FreeST: Context-free Session Types in a Functional Language

Bernardo Almeida Andreia Mordido
Vasco T. Vasconcelos
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

FreeST is an experimental concurrent programming language. Based on a core linear functional programming language, it features primitives to fork new threads, and for channel creation and communication. A powerful type system of context-free session types governs the interaction on channels. The compiler builds on a novel algorithm for deciding type equivalence of context-free session types. This abstract provides a gentle introduction to the language and discusses the validation process and runtime system.

1 Session types deserve to be free

Session types have been long subject to the shackles of tail recursion [11, 12]. Regular session-type languages bear the evident advantage of providing simple algorithms to check type equivalence and subtyping. Given two types, a fixed-point construction algorithm such as the one introduced by Gay and Hole builds, in polynomial time and space, a bisimulation relating the two types, or decides that no such relation exists [9]. The scenario darkens when one decides to let go of tail recursion, for now the fixed-point construction algorithm does not necessarily terminate. This is one of the main reasons why session types have been confined to (ω-) regular languages for so many years.

The discipline of conventional (that is, regular) session types provides guarantees not easily accessible to simpler languages such as Concurrent ML, where channels are unidirectional and transport values of a fixed type [21]. Session types, in turn, provide for the description of richer protocols, epitomised by the math client [9], which can be rendered in the SePi language [8] as follows:

```
MathClient = +{
  Plus: !Int.!Int.?Int.MathClient,
  Eq: !Int.!Int.?Bool.MathClient,
  Done: end
}
```

The type MathClient describes the client side of a protocol that introduces three choices: Plus, Eq, and Done. A client that chooses the Plus choice is supposed to send two integer values and to receive a further integer (possibly representing the sum of the two values sent), after which it goes back to the beginning. If the same client chooses instead the Eq option, it must subsequently send two integer values and expect a boolean result (possibly describing whether the two integers are equal), after which it must go back to the beginning. The Done option terminates the protocol, as described by type **end**.

The guarantees introduced by session type systems include the adherence of code to protocols and the related absence of runtime errors, including race conditions [12]. Some systems further guarantee progress (e.g. [4]). All this, under a rather expressive type language, that of (regular) session types.

There is one further characteristic of session types that attest for its flexibility: the ability to send channels on channels, often called delegation. This feature provides for the transmission of complex data on channels in a typeful manner. Suppose we want to transmit a tree of integer values on a channel. The tree may be described by data type Tree:

F. Martins, D. Orchard (Eds.): Programming Language Approaches
to Concurrency- & Communication-cEntric Software (PLACES 2019)
EPTCS 291, 2019, pp. 12–23, doi:10.4204/EPTCS.291.2

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

One has to choose between a) using multiple channels for sending the tree or b) using a single channel but incurring runtime checks to check adherence to the protocol. In the former scenario, trees are sent on channels of type

```
type TreeChannelC = +{Leaf:end, Node:!Int.!TreeChannelC.!TreeChannelC.end}
```

and we see that two new channels must be created and exchanged for each Node in a tree, so that $2n+1$ channels are needed to transmit an n-Node tree (equivalently, n channels are needed to transmit a tree with n Leaf and Node components). To see why, notice that type TreeChannelC is not recursive: a Leaf is sent on the given channel, and so is Node, but two the two subtrees are sent on newly created channels (witnessed by the parts !TreeChannelC in the type).

In the latter case, tree parts are sent on a single channel, but not necessarily in a "tree form". A suitable channel type in this case is

```
type TreeParts = +{Leaf: TreeParts, Node: !Int.TreeParts, EOS: end}
```

where EOS represents the end of transmission. In this case tree Node 1 (Node 2 Leaf Leaf) Leaf can be transmitted as Node 1 Node 2 Leaf Leaf Leaf EOS, when visited in a depth-first manner. It should be easy to see that type TreeParts allows transmitting many different tree parts that do not add up to a tree (such as, Node 1 Leaf Leaf Node 2 Leaf EOS), hence the necessary runtime checks to look over unexpected parts on the channel.

