Towards Context-Free Session Types

Abstract

Session types describe structured communication on heterogeneously typed channels at a high level. Their tail-recursive structure imposes a protocol that can be described by a regular language. The types of transmitted values are drawn from the underlying functional language, abstracting from the details of serializing values of structured data types.

Context-free session types extend session types by allowing nested protocols that are not restricted to tail recursion. Nested protocols correspond to deterministic context-free languages. Such protocols are interesting in their own right, but they are particularly suited to describe the low-level serialization of tree-structured data in a type-safe way.

We establish the metatheory of context-free session types, prove that they properly generalize standard (two-party) session types, and take first steps towards type checking by showing that type equivalence is decidable.

Keywords session types, semantics, type checking

1. Introduction

Session types have been discovered by Kohei Honda as a means to describe the structured interaction of processes via typed communication channels [13, 18]. While connections are homogeneously typed in languages like Concurrent ML [16], session types provide a heterogeneous type discipline for a protocol on a bidirectional connection: each message has an individual direction and type and there are choice points where a sender can make a choice and a receiver has to follow.

The type structure of a language with session types typically comes with two layers, regular types and session types:

$$T ::= S \mid \mathsf{unit} \mid B \mid T \to T \mid \dots$$

$$S ::= \mathsf{end} \mid ?\,T.S \mid !\,T.S \mid \&\{l_i\colon S_i\}_{i\in I} \mid \oplus\{l_i\colon S_i\}_{i\in I} \mid z \mid \mu z.S$$

A type T is either a session type S, a unit type, a base type B, a function type, and so on. Session types S are attached to communication channels. They denote denote different states of the channel. The type end indicates the end of a session, $?\,T.S$ ($!\,T.S$) indicates readiness to receive (send) a value of type T and continuing with S, the branching operators $\&\{l_i\colon S_i\}_{i\in I}$ and $\oplus\{l_i\colon S_i\}_{i\in I}$ indicate receiving and sending labels, where the label l_i selects the protocol S_i from a finite number of possibilities $i \in I$ for the subsequent communication on the channel. For example, the session type

$$\&\{\mathit{add}\colon ?\,\mathsf{int}.?\,\mathsf{int}.!\,\mathsf{int}.\mathsf{end}, \mathit{neg}\colon ?\,\mathsf{int}.!\,\mathsf{int}.\mathsf{end}\}$$

is the type of a server that accepts two commands *add* and *neg*, then reads the appropriate number of arguments and returns the result of the command. The session variable z and the operator $\mu z.S$ serve to introduce recursive protocols, for example, to read a list of numbers:

$$\mu z.\&\{\mathit{stop}\colon \mathsf{end}, \mathit{more}\colon ?\,\mathsf{int}.z\}$$

Session types are well suited to document high-level communication protocols and there is a whole range of extensions to make them amenable to deal with realistic situations, for example, multi-party session types [15], session types for distributed object-oriented programming [12], or for programming web services [6]. However, there is a fundamental limitation in their structure that makes it impossible to describe the low-level serialization (marshalling, pickling, ...) of tree structured data in a type-safe way, as we demonstrate with the following example.

Let's assume that a single communication operation can only transmit a label or a base type value to model the real-world restriction that data structures need to be serialized to a wire format before they can be sent over a network connection. Formally, it is sufficient to restrict the session type formation for sending and receiving data to base types: $!\,B.S$ and $?\,B.S$. Now suppose we want to transmit binary trees where the internal nodes contain a number. A recursive type for such trees can be defined as follows:

```
type Tree = Leaf
          | Node int Tree Tree
```

To serialize such a structure, we traverse it and transmit a sequence of labels `Leaf` and `Node` and **`int`** values as they are visited by the traversal. The set of serialization sequences corresponding to a pre-order traversal of a tree may be described by the following context-free grammar.

$$N ::= \mathsf{Leaf} \mid \mathsf{Node}\ \mathsf{int}\ N\ N \tag{1}$$

Listing 1 contains a function `sendTree` that performs a pre-order traversal of a tree and that sends the correctly serialized output on a channel. The function relies on typical operations in a functional session type calculus like GV [11]: the **`select`** operation takes a label and a channel, outputs the label, and returns the (updated)

```
sendTree : ∀α. Tree → TreeChannel;α → α
sendTree (Leaf) c =
  select Leaf c
sendTree (Node x l r) c =
  let c1 = select Node c
      c2 = send x c1
      c3 = sendTree l c2
      c4 = sendTree r c3
  in  c4
```

Listing 1. Type-safe serialization of a binary tree

channel. The send operation takes a value and a channel, outputs the value, and returns the channel. Ignore the type signature for a moment.

It turns out that the sendTree function cannot be typed in existing session-type calculi [10, 11, 13, 14, 18]. To see this, we observe that the language generated by the nonterminal N in (1) is context-free, but not regular. In contrast, the language of communication actions described by a traditional session type S is regular. More precisely, taking infinite executions into account, each traditional session type is related to the union of a regular language and an ω-regular language that describe the finite and infinite sequences of communication actions admitted by the type. A similar caveat applies to the language generated by a context-free grammar like (1), a point which we leave for future work.

It turns out that we can type functions like sendTree if we drop the restriction of being tail recursive from the language of session types. Here is an informal proposal for a revised session type structure replacing the previous one:

$$\begin{array}{rcl} S & ::= & ?B \mid !B \mid \&\{l_i\colon S_i\}_{i\in I} \mid \oplus\{l_i\colon S_i\}_{i\in I} \\ & \mid & z \mid \mu z.S \mid \mathsf{skip} \mid S;S \end{array}$$

That is, we remove the continuation from the primitive send and receive types and adopt a general sequence operator $\_;\_$ with unit skip. This change removes the restriction to tail recursion and enables a session type to express context-free communication sequences such as the ones required for the serialization example. However, the monoidal structure of skip and $\_;\_$ poses some challenges for the metatheory. We call this structure *context-free session types* and it is sufficient to assign a type to function sendTree. First, we define the recursive session type corresponding to the Tree datatype. Its definition follows the datatype definition, but it makes the sequence of communication operations explicit.

```
type TreeChannel = ⊕{
  Leaf: skip,
  Node: !int;TreeChannel;TreeChannel}
```

Now we are ready to explain the type signature for sendTree.

```
sendTree : ∀α.Tree → TreeChannel;α → α
```

It abstracts over the type α of the continuation channel, takes as input a Tree and a channel which first runs the recursive protocol TreeChannel followed by some other protocol specified by α. The α-typed channel is returned which leaves its processing to the continuation.

Polymorphism, as seen in this signature, is rarely considered in session types (with two exceptions [4, 9] discussed in the related work). However, it appears quite natural in this context as sending a tree generalizes sending a single value, which is naturally polymorphic over the continuation channel as in send : $\forall\alpha.B \to (!B;\alpha) \to \alpha$. Further study of the typing derivation of sendTree (in Section 2) makes it clear that polymorphism is absolutely essential to make context-free session types work in connection with recursive types. It turns out that the recursive calls happen at *instances* of the declared type, so that sendTree (as well as its receiving counterpart, recvTree) makes use of *polymorphic recursion*.

Contributions and overview

- We introduce context-free session types that extend the expressiveness of regular session types to capture the type-safe serialization of recursive datatypes and XML documents. They further enable the type-safe implementation of remote operations on recursive datatypes that either traverse the structure eagerly or on demand.
- Section 2 discusses the overall design and explains the requirements to the metatheory with examples.
- Section 3 formally introduces context-free session types. A kind system with subkinding guarantees well-formedness; the definition of contractiveness needs to be refined to deal with the moniodal structure of the type operators skip and $\_;\_$; we give a coinductive definition of type equivalence as a bisimulation of types and prove its decidability by reducing type equivalence to the equivalence of basic process algebra expressions (BPA).
- Section 4 formally introduces the term language along with its statics and dynamics. It is a synchronous, first-order version of Gay and Vasconcelos' linear type theory for asynchronous session types (GV) [11] extended with recursive types and variant types to model recursive datatypes. We establish type soundness and progress for the functional sublanguage. We prove in Section 4.5 that our system conservatively extends a regular (first-order) session-type system.
- We discuss related work in Section 5 and conclude.

2. Context-free session types in action

To understand the requirements for the metatheory of context-free session types, we first examine the type derivation of sendTree in Listing 1. Then we turn to further examples that underline the expressiveness and the usefulness of context-free session types.

2.1 Sending leaves

To typecheck the first alternative of the sendTree function for sending leaves, we need to derive type α for the code fragment

```
select Leaf c
```

given that c :: TreeChannel;α. Anticipating the formal definition in Section 4 (Figure 6), we sketch an informal typing rule for **select**, which is taken verbatim from GV[11]:

$$\frac{\vdash e : \oplus\{l_i\colon S_i\}_{i\in I} \quad j \in I}{\vdash \mathsf{select}\, l_j\, e\colon S_j} \tag{2}$$

The **select** operation expects a branch type $\oplus\{l_i\colon S_i\}$, but we are given the recursive TreeChannel type, which has to be unfolded first. Such unfolding is to be expected in the presence of recursive types. As unfolding is not indicated in the term, we require an *equi-recursive treatment of recursion in types*.

After unfolding, we obtain

```
c : ⊕{Leaf: skip,
      Node: !int;TreeChannel;TreeChannel};α
```

This type, a sequence of protocols, is still not in the form expected by **select**. Hence, we further need to enrich type equivalence to enable us to *commute the continuation type α inside the branches*.

After commutation, we obtain the typing

```
c : ⊕{Leaf: skip;α,
      Node: !int;TreeChannel;TreeChannel;α}
```

which is finally in a form acceptable to **select**. Applying the typing rule (2) yields

```
select Leaf c : skip;α
```

At this point, we need to apply the *monoid identity law* (which also needs to be part of type equivalence) to obtain the desired outcome.

```
select Leaf c : α
```

2.2 Sending nodes

We turn to typechecking the second alternative of the sendTree function

```
let c1 = select Node c
    c2 = send x c1
    c3 = sendTree l c2
    c4 = sendTree r c3
in  c4
```

given that

```
x : int, l : Tree, r : Tree, c : TreeChannel
```

Typechecking the **select** operation requires the same steps as for leaves. We skip over those and note the resulting typing for c1.

```
c1 : !int;TreeChannel;TreeChannel;α
```

The **send** operation just peels off the leading !**int** type, but our typing for c1 glosses over an important detail, namely the bracketing of the _;_ operator. After commuting α inside the branch type and applying the **select** rule, we are actually left with this type:

```
c1 : (!int;(TreeChannel;TreeChannel));α
```

Again, we need to appeal to type equivalence to reassociate the nesting of the sequence operator, that is, to apply the *monoidal associativity law*. The resulting type

```
c1 : !int;((TreeChannel;TreeChannel);α)
```

is compatible with the typing for **send** and we can proceed with

```
c2 : (TreeChannel;TreeChannel);α
```

Again, we need to reassociate:

```
c2 : TreeChannel;(TreeChannel;α)
```

At this point, we see the need for *polymorphic recursion*: the recursive call sendTree l c2 of

```
sendTree : ∀β.Tree → TreeChannel;β → β
```

must instantiate the type variable β to (TreeChannel;α). With this instantiation, we obtain

```
c3 : TreeChannel;α
```

The second recursive call instantiates β to α (it could be treated monomorphically) and we readily obtain the desired final outcome, which is equivalent to the outcome of the first alternative:

```
c4 : α
```

In summary, the type system for context-free session types requires polymorphism with polymorphic recursion. Furthermore, it relies on a nontrivial notion of type equivalence that includes unfolding of equi-recursive types, distributivity of branching over sequencing, and the monoidal structure of **skip** and sequencing (identity and associativity laws). Our technical treatment of type equivalence in Sections 3.2 and 3.3 relies on a terminating unraveling operation that normalizes the "head" of a session type with respect to these notions.

