Session Type Inference in Haskell

Keigo Imai Shoji Yuen Kiyoshi Agusa

IT Planning Inc., Japan Graduate School of Information Science, Nagoya University, Japan

imai@nagoya-u.jp yuen@is.nagoya-u.ac.jp agusa@is.nagoya-u.ac.jp

We present an inference system for a version of the π-calculus in Haskell for the session type proposed by Honda et al. The session type is very useful in checking if the communications are well-behaved. The full session type implementation in Haskell was first presented by Pucella and Tov, which is 'semi-automatic' in that the manual operations for the type representation was necessary. We give an automatic type inference for the session type by using a more abstract representation for the session type based on the 'de Bruijn levels'. We show an example of the session type inference for a simple SMTP client.

1 Introduction

The *Session-type system* [6] provides a way to statically check communication protocols. Incorporating session types in the existing programming languages eases the communication centric programming in that session typed components are guaranteed to behave correctly by their types. However, it is not apparent how to integrate the session typing discipline with the existing programming languages.

Several session-type implementations [13, 16, 17] have been proposed for Haskell. The *type-level programming* is shown to implement session types. It is natural to use the functional dependencies [8] for encoding the duality of session types. The indexed monad [1, 9] is used to propagate the session-type information through process constructs.

Currently, all existing implementations [13, 16, 17] of the session type implementation require some manual annotation in program code to infer types. The session types in [13] and [17] often make even a simple program to be unnecessarily verbose. The typechecking in [16] requires incomprehensible annotation when the number of channel increases. A fully-automatic type inference is essential as seen in other typed frameworks such as parser combinators [12] and database access [3].

Our goal is to provide a fully-automatic session-type inference in Haskell. We extend the work by Pucella and Tov [16] to infer types without manual operations. We show an implementation technique for the original session-type system [6] as the target language.

The issue of type-level representation The common idea in [16] and [17] is to track session types for multiple channels using the extra *symbol table* embedded in the Haskell's type. The inferred Haskell type for a process would be $\mathtt{P}\ \{c_1 \mapsto s_1; c_2 \mapsto s_2, \cdots\}$ where `P` is a process type constructor and $\{\cdots\}$ is the symbol table to assign each channel c_i to its session type s_i. This symbol table is represented in the *type-level*, hence the channels c_i is not a value, but a *type* which reflects an identity of a channel.

In the implementation in [16], *type variables* represents channels in a symbol table. To distinguish them from each other, such type variables are locally quantified at the position the channels are introduced. Such a type variable is matched against the symbol table every time a type inference rule is applied. Since the symbol table itself can only be represented as a type-level list of key-value pairs, matching on channels is unavoidable. But in the Haskell type-level programming, such matching operation is not provided.

To alleviate this difficulty, [16] devises the *stack manipulation* `dig` and `swap` on a symbol table for reordering. The stack restricts the communication primitives only to the first entry of the symbol table. `swap` swaps the first entry of the symbol table with the second one. `dig` p allows p to access on the second entry of the symbol table.

In [16], it is stated that the automatic application of these operations is not possible without adding extra information in the symbol table and the answer for this problem is not shown.

Main idea To resolve the type matching problem, we use the natural number based on *de Bruijn level* as the type-level representation for channels. The symbol table is represented as just the type-level list of session types, and accessed by the numbers. As the number-based access on the type-level list is possible in the existing technique [10], type inference is fully automatic.

Our main contribution is to show an automatic inference of the session-type inference in Haskell. [1] Although in [16] only the capability passing is possible, our calculus is possible to pass a channel. To show that our improvement is purely in the sense of matching, It is shown that by extending typing discipline in [16] it can have the same expressiveness as ours.

Related work Neubauer and Thiemann [13] implemented session types on a single communication channel in Haskell. Their implementation avoids aliasing by prohibiting explicit use of a channel.

Pucella and Tov [16] have shown a general technique to encode session types in languages like Haskell, ML, and C#. Their implementation based on manual stack manipulations `swap` and `dig` liberates from type-level programming which is only available in Haskell, hence their technique can be applied to other languages which have parametric or generic types. On the other hand, their implementation cannot enjoy fully-automatic type inference.

The implementation proposed by Sackman and Eisenbach [17] supports full functionality of session types. However, their library requires a manual construction of session types. There are trade-offs between such a manual handling and annotated type-inference approach in that while type-inference reduces unneeded annotations, explicit annotation with a rich set of syntax increases readability and expressiveness of types. We will discuss this aspect in the later section.

The difference of our implementation from the previous work is summarized in the following table.

	channel passing	annotation	portable
Neubauer et al. [13]	no	auto	no
Pucella et al. [16]	yes in a limited context[2]	stack based channel handling	yes
Sackman et al. [17]	yes	manual construction of session types	no
Our implementation	*yes*	*auto*	*no*

Paper Organization The rest of this paper is organized as follows. In Section 2, session types and the π-calculus is introduced from [6]. In Section 3, we show the session-type inference in Haskell, and compare it with other implementations. We describe an example of a SMTP client using our implementation

[1] A working implementation, `full-sessions` , which can be compiled by the Glasgow Haskell Compiler 6.10.2 or higher is available at: `http://hackage.haskell.org/package/full-sessions/`. Typing `cabal install full-sessions` in a shell will install `full-sessions` in your environment.

[2] See section 5

in Section 4. In Section 5, we show that our implementation is more expressive than [16] in the aspect of channel-passing and show the way to extend [16] to have the same expressive power as ours. In Section 7, we discuss a few aspects of usability of session-type implementation. Finally, Section 8 concludes the paper.

2 Session types and the π-calculus

2.1 The π-calculus

The syntax of our π-calculus processes is defined by the following grammar:

$$\begin{array}{rl} P ::= & \texttt{send}_\pi\ c\ e\ P \mid \texttt{recv}_\pi\ c\ (\lambda x.P) \mid \texttt{sel1}_\pi\ c\ P \mid \texttt{sel2}_\pi\ c\ P \mid \texttt{offer}_\pi\ c\ P_1\ P_2 \\ \mid & \texttt{sendS}_\pi\ c\ c'\ P \mid \texttt{recvS}_\pi\ c\ (\lambda c'.P) \mid P \mid\mid\mid Q \mid \texttt{inact}_\pi \mid \texttt{new}_\pi\ (\lambda c.P) \end{array}$$

We use λ-abstraction to represent bindings using higher-order abstract syntax [14]. x ranges over variables of basic values and channels, c and c' range over channels, and P and Q range over processes. We put the subscript π on each process constructor since they are overloaded in the later sections.