In 2016, Thiemann and Vasconcelos introduced the concept of context-free session types and proved that type equivalence remains decidable [22]. Context-free session types appear as a natural extension of conventional (regular) session-types. Syntactically, the changes are minor: rather than dot (.), the prefix operator, we use semi-colon (;), a new binary operator on types. We also take the chance to replace **end** by **Skip** to make it clear that it does not necessarily ends a session type, but else merely introduces a mark that can sometimes be omitted. In fact **Skip** is the identity element of sequential composition. The concurrent programming language we present, FreeST, is an implementation of the language proposed by Thiemann and Vasconcelos. FreeST types enjoy the monoidal axioms, **Skip**;S $\equiv$ S;**Skip** $\equiv$ S, associativity of sequential composition, S;(T;U) $\equiv$ (S;T);U, as well as the distributive property, +{l:S, m:T};U $\equiv$ +{l:S;U, m:T;U} (and similarly for external choice). Using the FreeST syntax, a channel that streams a tree can be written as follows:

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

The language FreeST, described in the remainder of the paper, provides for the best of both worlds: stream the tree on a *single* channel, *without* extraneous runtime checks.

There are a few experimental programming languages based on session types and there are many proposals for encoding session types in mainstream programming languages. We cannot cover them all in this short abstract; the interested reader is referred to a 2016 survey on behavioural types in programming languages [2]. Here, we briefly discuss a few prototypical programming languages using session types.

SePi is a programming language based on the pi-calculus whose channels are governed by regular session types refined with uninterpreted predicates [8]. The concrete syntax we choose for FreeST types is influenced by SePi.

SILL is a functional language with session typed concurrency, based on the Curry-Howard interpretation of intuitionistic linear logic [4], further extended with recursive types and processes [23, 24]. C1 is an imperative language [18] developed along the lines of SILL, featuring types that express sharing [3]. All these languages use regular session types. Compared to FreeST, they present stronger properties

(including progress), at the expense of imposing a particular form of programming (derived from the Curry-Howard interpretation) whereby each process uses zero or more channels and provides exactly one. Unlike these languages, at the time of this writing, FREEST does not allow for multiple threads to share a same channel.

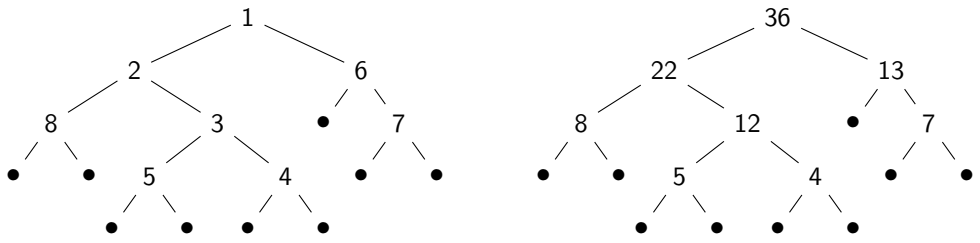
Links is a functional programming language for the web, later extended with session typing primitives [7, 14, 15]. Very much like SILL and C1, the base language is deadlock-free and terminating. Links also includes recursion and shared channels, the latter foregoing termination and deadlock-freedom. The kinding system of FREEST is quite similar to that of FST, proposed by Lindley and Morris [15].

The only other implementation of context-free session types we are aware of is that of Padovani [16], a language that admits equivalence, subtyping and, of particular interest, inference algorithms. It does however require a structural alignment between the process code and the session types, enforced by a *resumption* process operator that explicitly breaks a type $S;T$. The usage of resumptions requires additional annotations in programs—something we managed to avoid in FREEST—and, furthermore, it does not solve the type equivalence problem since the monoidal, associativity and distributivity properties typical of context-free session types are not accounted for. Padovani proposes the use of *explicit coercions* to overcome this limitation, but this requires greater efforts from programmers. Additional effort by programmers are always time consuming and error-prone, hence we have embedded a full-fledged type equivalence checker into the compiler.

