An Implementation of Session Types

Matthias Neubauer and Peter Thiemann[⋆]

Universität Freiburg
Georges-Köhler-Allee 079
D-79110 Freiburg, Germany

Abstract. A session type is an abstraction of a set of sequences of heterogeneous values sent and received over a communication channel. Session types can be used for specifying stream-based Internet protocols. Typically, session types are attached to communication-based program calculi, which renders them theoretical tools which are not readily usable in practice. To transfer session types into practice, we propose an embedding of a core calculus with session types into the functional programming language Haskell. The embedding preserves typing. A case study (a client for SMTP, the Simple Mail Transfer Protocol) demonstrates the feasibility of our approach.

Key Words functional programming, types, domain specific languages

1 Introduction

Much foundational work on calculi for concurrency is devoted to studying synchronous, one-shot communications, for example, CCS [20], the π calculus [21], the chemical abstract machine [5], the join calculus [9], and the M-calculus [32]. However, in particular in distributed systems, the cost of one-shot communications can be too high because a new connection must be established for each message and synchronous operation may be too restrictive. Hence, calculi and programming languages have been developed that are either based on asynchronous communication [14] or that incorporate channel-based communication primitives [12, 25, 29]. Once a channel has been created, many distinct messages may be communicated through it. Channels are often homogeneous, that is, all messages must have the same type.

Session types [10] have emerged as an expressive typing discipline for *heterogeneous*, bidirectional communication channels. In such a channel, each message may have a different type with the possible sequences of messages determined by the channel's session type. Such a type discipline subsumes typings for datagram communication as well as for homogeneous channels. Session types have been used to describe stream-based Internet protocols such as POP3 [10, 11].

A regular language on atomic communication actions describes the sequence of messages on each channel. The channel type specifies this language with a fixpoint expression. Each operation peels off the outermost action from the channel

⋆ {neubauer,thiemann}@informatik.uni-freiburg.de

$l \in$ Label
Definitions
$d ::= x = \mathtt{Op}(\tilde{x}) \mid \mathtt{rec}\ x(\tilde{x}) = e$
$\quad \mid \ \mathtt{Send}\ l(x) \mid \mathtt{Close}\ ()$
Expressions
$e ::= \mathtt{Halt} \mid \mathtt{Let}\ d\ \mathtt{in}\ e \mid \mathtt{If}\ x\ \mathtt{then}\ e\ \mathtt{else}\ e \mid x\ (\tilde{x})$
$\quad \mid \ \mathtt{Receive}\ [g]$
$g ::= l(x) \to e \mid g, g$

Types
$\tau ::= b \mid \tilde{\tau}\ , \gamma \to 0$
Type Environments
$\Gamma ::= \emptyset \mid \Gamma(x : \tau)$
Session types
$\gamma ::= \emptyset \mid \varepsilon \mid [\eta] \mid \beta \mid \mu\beta.\gamma$
$\eta ::= \ell(b) : \gamma \mid \eta, \eta$
$\ell ::= l \mid \bar{l}$

Fig. 1. Syntax

type so that each operation changes the channel's type. For that reason, channels should not be duplicated but rather be treated linearly by the type system.

The resulting type system is amenable to type inference using techniques developed for recursive types. Session types are compatible with polymorphic type inference [15]. However, thus far no mainstream programming language supports session types, so it is hard to take advantage of them in practice.

The present work demonstrates how to embed session types into a functional programming language with a sufficiently powerful type system. (Extended) Haskell fits this bill, but other languages with constrained type systems would be suitable, too [33]. In particular, type classes with functional dependencies [16] are required to model the progression of the current state of the channel and functions with polymorphic parameters are required to model client and server side of a communication with one specification [23].

The main contribution of the present work is an encoding of session types in terms of type classes with functional dependencies. A key technical problem is the encoding of fixpoint (μ) expressions that occur in the description of the regular language mentioned above. We define a typed translation from a calculus with session types into Haskell and prove its soundness. Finally, we demonstrate the practicability of our approach with a case study, a typed client for the Simple Mail Transfer Protocol (SMTP) [30].

The paper is structured as follows. Section 2 defines a small calculus for asynchronous communication with session types. Section 3 specifies the basic ideas for translating the calculus to Haskell and Section 4 gives a detailed overview of the translation. Section 5 contains excerpts from our type-safe implementation of an SMTP client. Section 6 briefly discusses related work on domain modeling with types and Section 7 concludes. Due to lack of space, we have to assume that the reader is reasonably fluent in Haskell [13].

