Session Types for Asynchronous Communication

Matthias Neubauer and Peter Thiemann

Universität Freiburg

Abstract. A session type is an abstraction of a sequence of heterogeneous values sent over one communication channel. Session types can be used for specifying stream-based Internet protocols.

We consider session types in a setting with asynchronous communication over buffered channels. Due to buffering, a process may write ahead before the reading process picks up the data. Thus the types of the two ends of a channel may differ.

We formalize the potential difference in types due to buffering, show that the ensuing relation on types is decidable, and prove the type soundness of the resulting type system with respect to a low-level operational semantics that models buffered communication channels accurately.

1 Introduction

Much foundational work on calculi for concurrency is devoted to studying synchronous, one-shot communications, for example, CCS [10], the π calculus [11], the chemical abstract machine [1], the join calculus [4], and the M-calculus [16]. 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 [8] or that incorporate channel-based communication primitives [7, 13, 15]. Once a channel has been created, many distinct messages may be communicated through it. Typically, channels are homogeneous, that is, all messages must have the same type.

Session types [5] have emerged as an expressive typing discipline for *heterogeneous* communication channels. In such a channel, each message may have a different type with the possible sequences of types determined by the channel's session type. Such a type discipline subsumes typings for one-shot communication as well as for homogeneous channels. Session types can describe stream-based Internet protocols such as SMTP, POP, IMAP, and many others [5].

Session types have been proposed to deal with client-server applications and synchronous communication [12, 17, 9, 5, 6]. A session type describes the sequencing of messages over a channel with a regular language on message types. Channels are first-class values that are created in pairs with complementary types, client and matching server, from a given port. Since a session type describes a sequence of messages, a channel changes its type with each operation. In consequence, channel values must be handled linearly by the type system.

Work on session types up to now has concentrated on either synchronous messaging [5] or has not established a formal connection between data sent and data received [12, 6]. The present work attempts to fill this gap by specifying a system of session types for buffered asynchronous communication channels. Honda et al [9] propose a superficially similar framework but specify a synchronous semantics. Their type system differs from ours in that they attach session types to processes (thus capturing the entire communication on all channels at once) whereas our session types are attached to single channels [6].

In our system, channels are created from a common port that prescribes a session type. However, due to buffering, the type of the client channel need not match up literally with the server channel's type. Instead of processing a message when it is received, a process may first perform a number of send operations. The only restriction is that incoming messages must be processed in FIFO order.

We specify the buffering relation as a relation on trace languages of the communication operations at both ends of a channel and approximate its restriction to regular languages (the trace languages of channel types) by a relation on finite automata (the channel types). This relation is shown to be decidable.

We prove type soundness for session types with respect to a low-level operational semantics using standard techniques [18]. The semantics models channels with bidirectional buffering.

Overview In Section 2, we give an example where asynchronous session types are useful. Then, we define the syntax (Section 3) and the type system for our language (Section 4). Section 5 defines the acceptance relation that specifies asynchrony. Section 6 is devoted to the operational semantics of the language and Section 7 provides typing rules for the configurations used in the semantics. In Section 8 we prove type soundness and Section 9 concludes.

2 Motivation

A typical application for heterongeneous channels is a crypto-server that might run on a smart card. The crypto-server maintains a database of service identifiers and private keys. It accepts connections from entities that seek to authorize the user of the smart card. Informally, the crypto-server's protocol might look as follows:

1. The client sends **AUTHORIZE** followed by the service name, a PIN, and a nonce. The server either answers **OK** followed by a number computed from the input and the associated private key, or it answers **FAILED**.
2. The client sends the **COMPROMISED** followed by a PIN. The server answers either **OK** or **FAILED**.
3. The client sends **QUIT** .