2 FREEST is for programming

FREEST is a basic implementation of the language introduced by Thiemann and Vasconcelos [22]. We have chosen the concrete syntax to be aligned with that of Haskell, as much as possible. A FREEST program is a collection of type abbreviations, datatype and function (or value) declarations. Function main runs the program.

Our example serializes a tree object on a channel. The aim is to transform a tree by interacting with a remote server. The client process streams a tree on a (single) channel. In addition, for each node sent, an integer is received. The server process reads a tree from the other end of the channel and, for each node received, sends back the sum of the integer values under (and including) that node. As an example, our program transforms the tree on the left into the one on the right, where we use • to abbreviate Leaf.



We need a channel capable of transmitting a tree. The type below describes a variant of the TreeChannel introduced in the previous section that not only sends a tree, but, for each Node, also receives back an integer value.

```
type TreeC = +{Leaf: Skip, Node: !Int;TreeC;TreeC;?Int}
```

The + type constructor introduces an internal choice (the process in possession of the channel end chooses) with two alternatives, labelled Leaf and Node. These two labels should not be confused with the

constructors of datatype Tree. The Leaf branch states that no further interaction is possible on the channel (denoted by **Skip**); the Node branch writes an integer value, followed by two trees, and terminates by reading an integer value (!**Int**;TreeC;TreeC;?**Int**). The type declaration introduces an *abbreviation* to a recursive type **rec** x. +{Leaf: **Skip**, Node: !**Int**;x;x;?**Int**}. This recursive type is not valid in conventional session type systems given the non tail-recursive nature of the occurrences of x.

The writer process, transform, writes a tree on a given channel. It receives a tree of type Tree and a channel of type TreeC;α, for a type variable α. It returns a Tree and the residual of the input channel, of type α. The type of transform is *polymorphic*: different calls to the function use different values for α, as we show below.

```
transform : forall α ⇒ Tree → TreeC;α → (Tree, α)
```

For each Node in the input tree, function transform reads an integer from the channel and returns a tree isomorphic to the input where the integer values in nodes are read from the channel. The function performs a **case** analysis on the Tree constructor (either Leaf or Node). In the former case, it selects the Leaf choice and returns a pair composed of the original tree and the residual of the channel. In the latter case, the function selects the Node choice, then sends the integer value, followed by the two subtrees (via recursive calls). Finally, it reads an integer y from the channel, assembles a tree with y at the root and returns this tree together with the residual of the channel.

```
1  transform tree c =
2    case tree of
3      Leaf →
4        (Leaf, select Leaf c)
5      Node x l r →
6        let c   = select Node c in
7        let c   = send x c in
8        let l,c = transform[TreeC;?Int;α] l c in
9        let r,c = transform[?Int;α] r c in
10       let y,c = receive c in
11       (Node y l r, c)
```

Notice that the language requires a continuous rebinding of channel c (lines 6–10), for its type changes at each interaction, as in Gay and Vasconcelos [10]. At this point in time, FreeST is not able to infer the appropriate types that instantiate the polymorphic type variable α in the type of function transform. We thus help the compiler by supplying these types (TreeC;?**Int**;α and ?**Int**;α) in lines 8 and 9.

The reader process, treeSum, reads a tree from a given channel, writes back on the channel the sum of the elements in the tree, and finally returns the sum. This process sees the channel from the other end: rather than performing an internal choice (+), it performs an external choice (&), rather than writing (!), it reads (?), and rather than reading, it writes. We abbreviate the thus obtained type, **rec** x. &{Leaf: **Skip**, Node: ?**Int**;x;x;!**Int**}, as TreeS. We say that TreeS is *dual* to TreeC and conversely. The signature of the reader process is as follows.

```
treeSum : forall α ⇒ TreeS;α → (Int, α)
```

Rather than performing a case analysis on a Tree as in the writer process, function treeSum matches the channel against its two possible choices (Leaf and Node). In the former case the function returns 0 (the sum of the integer values in an empty tree) and the residual channel. In the latter, the function reads an integer from the channel, then reads two subtrees (via recursive calls) and sends on the channel the sum of the values in the subtree. It finally returns this exact sum, together with the residual channel.

```
12 treeSum c =
13   match c with
14     Leaf c →
15       (0, c)
16     Node c →
17       let x, c = receive c in
18       let l, c = treeSum[TreeS;!Int;α] c in
19       let r, c = treeSum[!Int;α] c in
20       let c    = send (x + l + r) c in
21       (x + l + r, c)
```

Again notice the continuous rebinding of channel c (lines 17–20) and the explicit types that instantiate variable α in the two recursive calls (TreeS;!Int;α and !Int;α) in lines 18 and 19.