2 Calculus with Session Types

Figure 1 presents the syntax of a calculus with session types. For simplicity, the calculus only considers **one** end of **one** communication channel. The restriction to one end avoids the necessity and semantic complication of adopting an expression for concurrent execution in the syntax. It is adequate for the present purpose because the interest is in type checking the code for one peer (client or server), not it checking the consistency of a whole system of processes. The

$$\mathit{\Gamma}\,,\emptyset \vdash \texttt{Halt}$$

$$\frac{\mathit{\Gamma}\,,\gamma \vdash d \Rightarrow \mathit{\Gamma}'\,,\gamma' \qquad \mathit{\Gamma}'\,,\gamma' \vdash e}{\mathit{\Gamma}\,,\gamma \vdash \texttt{Let}\ d\ \texttt{in}\ e} \qquad \frac{\mathit{\Gamma}(x) : b \qquad \mathit{\Gamma}\,,\gamma \vdash e_1 \qquad \mathit{\Gamma}\,,\gamma \vdash e_2}{\mathit{\Gamma}\,,\gamma \vdash \texttt{If}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$\frac{\mathit{\Gamma}(x) = \tilde{\tau}\,,\gamma \to 0 \qquad \mathit{\Gamma}(\tilde{z}) = \tilde{\tau}}{\mathit{\Gamma}\,,\gamma \vdash x\ \tilde{z}} \qquad \frac{\gamma = [\overline{l_i}(b_i) : \gamma_i]_{i=1}^{n} \qquad \mathit{\Gamma}(x_i : b_i)\,,\gamma_i \vdash e_i}{\mathit{\Gamma}\,,\gamma \vdash \texttt{Receive}\ [l_i(x_i) \to e_i]}$$

Fig. 2. Typing rules for expressions

restriction to one communication channel transforms session types to communication effects [1], makes a linear treatment of the channel enforcable by syntactic means, and simplifies the translation in Section 4.

The calculus deals with three kinds of data, first-order base type values, functions, and labels. Labels have a status similar to labels in record and variant types, they occur in channel types and they can be sent and received via channels: each message is a base value tagged with a label.

The expressions of the calculus come in a sequentialized style reminiscent of continuation-passing style. More precisely, an expression e is a sequence of (let-) definitions which ends in either a `Halt` instruction, a conditional, a function call, or a receive instruction that branches on the received label. All arguments are restricted to variables. The notation $\tilde{x}$ stands for the sequence $x_1, \ldots, x_n$ where n derives from the context.

A definition d is either the application of a primitive operation, the definition of a recursive function, the send operation, and the close operation for closing the communication channel.

A type is either a base type or a function type. Due to the sequential style, functions do not return values. Instead they must take a continuation argument. The function type also includes an effect specification. It defines the latent communication that will take place when the function is applied [1].

A session type is either empty (the channel is closed), the empty word (the channel is depleted but not yet closed), a label-tagged alternative of different session types (the value may be sent $l(b)$ or received $\overline{l(b)}$), or a type variable which is used in constructing a recursive type with the μ operator. The μ operator constructs a fixpoint, *e.g.*, $\mu\beta.\gamma \approx \gamma[\beta \mapsto \mu\beta.\gamma]$. All uses of μ are *expansive*, that is, there can be no subterms of the form $\mu\beta_1 \ldots \mu\beta_n.\beta_1$.

The type system relies on two judgments, $\mathit{\Gamma}\,,\gamma \vdash e$, to check the consistency of an expression and prescribe its communication effect γ, and $\mathit{\Gamma}\,,\gamma \vdash d \Rightarrow \mathit{\Gamma}'\,,\gamma'$ to model the effect of a definition and its transformation of the environment.

Figure 2 contains the rules for expressions. `Halt` requires that the channel is closed. The let expression types the body after transforming the environment according to the definition. The conditional passes the environment unchanged to both branches. Applying a function requires that the function consumes the remaining effect. Receiving a tagged value eliminates a labeled alternative in the session type. The branches are typed with the remaining session type.

Figure 3 contains the typing rules for definitions. A primitive operation has base type arguments and result. It does not depend on the session type. Function formation is independent of the current session type, too. The body of the

$$\frac{\Gamma(x_i) = b}{\Gamma\ , \gamma \vdash x = \texttt{Op}(x_1, \ldots, x_n) \Rightarrow \Gamma(x : b)\ , \gamma} \qquad \frac{\Gamma(f : \tilde{\tau}\ , \gamma \to 0)(\tilde{x} : \tilde{\tau})\ , \gamma \vdash e}{\Gamma\ , \gamma' \vdash \texttt{rec}\ f(\tilde{x}) = e \Rightarrow \Gamma(f : \tilde{\tau}\ , \gamma \to 0)\ , \gamma'}$$

$$\frac{\Gamma(x_j) = b_j \qquad \gamma = [l_i(b_i) : \gamma_i]_{i=1}^{n}}{\Gamma\ , \gamma \vdash \texttt{Send}\ l_j(x_j) \Rightarrow \Gamma\ , \gamma_j}\ 1 \leq j \leq n \qquad \Gamma\ , \varepsilon \vdash \texttt{Close}\ () \Rightarrow \Gamma\ , \emptyset$$

Fig. 3. Typing rules for definitions

function must be checked with the session type prescribed by the call site of the function. Sending of a labeled value selects a part of the session type for the rest of the expression. Closing the communication channel requires that there are no exchanges left. The rule sets the remaining exchanges to the empty set.

The present paper does not define a semantics for the language. Suitable semantics (for extended languages) may be found in work of Gay et al [11].