After processing a request, the server is ready to process the next command. Such a protocol can be specified with session types. We consider the words as fixed labels and associate the data types `String`, `PIN`, and `Integer` with the other messages. With these conventions, the corresponding session type (from the client's point of view) looks as follows.[1]

$$\begin{aligned}\gamma_C = \mu\beta.[\,&\textbf{AUTHORIZE} : (\texttt{String}, \texttt{PIN}, \texttt{Integer}, [\,\overline{\textbf{OK}} : (\overline{\texttt{Integer}}, \beta), \overline{\textbf{FAILED}} : \beta]),\\ &\textbf{COMPROMISED} : (\texttt{PIN}, [\,\overline{\textbf{OK}} : \beta, \overline{\textbf{FAILED}} : \beta]),\\ &\textbf{QUIT} : \varepsilon]\end{aligned}$$

In this type, overlined items correspond to data items received by the client, whereas undecorated items are messages sent to the server. The square brackets denote a choice governed by sending or receiving one of the labels to the left of the colon. The $\mu\beta$ operator is a fixpoint operator. At each point, where β is mentioned in the session type, the communication pattern repeats the body of the $\mu\beta$. Swapping all decorations results in a suitable type for the server.

Once the **COMPROMISED** message is received and acknowledged, the server never returns **OK** anymore. In fact, the server may ignore whatever parameters are sent and just answer **FAILED** right away, thus changing its type to $\mu\beta.(\textbf{FAILED}, \beta)$. Unfortunately, this type of server will lead to a buffer overflow due to data items that are received but not processed. Hence, the server must eventually consume every item sent to it by the client, as expressed with the following type. It returns the answer as quickly as possible but *afterwards* receives the right number of data items to clear the buffer. (We omit the rest of the **OK** responses for readability.)

$$\begin{aligned}\gamma_S = \mu\beta.[\,&\overline{\textbf{AUTHORIZE}} : [\textbf{FAILED} : (\overline{\texttt{String}}, \overline{\texttt{PIN}}, \overline{\texttt{Integer}}, \beta), \textbf{OK} : \ldots],\\ &\overline{\textbf{COMPROMISED}} : [\textbf{FAILED} : (\overline{\texttt{PIN}}, \beta), \textbf{OK} : \ldots],\\ &\overline{\textbf{QUIT}} : \varepsilon]\end{aligned}$$

Our type system allows send operations to both ends of a channel "ahead of time", that is, without forcing a redenzvous with a receiving operation at the other end of the channel. It does so by defining a buffering relation $\preceq$ such that $\overline{\gamma_S} \preceq \gamma_C$ for the above example.

3 Syntax

We rely on standard notation for operations on sets. If A is a set, then $\mathcal{P}(A)$ is its power set. For a partial mapping $f : A \to B$ and $X \subseteq A$, the mapping $A \downarrow X$ is the restriction of f to X. The notation $f \setminus X$ denotes the corestriction of f with respect to X, that is, the elements of X are removed from f's domain. The notation $\tilde{x}$ denotes the sequence $x_1, \ldots, x_n$, where n depends on the context.

The syntax is given in an evaluation-order independent, linearized form. A channel may transfer either a label l or a first-order value bv taken from some base type. Expressions and definitions (see Fig. 1) make use of two kinds of variables, ordinary variables ranged over by $x, \ldots$ and linear variables ranged over by $s, \ldots$. An expression is either `Halt` for stopping evaluation, introduction of a definition, conditional, function call, reception of a label, or interleaved execution of two expressions. A definition is either a primitive operation on base-type values, the formation of a recursive function, creation of a fresh port,

[1] Our syntax resembles closely that in related work on session types[5].