2.3 Structure-preserving tree transformation

As another example for the expressiveness of context-free session types, we present client and server code for a remote structure-preserving tree transformation in Listing 2. It is based on the same tree datatype as before, but it introduces a new channel type XformChan that receives the transformed node value after sending the old value and the two subtrees. This code makes use of the **receive** operation that takes a channel of type !**int**;α and returns a linear pair of type **int**$\otimes\alpha$. The pair must be linear because channels in session-type calculi generally have linear types to cater for the change of their type at each operation.

The server function transform demonstrates the use of **receive**. It also uses pattern matching to deconstruct the linear pairs returned by recursive calls and by receiving integers. No new issues arise in typing this function compared to sendTree.

```
type XformChan = ⊕{
  Leaf: skip,
  Node: !int;XformChan;XformChan;?int}

transform : ∀α.Tree → XformChan;α → Tree ⊗ α
transform Leaf c =
  (Leaf, select Leaf c)
transform (Node x l r) c =
  let c1 = select Node c
      c2 = send c x
      l1,c3 = transform l c2
      r1,c4 = transform r c3
      x1,c5 = receive c4
  in (Node x1 l1 r1,c5)

treeSum : dualof XformChan; α ⊸ int ⊗ α
treeSum c =
  case c of
    Leaf: λc1.(0,c1)
    Node: λc1.let x,c2 = receive c1
                  l,c3 = treeSum c2
                  r,c4 = treeSum c3
                  c5 = send c4 (x+l+r)
              in (x+l+r,c5)

aTree = Node 3 Leaf (Node 4 Leaf Leaf)

go : Tree
go =
  let c s = new XformChan
  in fork (fst (treeSum s));
     fst (transform aTree c)
```

Listing 2. Remote tree transformation

The function treeSum is a suitable client for transformer channels. It computes the accumulated sum at each tree node, so that running transform and treeSum concurrently results in a tree where each node value is replaced by the sum of all node values below. The function treeSum takes an argument channel of type **dualof** XformChan; α where the **dualof** operator swaps sending and receiving types as usual. The **case** expression is the receiving counterpart of the **select** expression. It receives a label from a channel and dispatches according to this label. Each branch of the **case** is a function that takes the respective continuation of the channel and continues the interaction on that channel.

The final definition of go stitches it all together. Using **new** XformChan it creates a new pair of channels, the types of which are XFormChan and its dual, it forks a new process that runs treeSum on the server channel, and finally runs transform on an example tree and the client channel.

The example also illustrates how channels are closed. treeSum (transform) returns a *linear pair* of the accumulated sum (transformed tree) and a depleted channel of type **skip**. The function **fst** eliminates the linear pair and returns its first component. It implicitly closes the channel by discarding it, which is possible because **skip** is no longer restricted to be linear.

2.4 Expression server

An example that is quite often used in the literature on session types is an arithmetic server with a type like the one indicated in the introduction:

$$\&\{add\colon ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.\mathsf{end}, neg\colon ?\mathsf{int}.!\mathsf{int}.\mathsf{end}\}$$

```
type TermChan = ⊕{Const: !int,
                  Add:TermChan;TermChan,
                  Mult: TermChan;TermChan}

computeService : dualof TermChan;!int → skip
computeService c =
  let n1,c1 = receiveEval c
  in send n1 c1

receiveEval : ∀α.dualof TermChan;α → int ⊗ α
receiveEval c =
  case c of {
    Const: λc.receive c
    Add:   λc.let n1,c1 = receiveEval c
                  n2,c2 = receiveEval c1
              in (n1+n2,c2)
    Mult:  λc.let n1,c1 = receiveEval c
                  n2,c2 = receiveEval c1
              in (n1*n2,c2)
  }

client : TermChan;?int → int ⊗ skip
client c =
  let c1 = select Add c
      c2 = select Const c1
      c3 = send 5 c2
      c4 = select Mult c3
      c5 = select Const c4
      c6 = send 7 c5
      c7 = select Const c6
      c8 = send 9 c7
    in receive c8

go : int
go =
  let c s = new TermChan;?int
      _ = fork (computeService s);
      n,sk = client c
  in n
```

Listing 3. Arithmetic expression server

Exploiting context-free session types, we can extend the scope of such a server to receive and process arbitrary well-formed arithmetic expressions. As an example consider the arithmetic expression server for terms composed of constants, addition, and multiplication in Listing 3. The implementation of the protocol is straightforward using the techniques already described.

It is possible to extend this protocol to lazily traverse the term. In this case, the server requests from the client the parts of the term needed to complete the evaluation. For instance, if a factor in a multiplication is zero, the server can avoid to even ask for sending the other factor. We elucidate this idea with a simplified protocol to explore a binary tree lazily. No new features are required for its realization.

```
type XploreTreeChan = ⊕{
  Leaf: skip,
  Node: XploreNodeChan
}

type XploreNodeChan = ⊕{
  Value: !int;XploreNodeChan,
  Left:  XploreTreeChan;XploreNodeChan,
  Right: XploreTreeChan;XploreNodeChan,
  Exit: skip
}

exploreTree : Tree → XploreTreeChan;α → α
exploreTree Leaf c =
  select Leaf c
exploreTree (Node x l r) c =
  let c1 = select Node c in
  exploreNode x l r c1

exploreNode : int → Tree → Tree →
    XploreNodeChan;α → α
exploreNode x l r c1
  case c1 of {
    Value: λc2.let c3 = send c2 x
                in  exploreNode x l r c3,
    Left:  λc2.let c3 = exploreTree l c2
                in  exploreNode x l r c3,
    Right: λc2.let c3 = exploreTree r c2
                in  exploreNode x l r c3,
    Exit:  λc2.c2
  }
```

A client connecting to the server exploreTree first connects to the root node of a tree. First it must check whether the current node is a Leaf or a Node. If it is a Node, it can further explore the contents: it can ask for the value or traverse the left subtree or the right subtree as often as desired. Finally the client sends Exit to return to the parent node.

The type describing this interaction is mutually recursive. The "inner loop" described by XploreNodeChan is tail-recursive like a regular session type, but the "outer loop" corresponding to XploreTreeChan is not as its invocations are intertwined with the inner loop.

3. Types

This section introduces the notion of types and the machinery required for defining type equivalence which relies on bisimilarity.

3.1 Types and the kinding system

We rely on a few base sets: *recursion variables*, denoted by x, y, z; *type variables* denoted by α, β drawn from a set $\mathcal{TV}$; *labels* denoted by l drawn from $\mathcal{L}$, and *primitive types* denoted by B, which include unit and int, drawn from $\mathcal{B}$.

A *kinding* system establishes what constitutes a valid type, distinguishing between session types, general types, and type schemes. The kinding system further distinguishes linear from unrestricted types. *Prekinds*, denoted by υ, are session types $\mathcal{S}$, arbitrary types $\mathcal{T}$, or type schemes $\mathcal{C}$. *Multiplicities*, denoted by m, can be linear $\mathbf{l}$ or unrestricted $\mathbf{u}$. *Kinds* are of the form υ^m, describing types and their multiplicities. A partial order $<$ is defined on prekinds, which describes that a session type of kind $\mathcal{S}$ may be regarded as a type of kind $\mathcal{T}$, which in turn may be regarded as a type scheme of kind $\mathcal{C}$. Similarly, multiplicities establish that an unrestricted (use zero or more times) type can be regarded as a linear (use exactly once) type, that is $\mathbf{u} < \mathbf{l}$. The two order relations form a complete lattice on kinds. Its ordering is determined by $\upsilon_1^{m_1} \leq \upsilon_2^{m_2}$ iff $\upsilon_1 \leq \upsilon_2$ and $m_1 \leq m_2$.

A *kinding environment*, denoted by Δ, associates kinds κ to type variables α and to recursion variables x. When writing $\Delta, \alpha :: \kappa$ or $\Delta, x :: \kappa$ we assume that α and x do not occur in Δ. The notions of kinds and kind environments are summarized in Figure 1.

Kind assignment is defined by a judgment $\Delta \vdash T :: \kappa$ (see Figure 1) that ensures the good formation of types T, while classifying well-formed types in session types, general types, or type schemes, as well as assigning a multiplicity to session types and general types. The type scheme $\forall\alpha :: \kappa.T$ binds the type variable α and the recursive type $\mu x.T$ binds the recursion variables x. The sets of *bound* and *free variables* in types are defined in the usual way, and so is the *substitution* of a type variable α (resp. recursion variable x) by a type T in a type U, denoted by $U[T/\alpha]$ (resp. $U[T/x]$). We assume the variable convention that no type

$$
\begin{array}{lll}
\upsilon & ::= \mathcal{S} \mid \mathcal{T} \mid \mathcal{C} \quad \mathcal{S} < \mathcal{T} < \mathcal{C} & \text{Prekinds: session, type, or scheme} \\
m & ::= \mathbf{u} \mid \mathbf{l} \qquad \mathbf{u} < \mathbf{l} & \text{Multiplicity: unrestricted or linear} \\
\kappa & ::= \upsilon^m & \text{Kinds} \\
\Delta & ::= \cdot \mid \Delta, x :: \kappa \mid \Delta, \alpha :: \kappa & \text{Kind environments} \\
\gamma & ::= \mathsf{s} \mid \mathsf{p} & \text{Guardedness: skips or productive}
\end{array}
$$

$$
\frac{}{\Delta \vdash \mathsf{skip} :: \mathcal{S}^{\mathbf{u}}} \quad \frac{}{\Delta \vdash !B :: \mathcal{S}^{\mathbf{l}}} \quad \frac{}{\Delta \vdash ?B :: \mathcal{S}^{\mathbf{l}}}
$$

$$
\frac{\Delta \vdash T_1 :: \mathcal{S}^{m_1} \quad \Delta \vdash T_2 :: \mathcal{S}^{m_2}}{\Delta \vdash (T_1; T_2) :: \mathcal{S}^{\max(m_1, m_2)}}
$$

$$
\frac{(\forall i \in I)\ \Delta \vdash T_i :: \mathcal{S}^{\mathbf{l}}}{\Delta \vdash \oplus\{l_i : T_i\}_{i \in I} :: \mathcal{S}^{\mathbf{l}}} \quad \frac{(\forall i \in I)\ \Delta \vdash T_i :: \mathcal{S}^{\mathbf{l}}}{\Delta \vdash \&\{l_i : T_i\}_{i \in I} :: \mathcal{S}^{\mathbf{l}}}
$$

$$
\frac{}{\Delta, x :: \kappa \vdash x :: \kappa} \quad \frac{}{\Delta, \alpha :: \kappa \vdash \alpha :: \kappa} \quad \frac{}{\Delta \vdash B :: \mathcal{T}^{\mathbf{u}}}
$$

$$
\frac{\Delta \vdash_c T : \gamma \quad \Delta, x :: \kappa \vdash T :: \kappa \quad \kappa \leq \mathcal{T}^{\mathbf{l}}}{\Delta \vdash \mu x.T :: \kappa}
$$

$$
\frac{\Delta \vdash T_1 :: \mathcal{T}^{\mathbf{l}} \quad \Delta \vdash T_2 :: \mathcal{T}^{\mathbf{l}}}{\Delta \vdash T_1 \to T_2 :: \mathcal{T}^{\mathbf{u}}} \quad \frac{\Delta \vdash T_1 :: \mathcal{T}^{\mathbf{l}} \quad \Delta \vdash T_2 :: \mathcal{T}^{\mathbf{l}}}{\Delta \vdash T_1 \multimap T_2 :: \mathcal{T}^{\mathbf{l}}}
$$

$$
\frac{\Delta \vdash T_1 :: \mathcal{T}^{\mathbf{l}} \quad \Delta \vdash T_2 :: \mathcal{T}^{\mathbf{l}}}{\Delta \vdash T_1 \otimes T_2 :: \mathcal{T}^{\mathbf{l}}} \quad \frac{(\forall i \in I)\ \Delta \vdash T_i :: \mathcal{T}^{m}}{\Delta \vdash [l_i : T_i]_{i \in I} :: \mathcal{T}^{m}}
$$

$$
\frac{\Delta \vdash T :: \kappa_1 \quad \kappa_1 \leq \kappa_2}{\Delta \vdash T :: \kappa_2} \quad \frac{\Delta, \alpha :: \kappa \vdash T :: \mathcal{C}^{m} \quad \kappa \leq \mathcal{T}^{\mathbf{l}}}{\Delta \vdash \forall \alpha :: \kappa.T :: \mathcal{C}^{m}}
$$

Figure 1. Kinding system, $\Delta \vdash T :: \kappa$

or recursion variable occurs bound and free in the same type and tacitly rename variables accordingly.