3 Basic Framework

The mapping from the calculus with session types to Haskell rests on a few overloaded functions. Each message has its own specific type and it comes with functions to parse and unparse a message, as specific with the type class `Command`[1].

```
class Command command where
  parseCommand   :: ReadS command
  unparseCommand :: command -> ShowS
```

The underlying datatype is a function that maps the session type, a string, and a file handle to an `IO` action. The string contains the input and output is generated via the handle in the `IO` monad. Hence, the datatype is the composition of three reader monads and the `IO` monad.[2]

```
data Session st a = Session { unSession :: st -> String -> Handle -> IO a }
```

The attentive reader may wonder if this design of the session monad is the only possible. Consider the following two alternatives:

– Suppose that output is generated via some writer monad. Since the `IO` monad is still required because of `IO` actions that must be performed between the communication actions, the resulting type would look like `IO (a, Output)`. Unfortunately, a value in the `IO` monad does not return anything before all actions specified by the value have been completed [27]. That is, any output generated through the `Output` component in the middle of a transaction would only appear at the very end of that transaction. Too late for an interactive response!
– Suppose that input is obtained directly via the `IO` monad, for example, by having the parsers call `getChar :: IO Char` directly. Unfortunately, as

[1] `ReadS` and `ShowS` are predefined Haskell types for parsing and unparsing data.
[2] The obvious definitions to make `Session st` into a monad are omitted.

we will see, the parsers must be able to proceed speculatively and hence unget characters. While this option seems viable, it would have required the construction of an imperative parsing library based on `getChar` and `unGetChar`. We have not pursued this further due to lack of time.

The `send` operation takes the message to send (identified by its type) and a continuation to construct an action for the current session type. Executing `send` first sends the message, advances the session type as indicated by the typing rule, and then invokes the continuation on the new session type. The declaration of the operation (a type class) makes clear that the new type depends on the old type and the message by using a functional dependency.

```
class SEND st message nextst | st message -> nextst where
  send    :: message -> Session nextst () -> Session st ()
```

The `receive` operation takes a message continuation. It attempts to parse the message from the input, advances the session type, and invokes the continuation on the message and the new session type.

```
receive :: (RECEIVE st cont) => cont -> Session st ()
receive g = Session (\ st inp h -> case receive' g st inp h of
                                    Just action -> action
                                    Nothing     -> fail "unparsable")
```

When `receive` is applied directly to a message, `cont` has the form `message -> Session nextst ()` but this is not always the case: an alternative of different messages is also possible in which case the type is an alternative of the different continuations. An auxiliary function attempts to parse the expected message and returns either the continuation or nothing.

```
class RECEIVE st cont | st -> cont where
  receive' :: cont -> st -> String -> Handle -> (Maybe (IO ()))
```

The `io` operation serves to embed `IO` operations between the `send` and `receive` operations.

```
io :: IO a -> Session st a
io action = Session (\_ _ _ -> action)
```

Finally, the operation `close` closes the communication channel. It is only applicable if the session type is `EPS` (the translation of ε).

```
close :: Session NULL () -> Session EPS ()
close cont = Session (\_ _ _ -> unSession cont NULL [])
```

4 Translation

To make the information in a session type available to the Haskell type checker, all parts must be translated accordingly. Each labeled message is lifted to a separate type where the base types correspond to Haskell base types. Each constructor of a session type is mapped to a type constructor:

$$\begin{array}{lll}
[\![\emptyset]\!] \quad = \texttt{NULL} & [\![[\ell(b) : \gamma]]\!] = [\![\ell(b) : \gamma]\!] & [\![\bar{l}(x) : \gamma]\!] = \texttt{recv}\ [\![l(x)]\!]\ [\![\gamma]\!] \\
[\![\varepsilon]\!] \quad = \texttt{EPS} & [\![[g_1, g_2]]\!] \quad = \texttt{ALT}\ [\![[g_1]]\!]\ [\![[g_2]]\!] & [\![l(x) : \gamma]\!] = \texttt{send}\ [\![l(x)]\!]\ [\![\gamma]\!] \\
\mathcal{T}[\![\beta]\!]\tilde{\beta} = \beta & \mathcal{T}[\![\mu\beta.\gamma]\!]\tilde{\beta} = \texttt{REC (G send recv}\ \tilde{\beta}) & \\
 & \quad \text{where} \quad \texttt{data G send recv}\ \tilde{\beta}\ \beta = \texttt{G}(\mathcal{T}[\![\gamma]\!]\tilde{\beta}\beta) & \\
 & \quad \text{and} \quad \texttt{send}, \texttt{recv} :: \texttt{* -> * -> *} & \\
\mathcal{V}[\![\beta]\!] \quad = \beta & \mathcal{V}[\![\mu\beta.\gamma]\!] \quad = \texttt{let}\ \beta = \texttt{REC (G (}\mathcal{V}[\![\gamma]\!]))\ \texttt{in}\ \beta &
\end{array}$$

Fig. 4. Translation of Types and Values

```
data NULL         = NULL          -- the closed session
data EPS          = EPS           -- the empty session
data SEND_MSG m r = SEND_MSG m r  -- send message m, then session r
data RECV_MSG m r = RECV_MSG m r  -- receive message m, then session r
data ALT l r      = ALT l r       -- alternative session: either l or r
```

4.1 Translation of Session Types

The translation concerns types and values. It does not directly refer to SEND_MSG and RECV_MSG, instead it is parameterized over the send and receive operations, send and recv. This parameterization enables flipping the session type. Flipping is required to switch the point of view from one end of the channel to the other. In a session type it corresponds to exchanging all occurrences of $\bar{l}$ and l.