An input process $\texttt{recv}_\pi\ c\ (\lambda x.P)$ inputs a value via channel c, then binds it to x in P. An output process $\texttt{send}_\pi\ c\ e\ P$ first evaluates e, then emits the value via channel c, and becomes P. $\texttt{sel1}_\pi\ c\ P$ and $\texttt{sel2}_\pi\ c\ P$ denote the selection of branch label 1 or 2 on c. It first sends the selected label, and becomes P. $\texttt{offer}_\pi\ c\ P_1\ P_2$ receives a label. Then it becomes P_1 or P_2, depending on the received label. $\texttt{sendS}_\pi\ c\ c'\ P$ sends channel c' on c and becomes P. $\texttt{recvS}_\pi\ c\ (\lambda c'.P)$ receives a channel on c, binds it to c' in P, and becomes P. These operations enable higher-order session communications. $P \mid\mid\mid Q$ runs P and Q concurrently. $\texttt{inact}_\pi$ is the constant to denote the terminated (inactive) process. $\texttt{new}_\pi\ (\lambda c.P)$ generates a fresh channel c bound in P.

The operational semantics of the π-calculus is in Figure 1. Here, $e \downarrow v$ represents that e is evaluated to a value v. The structural congruence of processes is in Figure 2. The function $\mathrm{fn}(P)$ denotes free names in P. $\equiv_\alpha$ denotes α-equivalence.

$$\textsc{Com}: \frac{e \downarrow v}{\texttt{send}_\pi\ c\ e\ P \mid\mid\mid \texttt{recv}_\pi\ c\ (\lambda x.Q) \longrightarrow P \mid\mid\mid Q\{v/x\}}$$

$$\textsc{Label}_1: \frac{}{\texttt{sel1}_\pi\ c\ P \mid\mid\mid \texttt{offer}_\pi\ c\ Q_1\ Q_2 \longrightarrow P \mid\mid\mid Q_1}$$

$$\textsc{Label}_2: \frac{}{\texttt{sel2}_\pi\ c\ P \mid\mid\mid \texttt{offer}_\pi\ c\ Q_1\ Q_2 \longrightarrow P \mid\mid\mid Q_2}$$

$$\textsc{Pass}: \frac{}{\texttt{sendS}_\pi\ c\ c'\ P \mid\mid\mid \texttt{recvS}_\pi\ c\ (\lambda c'.Q) \longrightarrow P \mid\mid\mid Q}$$

$$\textsc{Scop}: \frac{P \longrightarrow P'}{\texttt{new}_\pi\ (\lambda c.P) \longrightarrow \texttt{new}_\pi\ (\lambda c.P')} \qquad \textsc{Par}: \frac{P \longrightarrow P'}{P \mid\mid\mid Q \longrightarrow P' \mid\mid\mid Q}$$

$$\textsc{Str}: \frac{P \equiv P' \wedge P' \longrightarrow Q' \wedge Q' \equiv Q}{P \longrightarrow Q}$$

Figure 1: The operational semantics of the π-calculus

$$P \equiv Q \text{ if } P \equiv_\alpha Q \qquad P \mid\mid\mid \texttt{inact} \equiv P \qquad P \mid\mid\mid Q \equiv Q \mid\mid\mid P \quad \texttt{new}_\pi\ (\lambda c.\texttt{inact}) \equiv \texttt{inact}$$
$$(P \mid\mid\mid Q) \mid\mid\mid R \equiv P \mid\mid\mid (Q \mid\mid\mid R) \qquad \texttt{new}_\pi\ (\lambda c.P) \mid\mid\mid Q \equiv \texttt{new}_\pi\ (\lambda c.P \mid\mid\mid Q) \ \text{ if } c \notin \text{fn}(Q)$$

Figure 2: Structural congruence of the π-calculus processes

2.2 Session types

In this subsection and following subsection, we review a session type system in [6].

A session type represents a protocol which is associated with an endpoint of a channel. v ranges over types for basic values, and u ranges over session types. The session types in this paper are defined by the following grammar:

$$\begin{array}{rcl} u & ::= & \texttt{Send}\ v\ u \mid \texttt{Recv}\ v\ u \mid \texttt{Select}\ u_1\ u_2 \mid \texttt{Offer}\ u_1\ u_2 \\ & \mid & \texttt{Throw}\ u_1\ u_2 \mid \texttt{Catch}\ u_1\ u_2 \mid \texttt{End} \mid \texttt{Bot} \end{array}$$

`Send` v u denotes a protocol to emit a value of type v followed by a behavior of type u. `Recv` v u denotes a protocol of receiving a value of type v followed by a behavior of type u. `Select` u_1 u_2 denotes to be either behavior of type u_1 or type u_2 after emitting a corresponding label 1 or 2. `Offer` u_1 u_2 denotes a behavior like either u_1 or u_2 according to the incoming label. `Throw` u_1 u_2 denotes a behavior to output of a channel with session type u_1 followed by a behavior of type u_2. `Catch` u_1 u_2 is the input of a channel with session type u_1 followed by a behavior of type u_2. `End` denotes a terminated session. `Bot` is the type for a channel whose endpoints are already engaged by two processes, so that no further processes can own that channel. For example, in $(\texttt{send}_\pi\ c\ e\ \texttt{inact} \mid\mid\mid \texttt{recv}_\pi\ c\ (\lambda x.\texttt{inact}))$, c has the session type `Bot`.

A session type u has the dual $\overline{u}$. The definition of dual is illustrated in Figure 3.2.2. A dual of a session on one end of a channel is the session on the other end of the same channel.

$$\begin{array}{rcl rcl rcl} \overline{\texttt{Send}\ v\ u} & = & \texttt{Recv}\ v\ \overline{u} & \overline{\texttt{Select}\ u_1\ u_2} & = & \texttt{Offer}\ \overline{u_1}\ \overline{u_2} & \overline{\texttt{Throw}\ u_1\ u_2} & = & \texttt{Catch}\ u_1 \overline{u_2} \\ \overline{\texttt{Recv}\ v\ u} & = & \texttt{Send}\ v\ \overline{u} & \overline{\texttt{Offer}\ u_1\ u_2} & = & \texttt{Select}\ \overline{u_1}\ \overline{u_2} & \overline{\texttt{Catch}\ u_1\ u_2} & = & \texttt{Throw}\ u_1\ \overline{u_2} \\ \overline{\texttt{End}} & = & \texttt{End} & \overline{\texttt{Bot}} & = & \textit{undefined} & & & \end{array}$$

Figure 3: Duality for session types

2.3 The typing rules

In the session-type system [6], there are two kinds of type judgments, value judgment $\Gamma \vdash e \triangleright v$ and process judgment $\Gamma \vdash P \triangleright \Delta$. A process P is *typeable* if there exists some Γ, Δ such that $\Gamma \vdash P \triangleright \Delta$. Γ denotes *sorting* that maps variables to types of basic values. Δ denotes *session type environment* or *session typing* that maps names to session types. A *completed* type environment is the one that assigns the type `End` to every name appearing in a process.

A process is *typeable* under Γ, iff $\Gamma \vdash P \triangleright \Delta$ for a given Δ. A typeable process never fails due to communication mismatch.