A session type may be skip indicating no communication (this type is unrestricted as it indicates a depleted channel that can be garbage collected), $!B$ for sending a base type value, $?B$ for receiving a base type value, or $(S_1; S_2)$ for the sequence of actions denoted by S_1 followed by those denoted by S_2. Further, there are branch types $\oplus\{l_i : T_i\}$ and choice types $\&\{l_i : T_i\}$ that either select and send a label l_i or branch on such a received label and then continue on the corresponding branch. The formation rule for sequence makes sure that its kind can only be unrestricted $\mathbf{u}$ if both S_1 and S_2 are. The formation rules for the branch and choice types are straightforward.

The formation rules for recursion variables, type variables, and base types are as expected. They furthermore require that their body is contractive in x using the judgment $\Delta \vdash_c T : \gamma$, which we define shortly. The formation rules of the remaining type constructions contain no surprises; the constituend types must not be type schemes. Finally, the kind subsumption rule is standard and the abstraction rule enables the formation of type schemes where abstraction is restricted to types by the constraint $\kappa \leq \mathcal{T}^{\mathbf{l}}$.

Regarding session types, the kinding system makes sure that the operators $\&$, $\oplus$, and $\_;\_$ are only applied to session types, that is, to types of kind $\mathcal{S}^m$. Types like $(\mathsf{int} \to \mathsf{int}); \mathsf{skip}$ and $\oplus\{l_1 : \mathsf{int}, l_2 : \mathsf{int} \otimes \mathsf{int}\}$ are not well formed. In addition, types essentially composed of skip may be assigned the unrestricted kind $\mathcal{S}^{\mathbf{u}}$, whereas all other session types are assigned the linear kind $\mathcal{S}^{\mathbf{l}}$. An example of a well-formed unrestricted session type is $\cdot \vdash \mu x.(\mathsf{skip}; \mathsf{skip}) :: \mathcal{S}^{\mathbf{u}}$; an example of a linear session type is $\cdot \vdash (\ !\mathsf{int}; \mathsf{skip}); ?\mathsf{int} :: \mathcal{S}^{\mathbf{l}}$. Recursion variables and type variables occurring free in types must be defined in the kinding environment.

The language of types includes recursion, hence we must pay particular attention to *contractivity* [8]. A type T is *contractive on a recursion variable* x, if $\Delta \vdash_c T : \gamma$ is derivable under an

$$
\frac{}{\Delta \vdash_c T_1 \to T_2 : \mathsf{p}} \quad \frac{}{\Delta \vdash_c T_1 \multimap T_2 : \mathsf{p}} \quad \frac{}{\Delta \vdash_c T_1 \otimes T_2 : \mathsf{p}}
$$

$$
\frac{}{\Delta \vdash_c [l_i : T_i] : \mathsf{p}} \quad \frac{}{\Delta \vdash_c B : \mathsf{p}} \quad \frac{}{\Delta \vdash_c !B : \mathsf{p}} \quad \frac{}{\Delta \vdash_c ?B : \mathsf{p}}
$$

$$
\frac{}{\Delta \vdash_c \oplus\{l_i : T_i\}_{i \in I} : \mathsf{p}} \quad \frac{}{\Delta \vdash_c \&\{l_i : T_i\}_{i \in I} : \mathsf{p}}
$$

$$
\frac{}{\Delta \vdash_c \mathsf{skip} : \mathsf{s}} \quad \frac{\Delta \vdash_c T_1 : \mathsf{p}}{\Delta \vdash_c (T_1; T_2) : \mathsf{p}} \quad \frac{\Delta \vdash_c T_1 : \mathsf{s} \quad \Delta \vdash_c T_2 : \gamma}{\Delta \vdash_c (T_1; T_2) : \gamma}
$$

$$
\frac{\Delta \vdash_c T : \gamma}{\Delta \vdash_c \mu x.T : \gamma} \quad \frac{}{\Delta, x : \kappa \vdash_c x : \mathsf{p}} \quad \frac{}{\Delta, \alpha : \kappa \vdash_c \alpha : \mathsf{p}}
$$

Figure 2. Contractivity, $\Delta \vdash_c T : \gamma$

environment Δ that does *not* contain x. The contractivity system classifies session types under *skips only*, s, and *productive*, p, both denoted by metavariable γ. If $\Delta \vdash_c T : \mathsf{s}$ is derivable then T is essentially composed of skips. On the other hand, if $\Delta \vdash_c T : \mathsf{p}$ is derivable then T describes a nontrivial interaction. The intuitive reading is that any use of recursion variable x must be preceded by a type construction that is different from skip. The kinding system makes sure that well-formed types are contractive, by calling the contractivity system in the rule for μ-types.

For example, types such as $\mu x.(\mathsf{skip}; x)$ or $\mu x.(; !\mathsf{int}; x)$ are disallowed whereas $\mu x.(!\mathsf{int}; x)$ is contractive. The interaction between the μ-operator and the semicolon is nontrivial: the type $\mu x.\mu y.(x; y)$ is ruled out because x would not be guarded after unrolling μy once. However, the type $\mu x.(!\mathsf{int}; \mu y.(x; y))$ is contractive because unrolling μx reveals that the recursive occurrence of y is guarded: $!\mathsf{int}; \mu y.(\mu x.(!\mathsf{int}; \mu y.(x; y)); y)$. Well-formedness of types is preserved under arbitrary unrolling of μ-operators, a result that we discuss below.

The definition of contractivity incorporates type variables α by assuming that they are always replaced by productive types. (Type variables labeled $\alpha : \mathsf{s}$ could be replaced by skip.) Type variables must be restricted in this way because contractivity of $\mu x.(\alpha; x)$ requires $\alpha :: \kappa \vdash_c \alpha : \mathsf{p}$ to be derivable so that $\alpha :: \kappa \vdash_c \mu x.(\alpha; x) : \mathsf{p}$ is derivable.

By abuse of notation let $\mathcal{T}$ denote the set of closed, well-formed types, that is, of types T such that $\Delta \vdash T :: \mathcal{T}^{\mathbf{l}}$, for some Δ that does not bind recursion variables. To avoid notational overhead, we let S range over (closed) *session types* with the understanding that $\Delta \vdash S :: \mathcal{S}^m$, for some Δ that does not bind recursion variables. We also write $\mathcal{S}$ for the set of all such session types.

Lemma 3.1 (Type substitution preserves kinding)**.** *If* $\Delta, \alpha :: \kappa_1 \vdash T_2 :: \kappa_2$ *and* $\Delta \vdash T_1 :: \kappa_1$ *then* $\Delta \vdash T_2[T_1/\alpha] :: \kappa_2$.

3.2 Type equivalence

Type equivalence is nontrivial in our system because the type language has a non-empty equational theory. This theory has two components. First, the skip and the sequence type constructors form a monoid and as such should respect the monoidal laws: skip is a left- and right-identity with respect to the sequence operator and the sequence operator is associative. Second, the reading of the μ-operator is equirecursive, which means that a μ-type is equal to its unrolling.

Our approach to defining type equivalence (for the session type fragment) relies on bisimulation. We regard two session types as equivalent if the exhibit the same communication behavior. Previous work follows a similar line, but syntactically restricts recursion to tail recursion which simplifies the definition of type equivalence and guarantees its decidability [10].

To define the bisimulation on session types, we first need to define a labeled transition system. Afterwards, we show that bisim-

$$\mathsf{skip}\checkmark \qquad \frac{S_1\checkmark \quad S_2\checkmark}{(S_1;S_2)\checkmark} \qquad \frac{S[\mu x.S/x]\checkmark}{\mu x.S\checkmark}$$

$$A \xrightarrow{A} \mathsf{skip} \qquad \star\{\overline{l_n\colon S_n}\} \xrightarrow{\star l_i} S_i$$

$$\frac{S_1 \xrightarrow{a} S_1'}{S_1;S_2 \xrightarrow{a} S_1';S_2} \qquad \frac{S_1\checkmark \quad S_2 \xrightarrow{a} S_2'}{S_1;S_2 \xrightarrow{a} S_2'} \qquad \frac{S[\mu x.S/x] \xrightarrow{a} S'}{\mu x.S \xrightarrow{a} S'}$$

Figure 3. Labelled transition system

ilarity for this system is decidable by reduction to basic process algebra (BPA), a well-studied system [2, 7]. BPA is known for generating context-free processes, which fits well with our context-free session type framework.

In our labeled transition system, we consider the following primitive actions:

- $!B$ and $?B$ for sending and receiving a base type value;
- $\oplus l$ and $\& l$ for sending and receiving a label from a choice or branch type;
- α (type variable) for an unknown, but nontrivial behavior.

In the following, let A range over α, $!B$, and $?B$; let $\star$ range over $\oplus$ and $\&$; and let a range over both A and $\star l$. The labelled transition system is given the set of (well-formed) types $\mathcal{T}$ as states, the set of *actions* ranged over by a, and the transition relation $\xrightarrow{a}$ defined by the rules in Figure 3. The transition relation makes use of an auxiliary judgment $S\checkmark$ that characterizes "terminated" session types that exhibit no further action [1]. The type skip has no action and a sequence $S_1;S_2$ has no action only if both S_1 and S_2 have no action. A μ-type has no action if that is the case for its unfolding. The rule for $\mu x.S\checkmark$ is inductive because $\mu x.S$ is contractive.

Apart from that, an A action reduces an A type to skip; $\star l_i$ selects branch l_i in a branch or choice type; and the remaining transitions define the standard left-to-right behavior of the sequence operator as well as the unrolling of the μ-operator.

Lemma 3.2. *For each S, either $S\checkmark$ or $\exists a, S'$ such that $S \xrightarrow{a} S'$.*

The labelled transition system is deterministic and thus image-finite and finitely branching, but it has infinite transition sequences ($\mu x.!B;x \xrightarrow{!B} \mu x.!B;x \xrightarrow{!B} \dots$) as well as transition sequences that visit infinitely many different states ($\mu y.?B;y;\alpha \xrightarrow{?B} \mu y.?B;y;\alpha;\alpha \xrightarrow{?B} \dots$).

Type bisimulation is defined in the standard way. We say that a binary relation $\mathcal{R}$ on types is a *bisimulation* if, whenever $S\mathcal{R}T$, for all a we have:

1. for all S' with $S \xrightarrow{a} S'$, there is T' such that $T \xrightarrow{a} T'$ and $S'\mathcal{R}T'$;
2. the converse, on transitions from T; i.e., for all T' with $T \xrightarrow{a} T'$, there is S' such that $S \xrightarrow{a} S'$ and $S'\mathcal{R}T'$.

