta \mid \Gamma_1 \vdash e_1 : U$ and $\Delta \mid \Gamma_2, x : U \vdash E[x] : T$, with $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$. By induction $\Delta \mid \Gamma_1 \vdash e_2 : U$. We conclude with context typing, now in the reverse direction. □

Theorem 23 *(Type preservation for process reduction). If $p \to q$ and $\Gamma \vdash p$, then $\Gamma \vdash q$.*

Proof. We proceed by rule induction on $p \to q$.

Case R-Exp. We invert P-Expto obtain $\varepsilon \mid \Gamma \vdash e : T$. Preservation for expression reduction (Theorem 22) gives $\varepsilon \mid \Gamma \vdash e' : T$. We conclude with rule P-Exp.

Case R-Fork. We invert P-Expto obtain

$$\varepsilon \mid \Gamma \vdash E[\mathsf{fork}\,[\_]\,e] : T \tag{31}$$

By Lemma 18, we can pick some $z : ()_m$ and $\varepsilon \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ such that

$$\varepsilon \mid \Gamma_1, z : ()_m \vdash E[z] : T \tag{32}$$
$$\varepsilon \mid \Gamma_2 \vdash \mathsf{fork}\,[\_]\,e : ()_m \tag{33}$$

By substitution and by inversion (for some $\varepsilon \vdash T' : \mathbf{T}^{\mathsf{un}}$)

$$\varepsilon \mid \Gamma_1 \vdash E[()_m] : T \tag{34}$$
$$\varepsilon \mid \Gamma_2 \vdash e : T' \tag{35}$$

By rules P-Exp and P-Par

$$\Gamma \vdash \langle E[()_m] \rangle \mid \langle e \rangle \tag{36}$$

Case R-New. We invert P-Expto obtain

$$\varepsilon \mid \Gamma \vdash E[\mathsf{new}\, S] : T \tag{37}$$

By Lemma 18, we can pick some $z : \{\mathsf{fst}\colon S, \mathsf{snd}\colon \overline{S}\}$ such that

$$\varepsilon \mid \Gamma, z : \{\mathsf{fst}\colon S, \mathsf{snd}\colon \overline{S}\} \vdash E[z] : T \tag{38}$$

$$\varepsilon \mid \varepsilon \vdash \mathsf{new}\, S : \{\mathsf{fst}\colon S, \mathsf{snd}\colon \overline{S}\} \tag{39}$$

We create a new typed value from two channel ends by

$$\varepsilon \mid x : S, y : \overline{S} \vdash (x, y) : \{\mathsf{fst}\colon S, \mathsf{snd}\colon \overline{S}\} \tag{40}$$

By substitution of this value for z, we obtain

$$\varepsilon \mid \Gamma, x : S, y : \overline{S} \vdash E[(x, y)] : T \tag{41}$$

By P-Exp and P-New, we find:

$$\Gamma \vdash \langle E[(x, y)]\rangle(\nu xy) \tag{42}$$

Case R-Com. We invert P-New to obtain

$$\Gamma, x\colon T, y\colon \overline{T} \vdash (\langle E_1[\mathsf{send}\,[\_]\, v\, [\_]\, x]\rangle \mid \langle E_2[\mathsf{receive}\,[\_]\,[\_]\, y]\rangle) \tag{43}$$

$$\vdash T : \mathbf{s}^{\mathsf{lin}} \tag{44}$$

We invert (43) using P-Par to obtain

$$\Gamma = \Gamma_1 \circ \Gamma_2 \tag{45}$$

$$\varepsilon \mid \Gamma_1, x\colon T \vdash E_1[\mathsf{send}\,[\_]\, v\, [\_]\, x] : ()_m \tag{46}$$

$$\varepsilon \mid \Gamma_2, y\colon \overline{T} \vdash E_2[\mathsf{receive}\,[\_]\,[\_]\, y] : ()_m \tag{47}$$

Applying Lemma 18 to judgement (46) yields some $\Gamma_1'', \Gamma_2'', S_1$ such that, for some type R and all z_1 not in Γ_2'',

$$\Gamma_1, x : T = \Gamma_1'' \circ (\Gamma_2'', x : T) \tag{48}$$

$$\varepsilon \mid \Gamma_2'', x : T \vdash \mathsf{send}\,[\_]\, v\, [\_]\, x : S_1 \tag{49}$$

$$\varepsilon \mid \Gamma_1'', z_1 : S_1 \vdash E_1[z_1] : ()_m \tag{50}$$

Inversion of the sending application yields

$$T = !\, R; S_1 \tag{51}$$

$$\varepsilon \mid \Gamma_2'' \vdash v : R \tag{52}$$

Applying Lemma 18 to (47) yields some $\Gamma_1''', \Gamma_2''', S_2$ such that, for all z_3 not in Γ_2''',

$$\Gamma_2, y : \overline{T} = \Gamma_1''' \circ (\Gamma_2''', y : \overline{T}) \tag{53}$$

$$\overline{T} = ?\, R; \overline{S_1} \text{ and } S_2 = \{\mathsf{fst}\colon R, \mathsf{snd}\colon \overline{S_1}\} \tag{54}$$

$$\varepsilon \mid \Gamma_2''', y : \overline{T} \vdash \mathsf{receive}\,[\_]\,[\_]\, y : S_2 \tag{55}$$

$$\varepsilon \mid \Gamma_1''', z_2 : S_2 \vdash E_2[z_2] : ()_m \tag{56}$$

To conclude, take $z_1 = x$ to obtain

$$\varepsilon \mid \Gamma_1'', x : S_1 \vdash E_1[x] : ()_m \tag{57}$$

From (52), we obtain

$$\varepsilon \mid \Gamma_2'', y : \overline{S_1} \vdash (v, y) : \{\mathsf{fst}\colon R, \mathsf{snd}\colon \overline{S_1}\} \tag{58}$$

Apply substitution (Lemma 21) for z_2 to (56) and (58) yielding

$$\varepsilon \mid \Gamma_1''' \circ \Gamma_2'', y : \overline{S_1} \vdash E_2[(v, y)] : ()_m \tag{59}$$

Apply P-Par to (57) and (59), exploiting that $\Gamma = \Gamma_1'' \circ \Gamma_2'' \circ \Gamma_1''' \circ \Gamma_2'''$

$$\Gamma, x : S_1, y : \overline{S_1} \vdash (\langle E_1[x] \rangle \mid \langle E_2[(v, y)] \rangle) \tag{60}$$

By (44) and (51) and inversion of K-Seq, we know that

$$\varepsilon \vdash S_1 : \mathbf{s}^{\mathsf{lin}} \tag{61}$$

Conclude by applying P-New to (61) and (60).

Case R-Ch. We proceed analogously.

Case R-Par. Conclude by induction.

Case R-Bind. Conclude by induction.

Case R-Cong. Apply Theorem 15 and conclude by induction. □

Absence of run-time errors We start with the definition of runtime errors. The *subject* of an expression e, denoted by $\mathrm{subj}(e)$, is z in the following cases and undefined in all other cases.

$$\mathsf{send}[T]v[U]z \qquad \mathsf{receive}[T][U]z \qquad \mathsf{select}\,\ell\,z \qquad \mathsf{match}\,z\,\mathsf{with}\,\{\ell \to e_\ell\}$$

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

- $\mathrm{agree}^{xy}(\mathsf{send}[T]v[U]x, \mathsf{receive}[T][\overline{U}]y)$;
- $\mathrm{agree}^{xy}(\mathsf{receive}[T][U]x, \mathsf{send}[T]v[\overline{U}]y)$;
- $\mathrm{agree}^{xy}(\mathsf{select}\,k\,x, \mathsf{match}\,y\,\mathsf{with}\,\{\ell \to e_\ell\}_{\ell \in L})$ and $k \in L$;
- $\mathrm{agree}^{xy}(\mathsf{match}\,x\,\mathsf{with}\,\{\ell \to e_\ell\}_{\ell \in L}, \mathsf{select}\,k\,y)$ and $k \in L$.

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

1. $\langle E[v\,e] \rangle$ where v is $x, c, \Lambda a{:}\,\kappa.v', \{\ell{=}v_\ell\}_{\ell \in L}, \ell\,v', \mathsf{send}[T]v'$ or $\mathsf{receive}[T]$;
2. $\langle E[v\,[T]] \rangle$ where v is x, $\lambda_m x{:}\,T.e$, $\{\ell{=}v_\ell\}_{\ell \in L}$, $\ell\,v'$, $\mathsf{send}[T]$, $\mathsf{send}[T]v'[U]$ or $\mathsf{receive}[T][U]$.
3. $\langle E[\mathsf{let}\,\{\ell{=}x_\ell\}_{\ell \in L} = v\,\mathsf{in}\,e] \rangle$ and $v \neq \{\ell{=}v_\ell\}_{\ell \in L}$;
4. $\langle E[\mathsf{match}\,v\,\mathsf{with}\,\{\ell \to e_\ell\}_{\ell \in L}] \rangle$ and $v \neq k\,v'$, for all $k \in L$;
5. $\langle E_1[e_1] \rangle \mid \langle E_2[e_2] \rangle$ and $\mathrm{subj}(e_1) = \mathrm{subj}(e_2)$;
6. $(\nu xy)(\langle E_1[e_1] \rangle \mid \langle E_2[e_2] \rangle \mid p)$ and $\mathrm{subj}(e_1) = x$ and $\mathrm{subj}(e_2) = y$ and $\neg\,\mathrm{agree}^{xy}(e_1, e_2)$.

The first four cases are typical of polymorphic functional languages with records, variants, and functional constants. Item 5 guarantees that no two threads hold references to the same channel end. If $\langle E_1[e_1] \rangle \mid \langle E_2[e_2] \rangle$ is closed for non channel-end variables (and given that the evaluation contexts E_1 and E_2 do not bind variables), then the subjects of e_1 and e_2 are channel ends. Item 6 says that channel ends agree at all times: if one thread is ready for sending, then the other is ready for receiving, and similarly for selection and branching.

Theorem 24 *(Absence of run-time errors). If $\vdash p$, then p is not an error.*

Proof. Suppose that $\vdash p$ and that p is an error. Then we know that for each subprocess q of p there exists Γ such that $\Gamma \vdash q$ with $\vdash \Gamma : \mathbf{s}^m$, since all entries in Γ are introduced by rule R-New. By Theorem 15, we may assume that q is one of the offending subprocesses. We analyse in turn each of the six cases in the definition of runtime errors.

1. Suppose $\varepsilon \mid \Gamma_1 \vdash v\,e : T$ where v is of one of the forms in the definition of run-time errors. By inversion (Lemma 17), we have $\vdash \Gamma_1 = \Gamma_1' \circ \Gamma_1''$ and $\varepsilon \mid \Gamma_1 \vdash v : U \to_m V$. We distinguish the various cases for v, extracting a contradiction in each case. If v is a variable, we find a contradiction because Γ_1, and hence Γ_1', contain no arrow types. If v is a constant, there is a contradiction because no constant is of an arrow type (see Fig. 11). If v is a type abstraction, a record, an injection, $\mathsf{send}[T]v'$ or $\mathsf{receive}[T]$, a contradiction arises because, by inversion, their types are not arrow types.

2, 3 and 4. Analogous to the previous case.

5. Suppose that $\Gamma \vdash \langle E_1[e_1] \rangle \mid \langle E_2[e_2] \rangle$ where $\mathrm{subj}(e_1) = \mathrm{subj}(e_2) = z$. By inversion of P-Parwe have $\Gamma_1 \vdash \langle E_1[e_1] \rangle$ and $\Gamma_2 \vdash \langle E_2[e_2] \rangle$ with $\varepsilon \vdash \Gamma = \Gamma_1 \circ \Gamma_2$. By Lemma 18 we know that $\varepsilon \mid \Gamma_1' \vdash e_1 : T_1$ with $\Gamma_1' \subseteq \Gamma_1$, and similarly for expression e_2. Analysing inversions (Lemma 17) for the various expressions e_1 such that $\mathrm{subj}(e_1)$ is defined, we conclude that $\varepsilon \vdash T_1 : \mathbf{s}^{\mathsf{lin}}$ and $\varepsilon \mid \Gamma_1'' \vdash z : T_1$ with $\Gamma_1'' \subseteq \Gamma_1$, and similarly for e_2. But this contradicts the existence of the splitting of Γ.

6. Suppose that $\Gamma \vdash (\nu xy)(\langle E_1[e_1] \rangle \mid \langle E_2[e_2] \rangle \mid q)$ with $\mathrm{subj}(e_1) = x$ and $\mathrm{subj}(e_2) = y$ and $\neg\,\mathrm{agree}^{xy}(e_1, e_2)$. By inversion of P-New we obtain $\vdash T : \mathbf{s}^{\mathsf{lin}}$ and $\Gamma, x{:}\,T, y{:}\,\overline{T} \vdash (\langle E_1[e_1] \rangle \mid \langle E_2[e_2] \rangle \mid q)$. By congruence and inversion of P-Par we obtain $\Gamma', x{:}\,T, y{:}\,\overline{T} \vdash \langle E_1[e_1] \rangle \mid \langle E_2[e_2] \rangle$. By further inversion of P-Par we get $\varepsilon \vdash \Gamma', x{:}\,T, y{:}\,\overline{T} = (\Gamma_1', x{:}\,T) \circ (\Gamma_2', y{:}\,\overline{T})$ and $\Gamma_1', x{:}\,T \vdash \langle E_1[e_1] \rangle$ and $\Gamma_2', y{:}\,\overline{T} \vdash \langle E_2[e_2] \rangle$. Now we have to consider the cases where $\neg\,\mathrm{agree}^{xy}(e_1, e_2)$, all of which contradict the definition of duality. We show two representative examples.

- e_1 has the form $\mathsf{send}[U_1]v_1[U_2]x$ and e_2 has the form $\mathsf{send}[V_1]v_2[V_2]y$. In this case, $T = !\,U_1; U_2$ and $\overline{T} = !\,V_1; V_2$, which contradicts the definition of duality.
- e_1 has the form $\mathsf{send}[U_1]e[U_2]x$ and e_2 has the form $\mathsf{select}\,\ell\,y$. In this case, $T = !\,U_1; U_2$ and $\overline{T} = \oplus\{\ell\colon T_\ell\}_{\ell\in L}$, which also contradicts the definition of duality. □

4.3. Progress for single-threaded processes

Progress for arbitrary processes cannot be guaranteed by typing, for threads may share more than one channel and interact on these in the "wrong" order. Yet, single-threaded processes enjoy the pleasing result of progress.