The typing rules are defined in Figure 4. The composition of type environments $\Delta \oplus \Delta'$ is defined by the component-wise extension of the type algebra which is defined as follows:

$$\begin{array}{ll} \texttt{End} \oplus u = u & u \oplus \texttt{End} = u \\ u \oplus u' = \texttt{Bot} \quad \text{if } \overline{u} = u' & \text{otherwise } \textit{undefined} \end{array}$$

The literature [6] defines an erroneous process using following notions: A *c-process* is a process prefixed by subject c, such as $\texttt{send}_\pi\ c\ e\ P$ and $\texttt{recvS}_\pi\ c\ (\lambda c'.P)$. A *c-redex* is the parallel composition of two c-processes in either of form $\texttt{send}_\pi\ c\ e\ P \mid\mid\mid \texttt{recv}_\pi\ c\ (\lambda x.Q)$, $\texttt{sel}_i\ c\ P \mid\mid\mid \texttt{offer}_\pi\ c\ Q_1\ Q_2$ for $i \in \{1,2\}$, or $\texttt{sendS}_\pi\ c\ c'\ P \mid\mid\mid \texttt{recvS}_\pi\ c\ (\lambda c'.Q)$.

Definition 1 (Error) *We shall say that P is an error if $P \equiv \texttt{new}_\pi\ (\lambda \tilde{c}.Q \mid R)$ where Q is, for some c, the parallel composition of either two c-processes that do not form a c-redex, or three or more c-processes.*

Then we quote the following theorem, which is also valid for our framework, from [6] [3]:

Theorem 1 (Type Safety) *A typeable program never reduces to an error.*

$$\frac{\Gamma \vdash e \rhd v \qquad \Gamma \vdash P \rhd \Delta \cdot c : u}{\Gamma \vdash \texttt{send}_\pi\ c\ e\ P \rhd \Delta \cdot c : \texttt{Send}\ v\ u}\ [\textsc{Send}] \qquad \frac{\Gamma, x : v \vdash P \rhd \Delta \cdot c : u}{\Gamma \vdash \texttt{recv}_\pi\ c\ (\lambda x.P) \rhd \Delta \cdot c : \texttt{Recv}\ v\ u}\ [\textsc{Rcv}]$$

$$\frac{\Gamma \vdash P \rhd \Delta \cdot c : u_1}{\Gamma \vdash \texttt{sel1}_\pi\ c\ P \rhd \Delta \cdot c : \texttt{Select}\ u_1\ u_2}\ [\textsc{Sel}] \qquad \frac{\Gamma \vdash P \rhd \Delta \cdot c : u_1 \qquad \Gamma \vdash Q \rhd \Delta \cdot c : u_2}{\Gamma \vdash \texttt{offer}_\pi\ c\ P\ Q \rhd \Delta \cdot c : \texttt{Offer}\ u_1\ u_2}\ [\textsc{Br}]$$

$$\frac{\Gamma \vdash P \rhd \Delta \cdot c : u_2}{\Gamma \vdash \texttt{sendS}_\pi\ c\ c'\ P \rhd \Delta \cdot c : \texttt{Throw}\ u_1\ u_2 \cdot c' : u_1}\ [\textsc{Thr}]$$

$$\frac{\Gamma \vdash P \rhd \Delta \cdot c : u_2, c' : u_1}{\Gamma \vdash \texttt{recvS}_\pi\ c\ (\lambda c'.P) \rhd \Delta \cdot c : \texttt{Catch}\ u_1\ u_2}\ [\textsc{Cat}]$$

$$\frac{\Gamma \vdash P \rhd \Delta \qquad \Gamma \vdash Q \rhd \Delta'}{\Gamma \vdash P \mid\mid\mid Q \rhd \Delta \oplus \Delta'}\ [\textsc{Conc}] \qquad \frac{\Gamma \vdash P \rhd \Delta \cdot c : \texttt{Bot}}{\Gamma \vdash \texttt{new}_\pi\ (\lambda c.P) \rhd \Delta}\ [\textsc{Cres}] \qquad \frac{\Delta \text{ completed}}{\Gamma \vdash \texttt{inact} \rhd \Delta}\ [\textsc{Inact}]$$

Figure 4: Typing rules for session types

3 Session-type inference on Haskell

We first introduce concurrency primitives in our implementation using the π-calculs defined in the Section 2.1. Then we present a few techniques to embed session types in Haskell as in [13] and [16]. Finally we show the session type reconstruction for multiple channels based on de Bruijn levels.

3.1 Concurrency primitives and session types in `full-sessions`

Our implementation, `full-sessions` , provides concurrency primitives using *monad* rather than the syntax provided in the Section 2. This is because monad is the most well-known way to describe communicating processes in Haskell.

[3]To show this we do not require type preservation, as stated in [5].

To keep connection between the original session-type system with our implementation, we show our primitives using continuation monad. The behaviour of each primitives is captured by the continuation-passing monad of type `((a -> Pi d`$_1$`) -> Pi d`$_2$`)` where `Pi d`$_i$ corresponds to the type of process term P in Section 2.1, and `d`$_i$ is a type-level representation of a session-type environment Δ. The meaning of each primitives are summarized in the Table 1. In the table we abuse the λ-notation of hoas syntax in Section 2 to represent a syntactic function from values or channels to processes. k ranges over continuations of type `a -> Pi d1`. For readability, we use the `ixdo` notation [16], which provides a syntactic sugar to write programs in an imperative style. For example, the term `ixdo send` c e`; recv` c and `ixdo fork (send` c e`); recv` c are interpreted as $\lambda k.\mathtt{send}_\pi\ c\ e\ (\mathtt{recv}_\pi\ c\ k)$ and $\lambda k.(\mathtt{recv}_\pi\ c\ k \mid\mid\mid \mathtt{send}_\pi\ c\ e)$, respectively.

Processes can be run using the function `runS`. `runS` p runs a π-calculus process $p(\lambda\_.\mathtt{inact}_\pi)$. Hereafter we call the all primitives in Table 1 as a *session* of type `Session`.

	Function	Meaning
Channel Creation	`new`	$\lambda k.\mathtt{new}_\pi\ k$
Value Output	`send` c e	$\lambda k.\mathtt{send}_\pi\ c\ e\ (k\ ())$
Value Input	`recv` c	$\lambda k.\mathtt{recv}_\pi\ c\ k$
Selection	`sel`i c $(i \in \{1,2\})$	$\lambda k.\mathtt{sel}i_\pi\ c\ (k\ ())$
Offering	`offer` c p_1 p_2	$\lambda k.\mathtt{offer}_\pi\ c\ (p_1\ k)\ (p_2\ k)$
Session Delegation	`sendS` c c'	$\lambda k.\mathtt{sendS}_\pi\ c\ c'\ (k\ ())$
Session Reception	`recvS` c	$\lambda k.\mathtt{recvS}_\pi\ c\ k$
Fork	`fork` p	$\lambda k.((k\ ()) \mid\mid\mid (p\ (\lambda\_.\mathtt{inact}_\pi)))$
Calling Haskell I/O	`io` m	(Execute Haskell's `IO` action m and pass the result to the continuation)
Recursion of a session	`recur1` f c	$\lambda k.f\ c\ k$ (Recursive call of a session $(f\ c)$ where c is a channel)
Recursive use of a channel	`unwind` c	$\lambda k.k\ ()$(Unwind a recursive session type `Rec` n u into u[`Var` $n \mapsto$ `Rec` n u] on c

Table 1: Primitives in the `full-sessions` library

3.2 Session type inference for a single channel

3.2.1 A single-threaded participant

Let us begin with a case of single channel in a single-threaded participant. In such a case a session type *advances* as a session proceeds. For example a type `Send Int End` advances to `End` when a channel of that type is used to send an integer. To track such an advance of a session type, we assign a pair of session types, called a *pre-type* and a *post-type*, to each occurrence of a channel. A pre-type denotes the session type *before* a session starts. Similarly, a post-type is the session type *after* a session ends.