Function main completes the program. It begins by creating a new channel (line 24). The **new** constructor takes a type S and returns a pair of channel ends of type (S, T), where T is a type dual to S. Then the main function forks a new thread to compute the treeSum (line 25). In the main thread, it transforms a given tree (aTree). Function treeSum uses the r end of the channel and transform uses w, the other end. In these calls, both functions are applied to type **Skip** (lines 25–26). Analysing the signatures of the two functions (they both return a channel of type α), we see that the channel ends r and w are both consumed to **Skip**. Type **Skip** is unrestricted in nature (of kind unrestricted) and hence its values can be safely discarded (cf. the two wilcards in the lets on lines 25–26). In addition to the residual of channel end w, function transform also returns a new tree t, which becomes the result of the main function.

```
22 main: Tree
23 main =
24   let w,r = new TreeC in
25   let _   = fork (treeSum[Skip] r) in
26   let t,_ = transform[Skip] aTree w in
27   t
```

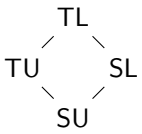
When run, function main prints on the console the textual representation of the tree on the right of the above diagram if aTree denotes the tree on the left.

3 Only valid programs may run

This section introduces the main syntactic categories of the language: kinds, types, expressions, and programs. It also discusses type equivalence.

Kinding validates types FREEST requires kinding; GV does. And the reason is on polymorphism, not on context-free types. !**Int** is undoubtedly a session type, and so is !**Int**;?**Bool**. On the other hand, **Int**→**Bool** and (**Int**,**Bool**) are certainly functional types. The GV language allows a stratified grammar with separate syntactic categories for functional types and for session types, even if mutually recursive. In FREEST this is not possible. To see why, consider a type variable α. Is !**Int**;α a session type? What about α itself? The former is a session type only when α represents a session type; the latter is a functional type when α denotes a functional type, and is a session type otherwise. If α does not denote a session type, then !**Int**;α is not a type, it is simply a piece of useless syntax.

To accommodate polymorphism, types are classified into *kinds*. Kinds are pairs composed of a *prekind* and a *multiplicity*. Prekinds distinguish functional types, T, from session types, S. Multiplicities



control the number of times a value may be used in a given context: exactly one—linear, L—or zero or more—unrestricted, U. Both prekinds and multiplicities come equipped with an ordering relation. Together they form the lattice in the diagram on the right. This is essentially the Lindley and Morris kinding system when restricted to prekinds and multiplicities [15] or the Thiemann and Vasconcelos system where kinds for type schemes are elided [22]. Types of kind TU can be used at kind TL. Similarly types of kind SL can be used at kind TL. Finally, any type can be seen of kind TL, the kind sitting at top of the hierarchy, the kind that carries the least information.

Equipped with a kinding system, FREEST features as functional types the following:

- Basic types, B : TU, that is, **Int**, **Bool**, **Char**, and (),
- Unrestricted and linear functions, T1 → T2 : TU and T1 ⊸ T2 : TL,
- Pairs (T1, T2) of kind depending on the kinds of T1 and T2, and
- Datatypes, [l1 : T1, ..., ln : Tn] of kind TU or TL depending on the kinds of the various types Ti.

The session types are the following:

- The terminated type, **Skip** : SU,
- Sequential composition, S1;S2 of kind SU or SL depending on the kinds of S1 and S2,
- Messages, !B, ?B, both of kind SL,
- Choices, +{l1: S1, ..., ln :Sn}, &{l1: S1, ..., ln :Sn}, both of kind SL,
- Recursive types, **rec** x.S, bearing the kind of S.