Before we embark on the proof for progress, we need to tell which values inhabit a few select types.

Lemma 25 *(Canonical forms). Let $\Delta \mid \Gamma \vdash v : T$ with $\Delta \vdash \Gamma : \mathbf{s}^{\mathsf{lin}}$.*

1. *If $T = U \to_m V$, then v is $\lambda_m x\colon W.e$, $\mathsf{select}\,k$, $\mathsf{send}[W]$, $\mathsf{send}[W]v'[X]$, or $\mathsf{receive}[W][X]$.*
2. *If $T = \forall a\colon \kappa\,.\,U$, then $v = \Lambda a\colon \kappa.v'$, send, $\mathsf{send}[W]v'[X]$, receive or $\mathsf{receive}[W]$.*
3. *If $T = ()_m$, then $v = ()_m$.*
4. *If $T = \{l\colon T_l\}_{l\in L}$, then $v = \{l{=}v_l\}_{l\in L}$.*
5. *If $T = \langle l\colon T_l\rangle_{l\in L}$, then $v = k\,v'$ and $k \in L$.*
6. *If $T = \star\{\ell\colon T_\ell\}_{\ell\in L}$ or $T = \sharp U; W$, then $v = x$.*

Proof. Given that $\Delta \vdash \Gamma : \mathbf{s}^{\mathsf{lin}}$, the only variables that can be read from Γ in item 6 are those of a session type. We start with an observation, which follows by inspection of the typing rules: For all syntactic values v, there is a derivation $\Delta \mid \Gamma \vdash v : T_0$ such that T_0 has one of the forms 1–6. Specifically, T_0 is not a recursive type. Second, we exploit Lemma 16 to obtain that the derivation of $\Delta \mid \Gamma \vdash v : T$ ends with a single use of the rule T-Eq. By inversion of that rule, we have $\Delta \mid \Gamma \vdash v : T_0$ and $\varepsilon \mid \Delta \vdash T_0 \simeq T : \kappa$, where T_0 has one of the forms 1–6. The last step in the derivation of $\varepsilon \mid \Delta \vdash T_0 \simeq T : \kappa$ cannot be Q-Fix(because $\Theta = \varepsilon$), nor Q-RecL(because T_0 is not a μ type), nor Q-RecR(because T is not a μ type).

Case $T = U \to_m V$. By rule Q-Arrow, $T_0 = U_0 \to_m V_0$ with $\varepsilon \mid \Delta \vdash U \simeq U_0 : \kappa'$ and $\varepsilon \mid \Delta \vdash V \simeq V_0 : \kappa'$. As rule T-Eqis not applicable to construct $\Delta \mid \Gamma \vdash v : T_0$, we find that this derivation may end with T-Const, T-Var, T-Abs, T-TApp, or T-Sel.

All constants have polymorphic types or type $()_m$, which rules out T-Const.

All variables have kind $\mathbf{s}^{\mathsf{lin}}$, which rules out T-Var.

Rule T-Absis applicable and yields $v = \lambda_m x\colon U_0.e$.

Rule T-Selis applicable and yields $v = \mathsf{select}\,k$.

Rule T-TAppis applicable and yields $v = \mathsf{send}[W]$, $v = \mathsf{send}[W]v'[X]$, or $v = \mathsf{receive}[W][X]$.

Case $T = \forall a\colon \kappa\,.\,U$. By rule Q-TAbs, $T_0 = \forall a\colon \kappa\,.\,U_0$ with $\varepsilon \mid \Delta, a\colon \kappa \vdash U \simeq U_0 : \kappa'$. As rule T-Eqis not applicable to construct $\Delta \mid \Gamma \vdash v : T_0$, we find that this derivation may end with T-Const, T-Var, T-TAbs, or T-TApp.

From rule T-Const, we obtain $v = \mathsf{send}$ or $v = \mathsf{receive}$.

All variables have kind $\mathbf{s}^{\mathsf{lin}}$, which rules out T-Var.

Rule T-TAbsyields $v = \Lambda a\colon \kappa.v'$.

Rule T-TAppis applicable and yields $v = \mathsf{send}[W]v'[X]$ or $v = \mathsf{receive}[W]$.

Case $T = ()_m$. By rule Q-Unit, $T_0 = ()_m$. As rule T-Eqis not applicable to construct $\Delta \mid \Gamma \vdash v : T_0$, we find that this derivation may end with T-Const, T-Var, or T-TApp.

From rule T-Const, we obtain $v = ()_m$.

All variables have kind $\mathbf{s}^{\mathsf{lin}}$, which rules out T-Var.

There is no polymorphic constant abstracting over unit, which rules out T-TApp.

Case $T = \{\ell\colon T_\ell\}_{\ell\in L}$. By rule Q-Record, $T_0 = \{\ell\colon T_{0\ell}\}_{\ell\in L}$ where $\varepsilon \mid \Delta \vdash T_\ell \simeq T_{0\ell} : \kappa'$, for all $\ell \in L$.

As rule T-Eqis not applicable to construct $\Delta \mid \Gamma \vdash v : T_0$, we find that this derivation may end with T-Const, T-Var, T-TApp, or T-Record.

There is no applicable constant, which rules out T-Const.

All variables have kind $\mathbf{s}^{\mathsf{lin}}$, which rules out T-Var.

There is no polymorphic constant abstracting over records, which rules out T-TApp.

Rule T-Recordyields $v = \{l{=}v_l\}_{l\in L}$.

Case $T = \langle l\colon T_l\rangle_{l\in L}$. Similar to the case for records.

Case $T = \star\{\ell\colon T_\ell\}_{\ell\in L}$ or $T = \sharp U; W$. In this case, $\kappa = \mathbf{s}^{\mathsf{lin}}$ and the equivalence derivation ends with rule Q-ST. By Lemma 6, $\Delta \vdash T_0 : \mathbf{s}^{\mathsf{lin}}$.

As rule T-Eqis not applicable to construct $\Delta \mid \Gamma \vdash v : T_0$, we find that this derivation may end with T-Const, T-Var, or T-TApp.

There is no constant typed with kind $\mathbf{s}^{\mathsf{lin}}$, which rules out rule T-Const.

Rule T-Varis applicable and yields $v = x$.

Rule T-TAppis not applicable as there are no polymorphic constants with kind $\mathbf{s}^{\mathsf{lin}}$. □

Theorem 26 *(Progress for the functional sub-language). Suppose that $\Delta \mid \Gamma \vdash e : T$ with $\Delta \vdash \Gamma : \mathbf{s}^{\mathsf{lin}}$. Then either*

1. *e is a value,*
2. *e reduces, or*
3. *e is of the form* $E[e']$ *with* $e' = \mathsf{new}\,U$, $\mathsf{send}[U]v[V]x$, $\mathsf{receive}[U][V]x$, $\mathsf{select}\,\ell\,x$ *or* $\mathsf{match}\,x\,\mathsf{with}\,\{\ell{\to}e_\ell\}_{\ell\in L}$, *in which case e is stuck.*

Proof. By rule induction on hypothesis $\Delta \mid \Gamma \vdash e : T$.

Cases T-Var, T-Const, T-Absand T-TAbs. Variables, constants, term and type abstractions are all values.

Case T-App. Then $e = e_1\,e_2$. The premises to the rule are $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ and $\Delta \mid \Gamma_1 \vdash e_1 : U \to_m T$ and $\Delta \mid \Gamma_2 \vdash e_2 : U$. By Lemma 9, $\Delta \vdash \Gamma_1 : \mathbf{s}^{\mathsf{lin}}$ and $\Delta \vdash \Gamma_2 : \mathbf{s}^{\mathsf{lin}}$. By induction on e_1, three cases may happen; we analyse each separately. Case e_1 is a value v_1. By induction, this time on e_2 we are faced with three subcases. Subcase e_2 is a value v_2: canonical forms (Lemma 25) for v_1 yield one of the following cases:

- $v_1 = \lambda_m x{:}\,U.e_1'$, in which case e reduces by rule E-App.
- $v_1 = \mathsf{select}\,k$. In this case inversion (Lemma 17) for v_1 gives $\varepsilon \mid \Delta \vdash \oplus\{\ell{:}\,T_\ell\}_{\ell\in L} \to_m T_k \simeq U \to_m V : \mathbf{T}^m$, hence $\varepsilon \mid \Delta \vdash \oplus\{\ell{:}\,T_\ell\}_{\ell\in L} \simeq U : \kappa$, for some kind κ. Given that $\simeq$ is commutative, by rule T-Eq, we have $\Gamma \vdash v_2 : \oplus\{\ell{:}\,T_\ell\}_{\ell\in L}$ and by canonical forms, $v_2 = x$, hence e is stuck at the empty context.
- $v_1 = \mathsf{send}[W]$, in which case $e = \mathsf{send}[W]v_2$ is a value.
- $v_1 = \mathsf{send}[W]v_1'[X]$. Similarly to $\mathsf{select}\,k$, by inversion of v_1 and canonical forms for v_2 we have $v_2 = x$, hence e is stuck at the empty context.
- $v_1 = \mathsf{receive}[W][X]$. As above.

Subcase e_2 reduces: take $E = v_1[]$ so that e reduces by rule E-Ctx. Subcase e_2 is stuck at $E[e_2']$: take $E' = v_1 E$ and e is stuck at E'.

Case T-TApp. Follows the lines of rule T-App.

Case T-Record. Then $e = \{\ell{=}e_\ell\}_{\ell\in L}$ and $T = \{\ell{:}\,T_\ell\}_{\ell\in L}$. The premises to the rule read $\Delta \vdash \Gamma = \circ\Gamma_\ell$ and $\Delta \mid \Gamma_\ell \vdash e_\ell : T_\ell$. By Lemma 9, $\Delta \vdash \Gamma_\ell : \mathbf{s}^{\mathsf{lin}}$. By induction three cases may happen, for each ℓ in L. If all e_ℓ are values, then e is a value. Otherwise let e_k ($k \in L$) be the first non-value. Expression e_k may reduce or be stuck. If it reduces, take for E the context $\{\ell_1 = v_1, \ldots, \ell_k = [], \ldots \ell_n = e_n\}$ and e reduces by rule E-Ctx. If e_k is stuck and is of the form $E[e_k']$, then e is stuck under context $\{\ell_1 = v_1, \ldots, \ell_k = E, \ldots \ell_n = e_n\}$.

Case T-RcdElim. Then $e = \mathsf{let}\,\{\ell{=}x_\ell\}_{\ell\in L} = e_1\,\mathsf{in}\,e_2$. The premises to the rule read $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ and $\Delta \mid \Gamma_1 \vdash e_1 : \{\ell{:}\,T_\ell\}_{\ell\in L}$ and $\Delta \mid \Gamma_2, (x_\ell{:}\,T_\ell)_{\ell\in L} \vdash e_2 : T$§. By Lemma 9, $\Delta \vdash \Gamma_1 : \mathbf{s}^{\mathsf{lin}}$. By induction on e_1 three cases may happen. If e_1 is a value, then canonical forms give $e_1 = \{\ell{=}e_\ell\}_{\ell\in L}$ and e_1 reduces by rule E-RcdElim. If e_1 reduces, then e reduces by rule E-Ctxunder context $\mathsf{let}\,\{\ell{=}x_\ell\}_{\ell\in L} = []\,\mathsf{in}\,e_2$. If e_1 is stuck under context E, then so is e under context $\mathsf{let}\,\{\ell{=}x_\ell\}_{\ell\in L} = E\,\mathsf{in}\,e_2$.

Cases T-UnitElim, T-Variant, T-Caseand T-Match. Similar.

Case T-Sel. Expression $\mathsf{select}\,k$ is a value.

Case T-New. Take the empty context for E. Then e is stuck under context $E[\mathsf{new}\,T]$.

Case T-Eq. By induction. □

We say that *variable x is an active channel end in expression e under contexts* $\Delta \mid \Gamma$ if x is free in e and $\Delta \mid \Gamma \vdash x : T$ with $\Delta \vdash T : \mathbf{s}^{\mathsf{lin}}$ but *not* $\Delta \vdash T : \mathbf{s}^{\mathsf{un}}$. Intuitively, this means that T is not (equivalent to) Skip.

Lemma 27. *Suppose that* $\Delta \mid \Gamma \vdash v : T$ *with* $\Delta \vdash \Gamma : \mathbf{s}^{\mathsf{lin}}$ *and* $\Delta \vdash T : \mathbf{T}^{\mathsf{un}}$. *Then v contains no active channel end under* $\Delta \mid \Gamma$.

Proof. By structural induction on v.

Case c. Constants are closed.

Case x. By inversion (Lemma 17), $\Delta \mid \Gamma \vdash x : U$ and $\varepsilon \mid \Delta \vdash U \simeq T : \kappa$. Because type equivalence is symmetric (Lemma 7), by agreement (Lemma 6) we have $\Delta \vdash T : \kappa$. Hence $\kappa = \mathbf{T}^{\mathsf{un}}$. Then, again by agreement, $\Delta \vdash U : \mathbf{T}^{\mathsf{un}}$. If x is active, then it must be that $\Delta \vdash U : \mathbf{s}^{\mathsf{lin}}$. Contradiction.

Case $\lambda_m x{:}\,U.e$. By inversion and agreement, $\Delta \vdash T : \mathbf{T}^m$ and $\Delta \vdash \Gamma : \mathbf{T}^m$. Hence $m = \mathsf{un}$. Then $\Delta \vdash \Gamma : \mathbf{T}^{\mathsf{un}}$ cannot contain active channel ends.

Case $\Lambda a{:}\,\kappa.v$. By inversion, $\Delta, a{:}\,\kappa \mid \Gamma \vdash v : U$ and $\varepsilon \mid \Delta \vdash T \simeq \forall a{:}\,\kappa\,.\,U : \mathbf{T}^{\mathsf{un}}$. By agreement $\Delta \vdash T : \mathbf{T}^m$, hence $m = \mathsf{un}$. Because type equivalence is symmetric, by agreement $\Delta \vdash \forall a{:}\,\mathbf{T}^{\mathsf{un}}\,.\,U : \mathbf{T}^{\mathsf{un}}$. Inverting type formation (using rules K-Suband K-TAbs), we have $\Delta, a{:}\,\mathbf{T}^{\mathsf{un}} \vdash U : \mathbf{T}^{\mathsf{un}}$. Weakening (Lemma 12) gives $\Delta, a{:}\,\mathbf{T}^{\mathsf{un}} \vdash \Gamma : \mathbf{s}^{\mathsf{lin}}$. The result follows by induction.