Hence, all types are parameterized over send and recv of kind `* -> * -> *` and all values are parameterized by *polymorphic functions* send and recv of type `forall x y . send x y` and `forall x y . recv x y`, respectively.

Figure 4 contains the definition of the translation. The $\mathcal{V}$ function defines the special cases for the value translation, $\mathcal{T}$ the cases for the type translation. The main difficulty is the treatment of the recursion operator. There are two problems due to restrictions on Haskell's type system. First, each use of recursion requires an explicit datatype definition. Fortunately, there is a generic encoding using a datatype that subsumes many recursively defined datatypes [19]:

```
data REC f = REC (f (REC f))
```

In this datatype, `f` is a constructor function of kind `* -> *`. Technically, REC constructs the fixpoint of this function `f`, up to lifting.

Second, instantiation of `f` is restricted to type constructor terms to keep the type system decidable. The ideal translation would be

$$\begin{array}{ll}
\mathcal{T}[\![\mu\beta.\gamma]\!] & = \texttt{REC}(\lambda\beta.\mathcal{T}[\![\gamma]\!]) \\
\mathcal{T}[\![\beta]\!] & = \beta
\end{array}$$

where $\lambda\beta\ldots$ is a lambda expression at the type level. However, such lambda expressions are not admissible in Haskell types and $(\lambda\beta.\mathcal{T}[\![\gamma]\!])$ cannot always be η-reduced to a pure constructor term: as an example consider that $\mu\beta.[l_1 : \beta, l_2 : \varepsilon]$

would translate to REC($\lambda\beta$.ALT(send $[\![l_1]\!]$ β)(send $[\![l_2]\!]$ EPS)). Hence, the translation of types needs to introduce a new parameterized datatype for each occurrence of $\mu\beta.\gamma$ in a session type. Furthermore, since datatype definitions cannot be nested, the translation needs to keep track of the pending type variables (corresponding to occurrences of $\mu\beta$'s in the context) and parameterize the new type over all these variables.

The corresponding part of the translation on the value level requires the introduction of a recursive definition, too. Technically, the two translations should be merged to fix the association of each $\mu\beta.\gamma$ to its corresponding datatype G. We keep them separate for readability.

For the remaining cases, $\mathcal{V}[\![]\!]$ and $\mathcal{T}[\![]\!]$ are equal to $[\![]\!]$. In the type translation, the parameters $\tilde{\beta}$ are always passed unchanged to the recursive calls.

Lemma 1. *Let Γ contain the types for the data constructors listed above and γ be a closed, expansive session type.*

$$\Gamma \vdash_{haskell} \mathcal{V}[\![\gamma]\!] : \mathcal{T}[\![\gamma]\!]$$

The variables send and receive are provided as polymorphic parameters:

```
\ (send :: (forall x y . x -> y -> send x y))
  (recv :: (forall x y . x -> y -> recv x y)) -> 𝒱⟦γ⟧
```

Similarly, the type translation refers to type variables send and recv so that the full type of the value produced by the translation is

```
forall send recv.
  (forall x y . x -> y -> send x y) ->
  (forall x y . x -> y -> recv x y) -> 𝒯⟦γ⟧
```

Additionally, the translation might restrict the instantiation of send and recv so that the supplied operations are always opposites. A further two-parameter type class would be required to specify this relation.

4.2 Connecting Session Types with Messages

The specification of the connection between the message to send or receive and the actual session type requires three ingredients. First, atomic messages must be matched with their specification. Second, when the specification prescribes an alternative (with the ALT operator), the matching must be properly dispatched to the alternatives. Third, recursion must be unwound whenever matching encounters the REC operator. The first part is straightforward, the second and third require careful consideration.