In many cases post-types act as a *placeholder*, which allows concatenation of two session types. For example, consider one of the simplest sessions, `send` c `True`. The pre-type of the channel c in this session is `Send Bool` u and the post-type of it is u, where u is a type variable. This means that another session which uses the channel c can be further concatenated after this session.

The concatenation of two session types are done by unification. In a concatenation $s_1;s_2$ of two sessions, the post-types of channels in s_1 is unified with the pre-types of ones in s_2. The pre-types of channels in the concatenated session $s_1;s_2$ is same as the ones in s_1. The post-types of channels in $s_1;s_2$ is the ones of s_2. Accordingly, `(send` c `True; send` c `"abc")` has `(Send Bool (Send String` u_2`))` as the pre-type and u_2 as the post-type on the channel c, where u_2 is a type variable distinct from u.

For a more complex example, the code below describes a simple calculator server.

```
server c = ixdo x ← recv c; y ← recv c; offer c (send c (x+y::Int)) (send c (x<y))
```

The `server` firstly receives two values of type `Int` and a branch label (here the label is either 1 or 2), then sends an answer either of type `Int` or of `Bool` according to the label.

The pre/post-type of the channel `c` in the `server` can be inferred by the GHC's typechecker via auxiliary function `channeltype1`. By showing the type of `(channeltype1 server)` using GHC's interactive environment, users will obtain the following response:

```
prompt> :t channeltype1 server
channeltype1 server :: (Recv Int (Recv Int (Offer (Send Int a) (Send Bool a))), a)
```

3.2.2 Duality of two session types

The `fork` primitive requires the *duality* between pre-types of two sessions. Here we explain it by using the previous example of a calculator server. Firstly, a client of the server would be like this:

```
client c = ixdo send c 123; send c 456; sel2 c; ans ← recv c;
                io (putStrLn (if ans then "Lesser" else "Greater or Equal"))
```

The pre-/post-type of `c` in `client` is `(Send Int (Send Int (Select` u_1 `(Recv Bool` u`)))` and u, respectively. By putting `server` and `client` in parallel by `fork`, and by generating a channel by `new`, we obtain the code below:

```
calc c    = ixdo fork (server c); client c;
startCalc = ixdo c ← new; calc c
```

The above code typechecks because the two usages of `c` in `client` and `server` are dual. The resulting pre-type is `Bot`, as the session-type algebra of [6] implies. The post-type is `End` since `fork` requires the usage of channels in the given session to be ended. [4] Here we confirm it:

```
prompt> :t channeltype1 calc
channeltype1 calc :: (Bot, End)
```

A session can be run by the function `runS`. Typing `runS startCalc` will produce the result "`Lesser`" on the console. The following is the result of the execution using the interpreter:

```
prompt> runS startCalc
Lesser
```

3.3 Tracking sessions with multiple channels by De Bruijn indexing

To track usages of multiple channels in type-level, a natural number of *de Bruijn level* is assigned to each channel. De Bruijn level represents the nesting depth of a variable binder. For example, in a λ-calculus term $\lambda x.\lambda y.x$ the level of the variable x is 0 whereas y is 1. Figure 5 shows the de Bruijn level indexing of a session. In the figure, the de Bruijn level of a variable is denoted by a superscript at the binding position. Note that we need to count on only channels, hence each variable c,d,e and f have an index but x does not.

De Bruijn levels are assigned to the type of channels. A channel has the type of the form `Channel` t n where n is a de Bruijn level of the channel and t is a "type-tag" [11]. We do not explain the type-tag, since it is out of scope of our paper. Natural numbers are represented by combinations of the two

[4]Such discipline can also be observed in session-type systems equipped with a thread-spawning construct, the "ended" condition of **Spawn** rule in [4].

```
       n           n+1
ixdo c ← new; d ← new;
                      n+2
      fork (ixdo e ←catch c; ...)
      x ← recv d;
       n+2
      f ← catch d;
      ...
```

Figure 5: De Bruijn *level* indexing in a session

types representing peano-numerals `Z` and `S` n where each of them denotes 0 and $n+1$ respectively. For example, a channel which has de Bruijn level 2 has type `Channel` t `(S (S Z))`. Each number points to a certain position of a type environment.

Session types of multiple channels are recorded in extra type environments. We need two type environments for pre-types and post-types. Hereafter we call them *pre-environment* and *post-environment*, respectively.

Such an environment is represented by a list of session types, and its elements are accessed by specifying the number of de Bruijn level. Figure 6 is an example of a session `send` c `"abc"; send` d `True` and its pre-/post-environment. Assuming that c and d have (Haskell-) type `Channel` t `Z` and `Channel` t `(S Z)` respectively, the pre- and post-environment of the session is inferred as shown in the figure. c and d has pre-type `Send String` u_1 and `Send Bool` u_2, and post-types of them are u_1 and u_2, respectively. Note that the figure also depicts the session-types in an intermediate step after `send` c `"abc"`. In that state, c has type u_1 and d has type `Send Bool` u_2.

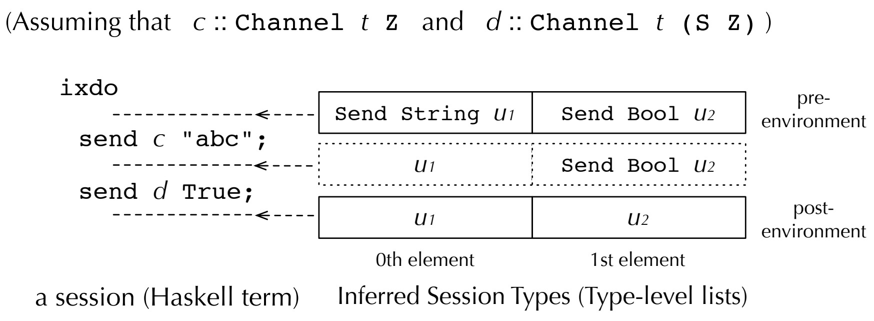

Figure 6: Tracking session types by numbers

The type of a session has the form of `Session` t ss tt a. The pre-/post-environments are at the position of ss and tt respectively. The parameter a is the type of a value returned by a session, and t is a type-tag.