One function, many forms We are now in a better position to understand the type signature for the functions in Section 2. Polymorphic variables are introduced solely with the **forall** construct. The non-abbreviated type for function treeSum spells out the kind for the polymorphic type variable α, as follows:

```
forall α:SL ⇒ Tree → TreeC;α → (Tree,α)
```

The absence of an explicit kind for a polymorphic variable is understood as SL. In this particular case, one can replace α by any type with kind SL or smaller, including types TreeC;?**Int**;α : SL (line 8 in the code for TreeSum in Section 2), ?**Int**;α : SL (line 9), and **Skip** : SU (line 25 in the code for main). The call to function transform on line 8 effectively calls the function at type

```
Tree → TreeC;TreeC;?Int;α → (Tree, TreeC;?Int;α).
```

making it clear that channel c is supposed to convey two trees (the left and the right subtrees of a non-empty tree) before it produces an integer value and continue as α. The function reads one tree from the channel and returns the unused part of the channel, namely TreeC;?**Int**;α.

Because we declared α of kind SL, functional types cannot replace α. If one tries to replace α by **Int**→**Bool**, one would get Tree → TreeC;(**Int**→**Bool**)→ (Tree,**Int**→**Bool**), which is not a well formed type for the semicolon operator is defined on session types only. At the time of this writing datatypes are monomorphic.

Repeated behavior A characteristic central to session types is their ability to describe unbounded behavior, usually captured by recursive types. Recursion is certainly useful in types such as TreeC in Section 2, meant do describe channels able to transmit binary trees of different sizes. We have seen that name TreeC abbreviates type **rec** x. +{Leaf: **Skip**, Node: !**Int**;x;x;?**Int**}. Type variable x, monomorphic, belongs to kind SU so that type +{Leaf: **Skip**, Node: !**Int**;x;x;?**Int**} may be deemed well formed (and of kind SL). Recursive types must be *contractive* or free from unguarded recursion. The following types are not contractive: **rec** x1. ... **rec** xn.x1 and **rec** x1. ... **rec** xn.(x1;S), for n>1 and for any session type S. Only session types can be recursive. Recursion in the functional part of the type language is obtained via datatypes as usual, and the fixed-point operator is built into function definition.

Expressions FREEST blends expressions typical of functional languages and of session types. In a way, it is not much different from GV. The expressions inspired from functional languages include:

- Basic values: characters, integer and boolean values, and the unit value, (),
- Term variables (as opposed to type variables),
- Lambda introduction, \x $\multimap$ E for linear abstractions and \x $\rightarrow$ E for unrestricted abstractions, and elimination, E1 E2,
- Pair introduction, (E1,E2), and elimination, **let** x, y = E1 **in** E2 (linear versions only, at the time of this writing),
- Datatype elimination, **case** E **of** C1 x11 ... x1k $\rightarrow$ E1, ..., Cn xn1 ... xnk $\rightarrow$ En, and
- Conditional, **if** E1 **then** E2 **else** E3, type application, x[T1 ,... Tn], and thread creation, **fork** E.

The session-type related expressions are:

- Channel creation, **new** S,
- Message sending and receiving, **send** E and **receive** E, and
- Branch selection, **select** C E, and match, **match** E **with** C1 x $\rightarrow$E1 ,..., Cn x $\rightarrow$ En.

Programs Programs are composed of

- Function signatures and declarations,
- Datatype declarations, and
- Type abbreviations.

Both functions and datatypes can be mutually recursive. Haskell code is generated for programs that contain a function named main with a non-function type. The result of the evaluation of this function is printed on the console.

Checking type equivalence is the main challenge of the compiler. If, on the one hand, the algorithm should be sound and complete, on the other hand it should have a running time compatible with a compiler.