Bisimilarity, written $\sim$, is the union of all bisimulations; thus $S \sim T$ holds if there is a bisimulation $\mathcal{R}$ such that $S\mathcal{R}T$. Basic properties of bisimilarity ensure that $\sim$ is an equivalence relation, and that $\sim$ is itself a bisimulation [17].

Two examples. Take $S_1 \triangleq \mu x.\oplus\{l\colon \alpha, m\colon x\}$ and $S_2 \triangleq \mu y.\oplus\{l\colon \mathsf{skip}, m\colon y\};\alpha$. We can easily show that $S_1 \sim S_2$ by exhibiting an appropriate bisimulation. Obviously the pair (S_1, S_2) must be in the relation. Then, using the rules in Figure 3, we conclude that $S_1 \xrightarrow{\oplus l} \alpha$ and $S_2 \xrightarrow{\oplus l} \alpha$. Then we add pair (α, α) to the relation. Because bisimulation is reflexive, we are done with this pair. The other transition from (S_1, S_2) is by label $\oplus m$; in this case we have $S_1 \xrightarrow{\oplus l} S_1$ and $S_2 \xrightarrow{\oplus l} S_2$. The bisimulation we seek is then $\{(S_1, S_2), (\alpha, \alpha)\}$.

Now take $T_1 \triangleq \mu x.?B;x$ and $T_2 \triangleq \mu y.?B;y;y$. We have $T_1 \xrightarrow{?B} T_1 \xrightarrow{?B} T_1 \dots$. We also have $T_2 \xrightarrow{?B} T_2;\alpha \xrightarrow{?B} T_2;\alpha;\alpha \dots$. Since these are the only available transitions, the relation $\{(T_1, T_2;\alpha^n) \mid n \geq 0\}$ is a bisimulation and contains the pair (T_1, T_2) when $n = 0$.

We now briefly explore the algebraic theory of bisimilarity, beginning with some basic laws.

Lemma 3.3 (Laws for terminated communication)**.** *If $S_1\checkmark$ and $S_2\checkmark$, then $S_1 \sim S_2$.*

Lemma 3.4 (Laws for sequential composition)**.**

$$(\mathsf{skip};S) \sim S$$
$$(S;\mathsf{skip}) \sim S$$
$$(S_1;S_2);S_3 \sim S_1;(S_2;S_3)$$
$$\star\{l_i\colon S_1\};S_2 \sim \star\{l_i\colon S_1;S_2\}$$

Proof. Each law is proved by exhibiting a suitable bisimulation. For example, for the distributivity law we use the relation that contains the identity relation as well as all pairs of the form $(\star\{l_i\colon S_1\};S_2, \star\{l_i\colon S_1;S_2\})$. □

Next we consider μ-types and substitution.

Lemma 3.5 (Laws for μ-types)**.**

$$\mu x.\mu y.S \sim \mu x.S[x/y]$$
$$\mu x.S \sim S \quad \textit{if} \quad x \notin \mathsf{free}(S)$$
$$\mu x.S \sim \mu y.S[y/x]$$
$$S[S'/x] \sim S[S''/x] \quad \textit{if} \quad S' \sim S''$$

Proof. Again, each law is proved by exhibiting a suitable bisimulation. For example, for the first case we use the relation that contain the identity relation as well as all pairs of the form $(\mu x.\mu y.S, \mu x.S[x/y])$ and $(\mu y.S[\mu x.\mu y.S/x], \mu x.S[x/y])$. □

Lemma 3.6. *Rewriting a type with one of the bisimilarities from Lemmas 3.4 and 3.5 does not affect well-formedness of a session type.*

Lemma 3.7 (Type equivalence preserves kinding)**.** *If $\Delta \vdash S_1 :: \mathcal{S}^m$ and $\Delta \vdash S_1 \sim S_2$ then $\Delta \vdash S_2 :: \mathcal{S}^m$.*

3.3 Type equivalence is decidable

It turns out that we can translate each well-formed session type into a guarded BPA (basic process algebra) process. The *expressions of recursive BPA processes* [2] are generated by the grammar

$$E ::= a \mid x \mid E_1 + E_2 \mid E_1;E_2 \mid \varepsilon$$

Here, a ranges over atomic actions, x over recursion variables, $E_1 + E_2$ denotes nondeterministic choice, $E_1;E_2$ stands for sequential composition, and ε stands for a terminated process. A *BPA process* is defined by a finite system of recursive process equations

$$\Theta = \{x_i = E_i \mid i, k \in \mathbf{N}, 0 \leq i \leq k\}$$

where the x_i are distinct and the E_i are BPA expressions with free variables in $\{x_1, \dots, x_k\}$. The variable x_0 is singled out as the root and the behavior of a process is defined as the behavior of x_0.

Definition 3.1. A BPA expression is *guarded* if every variable occurrence is within the scope of an atomic action. A system $\{x_i = E_i\}$ is guarded, if each E_i where x_i occurs in some E_j is guarded.

Definition 3.2. A guarded BPA process Θ defines a labelled transition system. The transition relation is the least relation $\xrightarrow{A}$ satisfying the following rules.

$$\varepsilon\checkmark \qquad \frac{E_1\checkmark \quad E_2\checkmark}{E_1; E_2\checkmark} \qquad \frac{E\checkmark}{x\checkmark}\ x = E \in \Theta$$

$$a \xrightarrow{a} \varepsilon \qquad \frac{E \xrightarrow{a} E'}{x \xrightarrow{a} E'}\ x = E \in \Theta$$

$$\frac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'}$$

$$\frac{E_1 \xrightarrow{a} E_1'}{E_1; E_2 \xrightarrow{a} E_1'; E_2} \qquad \frac{E_1\checkmark \quad E_2 \xrightarrow{a} E_2'}{E_1; E_2 \xrightarrow{a} E_2}$$

Theorem 3.8 ([7])**.** *Bisimilarity is decidable for guarded BPA processes.*

To reduce session type equivalence to bisimilarity of BPA processes, we need to exhibit a translation from (well-formed) sessions types to guarded BPA processes and show that this translation itself is a bisimulation.

To this end, we define an unraveling function for a session type S, $\mathsf{unr}(S)$, recursively by cases on the structure of S.

$$\mathsf{unr}(\mu x.S) = \mathsf{unr}(S[\mu x.S/x])$$

$$\mathsf{unr}(S; S') = \begin{cases} \mathsf{unr}(S') & \mathsf{unr}(S) = \mathsf{skip} \\ (\mathsf{unr}(S); S') & \mathsf{unr}(S) \neq \mathsf{skip} \end{cases}$$

$$\mathsf{unr}(S) = S \qquad \text{for all other cases}$$

The function unr is well-defined and terminating because we assume that the body of a recursive type is contractive.

To define the translation to BPA, we first show that, for a well-formed session type S, $\mathsf{unr}(S)$ is guarded.

Lemma 3.9 (Characterization of unr)**.** *Suppose that Δ does not bind recursion variables and that $\Delta \vdash S :: \kappa$ for $\kappa \leq \mathcal{S}^1$, then $\mathsf{unr}(S)$ is defined and yields either skip or a guarded type of the form O where*

$$O ::= A \mid \star\{\overline{l_i : S_i}\} \mid (O; S)$$

Now we define the translation of well-formed S to a BPA as follows. Assume that all recursion variable bindings are unique in the sense that the set $\{\mu x_1.S_1, \ldots, \mu x_n.S_n\}$ contains all μ-subterms of S with $S_i : \mathsf{p}$. Furthermore assume that for any free recursion variable $x_i \in \mathsf{free}(\mu x_j.S_j)$ it holds that $i < j$. That is, the μ-subterms are topologically sorted with respect to their lexical nesting.

Now define unrolled versions of the μ-subterms that have no free recursion variables. As we are just unrolling recursion, replacing $\mu_i.S_i$ by S_i' in S yields a term that is bisimilar to S.

$$\begin{aligned} S_1' &= S_1[\mu x_1.S_1/x_1] \\ S_2' &= S_2[\mu x_2.S_2/x_2][\mu x_1.S_1/x_1] \\ &\vdots \\ S_n' &= S_n[\mu x_n.S_n/x_n] \ldots [\mu x_1.S_1/x_1] \end{aligned}$$

Define the BPA process equations for S by

$$\begin{aligned} \mathsf{BPATop}(S) = \{ & \\ & x_0 = \mathsf{BPA}(S), \\ & x_1 = \mathsf{BPA}(\mathsf{unr}(S_1')), \quad \ldots \quad x_n = \mathsf{BPA}(\mathsf{unr}(S_n'))\} \end{aligned}$$

$$\begin{aligned} \mathsf{BPA}(\mathsf{skip}) &= \varepsilon \\ \mathsf{BPA}(A) &= A \\ \mathsf{BPA}(S_1; S_2) &= \mathsf{BPA}(S_1); \mathsf{BPA}(S_2) \\ \mathsf{BPA}(\star\{\overline{l_n : S_n}\}) &= (\star l_1; \mathsf{BPA}(S_1) + \cdots + \star l_n; \mathsf{BPA}(S_n)) \\ \mathsf{BPA}(\mu x.S) &= \begin{cases} x & S : \mathsf{p} \\ \varepsilon & S : \mathsf{s} \end{cases} \\ \mathsf{BPA}(x) &= x \end{aligned}$$

It is deliberate that we do **not** unravel the top-level type S in the defining equation for x_0. This equation need not be guarded because x_0 does not appear on the right-hand side of any equation. All other equations are translated from unraveled session types.

Lemma 3.10. *If $\mu x.S$ is a closed subterm of well-formed S_0 with $\Delta \vdash_c S : \mathsf{p}$, then $\mathsf{BPA}(\mathsf{unr}(S))$ is guarded with respect to $\mathsf{BPATop}(S_0)$.*

Proof. By Lemma 3.9, we know that $\mathsf{unr}(S)$ either yields skip or a term of the form O. The answer skip is ruled out by the assumption $\Delta \vdash_c S : \mathsf{p}$. Clearly, the translation of a type of shape O is guarded. □

It remains to show that S is bisimilar to its translation. Essentially, we want to prove that the function $\mathsf{BPA}(\cdot)$ is a bisimilation when considered as a relation.

Lemma 3.11. *If $\mathsf{unr}(S) = \mathsf{skip}$, then $S\checkmark$.*

Proof. Induction on the number n of recursive calls to unr.

Case $n = 0$. $S = \mathsf{skip}$ and $\mathsf{skip}\checkmark$.

Case $n > 0$.

Subcase $\mu x.S$. $\mathsf{unr}(\mu x.S) = \mathsf{skip}$ because $\mathsf{unr}(S[\mu x.S/x]) = \mathsf{skip}$. By induction, $S[\mu x.S/x]\checkmark$ and by applying the mu-DONE rule $\mu x.S\checkmark$.

Subcase $S_1; S_2$. $\mathsf{unr}(S_1; S_2) = \mathsf{skip}$ because $\mathsf{unr}(S_1) = \mathsf{unr}(S_2) = \mathsf{skip}$. By induction $S_1\checkmark$ and $S_2\checkmark$. By rule seq-DONE $S_1; S_2\checkmark$. □

Lemma 3.12. *Let S be closed, well-formed. Then $\mathsf{BPA}(S) \sim \mathsf{BPA}(\mathsf{unr}(S))$.*

Proof. Induction on the number n of recursive calls to unr.

Case $n = 0$. In this case, S must be skip, A, or $\star\{\overline{l_i : S_i}\}$ and the claim is immediate.

Case $n > 0$. There are two subcases.

Subcase $\mu x.S$. Then $\mathsf{BPA}(\mu x.S) = x$ and there is an equation $x = \mathsf{BPA}(\mathsf{unr}(S[\mu x.S]))$. Now, x is obviously bisimilar to $\mathsf{BPA}(\mathsf{unr}(S[\mu x.S]))$.