Case $\{\ell{=}v_\ell\}_{\ell\in L}$. Inversion gives $\Delta \vdash \Gamma = \circ\Gamma_\ell$, $\Delta \mid \Gamma_\ell \vdash v_\ell : T_\ell$ and $\varepsilon \mid \Delta \vdash T \simeq \{\ell{:}\,T_\ell\}_{\ell\in L} : \kappa$. Agreement gives $\Delta \vdash T : \kappa$, hence $\kappa = \mathbf{T}^{\mathsf{un}}$. From $\Delta \vdash \Gamma : \mathbf{s}^{\mathsf{lin}}$ and $\Delta \vdash \Gamma = \circ\Gamma_\ell$ we can show that $\Delta \vdash \Gamma_\ell : \mathbf{s}^{\mathsf{lin}}$. Inverting type formation (using rules K-Suband K-Record), we have $\Delta \vdash T_\ell : \mathbf{T}^{\mathsf{un}}$. The result follows by induction.

Case $k\,v$. Inversion gives $\Delta \mid \Gamma \vdash v : T_k$, $k \in L$ and $\varepsilon \mid \Delta \vdash T \simeq \langle\ell{:}\,T_\ell\rangle_{\ell\in L} : \kappa$. We proceed as above to establish that $\Delta \vdash T_k : \mathbf{T}^{\mathsf{un}}$ and the result follows by induction.

Cases $\mathsf{select}\,\ell$, $\mathsf{send}[U]$, $\mathsf{receive}[U]$ and $\mathsf{receive}[U][V]$. All these values are closed.

Case $\mathsf{send}[U]v$. Inversion gives $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$, $\Delta \mid \Gamma_1 \vdash \mathsf{send}[U] : V \to_m W$, $\Delta \mid \Gamma_2 \vdash v : V$ and $\varepsilon \mid \Delta \vdash T \simeq V : \kappa$. From $\Delta \vdash \Gamma : \mathbf{s}^{\mathsf{lin}}$ and $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$ by Lemma 9 we know that $\Delta \vdash \Gamma_2 : \mathbf{s}^{\mathsf{lin}}$. Because type equivalence is symmetric, by agree-

ment $\Delta \vdash V \to_{\mathsf{un}} W : \mathbf{T}^{\mathsf{un}}$. Inverting type formation (using rules K-Suband K-Arrow), we have $\Delta \vdash V : \mathbf{T}^{\mathsf{un}}$. Because $\mathsf{send}[U]$ is closed, the result follows by induction.

Case $\mathsf{send}[U]v[V]$. Similar to the above case. □

Corollary 28 *(Progress for single-threaded processes). Suppose that* $\Gamma \vdash \langle e \rangle$ *with* $\vdash \Gamma : \mathbf{S}^{\mathsf{lin}}$. *Then either*

1. *e is a value that contains no active channel end under* $\varepsilon \mid \Gamma$,
2. $\langle e \rangle$ *reduces, or*
3. $\langle e \rangle$ *is of the form* $\langle E[e'] \rangle$ *with* $e' = \mathsf{send}[U]v[V]x$, $\mathsf{receive}[U][V]x$, $\mathsf{select}\,\ell\,x$ *or* $\mathsf{match}\,x\,\mathsf{with}\,\{\ell \to e_\ell\}_{\ell \in L}$, *in which case process* $\langle e \rangle$ *is stuck.*

Proof. The premises of rule P-Expread $\varepsilon \mid \Gamma \vdash e : T$ and $\varepsilon \vdash T : \mathbf{T}^{\mathsf{un}}$. The statement mostly follows from Theorem 26. For item 1, we further use Lemma 27. For item 2, we further use rule R-Exp. For item 3, we note that while the expression $e = E[\mathsf{new}\,T]$ is stuck with respect to expression reduction, this expression reduces by rule R-Newwhen wrapped in a process: $\langle E[\mathsf{new}\,T] \rangle \to (\nu xy)\langle E[(x, y)] \rangle$. □

5. Type equivalence is decidable

This section focuses on the decidability of relation $T \simeq_s U$ under the following assumptions:

- $\Delta \vdash T : \mathbf{S}^m$ and $\Delta \vdash U : \mathbf{S}^m$, as per the premises of rule Q-ST, Fig. 7,
- In all subterms of T and U of the form $\mu a : \kappa . V$, type variable a occurs free in V, and
- Both types are α-renamed in such a way that they do not share bound variables.

Note that if a turns out not to occur free in V, then $\mu a : \kappa . V$ may be replaced by V, as per Lemma 5, item 4.

Decidability of type equivalence for context-free session types is inherited from the decidability of bisimilarity of basic process algebras (BPA) [17]. For this purpose, we translate context-free session types into BPA processes and we prove that the translation converts equivalent types to bisimilar processes.

Basic process algebra BPA processes are defined by the following grammar:

$$p ::= \alpha \mid X \mid p + p \mid p \cdot p \mid \varepsilon$$

where α ranges over a set of atomic actions, X is a BPA process variable, $+$ represents non-deterministic choice, and $\cdot$ represents sequential composition. The *terminated process* ε is the neutral element of sequential composition. We use $\sum_{i \in I} p_i$ when referring to an arbitrary number of non-deterministic choices.

Recursive BPA processes are defined by means of process equations $\Sigma = \{X_i \triangleq p_i\}_{i \in I}$, where X_i are distinct process variables. The empty process ε cannot occur in a process definition; we included it as part of the syntax because the operational semantics takes advantage of it [17] and, in the scope of our translation, ε is the natural candidate for the image of type Skip and other terminated types. A process equation is *guarded* if any variable occurring in p_i is in the scope of an atomic action. We are interested in guarded equations only.

The labelled transition relation on a set Σ of guarded equations is defined by the following rules.

$$\text{BPA-Var}\ \frac{}{\alpha \xrightarrow{\alpha} \varepsilon} \qquad \text{BPA-Ch1}\ \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \text{BPA-Ch2}\ \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'}$$

$$\text{BPA-Seq1}\ \frac{p \xrightarrow{\alpha} p' \quad p' \neq \varepsilon}{p \cdot q \xrightarrow{\alpha} p' \cdot q} \qquad \text{BPA-Seq2}\ \frac{p \xrightarrow{\alpha} \varepsilon}{p \cdot q \xrightarrow{\alpha} q} \qquad \text{BPA-Def}\ \frac{p \xrightarrow{\alpha} p' \quad X \triangleq p \in \Sigma}{X \xrightarrow{\alpha} p'}$$

The bisimulation associated to this labelled transition system is denoted by $\approx_\Sigma$. We omit the subscript when it is clear from context.

Translating context-free session types to BPA processes Types are translated in a three-step procedure: we start by identifying the recursive subterms of a given type, then we translate types to BPA processes and, finally, we introduce equations for BPA process variables.

The first step identifies the μ-subterms of a given type T. Assume that $\mathrm{sub}(T) = \{\mu a_1 : \kappa_1 . T_1, \ldots, \mu a_n : \kappa_n . T_n\}$ is the set of all μ-subterms in T.

The second step translates type T to a BPA process. To each recursive type $\mu a_i : \kappa_i . T_i$ in $\mathrm{sub}(T)$ we associate a distinct process variable X_i. The set of atomic actions of BPA processes is instantiated with $\sharp T$, $\star \ell$ and b, where b denotes a polymorphic variable.

To ensure that the processes obtained are well-defined, we need to remove any occurrences of the empty process ε from process definitions. For the purpose, we introduce a *special* sequential composition operator $\odot$ that filters occurrences of ε.

$$p \odot \varepsilon = p \qquad \varepsilon \odot q = q \qquad p \odot q = p \cdot q$$

Then, the translation is as follows.

$$[\![\mathsf{Skip}]\!] = \varepsilon \qquad [\![\sharp T]\!] = \sharp T \qquad [\![\star\{\ell\colon T_\ell\}_{\ell\in L}]\!] = \sum_{\ell\in L} \star\ell \odot [\![T_\ell]\!]$$

$$[\![T; U]\!] = [\![T]\!] \odot [\![U]\!] \qquad [\![b]\!] = b \qquad [\![a_i]\!] = X_i \qquad [\![\mu a_i\colon \kappa_i.T_i]\!] = X_i$$

The terminated type Skip is translated to the terminated process ε, message exchanges and polymorphic variables are translated to the corresponding atomic actions, choices are translated to process choices, sequential compositions are converted into sequential compositions through $\odot$ and each recursive type is translated to a corresponding process variable X_i.

The third step translates each of the bodies of the n μ-subterms in the type. To ensure that the equations are guarded, we perform a preliminary *unravel* of each type. Intuitively, the unravel function $\mathrm{unr}(\cdot)$ consists of unfolding recursive types until exposing a non-recursive type constructor. In the process we eliminate occurrences of type Skip.

$$\frac{\mathrm{unr}(T) = \mathsf{Skip}}{\mathrm{unr}(T; U) = \mathrm{unr}(U)} \qquad \frac{\mathrm{unr}(T) \neq \mathsf{Skip}}{\mathrm{unr}(T; U) = \mathrm{unr}(T); U}$$

$$\mathrm{unr}(\mu a\colon \kappa.T) = \mathrm{unr}(T) \qquad \frac{T \neq (U; V), \mu a\colon \kappa.U}{\mathrm{unr}(T) = T}$$

The equations for the μ-subterms of T are obtained by translating the unravelled μ-subterms:

$$\Sigma_T = \{X_1 \triangleq [\![\mathrm{unr}(T_1)]\!], \ldots, X_n \triangleq [\![\mathrm{unr}(T_n)]\!]\}$$

As an example of the translation procedure, consider type

$$T = !\,\mathsf{Char}; \mu a_2\colon \mathbf{s}^{\mathsf{lin}}.T_2$$

with $T_2 = \mathsf{Skip}; \mu a_1\colon \mathbf{s}^{\mathsf{lin}}.T_1$ and $T_1 = !\,\mathsf{Int}; a_1; ?\,\mathsf{Bool}; a_2$. The μ-subterms of T identified in step 1 are $\mathrm{sub}(T) = \{\mu a_1\colon \mathbf{s}^{\mathsf{lin}}.T_1, \mu a_2\colon \mathbf{s}^{\mathsf{lin}}.T_2\}$.

In step 2 we translate type T:

$$[\![!\,\mathsf{Char}; \mu a_2\colon \mathbf{s}^{\mathsf{lin}}.T_2]\!] = !\,\mathsf{Char} \cdot X_2$$

In step 3, we translate unravelled versions of bodies of the μ-types in $\mathrm{sub}(T)$:

$$[\![\mathrm{unr}(!\,\mathsf{Int}; a_1; ?\,\mathsf{Bool}; a_2)]\!] = !\,\mathsf{Int} \cdot X_1 \cdot ?\,\mathsf{Bool} \cdot X_2$$

$$[\![\mathrm{unr}(\mathsf{Skip}; \mu a_1\colon \mathbf{s}^{\mathsf{lin}}.T_1)]\!] = !\,\mathsf{Int} \cdot X_1 \cdot ?\,\mathsf{Bool} \cdot X_2$$

to obtain

$$\Sigma_T = \{X_1 \triangleq !\,\mathsf{Int} \cdot X_1 \cdot ?\,\mathsf{Bool} \cdot X_2,\ \ X_2 \triangleq !\,\mathsf{Int} \cdot X_1 \cdot ?\,\mathsf{Bool} \cdot X_2\}$$

Type equivalence is decidable The decidability of the type equivalence problem for context-free session types builds on the decidability of the bisimulation of BPA processes. We denote by $\Sigma_{T\cdot U}$ the system of BPA process equations resulting from the translation of types T and U. The α-identification convention allows choosing distinct recursion variables and hence distinct process variables.

Lemma 29. *If* $\Delta \vdash U : \mathbf{s}^m$ *then* $\mathrm{unr}(U)$ *terminates.*

Proof. By rule induction on the hypothesis. □

Lemma 30. *If* $\Delta \vdash U : \mathbf{s}^m$ *then* $\mathrm{unr}(U)$ *is* Skip *or a type of the form* $(\ldots(G; T_1); \ldots; T_n)$ *with* $n \geq 0$ *and*

$$G ::= \sharp T \mid \star\{\ell\colon T_\ell\}_{\ell\in L} \mid a$$

Proof. By rule induction on the hypothesis.

Case K-Msg. In this case, $U = \sharp T$ and the last unravelling rule applies. Cases K-Skip, K-Chand K-Varare similar and constitute the base cases for induction.

Case K-Seqfor $U = T; V$ has two subcases: if $\mathrm{unr}(T) = \mathsf{Skip}$ the proof follows by induction hypothesis on V, otherwise we apply induction hypothesis on T.

Case K-Rec. Since $\Delta \vdash \mu a\colon \mathbf{s}^{\mathsf{lin}}.T : \mathbf{s}^{\mathsf{lin}}$, by K-Recwe have $\Delta, a\colon \mathbf{s}^{\mathsf{lin}} \vdash T : \mathbf{s}^{\mathsf{lin}}$ and we proceed by induction hypothesis.