We have developed an algorithm to decide the equivalence of context-free session types [1]. It features three distinct stages. The *first stage* builds a context-free grammar in Greibach Normal Formal from a context-free session type in a way that bisimulation is preserved. The *second stage* prunes the grammar by removing unreachable symbols in unnormed sequences of non-terminal symbols. This stage

builds on the result of Christensen, Hüttel, and Stirling [6]. The *third stage* constructs an expansion tree, by alternating between expansion and simplification steps. During an expansion step, all children nodes are expanded according to the transitions in the grammar. Throughout the simplification phase, the reflexive, congruence and Basic Process Algebra (BPA) rules proposed by Jancăr and Moller in [13] are applied to each node, yielding a number of sibling nodes. The simplification phase promotes branching on the expansion tree. The tree is traversed using breadth-first search and the algorithm terminates as soon as it reaches an empty node—case in which it decides positively—or it fails to expand a node—case in which it decides negatively. The finite representation of bisimulations of BPA transition graphs [5, 6] is paramount for the soundness and completeness of the algorithm, a result that we show to hold.

Although the branching nature of the expansion tree confers an exponential complexity to the algorithm, we propose heuristics that allow constructing the relation in a reasonable time. For this purpose, we use a double-ended queue that allows prioritizing nodes with potential to reach an empty node faster. We have also speeded up the computation of the expansion tree by iterating the simplification phase until a fixed point is reached. These optimizations led to an improvement of more than $11,000,000\%$ on the runtime of the algorithm, so that it can now be effectively incorporated in the compiler for FreeST.

4 Let FreeST run

FreeST is written in Haskell and generates Haskell code that can be compiled with a conventional GHC compiler. The validation phase features a kinding system and a typing system briefly described in the previous section. Only declarative versions of these systems are described in Thiemann and Vasconcelos [22]; we have developed the corresponding algorithmic versions.

This section concentrates on the *runtime system*, a surprisingly compact system. We build on modules Control.Concurrent and Unsafe.Coerce, and make particular use of the monadic combinators below [26]. The **do** notation is built on top of these combinators.

```
(>>) :: Monad m ⇒ m a → m b → m b
(>>=) :: Monad m ⇒ m a → (a → m b) → m b
return :: Monad m ⇒ a → m a
```

To fork a new thread, we use the forkIO primitive. The primitive returns a threadId, which we discard and convert into the unit value.

```
fork :: IO () → IO ()
fork e = forkIO e >> return ()
```

Each channel is implemented with a pair of MVars first introduced in Concurrent ML [21] and made available in Haskell via module Concurrent.Control [17]. Each channel end is itself a pair of crossed MVars.

```
type ChannelEnd a b = (MVar a, MVar b)
```

The **new** function creates such a pair and returns the two channel ends in a pair.

```
new :: IO (ChannelEnd a b, ChannelEnd b a)
new = do
  m1 ← newEmptyMVar
  m2 ← newEmptyMVar
  return ((m1, m2), (m2, m1))
```

To write on a channel end, we use the second MVar in the given pair. Because the type of values transmitted on a channel vary over time, we use unsafeCoerce to bypass the Haskell type system, a technique common in standard implementations of session types, including that of Pucella and Tov [20]. To write on the channel end we use the putMVar primitive. If the MVar is currently full, putMVar waits until it becomes empty. The result of **send** is the original channel end, which must be rebound in programs, as discussed in the previous section.

```
send :: b -> ChannelEnd a b -> IO (ChannelEnd a b)
send x (m1, m2) = putMVar m2 (unsafeCoerce x) >> return (m1, m2)
```

Finally, to read from a channel end, we use the first element of the pair. The takeMVar primitive returns the contents of the MVar. If the MVar is empty, takeMVar waits until it is full. After a takeMVar, the MVar is left empty. The result of **receive** is a pair composed of the value read from the MVar and the original channel (which, again, should be rebound in the program).

```
receive :: ChannelEnd a b → IO (a, ChannelEnd a b)
receive (m1, m2) = takeMVar m1 >>= \x → return (unsafeCoerce x, (m1, m2))
```

Our implementation is *not* completely aligned with the operational semantics of *synchronous* session types (and that proposed by Thiemman and Vasconcelos in particular). We use MVars to implement *asynchronous* communication with buffers of size one. The discrepancy can be witnessed with the standard cross write-read on two channels. In the following program w1-r1 and w2-r2 are two channels.

```
writer :: !Char → !Bool → Skip
writer w1 w2 =
  let _ = send 'c' w1 in
  send False w2
reader :: ?Char → ?Bool → Bool
reader r1 r2 =
  let x, r2 = receive r2 in
  let _, r1 = receive r1 in
  x
```

The program does not deadlock in our current implementation but does so in a synchronous semantics. However, deadlock occurs in a simple adaptation of the program with two consecutive writes two consecutive reads on each channel (using a writer of type !**Char**;!**Char** → !**Bool**;!**Bool** → **Skip** and dually for reader).