The pre-/post-environments ss and tt are actually represented by *type-level list*s [10]. A type-level list is either ss `:>` u or `Nil`, where ss is another type-level list and u is a session type, and `Nil` is a empty list. Note that the type constructor `:>` is left-associative, for example ss`:>`a`:>`b is interpreted as `(`ss`:>`a`):>`b. Also note that the type environment is counted from left to right order. For example, the 0-th element of `Nil:>`a`:>`b`:>`c is a.

Provided that the type of c is `Channel` t `Z` and the type of d is `Channel` t `(S Z)`, a session of the previous example `(send` c `"abc"; send` d `True)` has type `Session` t `(Nil:>Send String` u_1`:> Send Bool` u_2`) (Nil:>`u_1`:>`u_2`) ()`.

In general, the de Bruijn levels can be a non-constant value, like $n+1$, $n+2$ and so on. For example, if the length of a session-type environment ss is n, and the type of c and d is `Channel` t n and `Channel` t `(S` n`)` respectively, a session `(send` c `1; send` d `True)` has type `Session` t `(`ss`:>Send Int` u_1`:> Send Bool` u_2`) (`ss`:>`u_1`:>`u_2`) ()`. Constraints for the length of a session-type environment is represented in the type-level by the type-class `SList` ss n, which represents that the length of ss is n. Observe that the existence of the placeholder ss in each of session-type environments. This makes possible to handle arbitrary numbers of channels by concatenation of sessions which introduce new channels, which involves unification between the post-environment of the earlier session and the pre-environment of the later session.

When a new channel is introduced, post-environments are *extended* to store the session type of the introduced channel. The primitive `new` and `catch` involve such a mechanism. `new` has pre-environment ss and post-environment ss`:>Bot`. At the same time `new` returns a channel of type `Channel` t n, where n is equal to the length of ss and points to the leftmost position of the post-environment, namely `Bot`. Hence the index of a generated name is assured to be fresh.

Figure 7 shows the pre-/post-environments of a session `(`c $\leftarrow$ `new; fork (send` c `True))`. The post-environment has an extra entry for the newly created channel. The post-type of the newly created channel is dual of `Send Bool End`, which is required to communicate with the forked session.

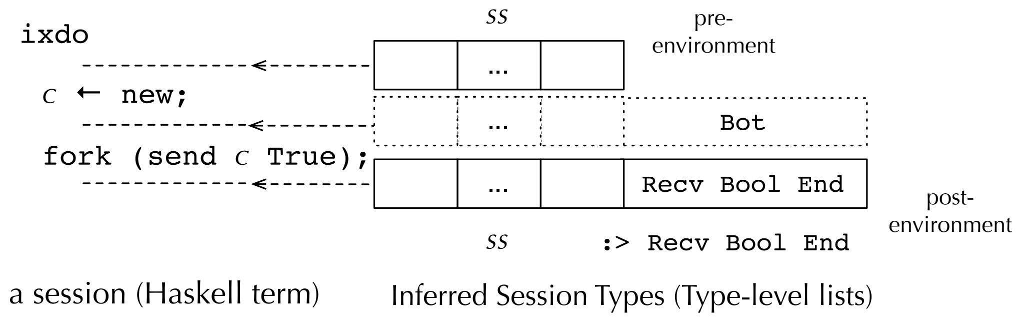

Figure 7: Extension of a type environment by `new` operation

Similarly, `catch` c has the pre-/post-environment ss and tt`:>`u', where n-th element of ss is `Catch` u' u and that of tt is u. Figure 8 shows such use of `catch` and the inferred session types.

3.4 Comparison of existing Haskell implementations of session types

Our encoding based on de Bruijn indexing reduces most of annotations which are required in the other works. We show that by giving a few examples of sessions.

Stack-based implementation The implementation by Pucella and Tov [16] applies a stack of session types as the representation of a type-environment. Communication primitives can only access the top of the stack, hence explicit manipulation of stack is required. The combinator `dig` and `swap` is provided for such purpose. The `swap` combinator swaps the top two channels on the stack. On the other hand, `dig` combinator converts a given session to operate on a deeper channel stack. Provided that the session type

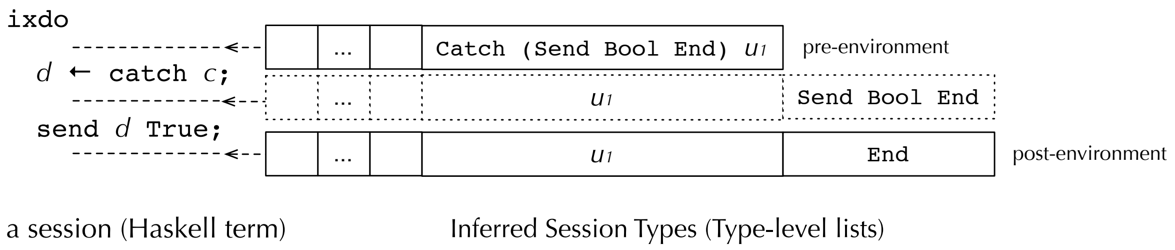

Figure 8: Extension of a type environment by `catch` operation

for *c* and *d* is on the top of the stack in this order, the code below is equivalent to `(send c "abc"; send d True)`:

```
ixdo send c "abc"; swap; send d True
```

or

```
ixdo send c "abc"; dig (send d True)
```

As a number of channels increases, more stack operations will be required. In [16] a few approaches to this problem are discussed, however the problem had been left open, and our number-based approach is not covered.

Manual construction of session types The implementation by Sackman and Eisenbach [17] provides a very rich set of communication primitives, at a cost of manual construction of session types. There are two communication media, channels and *Pid*s. We show the simplest case of communication via Pid. The example below passes an integer 10 to the other thread and terminates. It is equivalent to `runS (ixdo c <- new; fork (send c 10); recv c)`.

```
(s, a) = makeSessionType (
           newLabel ~>>= λa →
           a .= send (undefined :: Int) ~>> end
           ~>> sreturn a)

p = run s a (ssend 10) srecv
```

Here `makeSessionType` returns a collection of session types *s* and its fragment *a*. In the argument of `makeSessionType` the construction of a session type is described procedurally. Again, as a number of threads with different protocol increases, the more construction of session types will be required. The case of channel-based communication is similar.

4 An example SMTP client

We show the network functionality of the `full-sessions` by the example of a SMTP client with multiple channels. A single-channel version of SMTP client with session types has its origin from [13].

Table 2 shows additional primitives for network communication. To model network protocols, the *type-based branching*, `sel`*i*`N` and `offerN`, is provided in addition to the previous *label-based branching*. Note that the `sel`*i*`N` does nothing, but we need them to infer the session types for type-based selections.