Atomic Messages If the session type prescribes that an atomic message of type m be send, then the actual message type must be m and the type of the remaining session is the stripped session type. Similarly, if the session type prescribes that a message of type m be received then there must be a continuation that expects a message of that type and continues in the stripped session state.

```
instance Command m => SEND (SEND_MSG m b) m b where
  send mess cont =
    Session (\ st inp h -> do hPutStr h (unparseCommand mess "")
                              unSession cont (send_next st) inp h)

instance Command m => RECEIVE (RECV_MSG m x) (m -> Session x ()) where
  receive' g st inp h =
    case parseCommand inp of
      ((e, inp'):_) -> Just (unSession (g e) (recv_next st) inp' h)
      []            -> Nothing
```

The functions `send_next` and `recv_next` are just projections on the last argument of `SEND_MSG` and `RECV_MSG`:

```
send_next (SEND_MSG m s) = s
recv_next (RECV_MSG m s) = s
```

Alternatives When the current operation is a receive operation and the protocol prescribes an alternative of different messages, then the implementation attempts to parse the input according to the alternatives and selects the first successful parse. This trial is encoded in the `receive'` function (and is the main reason for keeping `receive` and `receive'` separate). It also serves as an example where the first argument of the `receive` operator is **not** a function.

```
instance (RECEIVE spec1 m1, RECEIVE spec2 m2) =>
         RECEIVE (ALT spec1 spec2) (ALT m1 m2) where
  receive' (ALT g1 g2) (ALT spec1 spec2) inp =
    case receive' g1 spec1 inp of
        Just action -> Just action
        Nothing     -> receive' g2 spec2 inp
```

When the current operation is a send operation and the protocol prescribes an alternative, then the automatic matching of the protocol specification against the message type would be fairly complicated [22]. For that reason, the alternatives must be explicitly selected prior to a send operation. Two primitive selector functions are provided for that task[3]:

```
left  :: Session l x -> Session (ALT l r) x
right :: Session r x -> Session (ALT l r) x

left  (Session g) = Session (\(ALT l r) inp -> g l inp)
right (Session g) = Session (\(ALT l r) inp -> g r inp)
```

This design has another positive effect: subtyping of the session type for send operations! If a protocol states that a certain message must be received, then the implementor must make sure that the message is understood. Hence, the matching for the `RECEIVE` class must be complete. From the sender's perspective, a protocol often states that a selection of messages may be sent at a certain point. So the sender's implementation may choose to omit some of the alternatives. The above arrangement makes this possible: a sender may choose just to send the left alternative and leave the other unspecified.

[3] They also require a type class for unwinding recursion.

$$
\begin{aligned}
&\llbracket \texttt{Halt} \rrbracket && = \texttt{return ()} \\
&\llbracket \texttt{If } x \texttt{ then } e_1 \texttt{ else } e_2 \rrbracket && = \texttt{if } x \texttt{ then } \llbracket e_1 \rrbracket \texttt{ else } \llbracket e_2 \rrbracket \\
&\llbracket x\ \tilde{z} \rrbracket && = x\ \tilde{z} \\
&\llbracket \texttt{Receive } [g] \rrbracket && = \texttt{receive } \llbracket g \rrbracket \\
&\llbracket g, g \rrbracket && = (\texttt{ALT } (\llbracket g \rrbracket)\ (\llbracket g \rrbracket)) \\
&\llbracket l(x) \to e \rrbracket && = (\lambda l(x) \to e) \\
&\llbracket \texttt{Let } x = \texttt{Op}(x_1, \ldots, x_n) \texttt{ in } e \rrbracket && = \texttt{io}(\texttt{Op}(x_1, \ldots, x_n)) \texttt{ >>= } (\lambda x \to \llbracket e \rrbracket) \\
&\llbracket \texttt{Let rec } f(\tilde{x}) = e \texttt{ in } e' \rrbracket && = \texttt{let } f(\tilde{x}) = \llbracket e \rrbracket \texttt{ in } \llbracket e' \rrbracket \\
&\llbracket \texttt{Let Send } l^{\gamma}(x) \texttt{ in } e' \rrbracket && = \llbracket \gamma \downarrow l \rrbracket (\texttt{send } (\llbracket l \rrbracket\ x)\ \llbracket e' \rrbracket) \\
&\llbracket \texttt{Let Close () in } e' \rrbracket && = \texttt{close } (\llbracket e' \rrbracket) \\
&\llbracket l(x) \to \gamma' \downarrow l \rrbracket && = \texttt{id} \\
&\llbracket \eta_1, \eta_2 \downarrow l \rrbracket && = \texttt{left} \circ \llbracket \eta_1 \downarrow l \rrbracket \cup \texttt{right} \circ \llbracket \eta_2 \downarrow l \rrbracket
\end{aligned}
$$

Fig. 5. Translation of Expressions

Recursion Matching against the recursion operator requires unwinding. Its implementation requires two steps. Due to the definition of `REC`, unwinding boils down to selecting the argument of `REC`. The second step derives from the observation that the body of each `REC` constructor has type `G` $\tilde{\beta}$, which was introduced by the translation. Now, `G` $\tilde{\beta}$ needs to be expanded to its definition so that matching can proceed. To achieve this expansion uniformly requires (yet) another type class, say, `RECBODY`:

```
class RECBODY t c | t -> c where recbody :: t -> c
```

with instances provided for each of the `G` $\tilde{\beta}$. If the definition is

$$\texttt{data G send recv } \tilde{\beta} = \texttt{G}(\mathcal{T}\llbracket\gamma\rrbracket\tilde{\beta})$$

then `recbody` is the selector function of type `G send recv` $\tilde{\beta} \to \mathcal{T}\llbracket\gamma\rrbracket\tilde{\beta}$:

```
instance RECBODY (G send recv β̃) T⟦γ⟧β̃ where
  recbody (G x) = x
```