FREEST is a standalone language whose types need not be in the runtime system of Haskell, since type checking is performed by the FREEST front-end. Coercions (that is, calls to the unsafeCoerce function) are then inserted so as to avoid typing errors when compiling the target Haskell code. The occurrences of unsafeCoerce are limited to *two* places: when reading and when writing from MVars, in runtime functions **receive** and **send**, respectively.

The original proposal of context-free session types feature a call-by-value semantics. In order to align FREEST with this semantics, we use Haskell's *bang patterns*. For example, the transform function is compiled into transform !tree !c = ..., forcing the evaluation of both parameters, prior to the execution of the function body, thus effectively implementing a call-by-value strategy.

5 The bright future of FREEST

We have developed a basic compiler for FREEST, a concurrent functional programming with context-free session types, based on the ideas of Thiemann and Vasconcelos [22]. There are many possible

extensions to the language. We discuss a few. Support for linear pairs and linear datatypes, as well for polymorphic datatypes, should not be difficult to incorporate. Because FREEST compiles to Haskell, a better interoperability is called for. We plan to add primitive support for lists, and for some functions in Haskell's prelude, namely rank-1 functions that we will have to annotate with FREEST types. We have chosen a buffered semantics with buffers of size one, for ease of implementation, but we plan to experiment with buffered channels of arbitrary size by simply replacing the runtime system. The original proposal of context-free sessions is based on a call-by-value operational semantics and we kept that strategy in FREEST. We however plan to experiment with call-by-need, taking advantage of the fact that FREEST compiles to Haskell. Shared channels allow for multiple readers and multiple writers, thus introducing (benign) races. There are several proposals in the literature [3, 8, 15, 25] on which we may base this extension. The **dualof** type operator is present in the SePi language [8]; we plan its incorporation in FREEST language.

We also plan to extend the expressivity of FREEST by allowing messages to convey arbitrary types, as opposed to basic types only. In this wider scope, the type equivalence algorithm for context-free session types must be intertwined with the type equivalence algorithm for functional types, for now the labels of the labeled-transition system are types themselves.

Last but not least, we plan to incorporate type inference on type applications in order to allow the automatic identification of the unifier matching a polymorphic type against a given type. This unification process should recognize the unifiers of two types up to type equivalence. However, dealing with type inference on type applications with recursive types might be challenging, as observed by Hosoya and Pierce [19].

Acknowledgements The authors would like to thank the anonymous reviewers for their extremely relevant comments and pointers. This work was supported by FCT through project Confident (PTDC/EEI-CTP/ 4503/2014), by the LASIGE Research Unit (UID/CEC/00408/2019) and by Cost Action CA15123 EUTypes, supported by COST (European Cooperation in Science and Technology).

References

[1] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2019): *Checking the equivalence of context-free session types*. Submitted.

[2] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos & Nobuko Yoshida (2016): *Behavioral Types in Programming Languages*. *Foundations and Trends in Programming Languages* 3(2-3), pp. 95–230, doi:10.1561/2500000031.

[3] Stephanie Balzer & Frank Pfenning (2017): *Manifest sharing with session types*. *PACMPL* 1(ICFP), pp. 37:1–37:29, doi:10.1145/3110281.

[4] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In: *CONCUR*, *LNCS* 6269, Springer, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[5] Didier Caucal (1986): *Décidabiité de l'égalité des Languages Algébriques Infinitaires Simples*. In: *STACS 86, 3rd Annual Symposium on Theoretical Aspects of Computer Science, Orsay, France, January 16-18, 1986, Proceedings*, Springer, pp. 37–48, doi:10.1007/3-540-16078-7_63.