Case K-Sub. Immediate by induction. □

Lemma 31. *Let $\vdash T : \mathbf{s}^m$. Then $[\![T]\!] = \varepsilon$ if and only if $\vdash T\ \checkmark$.*

Proof. The forward implication follows by rule induction on the formation of type T such that $[\![T]\!] = \varepsilon$, noting that $[\![T]\!] = \varepsilon$ if $T = \mathsf{Skip}$ or $T = T_1; T_2$ with $[\![T_1]\!] = [\![T_2]\!] = \varepsilon$. For the reverse direction, assume that $[\![T]\!] \neq \varepsilon$; we show that $\nvdash T\ \checkmark$. By definition of the translation, since $[\![T]\!] \neq \varepsilon$, T takes one of the following forms: $\sharp U$, $\star\{\ell\colon T_\ell\}_{\ell\in L}$, $\mu a_i\colon \kappa_i.T_i$ or $U; W$. (Notice that the cases for variables are not applicable because $\vdash T : \mathbf{s}^m$.) According to the rules in Fig. 2, cases $\sharp U$ and $\star\{\ell\colon T_\ell\}_{\ell\in L}$ immediately imply $\nvdash T\ \checkmark$. If $T = \mu a_i\colon \kappa_i.T_i$, since a_i occurs free in T_i (as assumed at the beginning of this section) and T_i is contractive on a_i, we also have $\nvdash T\ \checkmark$. The conclusion that $\nvdash U; W\ \checkmark$ follows from the previous observations, using Te-Seq. □

Lemma 32. *If $\nvdash T\ \checkmark$ and $\nvdash U\ \checkmark$, then the BPA process equations in $\Sigma_{T\cdot U}$ are guarded.*

Proof. Immediate from the definition of $\Sigma_{T\cdot U}$, using Lemmas 30 and 31. □

The following result helps in showing that the translation preserves bisimulation.

Lemma 33. *If $\mathcal{A} \mid \Delta \vdash T : \mathbf{s}^m$ and $[\![T]\!] \xrightarrow{\alpha} p$ then $p = [\![T']\!]$ for T' such that $T \xrightarrow{\alpha} T'$.*

Proof. By case analysis on the definition of the translation. If $T = \mu a_i\colon \kappa.U$, then $[\![T]\!] = X_i$ and $(X_i \triangleq [\![\mathrm{unr}(U)]\!]) \in \Sigma_T$. Note that, by rule BPA-Def, the labelled transitions of $[\![T]\!]$ are the transitions of $[\![\mathrm{unr}(U)]\!]$. By Lemma 30, we have four subcases to analyse.

Subcase $\mathrm{unr}(U) = \mathsf{Skip}$. Does not have any transition.

Subcase $\mathrm{unr}(U) = \sharp T; T_1; \cdots; T_n$. In this case we have $[\![\mathrm{unr}(U)]\!] = [\![\sharp T]\!] \odot [\![T_1]\!] \cdots [\![T_n]\!] \xrightarrow{\sharp T} [\![T_1]\!] \odot \cdots \odot [\![T_n]\!] = [\![T_1; \ldots; T_n]\!]$ and $T' = T_1; \ldots; T_n$.

Subcase $\mathrm{unr}(U) = \star\{\ell\colon U_\ell\}_{\ell\in L}; T_1; \ldots; T_n$. In this case we have $[\![\mathrm{unr}(U)]\!] = (\sum_{\ell\in L} \star\ell \odot [\![U_\ell]\!]) \odot [\![T_1]\!] \cdots [\![T_n]\!] \xrightarrow{\star\ell} [\![U_\ell]\!] \odot [\![T_1]\!] \cdots [\![T_n]\!] = [\![U_\ell; T_1; \ldots; T_n]\!]$ and we have $T' = U_\ell; T_1; \ldots; T_n$.

Subcase $\mathrm{unr}(U) = a_j; T_1; \ldots; T_n$, with $j \neq i$. In this case we have $[\![\mathrm{unr}(U)]\!] = X_j \odot [\![T_1]\!] \cdots [\![T_n]\!]$, with $(X_j \triangleq [\![\mathrm{unr}(U_j)]\!]) \in \Sigma_T$. Since types are contractive, we have $[\![\mathrm{unr}(U)]\!] = X_j \odot [\![T_1]\!] \cdots [\![T_n]\!] \xrightarrow{\alpha} [\![U'_j]\!] \odot [\![T_1]\!] \cdots [\![T_n]\!] = [\![U'_j; T_1; \ldots; T_n]\!]$ and $T' = U'_j; T_1; \ldots; T_n$.

In all these cases, $\mathrm{unr}(U) \xrightarrow{\alpha} T'$ and, thus, $T \xrightarrow{\alpha} T'[T/a_i]$. Noting that $[\![T']\!] = [\![T'[T/a_i]]\!]$, we conclude that $[\![T]\!] \xrightarrow{\alpha} [\![T'[T/a_i]]\!]$. If $T \neq \mu a\colon \kappa.U$, we proceed similarly. □

Lemma 34. *The translation $[\![\cdot]\!]$ is a fully abstract encoding, i.e., $T \simeq_s U$ if and only if $[\![T]\!] \approx_{\Sigma^*} [\![U]\!]$, where Σ^* is a system of process equations such that $\Sigma_{T\cdot U} \subseteq \Sigma^*$.*

Proof. For the forward direction, let $\mathcal{S}$ be a bisimulation for T and U. Consider the relation $\mathcal{R} = \{([\![T_0]\!], [\![U_0]\!]) \mid (T_0, U_0) \in \mathcal{S}\}$ and the set of process equations $\Sigma^* = \bigcup_{(T_0,U_0)\in\mathcal{S}} \Sigma_{T_0\cdot U_0}$. We have $([\![T]\!], [\![U]\!]) \in \mathcal{R}$. To prove that $\mathcal{R}$ is a bisimulation for $[\![T]\!]$ and $[\![U]\!]$, we just need to prove that given $([\![T_0]\!], [\![U_0]\!]) \in \mathcal{R}$, if $[\![T_0]\!] \xrightarrow{\alpha} p$, then $[\![U_0]\!] \xrightarrow{\alpha} p'$ with $(p, p') \in \mathcal{R}$, and vice-versa. To prove that U_0 mimics T_0, we use the definition of $[\![T_0]\!]$, Lemma 33, and the fact that $\mathcal{S}$ is a bisimulation. For instance, if $T_0 = \mu a_i\colon \kappa.T'_0$, we have $[\![T_0]\!] = X_i$ and $(X_i \triangleq [\![\mathrm{unr}(T'_0)]\!]) \in \Sigma^*$. The proof proceeds by case analysis on the structure of $\mathrm{unr}(T'_0)$, using Lemma 30. We detail the subcase $\mathrm{unr}(T'_0) = \sharp S; S_1; \ldots; S_k$. In this case, $\alpha = \sharp S$ and $p = [\![\mathsf{Skip}; S_1; \ldots; S_k]\!]$ and $T_0 \xrightarrow{\sharp S} (\mathsf{Skip}; S_1; \ldots; S_k)[S_0/a]$. Since $\mathcal{S}$ is a bisimulation then $U_0 \xrightarrow{\sharp S} U'_0$ and $((\mathsf{Skip}; S_1; \ldots; S_k)[T_0/a], U'_0) \in \mathcal{R}$. Hence, $\mathrm{unr}(U_0)$ is of the form $\sharp S; U'_0$ and $[\![\mathrm{unr}(U_0)]\!] \xrightarrow{\sharp S} [\![U'_0]\!]$. Furthermore, $([\![(\mathsf{Skip}; S_1; \ldots; S_k)[T_0/a]]\!], [\![U']\!]) \in \mathcal{R}$. Since, by definition of the translation, $[\![T_0]\!] = [\![a_i]\!]$, we have $[\![(\mathsf{Skip}; S_1; \ldots; S_k)[T_0/a]]\!] = [\![\mathsf{Skip}; S_1; \ldots; S_k]\!]$ and we conclude that $([\![\mathsf{Skip}; S_1; \ldots; S_k]\!], [\![U']\!]) \in \mathcal{R}$. The other subcases for $\mathrm{unr}(T'_0)$ and the case where $T_0 \neq \mu a\colon \kappa.T'_0$ are similar, as well as the analysis for U_0.

Reciprocally, let $\mathcal{R}$ be a bisimulation for $[\![T]\!]$ and $[\![U]\!]$ and let us fix $\mathrm{sub}(T) = \{\mu a_1\colon \kappa_1.T_1, \ldots, \mu a_n\colon \kappa_n.T_n\}$ and $\mathrm{sub}(U) = \{\mu b_1\colon \kappa'_1.U_1, \ldots, \mu b_m\colon \kappa'_m.U_m\}$. Consider the relation $\mathcal{S} = \{(T_0, U_0) \mid ([\![T_0]\!], [\![U_0]\!]) \in \mathcal{R}, T_0 \neq a_i, U_0 \neq b_j\}$. Obviously, $(T, U) \in$

$\mathcal{S}$. Hence, to prove that $\mathcal{S}$ is a bisimulation, we just need to prove that any simulation step of T_0, $T_0 \xrightarrow{\alpha} T_0'$, can be mimicked by U_0, $U_0 \xrightarrow{\alpha} U_0'$, with $(T_0', U_0') \in \mathcal{R}$, and vice-versa. The proof is done by case analysis on T_0 and U_0, using the rules of Fig. 6, Lemma 33, and the definition of $[\![\cdot]\!]$. □

Theorem 35. *The type equivalence problem for context-free session types is decidable.*

Proof. By Lemma 34, the full abstraction $[\![\cdot]\!]$ reduces the type equivalence problem of context-free session types to the bisimulation of BPA processes. The latter was proved to be decidable by Christensen et al. [17]. Hence, type equivalence for context-free session types is decidable. □

6. Algorithmic type checking

This section presents a bidirectional procedure [50] to decide expression formation as in Fig. 12.

6.1. Algorithmic kinding, algorithmic equivalence and type normalisation

Kinding The rules for algorithmic type formation are in Fig. 16. Sequents are of the form $\Delta_{\text{in}} \vdash T_{\text{in}} \Rightarrow \kappa_{\text{out}}$ for kind synthesis and $\Delta_{\text{in}} \vdash T_{\text{in}} \Leftarrow \kappa_{\text{in}}$ for the check against relation. Marks in and out describe the input/output mode of parameters. Algorithmic kinding relies on contractivity (Fig. 3), which in turn relies on the terminated relation for session (Fig. 2), both of which are algorithmic.

Rules mostly follow their declarative counterparts in Fig. 5. For example, the rule for sequential composition, KA-Seq, requires both types to be session types, and returns kind $\mathbf{S}^{m \sqcup n}$, where $m \sqcup n$ is the least upper bound of the multiplicities of the synthesised kinds. Rule KA-Msgchecks that the type is of message kind and returns kind $\mathbf{S}^{\text{lin}}$. Rule KA-Chchecks that all the elements of a choice type are themselves session types, by checking against kind $\mathbf{S}^{\text{lin}}$ and returns kind $\mathbf{S}^{\text{lin}}$.

As with the declarative rules, recursive types $\mu a\colon \kappa.T$ are both session and functional. Rule KA-Recchecks whether type T is contractive on type variable a. Then, it ensures that T has the kind of a by checking T against κ. Rule KA-TAbschecks whether type T is well formed under contexts $\mathcal{A}, a$ and $\Delta, a\colon \kappa$. Rules KA-Recordand KA-Variantboth check that all components are well-formed. The result is kind $\mathbf{T}^{\sqcup m_\ell}$, where $\sqcup m_\ell$ is the least upper bound of the multiplicities of the synthesised kinds. Rule KA-Varlooks for an entry $a\colon \kappa$ in Δ and returns κ. Rule KA-Arrowrequires both types T and U to be well-formed and return $\mathbf{T}^m$, where m stands for the function multiplicity. Skip has kind $\mathbf{S}^{\text{un}}$ and $()_m$ has kind $\mathbf{M}^m$.

We now look at the correctness of algorithmic type formation.

Lemma 36 *(Kinding correctness).*

1. *(Soundness, synthesis) If* $\Delta \vdash T \Rightarrow \kappa$, *then* $\Delta \vdash T : \kappa$.
2. *(Soundness, check against) If* $\Delta \vdash T \Leftarrow \kappa$, *then* $\Delta \vdash T : \kappa$.
3. *(Completeness, synthesis) If* $\Delta \vdash T : \kappa$, *then* $\Delta \vdash T \Rightarrow \kappa'$ *with* $\kappa' <: \kappa$.
4. *(Completeness, check against) If* $\Delta \vdash T : \kappa$, *then* $\Delta \vdash T \Leftarrow \kappa$.

Proof. Straightforward mutual rule induction on each hypothesis. □

Equivalence The proper integration of context-free session types in a compiler requires the design and implementation of a type equivalence algorithm. The rules in Fig. 7 are algorithmic provided an algorithm to decide $T \simeq_s U$ is available. A practical algorithm to check type equivalence for context-free session types can be found in Almeida et al. [4].

Context operations Context difference, $\Delta_{\text{in}} \vdash \Gamma_{\text{in}} \div x_{\text{in}} \Rightarrow \Gamma_{\text{out}}$ in Fig. 17, checks that x is not linear and removes it from the incoming context.

Lemma 37 *(Properties of context difference). Let* $\Delta \vdash \Gamma_1 \div x \Rightarrow \Gamma_2$.

1. $\Gamma_2 = \Gamma_1 \backslash \{x\colon T\}$
2. $\mathcal{L}(\Gamma_1) = \mathcal{L}(\Gamma_2)$
3. *If* $x\colon T \in \Gamma_1$, *then* $\Delta \vdash T : \mathbf{T}^{\text{un}}$ *and* $x \notin \text{dom}(\Gamma_2)$

Proof. By rule induction on the context difference hypothesis. □

Type normalisation Type normalisation, $T_{\text{in}} \Downarrow U_{\text{out}}$ in Fig. 18, is used in the elimination rules for algorithmic expression formation. Type normalisation exposes the top-level type constructor by unfolding recursive types, removing terminated types and adjusting associativity on sequential composition.