	Syntax	Meaning
Connect to a service	c ← `connectNw` s	Connect to a service s and bind a session channel to c
Type-based offering	`offerN` c p_1 p_2	Offer two receiving session p_1 and p_2 on c
Selection annotation	`sel`i`N` c $(i \in \{1,2\})$	Determine which branch of `Select` u_1 u_2 will be selected on c

Table 2: Additional primitives for network programming

Here we show our implementation of SMTP client in the simplest form. Firstly, the types for SMTP commands and replies are defined as follows:

```
-- Types for SMTP commands.
newtype EHLO = EHLO String
newtype MAIL = MAIL String
newtype RCPT = RCPT String
data    DATA = DATA
data    QUIT = QUIT
newtype MailBody = MailBody [String]

-- Types for SMTP server replies (200 OK, 500 error and 354 start mail input)
newtype R2yz = R2yz String; newtype R5yz = R5yz String; newtype R354 = R354 String
```

To deal with the stream-based communication of TCP, either a parser or a printer for each type of communicated values must be prepared. Provided such functions exist, the SMTP client is described as follows:

```
-- auxiliary functions
send_receive_200 c mes = ixdo send c mes; (R2yz _) ← recv c; ireturn ()
send_receive_354 c mes = ixdo send c mes; (R354 _) ← recv c; ireturn ()

sendMail c d = ixdo  -- the body of our SMTP client
  (R2yz _) ← recv c                    -- receive 220
  send_receive_200 c (EHLO "mydomain") -- send EHLO, then receive 250
  unwind0 c                            -- (annotation) unwind a recursion variable
  sel1N c                              -- (annotation) branch to send 'MAIL FROM'
  from ← recv d                           -- (1) input the sender's address on d
  send_receive_200 c (MAIL from)       -- send 'MAIL FROM', then receive 250
  unwind1 c                            -- (annotation) unwind a recursion variable
  sel1N c                              -- (annotation) branch to send 'RCPT TO'
  to ← recv d                             -- (2) input the recipient's address on d
  send c (RCPT to)                     -- send 'RCPT TO'
  offerN c (ixdo                       -- branch the session according to the reply
    (R2yz _) ← recv c                  -- if 250 OK is offered
    sel1 d; mail ← recv d                 -- (3) input the content of the mail on d
    unwind1 c                          -- (annotation) unwind a recursion variable
    sel2N c                            -- (annotation) branch to send 'DATA'
    send_receive_354 c DATA            -- send 'DATA' and receive 354
    send_receive_200 c (MailBody mail) -- send the content of the mail
    unwind0 c                          -- (annotation) unwind a recursion variable
    sel2N c                            -- (annotation) branch to send 'QUIT'
    send c QUIT; close c               -- send 'QUIT' and close the connection
   ) (ixdo
```

```
      (R5yz errmsg) ← recv c;           -- if 500 ERROR is offered
      sel2 d; send d errmsg;               -- (4) output the error message on d
      send c QUIT; close c)             -- send 'QUIT' and close the connection
    close d
```

The `sendMail` takes two channels `c` and `d` as its parameters. The former is used to communicate with the SMTP server while latter is used to prepare necessary information for sending a mail. By checking the type of `typecheck2 sendMail`, the following type is answered by GHC:

```
typecheck2 sendMail :: (SList ss l, IsEnded ss b1) ⇒ Session t
  (ss :> Recv R2yz (Send EHLO (Recv R2yz (Rec Z (SelectN
          (Send MAIL (Recv R2yz (Rec (S Z) (SelectN
            (Send RCPT (OfferN (Recv R2yz (Var (S Z))) (Recv R5yz (Send QUIT Close))))
            (Send DATA (Recv R354 (Send MailBody (Recv R2yz (Var Z)))))))))
          (Send QUIT Close)))))
      :> Recv String (Recv String (Select (Recv [String] Close) (Send [String] Close))))
  (ss :> End :> End) ()
```

The SMTP protocol is successfully represented in the pre-type of `c`. A server that have the dual of this type can communicate with this client.

Observe that the two channels are used with no annotation. On the other hand, the implementation of [16] requires the `swap` operation before and after the each occurrence of `d`, namely at `(1)`, `(2)`, `(3)` and `(4)`, and if we add more channels, more complicated bookkeeping operations will be required.

5 Expressiveness of the encoding based on de Bruijn levels

We discuss the expressiveness between our implementation and the others. The discussion goes around the feature of higher-order sessions originally provided in [6]. We show that the presentation provided in [16] has some limitation. Due to that fact, our implementation is more expressive than [16].

At the same time, we sketch that their `swap` and `dig` can provide the same expressive power as our de Bruijn based solution.

5.1 The limitation of capability-passing primitive

The implementation presented in [16] does not provide primitives for channel-passing, while they provide `send_cap` and `recv_cap` which communicate the *capability* of channels. We discuss here that capability-passing does not provide full-fledged feature of the higher-order session.

The primitives `send_cap` and `recv_cap` only synchronize on a given channel, but not communicate any run-time information. Instead, on the synchronization the sender's side delegates the capability of a channel to the receiver's side.

In several cases this capability-passing is succinct to simulate name-passing. Here we sketch their capability-passing primitives by rewriting the code in our implementation using `send_cap` and `recv_cap`. The following code in our implementation

```
pq c = ixdo fork (q c); p c
p c  = ixdo d ← new; throw c d; str ← recv d; io (putStrLn str)
q c  = ixdo d ← catch c; send d "Hello"
```

will be rewritten in their framework as follows[5]:

[5]Assume they provide `new` and `fork` primitive in their language.

```
pq' c  = ixdo d ← new; fork (q' c d); p' c d
p' c d = ixdo send_cap c; dig (ixdo str ← recv d; io (putStrLn str))
q' c d = ixdo recv_cap c; dig (send d "Hello")
```

Notice that we put the channel-creation primitive (`new`) *before* the forking.

The problem of the capability passing is that the communicated channel must be known *before* the run-time. This is fatal for the distributed application which can not communicate any information before run-time. In addition, the π-calculus theory says that under the existence of recursion (or replication), the rewriting shown above is not possible. See the following code. The process sends fresh channel on `c` repeatedly with sending ”Hello” on the thrown channel.

```
loop c = ixdo unwind c; d ← new; throw c d; send d "Hello"; recur1 loop c
```

The `new` is now put under the loop. Such repeated channel-creation cannot bring out of the loop, hence cannot be expressed in [16].

5.2 Implementing channel-passing with `dig` and `swap`

The problem of `send_cap` is that the type-signature of it requires *static* (type-level) identity of channels, which seems to be unneeded constraint with respect to the original Session-type system [6]. The following is the type signature for the both primitives.

```
send_cap :: Channel t → Session (Cap t e (Cap t' e' r' :!: r), (Cap t' e' r', x))
                                (Cap t e r, x)
                                ()
recv_cap :: Channel t → Session (Cap t e (Cap t' e' r' :?: r), x)
                                (Cap t e r, (Cap t' e' r', x))
                                ()
```

Here, the first arguments of them must have pre-type `Cap t e (Cap t' e' r' :!: r)` or `Cap t e (Cap t' e' r' :?: r)`, where `t'` denotes identity of communicated channel.