[6] Søren Christensen, Hans Hüttel & Colin Stirling (1995): *Bisimulation Equivalence is Decidable for All Context-Free Processes*. *Inf. Comput.* 121(2), pp. 143–148, doi:10.1006/inco.1995.1129.

[7] Ezra Cooper, Sam Lindley, Philip Wadler & Jeremy Yallop (2006): *Links: Web Programming Without Tiers*. In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, Springer, pp. 266–296, doi:10.1007/978-3-540-74792-5_12.

[8] Juliana Franco & Vasco Thudichum Vasconcelos (2013): *A Concurrent Programming Language with Refined Session Types*. In: *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*, Springer, pp. 15–28, doi:10.1007/978-3-319-05032-4_2.

[9] Simon J. Gay & Malcolm Hole (2005): *Subtyping for session types in the pi calculus*. *Acta Informaticæ* 42(2-3), pp. 191–225. Available at `http://dx.doi.org/10.1007/s00236-005-0177-z`.

[10] Simon J. Gay & Vasco Thudichum Vasconcelos (2010): *Linear type theory for asynchronous session types*. *J. Funct. Program.* 20(1), pp. 19–50, doi:10.1017/S0956796809990268.

[11] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *CONCUR*, *LNCS* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[12] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP*, *LNCS* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[13] Petr Jancar & Faron Moller (1999): *Techniques for Decidability and Undecidability of Bisimilarity*. In: *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, Springer, pp. 30–45, doi:10.1007/3-540-48320-9_5.

[14] Sam Lindley & J. Garrett Morris (2015): *A Semantics for Propositions as Sessions*. In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, Springer, pp. 560–584, doi:10.1007/978-3-662-46669-8_23.

[15] Sam Lindley & J. Garrett Morris (2017): *Behavioural Types: from Theory to Tools*, chapter Lightweight functional session types. River Publishers Series in Automation, Control and Robotics.

[16] Luca Padovani (2017): *Context-Free Session Type Inference*. In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, Springer, pp. 804–830, doi:10.1007/978-3-662-54434-1_30.

[17] Simon L. Peyton Jones, Andrew D. Gordon & Sigbjorn Finne (1996): *Concurrent Haskell*. In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, ACM Press, pp. 295–308, doi:10.1145/237721.237794.

[18] Frank Pfenning (2018): *Message-Passing Concurrency and Substructural Logics*. Tutorial at POPL.

[19] Benjamin C. Pierce & David N. Turner (2000): *Local type inference*. *ACM Trans. Program. Lang. Syst.* 22(1), pp. 1–44, doi:10.1145/345099.345100.

[20] Riccardo Pucella & Jesse A. Tov (2008): *Haskell session types with (almost) no class*. In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, ACM, pp. 25–36, doi:10.1145/1411286.1411290.

[21] John H. Reppy (1993): *Concurrent ML: Design, Application and Semantics*. In: *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, Springer, pp. 165–198, doi:10.1007/3-540-56883-2_10.

[22] Peter Thiemann & Vasco T. Vasconcelos (2016): *Context-free session types*. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, ACM, pp. 462–475, doi:10.1145/2951913.2951926.

[23] Bernardo Toninho (2015): *A Logical Foundation for Session-based Concurrent Computation*. Ph.D. thesis, Carnegie Mellon University and Universidade Nova de Lisboa.

[24] Bernardo Toninho, Luís Caires & Frank Pfenning (2013): *Higher-Order Processes, Functions, and Sessions: A Monadic Integration*. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, Springer, pp. 350–369, doi:10.1007/978-3-642-37036-6_20.

[25] Vasco T. Vasconcelos (2012): *Fundamentals of session types*. *Inf. Comput.* 217, pp. 52–70, doi:10.1016/j.ic.2012.05.002.

[26] Philip Wadler (1995): *Monads for Functional Programming*. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, Springer, pp. 24–52, doi:10.1007/3-540-59451-5_2.