Kind synthesis and check-against $\boxed{\Delta_{\text{in}} \vdash T_{\text{in}} \Rightarrow \kappa_{\text{out}}}$ $\boxed{\Delta_{\text{in}} \vdash T_{\text{in}} \Leftarrow \kappa_{\text{in}}}$

$$\textsc{KA-Type}\ \frac{\text{dom}(\Delta) \mid \Delta \vdash T \Rightarrow \kappa}{\Delta \vdash T \Rightarrow \kappa} \qquad \textsc{KA-Type-Against}\ \frac{\text{dom}(\Delta) \mid \Delta \vdash T \Leftarrow \kappa}{\Delta \vdash T \Leftarrow \kappa}$$

Kind synthesis (inner) $\boxed{\mathcal{A}_{\text{in}} \mid \Delta_{\text{in}} \vdash T_{\text{in}} \Rightarrow \kappa_{\text{out}}}$

$$\textsc{KA-Skip}\ \mathcal{A} \mid \Delta \vdash \mathsf{Skip} \Rightarrow \mathbf{s}^{\mathsf{un}} \qquad \textsc{KA-Msg}\ \frac{\mathcal{A} \mid \Delta \vdash T \Leftarrow \mathbf{m}^{\mathsf{lin}}}{\mathcal{A} \mid \Delta \vdash \sharp T \Rightarrow \mathbf{s}^{\mathsf{lin}}} \qquad \textsc{KA-Ch}\ \frac{\mathcal{A} \mid \Delta \vdash T_\ell \Leftarrow \mathbf{s}^{\mathsf{lin}}}{\mathcal{A} \mid \Delta \vdash \star\{\ell\colon T_\ell\}_{\ell \in L} \Rightarrow \mathbf{s}^{\mathsf{lin}}}$$

$$\textsc{KA-Seq}\ \frac{\mathcal{A} \mid \Delta \vdash T \Rightarrow \mathbf{s}^m \qquad \mathcal{A} \mid \Delta \vdash U \Rightarrow \mathbf{s}^n}{\mathcal{A} \mid \Delta \vdash (T; U) \Rightarrow \mathbf{s}^{m \sqcup n}} \qquad \textsc{KA-Unit}\ \mathcal{A} \mid \Delta \vdash ()_m \Rightarrow \mathbf{m}^m$$

$$\textsc{KA-Arrow}\ \frac{\mathcal{A} \mid \Delta \vdash T \Rightarrow \kappa_1 \qquad \mathcal{A} \mid \Delta \vdash U \Rightarrow \kappa_2}{\mathcal{A} \mid \Delta \vdash (T \to_m U) \Rightarrow \mathbf{t}^m}$$

$$\textsc{KA-Record}\ \frac{\mathcal{A} \mid \Delta \vdash T_\ell \Rightarrow v^{m_\ell} \qquad (\forall \ell \in L)}{\mathcal{A} \mid \Delta \vdash \{\ell\colon T_\ell\}_{\ell \in L} \Rightarrow \mathbf{t}^{\sqcup m_\ell}} \qquad \textsc{KA-Variant}\ \frac{\mathcal{A} \mid \Delta \vdash T_\ell \Rightarrow v^{m_\ell} \qquad (\forall \ell \in L)}{\mathcal{A} \mid \Delta \vdash \langle \ell\colon T_\ell \rangle_{\ell \in L} \Rightarrow \mathbf{t}^{\sqcup m_\ell}}$$

$$\textsc{KA-TAbs}\ \frac{\mathcal{A}, a \mid \Delta, a\colon \kappa \vdash T \Rightarrow v^m}{\mathcal{A} \mid \Delta \vdash \forall a\colon \kappa\,.\,T \Rightarrow \mathbf{t}^m} \qquad \textsc{KA-Rec}\ \frac{\mathcal{A} \vdash a\,.\,T \qquad \mathcal{A} \mid \Delta, a\colon \kappa \vdash T \Leftarrow \kappa}{\mathcal{A} \mid \Delta \vdash \mu a\colon \kappa.T \Rightarrow \kappa} \qquad \textsc{KA-Var}\ \frac{a\colon \kappa \in \Delta}{\mathcal{A} \mid \Delta \vdash a \Rightarrow \kappa}$$

Kind check-against (inner) $\boxed{\mathcal{A}_{\text{in}} \mid \Delta_{\text{in}} \vdash T_{\text{in}} \Leftarrow \kappa_{\text{in}}}$

$$\textsc{KA-Against}\ \frac{\mathcal{A} \mid \Delta \vdash T \Rightarrow \kappa_1 \qquad \kappa_1 <: \kappa_2}{\mathcal{A} \mid \Delta \vdash T \Leftarrow \kappa_2}$$

Fig. 16. Algorithmic type formation (kinding).

Context difference $\boxed{\Delta_{\text{in}} \vdash \Gamma_{\text{in}} \div x_{\text{in}} \Rightarrow \Gamma_{\text{out}}}$

$$\frac{x\colon T \notin \Gamma}{\Delta \vdash \Gamma \div x \Rightarrow \Gamma} \qquad \frac{\mathcal{A} \mid \Delta \vdash T \Leftarrow \mathbf{t}^{\mathsf{un}}}{\Delta \vdash (\Gamma_1, x\colon T, \Gamma_2) \div x \Rightarrow \Gamma_1, \Gamma_2}$$

Fig. 17. Context difference.

Type concatenation $\boxed{T_{\text{in}} +\!+ T_{\text{in}} = T_{\text{out}}}$

$$T +\!+ \mathsf{Skip} = T \qquad \frac{V \neq \mathsf{Skip}}{(T; U) +\!+ V = T; (U +\!+ V)} \qquad \frac{T \neq V; W \qquad U \neq \mathsf{Skip}}{T +\!+ U = T; U}$$

Type normalisation $\boxed{T_{\text{in}} \Downarrow T_{\text{out}}}$

$$\frac{\vdash T \checkmark \qquad U \Downarrow U'}{T; U \Downarrow U'} \qquad \frac{\not\vdash T \checkmark \qquad T \Downarrow T'}{T; U \Downarrow T' +\!+ U}$$

$$\frac{T \Downarrow T'}{\mu a\colon \kappa.T \Downarrow T'[\mu a\colon \kappa.T'/a]} \qquad \frac{T \neq (U; V), (\mu x\colon \kappa.U)}{T \Downarrow T}$$

Fig. 18. Type normalisation.

Lemma 38 *(Type normalisation). If $\Delta \vdash T : \kappa$, then $T \Downarrow U$ and $\varepsilon \mid \Delta \vdash T \simeq U : \kappa$. Furthermore, U does not start with μ and if $U = V; W$ then V is a message or a choice.*

Proof. By rule induction on the hypothesis. Consider the case where the derivation ends with rule K-Rec. We know that $\mathcal{A} \vdash a\,.\,T$ and $\mathcal{A} \mid \Delta, a\colon \kappa \vdash T : \kappa$. By induction hypothesis we obtain $T \Downarrow T'$ and $\varepsilon \mid \Delta, a\colon \kappa \vdash T \simeq T' : \kappa$ where T' does not start with μ and if $T' = V; W$, then V is either a message or a choice. By applying the third rule of type normalisation (Fig. 18) we get $\mu a\colon \kappa.T \Downarrow T'[\mu a\colon \kappa.T'/a]$. We conclude that $\varepsilon \mid \Delta, a\colon \kappa \vdash \mu a\colon \kappa.T \simeq T'[\mu a\colon \kappa.T'/a] : \kappa$ via the properties of type bisimilarity.

When the derivation ends with rule K-Seq, we consider two cases for $T = T_1; T_2$, regarding whether T_1 is terminated or not. Take the latter as an example. We have that $\mathcal{A} \mid \Delta \vdash T_1 : \mathbf{s}^{\mathsf{lin}}$ and $\mathcal{A} \mid \Delta \vdash T_2 : \mathbf{s}^{\mathsf{lin}}$. Induction hypothesis on the

Type synthesis $\boxed{\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} \Rightarrow T_{\text{out}} \mid \Gamma_{\text{out}}}$

$$\textsc{TA-Const}\quad \Delta \mid \Gamma \vdash c \Rightarrow \text{typeof}(c) \mid \Gamma$$

$$\textsc{TA-LinVar}\quad \frac{\Delta \vdash T \Rightarrow \upsilon^{\text{lin}}}{\Delta \mid \Gamma, x\colon T \vdash x \Rightarrow T \mid \Gamma} \qquad \textsc{TA-UnVar}\quad \frac{\Delta \vdash T \Rightarrow \upsilon^{\text{un}}}{\Delta \mid \Gamma, x\colon T \vdash x \Rightarrow T \mid \Gamma, x\colon T}$$

$$\textsc{TA-LinAbs}\quad \frac{\Delta \vdash T_1 \Rightarrow \_ \qquad \Delta \mid \Gamma_1, x\colon T_1 \vdash e \Rightarrow T_2 \mid \Gamma_2}{\Delta \mid \Gamma_1 \vdash (\lambda_{\text{lin}} x\colon T_1.e) \Rightarrow (T_1 \to_{\text{lin}} T_2) \mid \Gamma_2 \div x}$$

$$\textsc{TA-UnAbs}\quad \frac{\Delta \vdash T_1 \Rightarrow \_ \qquad \Delta \mid \Gamma_1, x\colon T_1 \vdash e \Rightarrow T_2 \mid \Gamma_2 \qquad \Gamma_1 = \Gamma_2 \div x}{\Delta \mid \Gamma_1 \vdash (\lambda_{\text{un}} x\colon T_1.e) \Rightarrow (T_1 \to_{\text{un}} T_2) \mid \Gamma_1}$$

$$\textsc{TA-App}\quad \frac{\Delta \mid \Gamma_1 \vdash e_1 \Rightarrow\Downarrow T \to_m U \mid \Gamma_2 \qquad \Delta \mid \Gamma_2 \vdash e_2 : T \Rightarrow \Gamma_3}{\Delta \mid \Gamma_1 \vdash e_1\, e_2 \Rightarrow U \mid \Gamma_3}$$

$$\textsc{TA-TAbs}\quad \frac{\Delta, a\colon \kappa \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2 \qquad a \notin \text{free}(\Gamma_2)}{\Delta \mid \Gamma_1 \vdash \Lambda a\colon \kappa.e \Rightarrow \forall a\colon \kappa\, .\, T \mid \Gamma_2}$$

$$\textsc{TA-TApp}\quad \frac{\Delta \mid \Gamma_1 \vdash e \Rightarrow\Downarrow \forall a\colon \kappa\, .\, U \mid \Gamma_2 \qquad \Delta \vdash T \Leftarrow \kappa}{\Delta \mid \Gamma_1 \vdash e\,[T] \Rightarrow U[T/a] \mid \Gamma_2}$$

Fig. 19. Algorithmic type checking (synthesis).

first premise yields $T_1 \Downarrow T_1'$ and $\varepsilon \mid \Delta \vdash T_1 \simeq T_1' : \kappa$. Since $\not\vdash T_1 \checkmark$, we apply the second rule of type normalisation to obtain $T_1; T_2 \Downarrow T_1' +\!\!+ T_2$. Now we have to prove three different cases depending on the nature of the type concatenation operation. The first two follow by induction hypothesis, using the properties of type bisimilarity (Lemma 5) and the last follows by reflexivity.

All the other cases follow directly by the last rule of type normalisation. □

6.2. Algorithmic type checking

The typing rules in Fig. 12 provide for a declarative description of well-typed programs. They cannot however be directly implemented due to two difficulties:

- The non-deterministic nature of context split, and
- The necessity of guessing the type for the injection expression operators, rules T-Variantand T-Selin Fig. 12.

We address the first difficulty by restructuring the typing rules so that we do not have to guess the correct split. Rather than guessing how to split the context for each subexpression, we use the entire context when checking the first subexpression, obtaining as a result the unused part, which is then passed to the next subexpression [43,64,69]. To avoid guessing the type for the two injection expressions, we assume these are explicitly typed, as in $\mathsf{select}\, k^{\oplus\{\ell\colon T_\ell\}_{\ell\in L}}\, e$ and $k^{\langle \ell\colon T_\ell\rangle_{\ell\in L}}\, e$.

The type synthesis and check-against rules are in Figs. 19 and 20. The judgements are now of form $\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} \Rightarrow T_{\text{out}} \mid \Gamma_{\text{out}}$ for the synthesis relation, and $\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} : T_{\text{in}} \Rightarrow \Gamma_{\text{out}}$ for the check against relation. We abbreviate type synthesis $\Delta \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$ followed by normalisation $T \Downarrow U$ as $\Delta \mid \Gamma_1 \vdash e \Rightarrow\Downarrow U \mid \Gamma_2$. Most of the rules are simple adaptations of the declarative typing rules to a bidirectional context. We describe the most relevant. Rule TA-LinAbssynthesises the kind of T_1. Then, it synthesises a type for the body of the abstraction under the extended context $\Gamma_1, x\colon T_1$. The result is a type and a context Γ_2, from which the variable x should be removed, represented by $\Gamma_2 \div x$. The rule TA-UnAbsis similar but it checks that no linear resources in the original context Γ_1 are used, a constraint ensured by $\Gamma_1 = \Gamma_2 \div x$.

The rule TA-Appfirst synthesises type T from expression e_1. Type normalisation exposes the top-level type constructor from type T which is expected to be a function type $T \to_m U$. Then, it checks function e_2 against argument T; the result is type U. Rule TA-Case starts by synthesising a type T for expression e that normalises to a type of the form $\langle \ell\colon T_\ell\rangle_{\ell\in L}$. The rule then synthesises a type for each branch; they should all be functions $T_\ell \to_{\text{lin}} U_\ell$. To ensure that all branches are equivalent, the rule selects a label k in L and checks that $\varepsilon \mid \Delta \vdash U_k \simeq U_\ell : \mathbf{T}^{\text{lin}}$ for all ℓ in L. To ensure that all branches consume the same linear resources, the rule also checks that the final contexts are equivalent, $\varepsilon \mid \Delta \vdash \Gamma_k \simeq \Gamma_\ell : \mathbf{T}^{\text{lin}}$.