Subcase $S_1; S_2$. If $\mathsf{unr}(S_1) = \mathsf{skip}$, then $S_1\checkmark$ and hence $\mathsf{BPA}(S_1)\checkmark$. Furthermore, $\mathsf{unr}(S_1; S_2) = \mathsf{unr}(S_2)$ and, by induction, $\mathsf{BPA}(S_2) \sim \mathsf{BPA}(\mathsf{unr}(S_2))$. The result follows because $\mathsf{BPA}(S_1; S_2) = \mathsf{BPA}(S_1); \mathsf{BPA}(S_2) \sim \mathsf{BPA}(S_2)$ and $\mathsf{BPA}(\mathsf{unr}(S_2)) = \mathsf{BPA}(\mathsf{unr}(S_1; S_2))$.

If $\mathsf{unr}(S_1) =: S_u \neq \mathsf{skip}$, then $\mathsf{unr}(S_1; S_2) = S_u; S_2$. By induction, we know that $\mathsf{BPA}(S_1) \sim \mathsf{BPA}(S_u)$ and as bisimilarity is a congruence $\mathsf{BPA}(S_1; S_2) \sim \mathsf{BPA}(S_u); \mathsf{BPA}(S_2) = \mathsf{BPA}(\mathsf{unr}(S_1; S_2))$. □

Lemma 3.13. *Suppose S is a well-formed closed session type. If $S \xrightarrow{a} S'$, then $\mathsf{BPATop}(S) \xrightarrow{a} \mathsf{BPATop}(S')$.*

Proof. By induction on $S \xrightarrow{a} S'$.

Case $A \xrightarrow{A} \mathsf{skip}$. In this case $\mathsf{BPATop}(A) = \{x_0 = A\} \xrightarrow{A} \{x_0 = \varepsilon\} = \mathsf{BPATop}(\mathsf{skip})$.

Case $\star\{\overline{l_i : S_i}\} \xrightarrow{\star l_i} S_i$. In this case $\mathsf{BPATop}(\star\{\overline{l_i : S_i}\}) = \{x_0 = (\cdots + \star l_i; \mathsf{BPA}(S_i) + \ldots)\} \xrightarrow{\star l_i} \{x_0 = \mathsf{BPA}(S_i)\} = \mathsf{BPATop}(S_i)$.

Case $\frac{S_1 \xrightarrow{a} S_1'}{S_1;S_2 \xrightarrow{a} S_1';S_2}$. In this case $\mathsf{BPATop}(S_1;S_2) = \{x_0 = E_1;E_2,\ldots\}$ where $E_i = \mathsf{BPA}(S_i)$ for $i = 1,2$. Because $S_1 \xrightarrow{a} S_1'$, we obtain by induction that $\mathsf{BPATop}(S_1) = \{x_0 = E_1,\ldots\} \xrightarrow{a} \mathsf{BPATop}(S_1') = \{x_0 = E_1',\ldots\}$. Therefore, $\{x_0 = E_1;E_2,\ldots\} \xrightarrow{a} \{x_0 = E_1';E_2,\ldots\} = \mathsf{BPATop}(S_1';S_2)$.

Case $\frac{S_1\checkmark \quad S_2 \xrightarrow{a} S_2'}{S_1;S_2 \xrightarrow{a} S_2'}$. In this case $\mathsf{BPATop}(S_1;S_2) = \{x_0 = E_1;E_2,\ldots\}$ where $E_i = \mathsf{BPA}(S_i)$ for $i = 1,2$. It is easy to see that $S_1\checkmark$ implies $\mathsf{BPA}(S_1)\checkmark$, that is, $E_1\checkmark$. Because $S_2 \xrightarrow{a} S_2'$, we obtain by induction that $\mathsf{BPATop}(S_2) = \{x_0 = E_2,\ldots\} \xrightarrow{a} \mathsf{BPATop}(S_2') = \{x_0 = E_2',\ldots\}$. Therefore, $\{x_0 = E_1;E_2,\ldots\} \xrightarrow{a} \{x_0 = E_2',\ldots\} = \mathsf{BPATop}(S_2')$.

Case $\frac{S[\mu x.S/x] \xrightarrow{a} S'}{\mu x.S \xrightarrow{a} S'}$. In this case $\mathsf{BPATop}(\mu x.S) = \{x_0 = x, x = E,\ldots\}$ with $E = \mathsf{BPA}(\mathsf{unr}(S[\mu x.S/x]))$. By induction, $\mathsf{BPATop}(S[\mu x.S/x]) \xrightarrow{a} \mathsf{BPATop}(S')$. Now $\mathsf{BPATop}(S[\mu x.S/x]) = \{x_0 = \mathsf{BPA}(S[\mu x.S/x]),\ldots\}$ which proves the claim because $x_0 \sim E$ by Lemma 3.12. □

Lemma 3.14. *Suppose that* $\mathsf{BPA}(\mathsf{unr}(S)) \xrightarrow{a} E'$. *Then* $S \xrightarrow{a} S'$ *and* $E' = \mathsf{BPATop}(S')$.

Proof. By induction on the number n of recursive calls of unr.

Case $n = 0$.

Subcase $S = \mathsf{skip}$. Contradictory.

Subcase $S = A$. Then $a = A$, $E' = \varepsilon$, and $S' = \mathsf{skip}$.

Subcase $S = \star\{\overline{l_i : S_i}\}$. Then $a = \star l_i$, $E' = \mathsf{BPA}(S_i)$, and $S' = S_i$.

Case $n > 0$.

Subcase $S = S_1;S_2$. If $\mathsf{unr}(S_1) = \mathsf{skip}$, then $\mathsf{unr}(S) = \mathsf{unr}(S_2)$ with less than n calls. As $\mathsf{BPA}(\mathsf{unr}(S_2)) \xrightarrow{a} E'$, induction yields that $S_2 \xrightarrow{a} S'$ and $E' = \mathsf{BPATop}(S')$. As $\mathsf{unr}(S_1) = \mathsf{skip}$, we know that $S_1\checkmark$. Hence, $S_1;S_2 \xrightarrow{a} S'$ and $E' = \mathsf{BPATop}(S')$.

If $\mathsf{unr}(S_1) \neq \mathsf{skip}$, then consider $\mathsf{BPA}(\mathsf{unr}(S_1);S_2) \xrightarrow{a} E'$ because $\mathsf{BPA}(\mathsf{unr}(S_1)) \xrightarrow{a} E_1'$, so that induction yields some S_1' such that $S_1 \xrightarrow{a} S_1'$ and $E_1' = \mathsf{BPATop}(S_1')$.

Subcase $\mu x.S$. $\mathsf{unr}(\mu x.S) = \mathsf{unr}(S[\mu x.S/x])$ with one less invocation. As $\mathsf{BPA}(\mathsf{unr}(S[\mu x.S/x])) \xrightarrow{a} E'$, induction yields that $S[\mu x.S/x] \xrightarrow{a} S'$ with $E' = \mathsf{BPATop}(S')$. □

Lemma 3.15. *Suppose that* S *is well-formed and let* $\Theta = \mathsf{BPATop}(S)$ *and* $\Theta \xrightarrow{a} \Theta'$.

There is some S' *such that* $S \xrightarrow{a} S'$ *and* $\Theta' = \mathsf{BPATop}(S')$.

Proof. By induction on S.

Case skip. Contradictory.

Case A. For Θ, $A \xrightarrow{a} \varepsilon$. Choose $S' = \mathsf{skip}$.

Case $\star\{\overline{l_i : S_i}\}$. For Θ, $\sum \overline{\star l_i;\mathsf{BPA}(S_i)} \xrightarrow{\star l_i} \mathsf{BPA}(S_i)$. Choose $S' = S_i$.

Case $S_1;S_2$. If $\mathsf{BPA}(S_1) \xrightarrow{a} E_1'$, then $S_1 \xrightarrow{a} S_1'$ and $E_1' = \mathsf{BPA}(S_1')$, by induction. Now, $\mathsf{BPA}(S_1;S_2) \xrightarrow{a} E_1';\mathsf{BPA}(S_2) = \mathsf{BPA}(S_1';S_2)$. Choose $S' = S_1';S_2$.

If $\mathsf{BPA}(S_1)\checkmark$ and $\mathsf{BPA}(S_2) \xrightarrow{a} E_2'$, then $S_1\checkmark$ and $S_2 \xrightarrow{a} S_2'$ and $E_2' = \mathsf{BPA}(S_2')$, by induction. Now, $\mathsf{BPA}(S_1;S_2) \xrightarrow{a} E_2' = \mathsf{BPA}(S_2')$. Choose $S' = S_2'$.

Case $\mu x.S$.
$\Theta = \mathsf{BPATop}(\mu x.S) = \{x_0 = x, x = \mathsf{BPA}(\mathsf{unr}(S[\mu x.S/x]))\}$.

$$
\begin{array}{rcl}
v & ::= & \mathsf{send} \mid \mathsf{receive} \mid () \mid \lambda a.e \mid (v,v) \mid \mathsf{in}\, l\, v \\
e & ::= & v \mid a \mid \mathsf{new} \mid ee \mid \mathsf{fix}\, a.e \mid (e,e) \mid \mathsf{in}\, l\, e \\
 & & \mid \mathsf{let}\, a,b = e\, \mathsf{in}\, e \mid \mathsf{fork}\, e \mid \mathsf{match}\, e\, \mathsf{with}\, [l_i \to e_i]_{i\in I} \\
 & & \mid \mathsf{select}\, l\, e \mid \mathsf{case}\, e\, \mathsf{of}\, \{l_i \to e_i\}_{i\in I} \\
p & ::= & e \mid p|p \mid (\nu a,b)p
\end{array}
$$

Figure 4. Values, expressions, and processes

If $\Theta \xrightarrow{a} \Theta'$, then it must be because $\mathsf{BPA}(\mathsf{unr}(S[\mu x.S/x])) \xrightarrow{a} E'$. Use Lemma 3.14 to establish the claim. □

Theorem 3.16. *Suppose that* S *is well-formed and let* $\Theta = \mathsf{BPATop}(S)$.

1. *If* $S \xrightarrow{a} S'$, *then* $\Theta \xrightarrow{a} \Theta'$ *with* $\Theta' = \mathsf{BPATop}(S')$.
2. *If* $\Theta \xrightarrow{a} \Theta'$, *then* $S \xrightarrow{a} S'$ *with* $\Theta' = \mathsf{BPATop}(S')$.

Proof. By Lemmas 3.13 and 3.15. □

4. Processes, statics, and dynamics

This section introduces our programming language, its static and dynamic semantics. It shows that typing is preserved by reduction and concludes by showing that our type system is a conservative extension of a conventional functional session type language.

4.1 Expressions and processes

Fix a base set of *term variables*, disjoint from those introduced before. Let a, b range over this set. The syntax of values, expressions, and processes is described in Figure 4.

Expressions, denoted by metavariable e, incorporate a *standard functional core* composed of term variables a, abstraction introduction $\lambda a.e$ and elimination ee, pair introduction (e,e) and elimination $\mathsf{let}\, a,b = e\, \mathsf{in}\, e$, datatype introduction $\mathsf{in}\, l\, e$ and elimination $\mathsf{match}\, e\, \mathsf{with}\, [l_i \to e_i]_{i\in I}$, as well as a fixed point construction $\mathsf{fix}\, e$. Further expressions support the usual *session operators*, in the form of channel creation new, message sending send and receiving receive, internal choice (or label selection) $\mathsf{select}\, l\, e$, and external choice (or branching) $\mathsf{case}\, e\, \mathsf{of}\, \{l_i \to e_i\}_{i\in I}$. Concurrency arises from a fork operator, spawning a new process.