Since there is no reason to have it, we put the alternative capability type `Cap2 e' r'` which does not have identity of a channel. By replacing `Cap` with `Cap2`, we get the following signature for sender's side:

```
send_chan :: Channel t → Channel t'
                       → Session (Cap t e (Cap2 e' r' :!: r), (Cap t' e' r', x))
                                 (Cap t e r, x)
                                 ()
```

On the other hand, the receiver's side is rather complicated. Since the identity of the communicated channel was lost at the sender's side, we must give the new one on the receiver's side. That was done by introducing an universally quantifying type variable on the signature. Since the type variable cannot escape from its scope, the continuation of the process must be given as the second argument.

```
recv_chan :: Channel t → (forall t'. Channel t' →
                              Session (Cap t e r, (Cap t' e' r', x))
                                      (Cap t e rr, y)
                                      () )
                       → Session (Cap t e (Cap2 e' r' :?: r), x)
                                 (Cap t e rr, y)
                                 ()
```

6 Other aspects of Session-type implementation

6.1 Representing recursion of session types

Many literature on session types takes *equi-recursive* view of types [15], which identifies $\mu t.T$ with its unfolded form $T\{\mu t.T/t\}$. Unfortunately, Haskell and many other languages do not support such typing. Hence ours and the other implementations of session types [13, 17, 16] take different approach, iso-recursive view of recursion on types. In this subsection we review each of them.

The first implementation of recursive session types [13] Neubauer et al. invented a representation of recursive type which requires a new type declaration for each session-type recursion. The type `Rec` is a fixpoint type constructor defined as follows:

```
newtype Rec f = MkRec (f (Rec f))
```

`Rec` has kind `(* → *) → *`. When a session repeatedly sends integers, the type corresponding to it must be declared first:

```
newtype G self = G (Send Int self)
```

where the type variable `self` is a placeholder for the recursion variable. By applying `Rec` on this type, the type `Rec G` is isomorphic to $\mu t.$`Send Int` t.

Such a type is unwound by declaring type classes. Consider expanding `Rec G` to `Send Int (Rec G)`. A type class `RECBODY` is declared as follows:

```
class RECBODY t c | t → c where ...
```

The first type parameter `t` is for folded form of a recursive type and the second type parameter `c` is for unfolded form. A functional dependency [8] `t → c` declares that Haskell's type checker can automatically infer `c` from `t`. The expanded form as an instance of `RECBODY` becomes following:

```
instance RECBODY (Rec G) (Send Int (Rec G)) where ...
```

However, declaring such types and instances for each recursion seems redundant. As [13] requires another explicit declaration of session types, such redundancy should be avoided. Our implementation and the other two implementations do not require such extra declarations. Hereafter we review that of ours and Pucella et al., though [17] do not give any account of it.

Expansion of recursive type representations by type-level computation In our implementation, such a recursive session type is represented in the form of `Rec Z (Send Int (Var Z))`. In `Rec` n r a type parameter n denotes the de Bruijn level of the binder and `Var` n is its occurrence. Then `Rec Z (Send Int (Var Z))` denotes $\mu t.$`Send Int` t. The primitive `unwind` expands the recursion. For example, `Rec Z (Send Int (Var Z))` is expanded as `Send Int (Rec Z (Send Int (Var Z)))`. The other primitive `recur1` f c is behaviourally equal to $f\,c$, yet this ensures that the pre-types of the all channels other than c is ended. Such annotation is required since sometimes the usage of some channels has no explicit end and it cannot be inferred by the typechecker.

Our encoding depends on Haskell's type-level computation. The encoding by Pucella and Tov [16] is not limited to Haskell, at a cost of a bit complex representation in types.

Deferred expansion of recursion body [16] In [16], a recursive type is represented by using de Bruijn *indices* (not levels) as a binder for recursion variable. Thus, the type for a session which repeatedly sends integers is `Rec (Send Int (Var Z))` (note that this `Rec` is different from previous one). Here `Var` *n* is a type-level recursion variable, where *n* is a peano-numeral of the de Bruijn index of the binder.

Their *capability type* has the form of `Cap t e r` where `t` is type tag [11] and `r` is a session type. The second parameter `e` represents a *stack* which is used for bookkeeping during recursion. Note that this stack is different from that of multiple channels in Section 3.4.

A recursive type is not immediately expanded, but deferred by using a notational trick. To see this, consider expanding a recursive session `Cap t e (Rec (Send Int (Var Z)))`. The recursion body is put at the second parameter, resulting `Cap t (Send Int (Var Z),e) (Send Int (Var Z)))`. When one met the recursion variable `Cap t (Send Int (Var Z),e) (Var Z)`, the substitution is actually done and it becomes `Cap t (Send Int (Var Z),e) (Send Int (Var Z))`.

By putting a notational trick, Pucella and Tov succeed to represent session-type recursion in a language-independent way. Since our implementation already uses heavy type-level computation, we have used full functionality of type-classes to represent recursions in a more direct way. In other words, our encoding does not require stacks for recursions.

6.2 Inter-process Communication

One notable difference between ours and [17] is that the communication primitives in their implementation are based on process identities, *Pid*s.. Providing both Pids and channels as communication media would be much convenient in view of scalability. However, since their framework requires much of annotations, usage of channels would become burden.

In [17], a typical inter-process communication example that a process `parent` forks a process `child` to send an integer 52 is written as follows:

```
(st, a) = makeSessionType (
            newLabel ~>>= λa →
            a .= recv int ~>> send bool ~>> end ~>>
            sreturn a)
  where
    int = undefined :: Int
    bool = undefined :: Bool

parent = fork a dual (cons (a, notDual) nil) child
         ~>>= λ(_, childPid) →
         createSession a dual childPid
         ~>>= λchildCh →
         withChannel childCh (ssend 52 ~>> srecv)
         ~>>= sliftIO . print

child _ parentPid
       = createSession a notDual parentPid
         ~>>= λparentCh →
         withChannel parentCh
           (srecv ~>>= ssend . ((==) 42))
```

where `a` is the value-level session type associated to the channel. `st` is the session type associated to a Pid, which is not used. The values `dual` and `notDual` is needed because the framework does not infer which side of the protocol it uses. Actual communication is described at the second argument of each call of `withChannel`.

This complication arises in order to maintain type-level symbol tables. Comparing with this, the same behaviour can simply be described in our framework as follows:

```
parent = ixdo
  ch ← new;
  fork (child ch)
  send ch 52 >>> recv ch >>>= io . print

child ch = recv ch >>>= send ch . ((==) 42)
```

Thanks to inference of the symbol table based on de Bruijn levels, our encoding requires essential communication primitives around the session type.

7 Usability

Here we discuss a few aspects of session type implementation.