Type synthesis $\boxed{\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} \Rightarrow T_{\text{out}} \mid \Gamma_{\text{out}}}$

TA-Record
$$\frac{\Delta \mid \Gamma_1 \vdash e_1 \Rightarrow T_1 \mid \Gamma_2 \qquad \cdots \qquad \Delta \mid \Gamma_n \vdash e_n \Rightarrow T_n \mid \Gamma_{n+1}}{\Delta \mid \Gamma_1 \vdash \{\ell{=}e_\ell\}_{\ell\in\{1,\ldots,n\}} \Rightarrow \{\ell\colon T_\ell\}_{\ell\in\{1,\ldots,n\}} \mid \Gamma_{n+1}}$$

TA-Proj
$$\frac{\Delta \mid \Gamma_1 \vdash e_1 \Rightarrow\Downarrow \{\ell\colon T_\ell\}_{\ell\in L} \mid \Gamma_2 \qquad \Delta \mid \Gamma_2, (x_\ell\colon T_\ell)_{\ell\in L} \vdash e_2 \Rightarrow U \mid \Gamma_3}{\Delta \mid \Gamma_1 \vdash \mathsf{let}\,\{\ell{=}x_\ell\}_{\ell\in L} = e_1\,\mathsf{in}\,e_2 \Rightarrow U \mid \Gamma_3 \div (x_\ell)_{\ell\in L}}$$

TA-UnitElim
$$\frac{\Delta \mid \Gamma_1 \vdash e_1 : ()_m \Rightarrow \Gamma_2 \qquad \Delta \mid \Gamma_2 \vdash e_2 \Rightarrow T \mid \Gamma_3}{\Delta \mid \Gamma_1 \vdash \mathsf{let}\,()_m = e_1\,\mathsf{in}\,e_2 \Rightarrow T \mid \Gamma_3}$$

TA-Variant
$$\frac{\Delta \vdash \langle \ell\colon T_\ell\rangle_{\ell\in L} \Leftarrow \mathbf{T}^{\mathsf{lin}} \qquad \Delta \mid \Gamma_1 \vdash e : T_k \Rightarrow \Gamma_2 \qquad k \in L}{\Delta \mid \Gamma_1 \vdash k^{\langle \ell\colon T_\ell\rangle_{\ell\in L}}\, e \Rightarrow \langle \ell\colon T_\ell\rangle_{\ell\in L} \mid \Gamma_2}$$

TA-Case
$$\frac{\begin{array}{l}\Delta \mid \Gamma_1 \vdash e \Rightarrow\Downarrow \langle \ell\colon T_\ell\rangle_{\ell\in L} \mid \Gamma_2 \qquad \Delta \mid \Gamma_2 \vdash e_\ell \Rightarrow\Downarrow T_\ell \to_{\mathsf{lin}} U_\ell \mid \Gamma_\ell \\ \varepsilon \mid \Delta \vdash U_k \simeq U_\ell : \mathbf{T}^{\mathsf{lin}} \qquad \varepsilon \mid \Delta \vdash \Gamma_k \simeq \Gamma_\ell : \mathbf{T}^{\mathsf{lin}} \qquad k \in L \qquad (\forall \ell \in L)\end{array}}{\Delta \mid \Gamma_1 \vdash \mathsf{case}\,e\,\mathsf{of}\,\{\ell \to e_\ell\}_{\ell\in L} \Rightarrow U_k \mid \Gamma_k}$$

TA-New
$$\frac{\varepsilon \vdash T \Leftarrow \mathbf{S}^{\mathsf{lin}}}{\Delta \mid \Gamma \vdash \mathsf{new}\,T \Rightarrow \{\mathsf{fst}\colon T, \mathsf{snd}\colon \overline{T}\} \mid \Gamma}$$

TA-Sel
$$\frac{\Delta \vdash \oplus\{\ell\colon T_\ell\}_{\ell\in L} \Leftarrow \mathbf{S}^{\mathsf{lin}} \qquad (k \in L)}{\Delta \mid \Gamma \vdash \mathsf{select}\,k^{\oplus\{\ell\colon T_\ell\}_{\ell\in L}} \Rightarrow (\oplus\{\ell\colon T_\ell\}_{\ell\in L} \to_m T_k) \mid \Gamma}$$

TA-Match
$$\frac{\begin{array}{l}\Delta \mid \Gamma_1 \vdash e \Rightarrow\Downarrow \&\{\ell\colon T_\ell\}_{\ell\in L} \mid \Gamma_2 \qquad \Delta \mid \Gamma_2 \vdash e_\ell \Rightarrow\Downarrow T_\ell \to_{\mathsf{lin}} U_\ell \mid \Gamma_\ell \\ \varepsilon \mid \Delta \vdash U_k \simeq U_\ell : \mathbf{T}^{\mathsf{lin}} \qquad \varepsilon \mid \Delta \vdash \Gamma_k \simeq \Gamma_\ell : \mathbf{T}^{\mathsf{lin}} \qquad k \in L \qquad (\forall \ell \in L)\end{array}}{\Delta \mid \Gamma_1 \vdash \mathsf{match}\,e\,\mathsf{with}\,\{\ell \to e_\ell\}_{\ell\in L} \Rightarrow U_k \mid \Gamma_k}$$

Type check-against $\boxed{\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} : T_{\text{in}} \Rightarrow \Gamma_{\text{out}}}$

TA-Eq
$$\frac{\Delta \mid \Gamma_1 \vdash e \Rightarrow U \mid \Gamma_2 \qquad \varepsilon \mid \Delta \vdash T \simeq U : \mathbf{T}^{\mathsf{lin}}}{\Delta \mid \Gamma_1 \vdash e : T \Rightarrow \Gamma_2}$$

Fig. 20. Algorithmic type checking (continued, synthesis and check against).

6.3. Correctness of algorithmic type checking

Soundness Algorithmic monotonicity relates the output against the input of the algorithmic type checking, and it is broadly used in the remaining proofs.

Lemma 39 *(Algorithmic monotonicity).*

1. *If* $\Delta \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$, *then* $\mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$ *and* $\mathcal{L}(\Gamma_2) \subseteq \mathcal{L}(\Gamma_1)$.
2. *If* $\Delta \mid \Gamma_1 \vdash e : T \Rightarrow \Gamma_2$, *then* $\mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$ *and* $\mathcal{L}(\Gamma_2) \subseteq \mathcal{L}(\Gamma_1)$.

Proof. The proof follows by mutual rule induction on the hypothesis, using the properties of context difference (Lemma 37). For item 1, we show the case when the derivation ends with TA-LinAbs. To show that $\mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2 \div x)$ and $\mathcal{L}(\Gamma_2 \div x) \subseteq \mathcal{L}(\Gamma_1)$, we start by applying induction hypothesis on the second premise and get $\mathcal{U}(\Gamma_1, x\colon T) = \mathcal{U}(\Gamma_2)$ and $\mathcal{L}(\Gamma_2) \subseteq \mathcal{L}(\Gamma_1, x\colon T)$. Then, we have two cases regarding the un/lin nature of T. Consider the case in which we have $\Delta \vdash T \Rightarrow \mathbf{T}^{\mathsf{un}}$. We know that $\mathcal{U}(\Gamma_1, x\colon T) \div x = \mathcal{U}(\Gamma_2) \div x$ and so, $\mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2 \div x)$. Since we have that $\mathcal{L}(\Gamma_2) = \mathcal{L}(\Gamma_2 \div x) \subseteq \mathcal{L}(\Gamma_1, x\colon T) = \mathcal{L}(\Gamma_1)$, we conclude $\mathcal{L}(\Gamma_2 \div x) \subseteq \mathcal{L}(\Gamma_1)$. The remaining cases are similar. □

Algorithmic linear strengthening allows removing unused linear assumptions from both the input context and the output contexts.

Lemma 40 *(Algorithmic linear strengthening). Suppose that* $\Delta \vdash U : \mathbf{T}^{\mathsf{lin}}$.

1. *If* $\Delta \mid \Gamma_1, x\colon U \vdash e \Rightarrow T \mid \Gamma_2, x\colon U$, *then* $\Delta \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$.

2. *If* $\Delta \mid \Gamma_1, x\colon U \vdash e : T \Rightarrow \Gamma_2, x\colon U$, *then* $\Delta \mid \Gamma_1 \vdash e : T \Rightarrow \Gamma_2$.

Proof. The proof is by mutual rule induction on both hypotheses and uses the properties of context difference (Lemma 37) and monotonicity (Lemma 39). Let us show one example. When the derivation ends with rule TA-App, suppose that $\Delta \mid \Gamma_1, x\colon U \vdash e_1\, e_2 \Rightarrow T_2 \mid \Gamma_3, x\colon U$. The premises to the rule are *(a)* $\Delta \mid \Gamma_1, x\colon U \vdash e_1 \Rightarrow\Downarrow T_1 \to_m T_2 \mid \Gamma_2$ and *(b)* $\Delta \mid \Gamma_2 \vdash e_2 : T_1 \Rightarrow \Gamma_3, x\colon U$. The first premise is an abbreviation for *(c)* $\Delta \mid \Gamma_1, x\colon U \vdash e_1 \Rightarrow T \mid \Gamma_2$ and *(d)* $T \Downarrow T_1 \to_m T_2$. Monotonicity gives $\mathcal{L}(\Gamma_3, x\colon U) \subseteq \mathcal{L}(\Gamma_2)$ and thus $x\colon T_1 \in \Gamma_2$. Let $\Gamma_2 = \Gamma_2', x\colon U$. Induction on items *(b)* and *(c)* yields *(e)* $\Delta \mid \Gamma_2' \vdash e_2 : T_1 \Rightarrow \Gamma_3$ and *(f)* $\Delta \mid \Gamma_1 \vdash e_1 \Rightarrow T \mid \Gamma_2'$, respectively. Conclude with rule TA-Append items *(d)* to *(f)*. □

Theorem 41 *(Algorithmic soundness). Suppose that* $\Delta \vdash \Gamma_2 : \mathbf{T}^{\mathsf{un}}$.

1. *If* $\Delta \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$, *then* $\Delta \mid \Gamma_1 \vdash e : T$.
2. *If* $\Delta \mid \Gamma_1 \vdash e : T \Rightarrow \Gamma_2$, *then* $\Delta \mid \Gamma_1 \vdash e : T$.

Proof. By mutual rule induction on the hypotheses.

Item 1. When the derivation ends with TA-LinVar, TA-UnVar, TA-New, or TA-Selwe use Algorithmic Kinding Soundness (Lemma 36) and the corresponding expression formation rules. Rules TA-App, TA-Record, TA-Proj, TA-Case, and TA-Matchrequire Monotonicity (Lemma 39) and Algorithmic Linear Strengthening (Lemma 40) to push through the result.

We detail the case for rule TA-Case. We strengthen the first premise to obtain $\Delta \mid \Gamma_1 - \mathcal{L}(\Gamma_2) \vdash e \Rightarrow T \mid \Gamma_2 - \mathcal{L}(\Gamma_2)$, then by induction hypothesis we get $\Delta \mid \Gamma_1 - \mathcal{L}(\Gamma_2) \vdash e : T$. Properties of type normalisation (Lemma 38) yield $\varepsilon \mid \Delta \vdash T \simeq \langle \ell\colon T_\ell \rangle_{\ell \in L} : \mathbf{T}^m$. Then, by applying rule T-Eq, we get $\Delta \mid \Gamma_1 - \mathcal{L}(\Gamma_2) \vdash e : \langle \ell\colon T_\ell \rangle_{\ell \in L}$. Using agreement for type equivalence lifted to typing contexts we get $\Delta \vdash \Gamma_\ell : \mathbf{T}^{\mathsf{un}}$, given that $\Delta \vdash \Gamma_k : \mathbf{T}^{\mathsf{un}}$ and $\varepsilon \mid \Gamma_k \vdash U_k \simeq U_\ell : \mathbf{T}^{\mathsf{un}}$. Then, we can apply induction hypothesis to the second premise, obtaining $\Delta \mid \Gamma_2 \vdash e_\ell : T_\ell \to_{\mathsf{lin}} U_\ell$. Since $\varepsilon \mid \Gamma \vdash U_k \simeq U_\ell : \mathbf{T}^{\mathsf{un}}$, we get $\Delta \mid \Gamma_2 \vdash e_\ell : T_\ell \to_{\mathsf{lin}} U_k$ for $k \in L$. Before concluding with rule T-Casewe need to establish that $\Gamma_1 = (\Gamma_1 - \mathcal{L}(\Gamma_2)) \circ \Gamma_2$ is in the context split relation. For the unrestricted portion monotonicity yields $\mathcal{U}(\Gamma_1 - \mathcal{L}(\Gamma_2)) = \mathcal{U}(\Gamma_2 - \mathcal{L}(\Gamma_2))$ which gives $\mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$. Regarding the linear entries we know that $\mathcal{L}(\Gamma_1 - \mathcal{L}(\Gamma_2)) \cap \mathcal{L}(\Gamma_2) = \emptyset$. Hence, $\mathcal{L}(\Gamma_1) = \mathcal{L}(\Gamma_1 - \mathcal{L}(\Gamma_2)) \cup \mathcal{L}(\Gamma_2)$.

Item 2. From the hypothesis, using TA-Eq, we know that $\Delta \mid \Gamma_1 \vdash e \Rightarrow U \mid \Gamma_2$ and $\varepsilon \mid \Delta \vdash T \simeq U : \mathbf{T}^{\mathsf{lin}}$. By item 1, we get $\Delta \mid \Gamma_1 \vdash e : U$. Conclude with T-Eq. □

Completeness Weakening allows adding new entries to both the input and output contexts.

Lemma 42 *(Algorithmic weakening).*

1. *If* $\Delta \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$, *then* $\Delta \mid \Gamma_1, x\colon U \vdash e \Rightarrow T \mid \Gamma_2, x\colon U$.
2. *If* $\Delta \mid \Gamma_1 \vdash e : T \Rightarrow \Gamma_2$, *then* $\Delta \mid \Gamma_1, x\colon U \vdash e : T \Rightarrow \Gamma_2, x\colon U$.

Proof. By mutual rule induction on the hypotheses. We show the case TA-LinAbs for item 1. By induction hypothesis we know that $\Delta \mid \Gamma_1, x\colon U, y\colon V \vdash e \Rightarrow T_2 \mid \Gamma_2, y\colon V$. From the properties of context difference (Fig. 17), we have $(\Gamma_2 \div x), y\colon V = (\Gamma_2, y\colon V) \div x$. Conclude applying the rule TA-LinAbs. □

Theorem 43 *(Algorithmic completeness).*

1. *If* $\Delta \mid \Gamma_1 \vdash e : T$, *then* $\Delta \mid \Gamma_1 \vdash e \Rightarrow U \mid \Gamma_2$ *and* $\Delta \vdash \Gamma_2 : \mathbf{T}^{\mathsf{un}}$ *and* $\varepsilon \mid \Delta \vdash T \simeq U : \kappa$.
2. *If* $\Delta \mid \Gamma_1 \vdash e : T$, *then* $\Delta \mid \Gamma_1 \vdash e : T \Rightarrow \Gamma_2$ *and* $\Delta \vdash \Gamma_2 : \mathbf{T}^{\mathsf{un}}$.

Proof. By mutual rule induction on the hypotheses.