Hence, unwinding boils down to two selection operations. In this case, there is no conceptual difference between the send and receive operations.

```
instance (RECEIVE t c, RECBODY (f (REC f)) t) => RECEIVE (REC f) c where
  receive' g = Session (\ (REC fRECf) inp ->
                         unSession (receive' g) (recbody fRECf) inp)
instance (SEND t x y, RECBODY (f (REC f)) t) => SEND (REC f) x y where
  send mess cont = Session (\ (REC fRECf) inp ->
                             unSession (send mess cont) (recbody fRECf) inp)
```

4.3 Translation of Expressions

The translation of expressions is given by the table in Figure 5. It assumes that primitive operations (may) have side effects. When sending a message, the translation requires information about the current session type. This information is needed to inject the `send` operation into the corresponding "slot" in the translated session type. The formulation of this injection (the last two lines) is nondeterministic: it has to search for the matching label in the tree of alternatives. However, the result is deterministic because each sending label appears exactly once in the session type. It can be shown that a typed expression in the session calculus is mapped to a typed Haskell program.

Lemma 2. *Suppose that* $\emptyset\, , \gamma \vdash e$*. Then* $\Gamma \vdash_{haskell} [\![e]\!] : \texttt{Session}\ (\mathcal{T}[\![\gamma]\!])\ ()$*, where* Γ *is as in Lemma 1.*

5 Case Study: A simple SMTP client

An excerpt of a real world application, our type-safe implementation of the *Simple Mail Transfer Protocol* (SMTP) [30], demonstrates the practicability of our Haskell encoding of session types. Our implementation expects an email as input and tries to deliver it to a specific SMTP server. It is automatically derived from the protocol specification in terms of a session type.

5.1 A simplified SMTP session type

A session type corresponding to full SMTP is quite unreadable. Hence, we only consider a fragment of SMTP relevant to our application. The type is given from the client's point of view, flipping all decorations results in the server's type.

$$\begin{array}{l}
\gamma_C = [\,\overline{\texttt{220}} : \mu\beta_0.[\,\texttt{EHLO} : [\,\overline{\texttt{250}} : \mu\beta_1.[\,\texttt{MAIL} : [\,\overline{\texttt{250}} : [\,\texttt{RCPT} : \mu\beta_2.[\,\overline{\texttt{250}} : [\,\texttt{DATA} : [\,\overline{\texttt{3yz}} : (\texttt{LINES}, [\,\overline{\texttt{250}} : \beta_1,\\
\hspace{24em} \ldots]),\\
\hspace{20em} \ldots],\\
\hspace{18em} \texttt{RCPT} : \beta_2,\\
\hspace{18em} \ldots],\\
\hspace{16em} \ldots],\\
\hspace{13em} \ldots],\\
\hspace{11em} \ldots],\\
\hspace{9em} \texttt{QUIT} : \emptyset,\\
\hspace{9em} \ldots],\\
\hspace{6em} \overline{\texttt{5yz}} : \beta_0],\\
\hspace{4em} \ldots],\\
\hspace{1em} \ldots]
\end{array}$$

After receiving a greeting from the SMTP server (a $\overline{\texttt{222}}$ reply), the client initiates a mail session sending `EHLO` to perform several, consecutive email transactions with the server. A mail transactions starts with a `MAIL` command, followed by at least one `RCPT` command telling the server the recipients of the mail, by a `DATA` command announcing the mail content, and by the actual content (`LINES`). The client terminates the session issuing a `QUIT` command. The server usually acknowledges successful commands by sending a $\overline{\texttt{250}}$ reply. In the full protocol, different kinds of error replies can return after client commands, the server understands several administrative commands anytime during a session, and the client may always quit the session.

5.2 Haskell encoding of the SMTP session type

To implement the session specification in Haskell, we first translate the SMTP commands and replies to Haskell data types.

```
-- SMTP commands
data EHLO = EHLO Domain
instance Command EHLO where ...

cEHLO = EHLO undefined                 -- for the protocol specification

-- SMTP replies
data Reply220 = Reply220 Domain [String]
instance Command Reply220 where ...

r220 = Reply220 undefined undefined     -- for the protocol specification
```

The above declarations show the encoding of the first two messages occurring during an SMTP session, the 220 reply holding the Internet domain and additional information about the mail server, and the `EHLO` command, holding the Internet domain of the client. The `Domain` datatype represents either IP addresses or domain names. To make commands and replies amenable both to parsing and to printing, we make them instances of the `Command` type class introduced in Section 3. The remaining SMTP commands are implemented analogously.