Processes, denoted by metavariable p, are expressions e, the parallel composition of two processes $p|q$, and the scope restriction $(\nu a,b)p$ of a channel described by its two end points, a and b.

4.2 Operational semantics

The *binding* occurrences for term variables a and b are expressions $\lambda a.e$, $\mathsf{fix}\, a.e$, and $\mathsf{let}\, a,b = e_1\, \mathsf{in}\, e_2$. The sets of free and bound term variables are defined accordingly, and so is the *capture avoiding substitution* of a variable a by a value v in a term e, denoted by $e[v/a]$.

The operational semantics makes use of a *structural congruence* relation on processes, $\equiv$, defined as the smallest relation that includes the commutative monoid rules—binary operator $|$ and neutral $()$—and scope extrusion:

$$(\nu a,b)p|q \equiv (\nu a,b)(p|q) \quad \text{if } a,b \text{ not free in } q$$

The operational semantics is call-by-value: expressions are reduced to values before being "applied". The syntax of values is in Figure 4 enumerates the values send, receive, unit, lambda abstraction, pair of values, and injection of a value in a sum type. The semantics combines a standard reduction relation for the functional part with an also standard message passing semantics of the π-calculus. The rules are in Figure 5.

$$(\lambda a.e)v \to e[v/a]$$
$$\mathsf{let}\, a, b = (u, v)\, \mathsf{in}\, e \to e[u/a][v/b]$$
$$\mathsf{match}\, (\mathsf{in}\, l_j\, v)\, \mathsf{with}\, [l_i \to e_i] \to e_j v$$
$$\mathsf{fix}\, a.e \to e[\mathsf{fix}\, a.e/a]$$
$$\frac{e_1 \to e_2}{E[e_1] \to E[e_2]}$$
$$E[\mathsf{fork}\, e] \to E[()]|e$$
$$E[\mathsf{new}] \to (\nu a, b)E[(a, b)]$$
$$(\nu a, b)(E_1[\mathsf{send}\, v\, a]|E_2[\mathsf{receive}\, b]) \to (\nu a, b)(E_1[a]|E_2[(v, b)])$$
$$(\nu a, b)(E_1[\mathsf{select}\, l_j\, a]|E_2[\mathsf{case}\, b\, \mathsf{of}\, \{l_i \to e_i\}]) \to (\nu a, b)(E_1[a]|E_2[e_j b])$$
$$\frac{p \to p'}{p|q \to p'|q} \qquad \frac{p \to p'}{(\nu\, a, b)p \to (\nu\, a, b)p'} \qquad \frac{p \equiv q \quad q \to q'}{p \to q'}$$

Context E_1 (resp. E_2, resp. E) does not bind a (resp. b, resp. a and b).
Dual $(\nu b, a)$ rules for send/receive and select/case omitted.

Figure 5. Evaluation contexts and reduction relation

The first four axioms are standard in functional call-by-value languages, and comprise β-reduction, (linear) pair elimination, data type elimination, and fixed-point unrolling. The first rule in the figure allows reduction to happen underneath (functional) evaluation contexts E as defined by the grammar below.

$$\begin{aligned} E \;::=\; & (E, e) \mid (v, E) \mid Ee \mid vE \mid \mathsf{let}\, a, b = E\, \mathsf{in}\, e \\ & \mid \mathsf{case}\, E\, \mathsf{of}\, \{l_i \to e_i\} \mid \mathsf{select}\, l\, E \\ & \mid \mathsf{match}\, E\, \mathsf{with}\, [l_i \to e_i] \mid \mathsf{in}\, l\, E \end{aligned}$$

The fork operator created new threads: the expression $\mathsf{fork}\, e$ evaluates to $()$, the unit value, while creating a new thread to run expression e concurrently.

The following three axioms in the figure deal with session operations. The new operator creates a new channel. Channels are denoted by their two end points, a and b in this case. We require that context E does not bind variables a, b, so that these are bound by the outermost channel binding, $(\nu a, b)$.

The send-receive rule captures message passing: the sending process writes value v in channel end point a whereas the receiving process reads it from channel end point b. That the pair a-b forms the two end points of a channel is captured by the outermost $(\nu a, b)$ binding. The result of sending a value on channel end a is a itself; that of receiving on b is the pair (v, b). In this way both threads are able to use their channel ends for further interaction. This "rebinding" of channel ends provide for a standard treatment of a and b as linear values. It is also the type system that makes sure that, in a given process, there is exactly one thread holding a copy of a given channel end, thus allowing a simplified reduction rule where one finds exactly two threads underneath channel binder $(\nu a, b)$.

The rule for select-case is similar in spirit. One thread selects an l-labeled option on a channel end, whereas another offers a choice on the other channel end. After successful interaction, the selecting thread is left with its channel end, a, whereas the choice-offering thread is left with an application of the branch that was selected, e_j, to its channel end, b.

The remaining three rules are standard in the π-calculus. The first two allow reduction underneath parallel composition and scope restriction; the last incorporates structural congruence in the reduction relation.

As an example consider an expression that creates a new channel, forks a thread that writes an integer on the channel and reads back its successor. The original thread, in turn, waits for an integer value and writes back its successor. We depict the reduction below, where we make use of the conventional semicolon operator $e_1; e_2$ defined as $(\lambda a.e_2)e_1$, where a is a variable not occurring free in e_2. We also write $\mathsf{let}\, a = e_1\, \mathsf{in}\, e_2$ for $(\lambda a.e_2)e_1$.

$$\begin{aligned} &\mathsf{let}\, a, b = \mathsf{new}\, \mathsf{in}\, \mathsf{fork}\, (\mathsf{let}\, c = \mathsf{send}\, 5\, a\, \mathsf{in}\, \mathsf{receive}\, c); \\ &\qquad \mathsf{let}\, n, d = \mathsf{receive}\, b\, \mathsf{in}\, \mathsf{send}\, (n+1)\, d \to \\ &\quad (\nu e, f)\mathsf{let}\, a, b = (e, f)\, \mathsf{in} \ldots \to \\ &(\nu e, f)\mathsf{fork}\, (\mathsf{let}\, c = \mathsf{send}\, 5\, e\, \mathsf{in} \ldots); \mathsf{let}\, n, d = \mathsf{receive}\, f\, \mathsf{in} \ldots \to\to \\ &\quad (\nu e, f)(\mathsf{let}\, c = \mathsf{send}\, 5\, e\, \mathsf{in} \ldots | \mathsf{let}\, n, d = \mathsf{receive}\, f\, \mathsf{in} \ldots) \to \\ &\quad (\nu e, f)(\mathsf{let}\, c = e\, \mathsf{in} \ldots | \mathsf{let}\, n, d = (5, f)\, \mathsf{in} \ldots) \to\to \\ &\quad (\nu e, f)(\mathsf{receive}\, e | \mathsf{send}\, (5+1)\, f) \to \\ &\quad (\nu e, f)((e, 6)|f) \end{aligned}$$

We complete this section by discussing the notion of *run-time errors*. The *subject* of an expression e, denoted by $\mathsf{subj}(e)$, is a in the following cases and undefined in all other cases.

$$\mathsf{send}\, e\, a \qquad \mathsf{receive}\, a \qquad \mathsf{select}\, e\, a \qquad \mathsf{case}\, a\, \mathsf{of}\, e$$

We say that two expressions e_1 and e_2 *agree* on channel ab, denoted $\mathsf{agree}^{ab}(e_1, e_2)$, in the following four cases.

$$\mathsf{agree}^{ab}(\mathsf{send}\, e\, a, \mathsf{receive}\, b) \qquad \mathsf{agree}^{ab}(\mathsf{receive}\, a, \mathsf{send}\, e\, b)$$
$$\mathsf{agree}^{ab}(\mathsf{select}\, e\, a, \mathsf{case}\, b\, \mathsf{of}\, e) \qquad \mathsf{agree}^{ab}(\mathsf{case}\, a\, \mathsf{of}\, e, \mathsf{select}\, e\, b)$$

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

1. $\mathsf{let}\, a, b = v\, \mathsf{in}\, e$ and v is not a pair;
2. $\mathsf{match}\, v\, \mathsf{with}\, [l_i \to e_i]_{i \in I}$ and $v \neq (\mathsf{in}\, l_j\, v')$ for some v' and some $j \in I$;
3. $E_1[e_1]|E_2[e_2]$ and $\mathsf{subj}(e_1) = \mathsf{subj}(e_2) = a$ where neither E_1 nor E_2 bind a;
4. $(\nu\, ab)(E_1[e_1]|E_2[e_2]|p)$ and $\mathsf{subj}(e_1) = a$ and $\mathsf{subj}(e_2) = b$ and $\neg\mathsf{agree}^{ab}(e_1, e_2)$ where E_1 does not bind a and E_2 does not bind b;
5. $(\nu\, ab)(E_1[\mathsf{select}\, l_j\, a]|E_2[\mathsf{case}\, b\, \mathsf{of}\, \{l_i \to e_i\}_{i \in I}]|p)$ where E_1 does not bind a and E_2 does not bind b and $j \notin I$ (or the same with a and b swapped).

4.3 Type assignment system

Duality is a central notion in session types. It allows to "switch" the point of view from one end of a channel (say, the client side) to the other (the server side). The duality function on session types, $\overline{S}$, is defined as follows.

$$\overline{\alpha} = \alpha \qquad \overline{\mathsf{skip}} = \mathsf{skip} \qquad \overline{!B} = ?B \qquad \overline{?B} = !B$$
$$\overline{\&\{l_i\colon S_i\}} = \oplus\{l_i\colon \overline{S_i}\} \qquad \overline{\oplus\{l_i\colon S_i\}} = \&\{l_i\colon \overline{S_i}\}$$
$$\overline{S_1; S_2} = \overline{S_1}; \overline{S_2} \qquad \overline{\mu x.S} = \mu x.\overline{S} \qquad \overline{x} = x$$

This simple definition is justified by the fact that the types we consider are first order. Hence, we avoid a complication known to arise in the presence of higher-order recursion [3].

To check whether S_1 is dual to S_2 we compute $S_3 = \overline{S_1}$ and check S_2 and S_3 for equivalence. Duality is clearly an involution ($\overline{\overline{S}} = S$), hence we can alternatively compute $\overline{S_2}$ and check that S_1 is equivalent to $\overline{S_2}$. For example, to check that $!B; \mu x.(\mathsf{skip}; !B; x)$ is dual to $\mu y.(?B; y)$, we compute $\overline{\mu y.(?B; y)}$

to obtain $\mu y.(!B; y)$, and check that this type is equivalent to $!B; \mu x.(\mathsf{skip}; !B; x)$.

We can easily show that duality preserves kinding.

Lemma 4.1. *If* $\Delta \vdash S :: \kappa$ *then* $\Delta \vdash \overline{S} :: \kappa$.

Proof. By rule induction on the hypothesis. □

Typing environments are generated by the following grammar.

$$\Gamma \ ::= \ \cdot \ \mid \ \Gamma, a\colon T$$

As before we consider environments up to reordering of their entries. However, unlike conventional environments (Δ for example), we allow duplicated variables in environments, but subject to two restrictions: if both $a\colon T$ and $a\colon U$ are in Γ then: i) $T = U$ and ii) $\Delta \vdash T :: \mathcal{T}^{\mathbf{u}}$.

The un_Δ predicate, on types T defined on Δ, is an abbreviation of $\Delta \vdash T :: \mathcal{T}^{\mathbf{u}}$. The un predicate is also true of environments of the form $x_1\colon T_1, \ldots, x_n\colon T_n$ if it is true of all types $T_1, \ldots, T_n$. We often omit the Δ in un_Δ when this is obvious from the context.

Figure 6 contains the typing rules fo expressions and for processes. Judgements for expressions and processes take the usual forms of $\Delta; \Gamma \vdash e : T$ and $\Delta; \Gamma \vdash p$. We now briefly describe the rules.