Item 1. We detail the case for T-Case. Induction on the first premise gives *(a)* $\Delta \mid \Gamma_1 \vdash e \Rightarrow U \mid \Gamma_2'$, *(b)* $\Delta \vdash \Gamma_2' : \mathbf{T}^{\mathsf{un}}$, and *(c)* $\varepsilon \mid \Delta \vdash U \simeq \langle \ell\colon T_\ell \rangle_{\ell \in L} : \kappa$ Rules of Figs. 7 and 18 with *(a)* and *(c)* yield $U \Downarrow \langle \ell\colon T_\ell \rangle_{\ell \in L}$ and thus *(d)* $\Delta \mid \Gamma_1 \vdash e \Rightarrow\Downarrow \langle \ell\colon T_\ell \rangle_{\ell \in L} \mid \Gamma_2'$ Induction on the second premisse of T-Caseyields *(e)* $\Delta \mid \Gamma_2 \vdash e \Rightarrow V \mid \Gamma_\ell$, *(f)* $\Delta \vdash \Gamma_\ell : \mathbf{T}^{\mathsf{un}}$, and *(g)* $\varepsilon \mid \Delta \vdash V \simeq T_\ell \to_{\mathsf{lin}} T : \kappa$ Again, rules of Figs. 7 and 18 with *(e)* and *(g)* yield $V \Downarrow T_\ell \to_{\mathsf{lin}} T$ and thus *(h)* $\Delta \mid \Gamma_2 \vdash e_\ell \Rightarrow\Downarrow T_\ell \to_{\mathsf{lin}} T \mid \Gamma_\ell$ We have to prove that $\Gamma_2' = \Gamma_2$. Since $\Gamma_1 \cup \mathcal{L}(\Gamma_2) = \Gamma_1 \circ \Gamma_2$, we weaken *(d)* to obtain $\Delta \mid \Gamma_1 \circ \Gamma_2 \vdash e \Rightarrow\Downarrow \langle \ell\colon T_\ell \rangle_{\ell \in L} \mid \Gamma_2' \cup \mathcal{L}(\Gamma_2)$. We have to show that $\Gamma_2' \cup \mathcal{L}(\Gamma_2) = \Gamma_2$. We know that $\mathcal{L}(\Gamma_2' \cup \mathcal{L}(\Gamma_2)) = \mathcal{L}(\Gamma_2)$ since $\mathcal{L}(\Gamma_2') = \emptyset$. From monotonicity and properties of context split (Lemmas 39 and 10), we know that $\mathcal{U}(\Gamma_2' \cup \mathcal{L}(\Gamma_2)) = \mathcal{U}(\Gamma_2') = \mathcal{U}(\Gamma_1 \circ \Gamma_2) = \mathcal{U}(\Gamma_2)$. From *(h)* and reflexivity of type equivalence (Lemma 7) we obtain $\Theta \mid \Delta \vdash T \simeq T : \kappa$. Before concluding with rule TA-Casewe still have to show that $\varepsilon \mid \Delta \vdash \Gamma_k \simeq \Gamma_\ell : \mathbf{T}^{\mathsf{lin}}$. From monotonicity and *(h)* we get $\mathcal{U}(\Gamma_2) = \mathcal{U}(\Gamma_\ell)$. Since $\Delta \vdash \Gamma_\ell : \mathbf{T}^{\mathsf{un}}$ we can pick a $k \in L$ such that $\varepsilon \mid \Delta \vdash \Gamma_k \simeq \Gamma_\ell : \mathbf{T}^{\mathsf{un}}$. By subkinding we get $\varepsilon \mid \Delta \vdash \Gamma_k \simeq \Gamma_\ell : \mathbf{T}^{\mathsf{lin}}$.

Item 2. From item 2 we know that $\Delta \mid \Gamma_1 \vdash e \Rightarrow U \mid \Gamma_2$ and $\Delta \vdash \Gamma_2 : \mathbf{T}^{\mathsf{un}}$ and $\varepsilon \mid \Delta \vdash T \simeq U : \mathbf{T}^{\mathsf{lin}}$. Conclude with rule TA-Eq. □

Pragmatics and FreeST In the injection rules, TA-Variantand TA-Selexpressions are type-annotated. FreeST handles this by requiring variants to be declared in advance so that each datatype constructor becomes a function symbol in the symbol table, as usual in functional programming languages where variants and records are treated nominally.

The check against relation, defined in Fig. 20, may include more rules to produce better error messages. The following two rules are derivable, and we use them in the compiler.

$$\frac{\Delta \mid \Gamma_1 \vdash e_1 \Rightarrow T_1 \mid \Gamma_2 \qquad T_1 \Downarrow \{\mathsf{fst}\colon U_1, \mathsf{snd}\colon U_2\} \qquad \Delta \mid \Gamma_2, x\colon U_1, y\colon U_2 \vdash e_2 : T_2 \Rightarrow \Gamma_3}{\Delta \mid \Gamma_1 \vdash \mathsf{let}\,(x, y) = e_1\,\mathsf{in}\,e_2 : T_2 \Rightarrow \Gamma_3 \div x \div y}$$

$$\frac{\Delta \mid \Gamma_1 \vdash e_1 : \mathsf{Bool} \Rightarrow \Gamma_2 \qquad \Delta \mid \Gamma_2 \vdash e_2 : T \Rightarrow \Gamma_3 \qquad \Delta \mid \Gamma_2 \vdash e_3 : T \Rightarrow \Gamma_4 \qquad \Delta \vdash \Gamma_3 \simeq \Gamma_4}{\Delta \mid \Gamma_1 \vdash \mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3 : T \Rightarrow \Gamma_3}$$

Similarly to the translation to BPA (section 5), the algorithm to decide type equivalence [4] requires μ-introduced type variables to occur free in the body of the type, hence FreeST replaces types of the form $\mu a\colon \kappa.T$ by T when $a \notin \mathrm{free}(T)$. Furthermore, programs must be kept renamed so that the types inspected by type equivalence do not share bound variables. Substitution preserves neither of these two properties. The current version of FreeST ensures that the properties remain invariant throughout type checking by renaming and replacing μ-types at parsing and whenever it performs a substitution.

7. Related work

The related work on session types is vast. The BETTY book [26] provides a recent overview. Here we concentrate on direct influences to our design, systems that support polymorphism, and systems that go beyond tail recursive types for communication channels.

Binary session types Session types were introduced by Honda et al. [38,39,57] on variants of the π-calculus. These early ideas were later incorporated in functional languages by Gay et al. [32,65,66], who also enunciate the importance of linearity and linear types for session types. Our language is particularly close to that of Gay and Vasconcelos [32], which owes much of its elegance and generality to its reliance on linear types. At the level of the operational semantics we use a synchronous semantics for communication in place of a buffered asynchronous one. This choice simplifies the technical treatment, even if we believe that a buffered semantics can be reintroduced without compromising the metatheory of the language. At the level of types we incorporate record and variant types, general recursive types (not restricted to session types), a sequential composition operator for session types, and impredicative polymorphism. We omit subtyping as it requires a significant treatment on its own. Vasconcelos [63,64] introduces the syntactic distinction of the two ends of a channel, related by a single ν-binding, a technique we follow in this paper.

Languages with conventional (that is, regular) recursive session types [14,32,38,39,57,62–66] can still describe some of the protocols discussed in this paper, but they require higher-order session types (the ability to send channels on channels). For example, the session type TreeChannel introduced in section 2.3 can be emulated by the type

$$\mu a.\, \oplus\{\mathsf{Leaf}\colon \mathsf{end}, \mathsf{Node}\colon\, !\mathsf{Int}.!a.!a\}$$

where two new channels must be created to transmit the two subtrees at every node. In comparison, our calculus is intentionally closer to a low level language: it only supports the transmission of non-session typed values. Furthermore its implementation is simpler and more efficient: only one channel is created and used for the transmission of the whole tree; thus, avoiding the overhead of multiple channel creation.

Context-free session types As an alternative to checking type equivalence for arbitrary context-free types, Padovani [46, 47] proposes a language that requires explicit annotations in the source code. These annotations result in the structural alignment between code and types, thus simplifying the problem of type checking. Our system requires no annotations and relies directly on the algorithm for type equivalence described in section 5. Aagaard et al. [1] adapt the concept of context-free session types from our previous work [58] to the applied pi calculus. They also account for unrestricted (that is, arbitrarily shared) channels in the style of Vasconcelos [63], a feature that we decided not to include in this paper. They prove session fidelity by mapping their calculus to the psi calculus and defining types up to equivalence with respect to session type bisimulation. Gay et al. recently showed that the equivalence for higher-order context-free session types is decidable [30].

Context-free protocols Early attempts to describe non-regular protocols as types include proposals by Ravara and Vasconcelos [52] using directed labelled graphs as types, by Puntigam [51] on non-regular process types to describe component interfaces, and by Südholt [56] who investigates some use cases with an approach based on Puntigam's types. Ravara and Vasconcelos define a concurrent object calculus and establish some metatheoretical results culminating in subject reduction for their type system, but do not address type checking. Puntigam studies type equivalence and subtyping, claims decidability for both, and gives a trace-based characterization. No details are given on the underlying calculus. These proposals appear in disparate formalisms based on process calculi and none of them supports polymorphism, so they are

quite distinct from the system in this paper, which is based on System F. Context-free session types were introduced by Thiemann and Vasconcelos [58]; we follow their formulation and extend it rigorously to System F. Das et al. [22,23] propose the theory of nested session types, featuring recursive types with polymorphic type constructors and nested types. Nested session types enable expressing protocols described by context-free languages recognized by multi-state deterministic pushdown automata, whereas context-free session types capture communication patterns recognized by single-state deterministic pushdown automata [30]. Although covering a more restrictive class of context-free languages, context-free session types enjoy a practical, sound and complete, type equivalence algorithm [4]. In turn, type equivalence for nested session types is decidable, the authors present a sound equivalence algorithm, but a complete and practical type equivalence algorithm is not yet known [22].

Linear logic propositions as sessions Caires and Pfenning [15,16] pioneered the interpretation of intuitionistic linear logic propositions as sessions. This interpretation, indicated by $[\_]$, maps receiving and sending session types for base types T as payload to tensor and lollipop as follows:

$$[!T.U] = T \otimes [U] \qquad [?T.U] = T \multimap [U]$$

Base types T need not be interpreted. The semantics of these systems, given directly by the cut elimination rules, ensures deadlock freedom. An analogous interpretation applies to our session type language, but to compensate for the presence of the sequencing operator, the interpretation must be formulated in CPS and invoked as in $[S](\mathbf{1})$:

$$[!T](U) = T \otimes U \qquad [?T](U) = T \multimap U \qquad [T_1; T_2](U) = [T_1]([T_2](U))$$

The interpretation of propositions as sessions is extended to dependent types [59], encompassing polymorphism since the resulting session type discipline includes explicit operators to send and receive types.

Wadler [67,68] proposes a typing preserving translation of a variant of the aforementioned language by Gay and Vasconcelos [32] into a process calculus that translates Caires and Pfenning's work [15,16] to classical linear logic, thus ensuring deadlock freedom. Even though our system ensures progress for the functional part of the language, the unrestricted interleaving of channel read/write on multiple channels may lead to deadlocked situations. That is the price to pay for the flexibility our language offers compared to the accounts of session types based on linear logic.

Polymorphism for session types An early version of this work [58] uses predicative (Damas-Milner [19]) polymorphism, in a form closely related to that offered by Bono et al. [13] for functional session types. The current paper extends the predicative system to System F, with the associated extra flexibility.

A different form of polymorphism—bounded polymorphism on the values transmitted on channels—was introduced by Gay [27] in the realm of session types for the π-calculus. The intention of this work was to gain flexibility over subtyping: a protocol like ?Byte; !Byte is not a subtype of ?Int; !Int because sending behaves contravariantly whereas receiving is covariant. Bono and Padovani [10,11] present a calculus to model process interactions based on copyless message passing. Their type system includes subtyping and recursion. An extension [12] features polymorphic endpoint types using bounded polymorphism in a similar way as Gay [27]. Dardha et al. [21] extend their encoding of session types into π-types with parametric and bounded polymorphism, but recursive types are not considered; Dardha [20] extends the encoding to recursive types, but polymorphism is only referred to as future work.

Goto et al. [34] study a type system that provides polymorphism over sessions in the context of the π calculus. Their calculus supports match processes which are guarded on a check for token (in)equality and serve to implement the usual branch session types. The goal of their system is to enable the typing of forwarders and session transducers using polymorphism and (session) type functions provided by the programmer as part of the definition of the type system. Their system is quite different from ours (and from most other polymorphic type systems) in that there is no syntax for quantification in their type language. They rely on polymorphism in their metalanguage (Coq), instead.

Wadler [68] supports polymorphism on session types, introducing dual quantifiers, $\forall$ and $\exists$, interpreted as sending and receiving types, similar to the polymorphic π-calculus by Turner [61]. Explicit higher-rank quantifiers on session types were pursued by Caires et al. [14,48]. Caires et al. provide a logically motivated theory of parametric polymorphism with higher-rank quantifiers but without recursive types. Griffith explores the inclusion of parametric polymorphism without nested types in the language SILL [36]. This paper considers recursive types but restricts polymorphism to the functional layer (see rule K-TAbsin Fig. 5), leaving polymorphism via type passing to future work.

In recent work, Das et al. explore recursive session type definitions with type parameters and nested type instantiation [22]. Type quantification and subtyping were added to this theory separately [23]. In this paper, we only consider nested types in the functional layer; incorporating explicit polymorphism and nested types on sessions constitutes future work.

FreeST *and the decidability of type equivalence* Almeida et al. [3] describe the implementation of the predicative version of FreeST (FreeST 1). At the core of all FreeST versions lies an algorithm to decide type equivalence. Unfortunately the proof of decidability of type equivalence (in section 3.3, based on the reduction to the decidability of bisimilarity of basic process algebras [17]) does not directly lead to a practical algorithm. Type equivalence in FreeST (1 and 2) is based on the algorithm developed by Almeida et al. [4].