The translation of the previously specified SMTP session type arises from applying the translation specified in Section 4.

```
smtpSpec
  (send:: (forall x y . x -> y -> s x y))
  (recv:: (forall x y . x -> y -> r x y)) =
  recv p220
    (let a0 = REC (Ga0
                    (send cEHLO
                     (ALT
                      (recv p250
                       (let a1 = REC (Ga1
                                      (ALT
                                       (send cMAIL
                                        (recv p250
                                         (send cRCPT
                                          (let a2 = REC (Ga2
                                                         (recv p250
                                                          (ALT
                                                           (send cDATA
                                                            (recv p354
                                                             (send cMESG
                                                              (recv p250 a1))))
                                                           (send cRCPT a2))))
                                           in a2))))
                                       (send cQUIT NULL)))
                        in a1))
                      (recv p5yz a0))))
         in a0)
```

For each occurrence of a $\mu\beta.\gamma$ in the session type, the translation introduces additional parameterized datatypes as explained in Section 4. Here, we only show the datatype declaration corresponding to β_0 and its instance declaration of the `RECBODY` type class.

```
data Ga0 send recv a0 =
  Ga0 (send EHLO
       (ALT (recv Reply2 (REC (Ga1 send recv a0))) (recv Reply5 a0)))
```

```
instance RECBODY
    (Ga0 send recv a0)
    (send EHLO (ALT (recv Reply2 (REC (Ga1 send recv a0))) (recv Reply5 a0)))
  where
  recbody (Ga0 x) = x
```

5.3 The SMTP client

With the specification of SMTP sessions in place, we now encode the main function of an email client, `sendMessage`. The functions adhere to our SMTP session specification which is statically guaranteed by the Haskell type system.

```
sendMessage :: Client -> Server ->
               ReversePath -> [ForwardPath] -> [String] -> IO ()
sendMessage client server sender rcpts message =
  withSocketsDo $ do
  h   <- connectTo (showDomain (name server)) (Service "smtp")
  str <- hGetContents h
  let recv220  = receive (\ (Reply220 server_domain text_220) -> sendEHLO)
      sendEHLO = send (EHLO (cname client)) recvEHLO
      recvEHLO = receive (ALT (\ (Reply2 y z text_250) -> sendMail)
                              (\ (Reply5 y z text_5yz) -> sendEHLO))
      sendMail = left (send (MAIL sender []) (recv250 (sendRCPT rcpts)))
      recv250   cont         = receive (\ (Reply2 y z text_250) -> cont)
      sendRCPT  (rcpt:rcpts) = send (RCPT rcpt []) (recvRCPT rcpts)
      recvRCPT  rcpts        = recv250 (sendRCPT' rcpts)
      sendRCPT' []           = left sendDATA
      sendRCPT' (rcpt:rcpts) = right (send (RCPT rcpt []) (recvRCPT rcpts))
      sendDATA = send DATA recv354
      recv354  = receive (\ (Reply3 y z text_354) -> sendMESG)
      sendMESG = send (LINES message) (recv250 sendQUIT)
      sendQUIT = right (send QUIT finish)
  runSession h recv220 (smtpSpec SEND_MSG RECV_MSG) str
  hClose h
```

The `sendMessage`function takes five arguments: `client` and `server` arguments hold information about the parties involved, `sender` and `rcpts` encode the email addresses of the sender and the recipients of the message, and `message` holds the message body itself.

After opening a socket connection to the SMTP server and after getting a handle to the socket to read from it lazily using `hGetContents`, we first specify the different interaction steps of the client-server communication using `send` and `receive`. The first step, `recv220`, receives the greeting message and transfers the control to the next step, `sendEHLO`. The `sendEHLO` step simply sends out the initial command introducing the client to the server. Its continuation, `recvHELLO`, shows how to handle a choice of two possibly incoming messages: instead of a function, we apply an `ALT` value holding two alternative handlers branching to two different continuations to `receive`. The functions `recvRCPT` and `sendRCPT'` are recursive functions handling the transmission of a list of recipients. They show how to send a message of two possible alternatives. To continue with the first alternative, we wrap `left` around the continuation `sendDATA`, otherwise, we apply `right` to the second alternative continuation. `Close` in the continuation to the `QUIT` command terminates the session.

The function `runSession` starts the client/server interaction taking `recv220` as entry point, `smtpSpec` as SMTP specification, and the socket both as input stream and output stream.

6 Related Work

The introduction already mentioned some related work on concurrency and session types. A particular system close to session types is Armstrong's UBF [2]. Also relevant to the present work are other applications of domain modeling using type systems.

There are a number of applications, ranging from general techniques to modeling DTD's [22, 18]. Also work on modeling type-safe casting [34], the encoding of type equality predicates [7, 3], and the representation of type-indexed values [35] is relevant.

A foundational work by Rhiger [31] considers typed encodings of the simply typed lambda calculus in Haskell and clarifies the necessary prerequisites for such an encoding to be sound and complete. An example encoding is developed by Danvy and others [8]. Later work by Chen and Xi [6] extends their approach to a meta-programming setting.

Another application area is modeling external concepts like relational schemes in databases for obtaining a type-safe query language [17], wrapping accesses to COM components [26], and integrating access to Java library [4].

None of the listed works specifies an encoding of recursion as we do in our translation neither do they exploit polymorphism as in our guarantee that client and server specifications match up.