The first group of rules in the figure deals with the functional part of the language. The rule for the unit value requires an environment free from term variables. If needed, unrestricted term variables are introduced by an explicit weakening rule. The typing rule for variables reads the type of the variable from the environment. We require that the term environment contains no other entry, and that the type is well-formed against Δ, ensuring that types introduced in a derivation are well-formed. The rule for the fixed point is standard.

The type system comprises two rules for abstraction, introducing unrestricted ($\to$) and linear ($\multimap$) functions. In the former case we require an unrestricted context. The rule for function elimination is standard. Two points related to linear type systems are worth remarking: i) the term context is split in two parts, one to type the function e_1, the other to type the argument e_2, and ii) the rule requires a linear function. The next rule in the same line allows unrestricted functions to be converted into linear functions, so that the rule for function elimination may apply.

The subsequent four rules are all standard and provide for the introduction and the elimination of pairs ($T_1 \otimes T_2$) and variants ($[l_i\colon T_i]$). The rules for the introduction and elimination of type abstraction are also conventional; the extra premises on kindings are meant to ensure that types introduced in derivations are well-formed.

We now come to the channel communication rules. The rule for channel creation introduces a pair of dual session types, one for each end point. The send operator is a function that expects a value to be sent B, then a channel on which to send the value $!B; T$, and returns the rest of the channel T. The receive operator expects a channel from which a value can be read $?B; T$ and returns a pair composed of the value and the rest of the channel, $B \otimes T$. The premises ensure that T is a session type. The rule for label selection requires that expression e denotes a channel offering an internal choice, $\oplus\{l_i\colon T_i\}$. Expression $\mathsf{select}\, l_j\, e$ evaluates to the rest of the channel, hence its type is T_j. A case expression expects a channel offering an external choice, $\&\{l_i\colon T_i\}$. The expression in each branch must be function expecting the rest of the channel T_i. All such functions must produce a value of a common type T, which becomes the type of the case expression.

The last three rules are structural. The first two —weakening and copy—manipulate the term environment. In both cases we require the type to be unrestricted. The last rule incorporates type equivalence in typing.

The rules for processes should be easy to understand. An expression, when seen as a process must be of an unrestricted type. This implies that linear resources, channels in particular, are fully consumed. The rule for parallel composition splits the environment in two, using each part for one of the processes. Finally, the rule for channel creation introduces two entries in the context, of types dual to each other, one for each end of the channel.

We complete this section with a result that relates the type system to the kinding system.

Lemma 4.2 (Agreement)**.** *If* $\Delta; a_1\colon T_1, \ldots, a_n\colon T_n \vdash e : T_0$ *then, for all* $0 \le i \le n$*, there are kinds* κ_i *such that* $\Delta \vdash T_i :: \kappa_i$.

Typing for expressions, $\Delta; \Gamma \vdash e : T$

$$\frac{}{\Delta; \cdot \vdash ()\colon \mathsf{unit}} \qquad \frac{\Delta \vdash T :: \kappa}{\Delta; a\colon T \vdash a : T} \qquad \frac{\Delta; \Gamma, a\colon T \vdash e : T}{\Delta; \Gamma \vdash \mathsf{fix}\, a.e : T}$$

$$\frac{\Delta; \Gamma, a\colon T_1 \vdash e : T_2 \quad \mathsf{un}(\Gamma)}{\Delta; \Gamma \vdash \lambda a.e : T_1 \to T_2} \qquad \frac{\Delta; \Gamma, a\colon T_1 \vdash e : T_2}{\Delta; \Gamma \vdash \lambda a.e : T_1 \multimap T_2}$$

$$\frac{\Delta; \Gamma_1 \vdash e_1 : T_1 \multimap T_2 \quad \Delta; \Gamma_2 \vdash e_2 : T_1}{\Delta; \Gamma_1, \Gamma_2 \vdash e_1 e_2 : T_2} \qquad \frac{\Delta; \Gamma \vdash e : T_1 \to T_2}{\Delta; \Gamma \vdash e : T_1 \multimap T_2}$$

$$\frac{\Delta; \Gamma_1 \vdash e_1 : T_1 \quad \Delta; \Gamma_2 \vdash e_2 : T_2}{\Delta; \Gamma_1, \Gamma_2 \vdash (e_1, e_2) : T_1 \otimes T_2}$$

$$\frac{\Delta; \Gamma_1 \vdash e_1 : T_1 \otimes T_2 \quad \Delta; \Gamma_2, a\colon T_1, b\colon T_2 \vdash e_2 : U}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathsf{let}\, a, b = e_1\, \mathsf{in}\, e_2 : U}$$

$$\frac{\Delta; \Gamma \vdash e : T_j \quad j \in I \quad \Delta \vdash T_i :: \mathcal{T}^m}{\Delta; \Gamma \vdash \mathsf{in}\, l_j\, e : [l_i\colon T_i]_{i \in I}}$$

$$\frac{\Delta; \Gamma_1 \vdash e : [l_i\colon T_i] \quad \Delta; \Gamma_2 \vdash e_i : T_i \multimap T}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathsf{match}\, e\, \mathsf{with}\, [l_i\colon e_i] : T}$$

$$\frac{\Delta; \Gamma \vdash e : T \quad \alpha \notin \Delta, \Gamma \quad \Delta, \alpha :: \kappa \vdash T :: \mathcal{C}^m \quad \kappa \le \mathcal{T}^{\mathbf{l}}}{\Delta; \Gamma \vdash e : \forall \alpha :: \kappa.T}$$

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha :: \kappa.T_1 \quad \Delta \vdash T_2 :: \kappa}{\Delta; \Gamma \vdash e : T_1[T_2/\alpha]}$$

$$\frac{\Delta \vdash T :: \mathcal{S}^m}{\Delta; \cdot \vdash \mathsf{new} : T \otimes \overline{T}} \qquad \frac{\Delta \vdash T :: \mathcal{S}^m}{\Delta; \cdot \vdash \mathsf{send} : B \to\, !B; T \to T}$$

$$\frac{\Delta \vdash T :: \mathcal{S}^m}{\Delta; \cdot \vdash \mathsf{receive} : ?B; T \to (B \otimes T)}$$

$$\frac{\Delta; \Gamma \vdash e : \oplus\{l_i\colon T_i\}_{i \in I} \quad j \in I}{\Delta; \Gamma \vdash \mathsf{select}\, l_j\, e\colon T_j}$$

$$\frac{\Delta; \Gamma \vdash e : \&\{l_i\colon T_i\} \quad \Delta; \Gamma_2 \vdash e_i : T_i \multimap T}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathsf{case}\, e\, \mathsf{of}\, l_i\colon e_i : T}$$

$$\frac{\Delta; \Gamma \vdash e : T \quad \mathsf{un}(\Gamma, T)}{\Delta; \Gamma \vdash \mathsf{fork}\, e : \mathsf{unit}} \qquad \frac{\Delta; \Gamma \vdash e : T_1 \quad \mathsf{un}(T_2) \quad a \notin \Gamma}{\Delta; \Gamma, a\colon T_2 \vdash e : T_1}$$

$$\frac{\Delta; \Gamma, a\colon T_1, a\colon T_1 \vdash e : T_2 \quad \mathsf{un}(T_1)}{\Delta; \Gamma, a\colon T_1 \vdash e : T_2}$$

$$\frac{\Delta; \Gamma \vdash e : T_1 \quad \Delta \vdash T_1 \sim T_2}{\Delta; \Gamma \vdash e : T_2}$$

Typing for processes, $\Delta; \Gamma \vdash P$

$$\frac{\Delta; \Gamma \vdash e : T \quad \mathsf{un}(T)}{\Delta; \Gamma \vdash e}$$

$$\frac{\Delta; \Gamma_1 \vdash P_1 \quad \Delta; \Gamma_2 \vdash P_2}{\Delta; \Gamma_1, \Gamma_2 \vdash p_1 \mid p_2} \qquad \frac{\Delta; \Gamma, a\colon T, b\colon \overline{T} \vdash p}{\Delta; \Gamma \vdash (\nu\, a, b)p}$$

Figure 6. Typing for expressions and processes

Proof. By rule induction on the hypothesis using the various kinding preservation lemmas (3.1, 3.7, and 4.1). □

4.4 Soundness and type safety

Lemma 4.3 (Strengthening)**.** *If* $\Delta; \Gamma, a\colon T \vdash p$ *and* a *not free in* p *then* $\Delta; \Gamma \vdash p$ *and* $\mathsf{un}(T)$.

Proof. By rule induction on the first hypothesis. □

Lemma 4.4 (Congruence)**.** *If* $\Delta; \Gamma \vdash p$ *and* $p \equiv q$ *then* $\Delta; \Gamma \vdash q$.

Proof. By rule induction on the first hypothesis, using strengthening (Lemma 4.3). □

Lemma 4.5 (Substitution)**.** *If* $\Delta; \Gamma_1, a\colon T_2 \vdash e_1 : T_1$ *and* $\Delta; \Gamma_2 \vdash e_2 : T_2$ *then* $\Delta; \Gamma_1, \Gamma_2 \vdash e_1[e_2/a] : T_1$.

Proof. By rule induction on the first hypothesis. □

Lemma 4.6 (Sub-derivation introduction)**.** *If* $\mathcal{D}_1$ *is a derivation of* $\Delta; \Gamma \vdash E[e] : T_1$ *then* $\Gamma = \Gamma_1, \Gamma_2$ *and* $\mathcal{D}_1$ *has a sub-derivation* $\mathcal{D}_2$ *concluding* $\Delta; \Gamma_2 \vdash e : T_2$ *such that the position of* $\mathcal{D}_2$ *in* $\mathcal{D}_1$ *corresponds to the hole in* $E[]$.

Lemma 4.7 (Sub-derivation elimination)**.** *If* $\mathcal{D}_1$ *is a derivation of* $\Delta; \Gamma_1, \Gamma_2 \vdash E[e_1] : T_1$ *and* $\mathcal{D}_2$ *is a sub-derivation of* $\mathcal{D}_1$ *concluding* $\Delta; \Gamma_2, \Gamma_3 \vdash e_1 : T_2$ *and the position of* $\mathcal{D}_2$ *in* $\mathcal{D}_1$ *corresponds to the hole in* $E[]$ *and* $\Delta; \Gamma_2 \vdash e_2 : T_2$ *then* $\Delta; \Gamma_1, \Gamma_2, \Gamma_2 \vdash E[e_2] : T_1$.

The structural rules (weak, copy, and $\sim$) commute. The following fat rule is admissible.

$$\frac{\Delta; \Gamma_1, \Gamma_2, \Gamma_2 \vdash e : T_1 \quad \mathsf{un}(\Gamma_2, \Gamma_3) \quad \Delta \vdash T_1 \sim T_2}{\Delta; \Gamma_1, \Gamma_2, \Gamma_3 \vdash e : T_2}$$

Notice that if we replace, in the weak rule, the proviso $a \notin \Gamma$ by $a : U \in \Gamma \Rightarrow \Delta \vdash U \sim T$, then copy and weak do not commute anymore. This fat rule forms the basis for the inversion lemma below.