Trade-offs between type inference and manual construction of session types As we have shown in Section 3.4, annotations required by our implementation is not more than any of the other implementations. However, there seems to be a few advantages in [17] in a few points. (1) Recursion of a session type is treated more naturally in [17]. By using term-level operation for constructing session types, [17] offers more readable formulation of recursion via labels. As you can observe in the SMTP example of the previous section, recursion on a session type require a few of not so intuitive annotations $\texttt{unwind}_i$ on the term-level to represent a recursive protocol. (2) Manual construction of session types in term-level offers chance of subtyping. It is difficult to allow subtyping of session types in the parallel composition, because of our bijective encoding of duality to extract more information in a parallel composition of a session.

Readability of type error messages If the duality check of two session types fails, the type error would be reported. For example, by replacing the occurrence of an integer `456` in Section 3.2.2 with a string `"456"`, the following error is obtained :[6]

```
examples/calc.hs:<xx>:0:
    Couldn't match expected type '[Char]' against inferred type 'Int'
      Expected type: tt' :> Send [Char] a
      Inferred type: tt' :> Send Int (Select (Recv Int End) (Recv Bool End))
    When generalising the type(s) for 'plus'
```

The error reports that the inferred pre-type of `client` is not compatible with the expected one. The position `<xx>` of the reported error is not at `send c "456"` itself, but at the position where the dual of the session type is calculated, namely the occurrence of the `fork`. Thus, this error message directly shows which session types are not compatible. Even the type-level hackery we depend tends to produce large type signatures, the type error itself can be concisely represented.

[6]Here, `[Char]` is a type synonym of `String`.

8 Concluding remarks

This paper showed a Haskell implementation of the session-type inference. Our implementation infers session types fully automatic without any manual operations such as stack operations in [16].

The treatment of binders is the key issue for embedding one language into another, as stated in [2]. In our implementation, we took a separated approach for the *term-level computation* and *type-level* (compile-time) computation. In the term-level computation, a fresh channel is represented by λ-abstraction (the technique usually called Higher order abstract syntax), utilizing the power of variable-bindings in the host language of Haskell. In the type-level computation, *de Bruijn levels* represent in channel types to compare names. These are the key to automate the session-type inference. However, since the current technique depends on the type-level programming functionality of Haskell, it is not easy to export this technique to the other programming languages yet.

Our technique using de Bruijn level can be applied to other substructural type systems for the π-calculus, such as linear type systems and multiparty session types [7]. In particular, encoding of multiparty session types is promising. The end-point session types of [7] is much similar with the original binary session types [6], hence our technique can be effectively used. In Haskell, types cannot have different concrete representation of types since Haskell's type inference goes through unification. However, due to asynchronous nature of multiparty session types, a end-point type $k\langle U\rangle;k'\langle U'\rangle;T$ can have different concrete representation $k'\langle U'\rangle;k\langle U\rangle;T$ if $k \neq k'$ where two first components can be exchanged. To express such type in a unique form, again our de Bruijn encoding of channels might play a key role. That is, ordering such asynchronous sequencing by the de Bruijn level, one can obtain the unique representation of a end-point type. Yet much remains to be done in making such ideas in a real code.

Acknowledgments The first author thanks his colleagues in IT Planning Inc. for supporting him during the writing of this paper. This work was partially supported by the Grant-in-Aid (Scientific Research (B) 20300009 and Scientific Research(C) 19500026) from the Ministry of Education, Culture, Sports, Science, and Technology of Japan.

References

[1] Robert Atkey (2009): *Parameterized Notions of Computation*. Journal of Functional Programming 19(3-4), pp. 335–376, doi:10.1017/S095679680900728X.

[2] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The POPLMARK Challenge*. In: Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science 3603, Springer-Verlag, pp. 50–65, doi:10.1007/11541868_4.

[3] Björn Bringert, Anders Höckersten, Conny Andersson, Martin Andersson, Mary Bergman, Victor Blomqvist & Torbjörn Martin (2004): *Student paper: HaskellDB improved*. In: Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell, ACM, pp. 108–115, doi:10.1145/1017472.1017473.

[4] Mario Coppo, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2007): *Asynchronous Session Types and Progress for Object Oriented Languages*. In: Formal Methods for Open Object-Based Distributed Systems, Lecture Notes in Computer Science 4468, Springer-Verlag, pp. 1–31, doi:10.1007/978-3-540-72952-5_1.

[5] Marco Giunti, Kohei Honda, Vasco T. Vasconcelos & Nobuko Yoshida (2009): *Session-Based Type Discipline for Pi Calculus with Matching*. In: In the preproceedings of PLACES '09: Programming Language Approaches to Concurrency and Communication-cEntric Software. Available at `http://places09.di.fc.ul.pt/`.

[6] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP '98: Proceedings of the 7th European Symposium on Programming*, *Lecture Notes in Computer Science* 1381, Springer-Verlag, pp. 122–138, doi:10.1007/BFb0053567.

[7] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. *SIGPLAN Notices* 43(1), pp. 273–284, doi:10.1145/1328438.1328472.

[8] Mark P. Jones (2000): *Type Classes with Functional Dependencies*. In: *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, Springer-Verlag, pp. 230–244, doi:10.1007/3-540-46425-5_15.

[9] Oleg Kiselyov (2006): *Simple variable-state monad*. Available at `http://www.haskell.org/pipermail/haskell/2006-December/018917.html`. Mailing list message.

[10] Oleg Kiselyov, Ralf Lämmel & Keean Schupke (2004): *Strongly Typed Heterogeneous Collections*. In: *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, ACM Press, pp. 96–107, doi:10.1145/1017472.1017488.

[11] Oleg Kiselyov & Chung C. Shan (2008): *Lightweight monadic regions*. In: *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, ACM, pp. 1–12, doi:10.1145/1411286.1411288.

[12] Daan Leijen & Erik Meijer (2001): *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Technical Report, Departement of Computer Science, Universiteit Utrecht. Available at `http://www.cs.uu.nl/~daan/parsec.html`.

[13] Matthias Neubauer & Peter Thiemann (2004): *An Implementation of Session Types*. In: *PADL'04 : Practical Aspects of Declarative Languages*, *Lecture Notes in Computer Science* 3057, Springer-Verlag, pp. 56–70, doi:10.1007/978-3-540-24836-1_5.

[14] Frank Pfenning & Conal Elliot (1988): *Higher-Order Abstract Syntax*. In: *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, ACM Press, pp. 199–208, doi:10.1145/53990.54010.

[15] Benjamin C. Pierce (2002): *Types and Programming Languages*. MIT Press.

[16] Riccardo Pucella & Jesse A. Tov (2008): *Haskell Session Types with (Almost) No Class*. In: *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, ACM Press, pp. 25–36, doi:10.1145/1411286.1411290.

[17] Matthew Sackman & Susan Eisenbach (2008): *Session Types in Haskell: Updating Message Passing for the 21st Century*. Technical Report, Imperial College London. Available at `http://pubs.doc.ic.ac.uk/session-types-in-haskell/`.