7 Conclusion

A calculus with session types can be embedded into the programming language Haskell in a type-safe way. We give a detailed account of the translation and prove its type safety. Our case study demonstrates that the embedding is practical and exhibits the benefits of declarative programming. The resulting program is straightforward to read and understand.

It would be interesting to further investigate alternative designs for the `Session` monad. In particular, pushing the idea character-based parsing to the extreme appears to lead to a pure implementation on the basis of stream transformers. This implementation would completely decouple the processing of the protocol from the underlying IO actions.

References

1. Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency.* Imperial College Press, 1999.
2. Joe Armstrong. Getting erlang to talk to the outside world. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 64–72. ACM Press, 2002.
3. Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In Peyton-Jones [24], pages 157–166.
4. Nick Benton and Andrew Kennedy. Interlanguage working without tears: Blending SML with Java. In Peter Lee, editor, *Proc. International Conference on Functional Programming 1999*, pages 126–137, Paris, France, September 1999. ACM Press, New York.
5. Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96, 1992.
6. Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In Olin Shivers, editor, *Proc. International Conference on Functional Programming 2003*, pages 275–286, Uppsala, Sweden, August 2003. ACM Press, New York.
7. James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 90–104. ACM Press, 2002.
8. Olivier Danvy, Morten Rhiger, and Kristoffer Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.
9. Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In POPL 1996 [28], pages 372–385.
10. Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In Doaitse Swierstra, editor, *Proceedings of the 1999 European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 74–90, Amsterdam, The Netherlands, April 1999. Springer-Verlag.
11. Simon Gay, Vasco Vasconcelos, and Antonio Ravara. Session types for inter-process communication. Technical Report TR-2003-133, Department of Computing Science, University of Glasgow, 2003.
12. Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. FACILE: A symmetric integration of concurrent and functional programming. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, number 351,352 in Lecture Notes in Computer Science, pages II, 184–209, Barcelona, Spain, March 1989. Springer-Verlag.
13. Haskell 98, a non-strict, purely functional language. http://www.haskell.org/definition, December 1998.
14. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *5th European Conference on Object-Oriented Programming (ECOOP '91)*, number 512 in Lecture Notes in Computer Science, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag.
15. Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Proc. 7th European Symposium on Programming*, number 1381 in Lecture Notes in Computer Science, pages 122–138, Lisbon, Portugal, April 1998. Springer-Verlag.
16. Mark P. Jones. Type classes with functional dependencies. In Gert Smolka, editor, *Proc. 9th European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 230–244, Berlin, Germany, March 2000. Springer-Verlag.

17. Daan Leijen and Erik Meijer. Domain-specific embedded compilers. In *2nd Conference on Domain-Specific Languages*, Austin, Texas, USA, October 1999. USENIX. http://usenix.org/events/dsl99/index.html.
18. Conor McBride. Faking it—simulating dependent types in Haskell. http://www.dur.ac.uk/ dcs1ctm/faking.ps, 2001.
19. Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In Simon Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, pages 324–333, La Jolla, CA, June 1995. ACM Press, New York.
20. Robin Milner. *Communication and Concurrency.* Prentice Hall, Englewood Cliffs, NJ, 1989.
21. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I + II. *Information and Control*, 100(1):1–77, 1992.
22. Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. A functional notation for functional dependencies. In Ralf Hinze, editor, *Proceedings of the 2001 Haskell Workshop*, 2001. to appear.
23. Martin Odersky and Konstantin Läufer. Putting type annotations to work. In POPL 1996 [28], pages 54–67.
24. Simon Peyton-Jones, editor. *International Conference on Functional Programming*, Pittsburgh, PA, USA, October 2002. ACM Press, New York.
25. Simon Peyton Jones, Andrew Gordon, and Sigbjørn Finne. Concurrent Haskell. In POPL 1996 [28], pages 295–308.
26. Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components in Haskell. In *Proc. International Conference of Software Reuse*, 1998.
27. Simon L. Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.
28. *Proceedings of the 1996 ACM SIGPLAN Symposium on Principles of Programming Languages*, St. Petersburg, FL, USA, January 1996. ACM Press.
29. John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.
30. Simple mail transfer protocol. http://www.faqs.org/rfcs/rfc2821.html, April 2001.
31. Morten Rhiger. A foundation for embedded languages. *ACM Transactions on Programming Languages and Systems*, 25(3):291–315, 2003.
32. Alan Schmitt and Jean-Bernard Stefani. The m-calculus: a higher-order distributed process calculus. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 50–61. ACM Press, 2003.
33. Peter J. Stuckey and Martin Sulzmann. A theory of overloading. In Peyton-Jones [24], pages 167–178.
34. Stephanie Weirich. Type-safe cast: Functional pearl. In Philip Wadler, editor, *Proc. International Conference on Functional Programming 2000*, pages 58–67, Montreal, Canada, September 2000. ACM Press, New York.
35. Zhe Yang. Encoding types in ML-like languages. In Paul Hudak, editor, *Proc. International Conference on Functional Programming 1998*, pages 289–300, Baltimore, USA, September 1998. ACM Press, New York.