Lemma 4.8 (Inversion)**.**

- *If* $\Delta; \Gamma \vdash e$ *then* $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$ *and* $\mathsf{un}(\Gamma_2, \Gamma_3)$ *and* $\Delta; \Gamma_1, \Gamma_2, \Gamma_2 \vdash e\colon T$ *and* $\mathsf{un}(T)$.
- *If* $\Delta; \Gamma \vdash \mathsf{let}\, a, b = e_1 \,\mathsf{in}\, e_2 : T$ *then* $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$ *and* $\mathsf{un}(\Gamma_2, \Gamma_3)$ *and* $\Gamma_4, \Gamma_5 = \Gamma_1, \Gamma_2, \Gamma_2$ *and* $\Delta; \Gamma_4 \vdash e_1 : T_1 \otimes T_2$ *and* $\Delta; \Gamma_5, a\colon T_1, b\colon T_2 \vdash e_2 : \tau = T$.
- *If* $\Delta; \Gamma \vdash (e_1, e_2) : T$ *then* $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$ *and* $\mathsf{un}(\Gamma_2, \Gamma_3)$ *and* $\Gamma_4, \Gamma_5 = \Gamma_1, \Gamma_2, \Gamma_2$ *and* $\Delta; \Gamma_4 \vdash e_1 : T_2$ *and* $\Delta; \Gamma_5 \vdash e_2 : T_2$ *and* $T_1 \otimes T_2 = T$
- *If* $\Delta; \Gamma \vdash \mathsf{fork}\ e : T$ *then* $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$ *and* $\Delta; \Gamma_1, \Gamma_2, \Gamma_2 \vdash e : \mathsf{unit}$ *and* $\mathsf{un}(\Gamma)$ *and* $\Delta \vdash T \sim \mathsf{unit}$.
- *If* $\Delta; \Gamma \vdash \mathsf{new} : T$ *then* $\mathsf{un}(\Gamma)$ *and* $\Delta \vdash T \sim S \otimes \overline{S}$.
- *If* $\Delta; \Gamma_1 \vdash \mathsf{receive}\, a : T_1$ *then* $\Delta \vdash T_1 \sim T \otimes S$ *and* $\Gamma_1 = \Gamma_2, a\colon T_2$ *and* $a \notin \Gamma_2$ *and* $\Delta \vdash T_2 \sim ?B; S$.
- *If* $\Delta; \Gamma \vdash \mathsf{send}\, e\, a : T$ *then* $\Delta \vdash T \sim S$ *and* $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3, a\colon T_2$ *and* $\mathsf{un}(\Gamma_2, \Gamma_3)$ *and* $\Delta \vdash T_2 \sim !B; S$ *and* $\Delta; \Gamma_1, \Gamma_2, \Gamma_2 \vdash e\colon T_1$ *and* $\Delta \vdash T_1 \sim T$.

Theorem 4.9 (Soundness)**.** *If* $\Delta; \Gamma \vdash p$ *and* $p \to q$ *then* $\Delta; \Gamma \vdash q$.

Proof. By rule induction on the second premise, using the congruence and the substitution lemmas, sub-derivation introduction and elimination, and inversion (Lemmas 4.4, 4.5, 4.6, 4.7, 4.8).

Case the derivation ends with the reduction rule for communication, use inversion (new, par, process twice), sub-derivation intro (twice), inversion (send, receive), typing rules for variables and $\sim$, sub-derivation elim (twice), typing rules for processes (twice) and par and weak and copy, definition of $\overline{S}$, and typing rule new.

Case the derivation ends with the rule for branching: similar to the above, but simpler.

Case the derivation ends with channel creation: inversion, sub-derivation intro, inversion, var axiom, $\otimes$ intro, sub-derivation elimination, rules proc and new.

Case the derivation ends with fork: sub-derivation intro, inversion, copy, sub-derivation elimination, rules proc and par and copy.

Case the derivation ends with par: inversion, induction, typing rule for parallel composition.

Case the derivation ends with reduction under new: inversion, induction, rule new.

Case the derivation ends with $\equiv$: congruence lemma, induction.

Case the derivation ends with context: sub-derivation intro, induction, sub-derivation elim.

Case the derivation ends with β: inversion, substitution lemma.

Case the derivation ends with let: rule proc, inversion, substitution lemma (twice), rules weak and copy.

Case the derivation ends with match: inversion, application rule.

Case the derivation ends with fix: inversion, substitution lemma, contraction (copy rule). □

Theorem 4.10 (Type safety)**.** *If* $\Delta; \Gamma \vdash p$ *then* p *is not an error.*

Proof. A simple analysis of the typing derivation for the hypothesis. We analyse one such case. If $(\nu\, ab)(E_1[e_1]|E_2[e_2]|p)$ and $\mathsf{subj}(e_1) = a$ and $\mathsf{subj}(e_2) = b$ where E_1 does not bind a and E_2 does not bind b, then it must be the case that $\mathsf{agree}^{ab}(e_1, e_2)$. In fact, the structural typing rules and those for new and for parallel composition guarantee that $\Delta; \Gamma_1, a\colon S \vdash E_1[e_1]$ and $\Delta; \Gamma_2, b\colon \overline{S} \vdash E_2[e_2]$, for some $\Delta, \Gamma_1, \Gamma_2$. When e_1 is $\mathsf{send}\, e_1'\, a$, sub-derivation introduction and inversion (lemmas 4.6 and 4.8) allow to conclude that $\Delta \vdash S \sim !B.S'$, hence $\Delta \vdash \overline{S} \sim ?B.\overline{S'}$. Of all the cases terms with subject b only $\mathsf{receive}\, b$ has a type of the form $?B.\overline{S'}$, hence $\mathsf{agree}^{ab}(\mathsf{send}\, e_1'\, a, \mathsf{receive}\, b)$. □

Corollary 4.11 (Progress for the functional sub-language)**.** *If* $\Delta; \Gamma \vdash (\nu\vec{a}, \vec{b})(E[e]|p)$ *then either* $e \to e'$ *or* e *is a value or* e *is of one of the following forms:* $\mathsf{send}\, v\, a$, $\mathsf{receive}\, a$, $\mathsf{select}\, l\, a$ *or* $\mathsf{case}\, a\, \mathsf{of}\, \{l_i \to e_i\}$.

4.5 Conservative extension

Our system is a conservative extension of previous session type systems. In those systems, the session type language is restricted to tail recursion, the μ operator works with a much simpler notion of contractivity, and equivalence is defined modulo unfolding. We take the definitions from the functional session type calculus [11] as a blueprint. The first-order part of the session type language from that paper may be defined by S' in the following grammar. Henceforth, we call that language *regular session types*.

$$\begin{aligned} S'_X &::= \mathsf{end} \mid !B.S''_X \mid ?B.S''_X \mid \oplus\{l_i\colon S''_{iX}\} \mid \&\{l_i\colon S''_{iX}\} \\ &\quad \mid \mu x.S'_{X\cup\{x\}} \\ S''_X &::= x \in X \mid S'_X \end{aligned}$$

The translation $()^\dagger$ into our system is defined as follows.

$$\begin{aligned} (\mathsf{end})^\dagger &= \mathsf{skip} \\ (!B.S'')^\dagger &= (!B; (S'')^\dagger) & (?B.S'')^\dagger &= (?B; (S'')^\dagger) \\ (\oplus\{l_i\colon S''_i\})^\dagger &= \oplus\{l_i\colon (S''_i)^\dagger\} & (\&\{l_i\colon S''_i\})^\dagger &= \&\{l_i\colon (S''_i)^\dagger\} \\ (\mu x.S')^\dagger &= \mu x.(S')^\dagger & (x)^\dagger &= x \end{aligned}$$

Lemma 4.12. *For all $S'_\emptyset$, $\cdot \vdash (S'_\emptyset)^\dagger :: \mathcal{S}^1$.*

Proof. We need to prove a more general property. Define $\Delta_X = x : \mathcal{S}^1 | x \in X$ and show that for all S'_X, $\Delta_X \vdash (S'_X)^\dagger :: \mathcal{S}^1$. The proof is by straightforward induction. □

Lemma 4.13. *Let $\vdash_{GV}$ be the typing judgment for expressions from Gay and Vasconcelos [11]. If $\Gamma \vdash_{GV} e : T$ then $\cdot; \Gamma \vdash e : (T)^\dagger$.*

5. Related work

The system we propose is ultimately rooted in the work of Honda et at. on session types [13, 14, 18]. The particular language of this paper is closely related to the one proposed by Gay and Vasconcelos [11]. The main difference is at the level of semantics: we use a synchronous semantics in place of a buffered one. We make this choice to simplify the technical treatment of the operational semantics. We believe that a buffered semantics can be derived without compromising the most important properties of the paper. At the level of the language, and in addition to Gay and Vasconcelos, we incorporate variant types and recursion on functional types. The linear treatment of session types is however identical, including the syntactic distinction of the two ends of a channel, related by a ν-binding.

The predicative polymorphism we employ is closely related to that of Bono et al. [4], including the kinding system for type variables. The extra complexity of context-free types lead us to a more elaborate kinding system, allowing to distinguish session (or end point) types, from functional types and type schemes (Bono et al. rely on different syntactic categories). Predicative polymorphism for the π-calculus was introduced by Vasconcelos [20]. A different form of polymorphism—bounded polymorphism on the values transmitted on channels—was introduced by Gay [9] in the realm of session types for the π-calculus.

Wadler [21] gives a typing preserving translation of the Gay and Vasconcelos calculus mentioned before to a process calculus inspired by the work of Caires and Pfenning [5]. The semantics of these systems, given directly by the cut elimination rules of linear logic, ensure 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 with respect to the work of Caires, Pfenning, and Wadler [5, 21].

The Sill language described by Toninho, Caires, and Pfenning [19] can describe a type-safe protocol to transmit trees. However, to do so requires a higher-order recursive session type of the following shape:

```
TreeC = ⊕{Leaf: end, Node: !int.!TreeC.!TreeC}
```

That is, to transmit a node `Node(i,t1,t2)` on channel c, the originating process first sends the integer `i` on c. But then it creates two new channels c_1 and c_2, sends their receiving ends on c, and closes c. Finally, it recursively transmits `t1` on c_1 and `t2` on c_2. In comparison, our calculus is intentionally closer to a low level language: it only supports the transmission of base type values. We furthermore believe that its run-time implementation is simpler and more efficient: only one channel is created and used for the transmission of the tree; thus, it avoids the overhead of multiple channel creation and channel passing.

6. Conclusion

Context-free session types extend the expressiveness of regular session types by generalizing the type structure from regular to context-free processes. This extension enables the implementation of type-safe serialization of recursive datatypes and XML documents among other examples.

The price to pay is a considerable complication of the metatheory. We established decidability of type equivalence, which is the bare foundation for type checking of context-free session types. There is more work to do towards a practical type-checking algorithm as the algorithm resulting from the decidability proof is very hard to implement. We plan to proceed in this direction by giving up on completeness for type checking, but going for a sound approximation instead.

References

[1] L. Aceto and M. Hennessy. Termination, deadlock, and divergence. *J. ACM*, 39(1):147–187, 1992.

[2] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *J. ACM*, 40(3):653–682, 1993.

[3] G. Bernardi and M. Hennessy. Using higher-order contracts to model session types. Unpublished, 2015.

[4] V. Bono, L. Padovani, and A. Tosatto. Polymorphic types for leak detection in a session-oriented functional language. In *FORTE*, volume 7892 of *LNCS*, pages 83–98. Springer, 2013.

[5] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[6] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.

[7] S. Christensen, H. Hüttel, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. *Inf. Comput.*, 121(2):143–148, 1995.

[8] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Comput. Sci.*, 25:95–169, 1983.

[9] S. J. Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.

[10] S. J. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informaticæ*, 42(2-3):191–225, 2005.

[11] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[12] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312. ACM, 2010.

[13] K. Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

[14] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[15] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

[16] J. H. Reppy. CML: A higher-order concurrent language. In *PLDI*, pages 293–305. ACM, 1991.

[17] D. Sangiorgi. *An Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2014. ISBN 9781139161381.

[18] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[19] B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2013.

[20] V. T. Vasconcelos. Predicative polymorphism in pi-calculus. In *PARLE*, volume 817 of *LNCS*, pages 425–437. Springer, 1994.

[21] P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.