Linear and recursive system f Bierman et al. [9] propose a polymorphic linear lambda calculus with recursive types, supporting both call by value and call by name semantics, and featuring a ! constructor to account for intuitionistic terms. Mazurak et al. [44] use kinds to qualify linearity of types in a variant of System F with linear types. In their system, type qualifiers are fixed as in our system. This approach was taken further by Lindley and Morris [42] in a sophisticated kind structure with linear and unrestricted kinds as opposed to type qualifiers (as in Walker [69] and in Vasconcelos [63]), the $\mathsf{un} <: \mathsf{lin}$ kind subsumption rule, distinct lin/un arrow types (this distinction is already present in Gay and Vasconcelos [32] in the context of functional session types, but their work does not support polymorphism). We follow the latter approach path quite closely: we have an additional message kind besides session and functional, whereas Lindley and Morris model records, variants, and branching using row types. Moreover, Lindley and Morris support polymorphism over kinds and rows, which we do not.

Alms [60] is a polymorphic functional programming language that supports affine types and modules. The mode of a type (affine or unrestricted) is indicated with a kind that may depend on a type, which serves the same purpose as Lindley and Morris' kind polymorphism (which we do not support). Alms comes with recursive algebraic datatypes, but does not offer equirecursive types. A library implementation of (regular) session types without recursion is among the examples provided with the Alms implementation.

API protocols Ferles et al. [24] verify the correct use of context-free API protocols by abstracting the program as a context-free grammar (representing feasible sequences of API calls) and checking whether the thus obtained grammar is included in that of a specification. Their tool is restricted to LL(k) grammars, given the undecidability of inclusion checking for arbitrary context-free languages. Our approach, instead, directly checks programs against protocol specifications in the form of context-free types.

8. Conclusion

This paper proposes $F^{\mu;}$, a functional language with support for polymorphic equi-recursive context-free session types extended with multi-threading and inter-process communication. We prove that type equivalence is decidable for context-free session types, and we present a bidirectional type checking algorithm that we prove correct with respect to the declarative system. The proposed algorithms are incorporated in a typechecker and interpreter for FreeST [2].

FreeST has a past and a future. FreeST 1 is based on a predicative type system; essentially the proposal by Thiemann and Vasconcelos [58]. FreeST 2, the current release, is a language based on this paper. For future releases we plan to extend the FreeST with higher-order kinds (providing for a convenient dualof type operator), higher-order channels (allowing to pass channels on channels), and polymorphism over session types (in addition to polymorphism over functional types). We also plan to explore type inference at type application points and the incorporation of shared channels [6,54,63].

Type equivalence for higher-order channels poses interesting challenges, for both the bisimulation-based declarative definition [58] and its grammar-based algorithmic version [4] rely on first-order messages. A recent work by Costa et al. [18] extends both the bisimulation and the grammar construction to higher-order session types. Furthermore, we do not expect difficulties in type duality for higher-order types, given that the results in this paper rely on building a type dual to a given type, a simple operation when contrasted with checking the duality of session types.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank the anonymous reviewers for their detailed comments that greatly contributed to improve the paper. This work was supported by FCT through project SafeSessions, ref. PTDC/CCI-COM/6453/2020, and the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

References

[1] Jens Aagaard, Hans Hüttel, Mathias Jakobsen, Mikkel Kettunen, Context-free session types for applied pi-calculus, in: EXPRESS/SOS, in: EPTCS, vol. 276, 2018, pp. 3–18.
[2] Bernardo Almeida, Andreia Mordido, Vasco T. Vasconcelos, FreeST, a functional programming language with context-free session types, http://rss.di.fc.ul.pt/tools/freest/, 2019, Last accessed 2022.
[3] Bernardo Almeida, Andreia Mordido, Vasco T. Vasconcelos, Freest: context-free session types in a functional language, in: PLACES, in: EPTCS, vol. 291, 2019, pp. 12–23.
[4] Bernardo Almeida, Andreia Mordido, Vasco T. Vasconcelos, Deciding the bisimilarity of context-free session types, in: TACAS, in: LNCS, vol. 12079, Springer, 2020, pp. 39–56.
[5] Roberto M. Amadio, Luca Cardelli, Subtyping recursive types, ACM Trans. Program. Lang. Syst. 15 (4) (1993) 575–631.
[6] Stephanie Balzer, Frank Pfenning, Manifest sharing with session types, Proc. ACM Program. Lang. 1 (ICFP) (2017) 37.

[7] Giovanni Bernardi, Matthew Hennessy, Using higher-order contracts to model session types (extended abstract), in: CONCUR, in: LNCS, vol. 8704, Springer, 2014, pp. 387–401.
[8] Giovanni Bernardi, Matthew Hennessy, Using higher-order contracts to model session types, Log. Methods Comput. Sci. 12 (2) (2016).
[9] Gavin M. Bierman, Andrew M. Pitts, Claudio V. Russo, Operational properties of Lily, a polymorphic linear lambda calculus with recursion, Electron. Notes Theor. Comput. Sci. 41 (3) (2000) 70–88.
[10] Viviana Bono, Chiara Messa, Luca Padovani, Typing copyless message passing, in: ESOP, in: LNCS, vol. 6602, Springer, 2011, pp. 57–76.
[11] Viviana Bono, Luca Padovani, Polymorphic endpoint types for copyless message passing, in: ICE, in: EPTCS, vol. 59, 2011, pp. 52–67.
[12] Viviana Bono, Luca Padovani, Typing copyless message passing, Log. Methods Comput. Sci. 8 (1) (2012).
[13] Viviana Bono, Luca Padovani, Andrea Tosatto, Polymorphic types for leak detection in a session-oriented functional language, in: FMOODS/FORTE, in: LNCS, vol. 7892, Springer, 2013, pp. 83–98.
[14] Luís Caires, Jorge A. Pérez, Frank Pfenning, Bernardo Toninho, Behavioral polymorphism and parametricity in session-based communication, in: ESOP, in: LNCS, vol. 7792, Springer, 2013, pp. 330–349.
[15] Luís Caires, Frank Pfenning, Session types as intuitionistic linear propositions, in: CONCUR, in: LNCS, vol. 6269, Springer, 2010, pp. 222–236.
[16] Luís Caires, Frank Pfenning, Bernardo Toninho, Linear logic propositions as session types, Math. Struct. Comput. Sci. 26 (3) (2016) 367–423.
[17] Søren Christensen, Hans Hüttel, Colin Stirling, Bisimulation equivalence is decidable for all context-free processes, Inf. Comput. 121 (2) (1995) 143–148.
[18] Diana Costa, Andreia Mordido, Diogo Poças, Vasco T. Vasconcelos, Higher-order context-free session types in system F, in: PLACES, in: EPTCS, vol. 356, 2022, pp. 24–35.
[19] Luís Damas, Robin Milner, Principal type-schemes for functional programs, in: POPL, ACM Press, 1982, pp. 207–212.
[20] Ornela Dardha, Recursive session types revisited, in: BEAT, in: EPTCS, vol. 162, 2014, pp. 27–34.
[21] Ornela Dardha, Elena Giachino, Davide Sangiorgi, Session types revisited, Inf. Comput. 256 (2017) 253–286.
[22] Ankush Das, Henry DeYoung, Andreia Mordido, Frank Pfenning, Nested session types, in: ESOP, in: LNCS, vol. 12648, Springer, 2021, pp. 178–206.
[23] Ankush Das, Henry DeYoung, Andreia Mordido, Frank Pfenning, Subtyping on nested polymorphic session types, arXiv preprint, arXiv:2103.15193, 2021.
[24] Kostas Ferles, Jon Stephens, Isil Dillig, Verifying correct usage of context-free API protocols, Proc. ACM Program. Lang. 5 (POPL) (2021) 1–30.
[25] Simon Fowler, Sam Lindley, J. Garrett Morris, Sára Decova, Exceptional asynchronous session types: session types without tiers, Proc. ACM Program. Lang. 3 (POPL) (2019) 28.
[26] Simon Gay, António Ravara (Eds.), Behavioural Types: From Theory to Tools, River Publishers, 2017.
[27] Simon J. Gay, Bounded polymorphism in session types, Math. Struct. Comput. Sci. 18 (5) (2008) 895–930.
[28] Simon J. Gay, Nils Gesbert, António Ravara, Vasco Thudichum Vasconcelos, Modular session types for objects, Log. Methods Comput. Sci. 11 (4) (2015).
[29] Simon J. Gay, Malcolm Hole, Subtyping for session types in the pi calculus, Acta Inform. 42 (2–3) (2005) 191–225.
[30] Simon J. Gay, Diogo Poças, Vasco T. Vasconcelos, The different shades of infinite session types, in: FOSSACS, 2012.
[31] Simon J. Gay, Peter Thiemann, Vasco T. Vasconcelos, Duality of session types: the final cut, in: PLACES, in: EPTCS, vol. 314, 2020, pp. 23–33.
[32] Simon J. Gay, Vasco Thudichum Vasconcelos, Linear type theory for asynchronous session types, J. Funct. Program. 20 (1) (2010) 19–50.
[33] The Go team, The Go programming language specification, https://golang.org/ref/spec, 2020, Last accessed 2021.
[34] Matthew A. Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, James Riely, An extensible approach to session polymorphism, Math. Struct. Comput. Sci. 26 (3) (2016) 465–509.
[35] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, Nobuko Yoshida, Featherweight Go, Proc. ACM Program. Lang. 4 (OOPSLA) (2020) 149.
[36] Dennis Edward Griffith, Polarized substructural session types, PhD thesis, University of Illinois at Urbana-Champaign, 2016.
[37] Robert Harper, Practical Foundations for Programming Languages, 2nd ed., Cambridge University Press, 2016.
[38] Kohei Honda, Types for dyadic interaction, in: CONCUR, in: LNCS, vol. 715, Springer, 1993, pp. 509–523.
[39] Kohei Honda, Vasco Thudichum Vasconcelos, Makoto Kubo, Language primitives and type discipline for structured communication-based programming, in: ESOP, in: LNCS, vol. 1381, Springer, 1998, pp. 122–138.
[40] Kohei Honda, Nobuko Yoshida, Marco Carbone, Multiparty asynchronous session types, J. ACM 63 (1) (2016) 9.
[41] Julien Lange, Nobuko Yoshida, Characteristic formulae for session types, in: TACAS, in: LNCS, vol. 9636, Springer, 2016, pp. 833–850.
[42] Sam Lindley, J. Garrett Morris, Lightweight functional session types, in: Behavioural Types: From Theory to Tools, River Publishers, 2017.
[43] Ian Mackie Lilac, A functional programming language based on linear logic, J. Funct. Program. 4 (4) (1994) 395–433.
[44] Karl Mazurak, Jianzhou Zhao, Steve Zdancewic, Lightweight linear types in system fdegree, in: TLDI, ACM, 2010, pp. 77–88.
[45] Robin Milner, Joachim Parrow, David Walker, A calculus of mobile processes, I, Inf. Comput. 100 (1) (1992) 1–40.
[46] Luca Padovani, Context-free session type inference, in: ESOP, in: LNCS, vol. 10201, Springer, 2017, pp. 804–830.
[47] Luca Padovani, Context-free session type inference, ACM Trans. Program. Lang. Syst. 41 (2) (2019) 9.
[48] Jorge A. Pérez, Luís Caires, Frank Pfenning, Bernardo Toninho, Linear logical relations and observational equivalences for session-based concurrency, Inf. Comput. 239 (2014) 254–302.
[49] Benjamin C. Pierce, Types and Programming Languages, MIT Press, 2002.
[50] Benjamin C. Pierce, David N. Turner, Local type inference, ACM Trans. Program. Lang. Syst. 22 (1) (2000) 1–44.
[51] Franz Puntigam, Non-regular process types, in: Euro-Par, in: LNCS, vol. 1685, Springer, 1999, pp. 1334–1343.
[52] António Ravara, Vasco Thudichum Vasconcelos, Behavioural types for a calculus of concurrent objects, in: Euro-Par, in: LNCS, vol. 1300, Springer, 1997, pp. 554–561.
[53] John H. Reppy, CML: a higher-order concurrent language, in: PLDI, ACM, 1991, pp. 293–305.
[54] Pedro Rocha, Luís Caires, Propositions-as-types and shared state, Proc. ACM Program. Lang. 5 (ICFP) (2021) 1–30.
[55] Sangiorgi Davide, An Introduction to Bisimulation and Coinduction, Cambridge University Press, 2014.
[56] Mario Südholt, A model of components with non-regular protocols, in: SC, in: LNCS, vol. 3628, Springer, 2005, pp. 99–113.
[57] Kaku Takeuchi, Kohei Honda, Makoto Kubo, An interaction-based language and its typing system, in: PARLE, in: LNCS, vol. 817, Springer, 1994, pp. 398–413.
[58] Peter Thiemann, Vasco T. Vasconcelos, Context-free session types, in: ICFP, ACM, 2016, pp. 462–475.
[59] Bernardo Toninho, Luís Caires, Frank Pfenning, Dependent session types via intuitionistic linear type theory, in: PPDP, ACM, Odense, Denmark, 2011, pp. 161–172.
[60] Jesse A. Tov, Riccardo Pucella, Practical affine types, in: POPL, ACM, Austin, TX, USA, 2011, pp. 447–458.
[61] David N. Turner, The polymorphic Pi-calculus: theory and implementation, PhD thesis, University of Edinburgh, UK, 1996.
[62] Vasco T. Vasconcelos, Sessions, from types to programming languages, Bull. Eur. Assoc. Theor. Comput. Sci. 103 (2011) 53–73.
[63] Vasco T. Vasconcelos, Fundamentals of session types, Inf. Comput. 217 (2012) 52–70.
[64] Vasco Thudichum Vasconcelos, Fundamentals of session types, in: SFM, in: LNCS, vol. 5569, Springer, 2009, pp. 158–186.
[65] Vasco Thudichum Vasconcelos, Simon J. Gay, António Ravara, Type checking a multithreaded functional language with session types, Theor. Comput. Sci. 368 (1–2) (2006) 64–87.

[66] Vasco Thudichum Vasconcelos, António Ravara, Simon J. Gay, Session types for functional multithreading, in: CONCUR, in: LNCS, vol. 3170, Springer, 2004, pp. 497–511.
[67] Philip Wadler, Propositions as sessions, in: ICFP, ACM, 2012, pp. 273–286.
[68] Philip Wadler, Propositions as sessions, J. Funct. Program. 24 (2–3) (2014) 384–418.
[69] David Walker, Substructural type systems, in: Advanced Topics in Types and Programming Languages, MIT Press, 2005.
[70] Andrew K. Wright, Simple imperative polymorphism, LISP Symb. Comput. 8 (4) (1995) 343–355.