$l \in$ Label
Definitions
$d ::= x = \mathtt{Op}(\tilde{x}) \mid \mathtt{rec}\ \tilde{x}[\tilde{s}](x) = e$
$\quad \mid\ x = \mathtt{NewPort} \mid s = \mathtt{Connect}(x) \mid s = \mathtt{Listen}(x)$
$\quad \mid\ \mathtt{Send}\ s(x) \mid x = \mathtt{Receive}(s) \mid \mathtt{Close}\ (s)$
Expressions
$e ::= \mathtt{Halt} \mid \mathtt{Let}\ d\ \mathtt{in}\ e \mid \mathtt{If}\ x\ \mathtt{then}\ e\ \mathtt{else}\ e \mid x[\tilde{s}]\ \tilde{x}$
$\quad \mid\ \mathtt{RecCase}\ x\ \mathtt{of}\ [l_i \to e_i] \mid e \parallel e$

Types
$\tau ::= b \mid \pi \mid [\tilde{\gamma}]\tilde{\tau} \to 0$
Port types
$\pi ::= \mathtt{Port}\ \gamma$
Session types
$\gamma ::= \varepsilon \mid (\iota, \gamma) \mid [\ell_i : \gamma_i] \mid \beta \mid \mu\beta.\gamma$
$\iota ::= b \mid \overline{b}$
$\ell ::= l \mid \overline{l}$

Fig. 1. Syntax

connecting a client to a port (binding to a channel opened by some server), connecting a server to a port (thus creating a channel), sending a data item or a label, receiving a data item, and closing a channel. Functions take two kinds of arguments, linear channel values and standard values.

The syntax of types has three layers. The type of a value is either a base type, a port type, or a function type. Due to the linearized syntax, functions invoke a continuation to deliver their results. They do not have a return type. A port type, π, describes a communication port and has the port's protocol attached to it. A session type, γ, can specify either no communication, a base type communication followed by further interaction, communication of a label followed by further interaction depending on the label, or a recursive behavior. Transport items, ι, indicate the direction of the communication. It may be outbound, as in b, or inbound, as in $\overline{b}$. The meaning of label items, ℓ, is defined analogously.

The use of the recursion operator is restricted to expansive session types, as usual. A session type is expansive if it does not contain subterms of the form $\mu\beta_1 \dots \mu\beta_n.\beta_1$. Thus, each type corresponds to a deterministic finite automaton on the alphabet $\Sigma \cup \overline{\Sigma}$ where $\Sigma = \{b, \dots, l, \dots\}$, b ranges over the base types and $l \in$ Label, and $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$. The operator $\overline{\gamma}$ swaps all inbound and outbound annotations in γ, converting the view on the type from server to client.

4 Static semantics

The static semantics is specified with two judgments. The judgment for expressions, $\Gamma|\Theta \vdash e$, relates a typing environment, Γ, a linear typing environment, Θ, and an expression, e. It checks the use of variables for consistency, since linearized expressions do not compute values. The judgment for definitions, $\Gamma|\Theta \vdash d \Rightarrow \Gamma'|\Theta'$, specifies the transformation that d performs on a pair of typing environments. Typing environments are defined as usual with a binary $+$ operator denoting disjoint union.

Figure 2 defines the typing rules for expressions. The rules are fairly standard, but there are some fine points regarding the use of the linear type environment. `Halt` requires that all linear variables (channel) have been used up. `Let` and the conditional are standard. A function call requires that all remaining linear variables are passed to the function. `RecCase` receives a label and branches to one of the expressions according to the label. Receiving a value changes the current

$$TEnv \ni \Gamma ::= \emptyset \mid \Gamma, x : \tau \qquad STEnv \ni \Theta ::= \emptyset \mid \Theta, s : \gamma$$

$$\Gamma|\emptyset \vdash \mathtt{Halt} \qquad \frac{\Gamma|\Theta \vdash d \Rightarrow \Gamma'|\Theta' \quad \Gamma'|\Theta' \vdash e}{\Gamma|\Theta \vdash \mathtt{Let}\ d\ \mathtt{in}\ e}$$

$$\frac{\Gamma(x) : b \quad \Gamma|\Theta \vdash e_1 \quad \Gamma|\Theta \vdash e_2}{\Gamma|\Theta \vdash \mathtt{If}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2} \qquad \frac{\Gamma(x) = [\tilde{\gamma}]\tilde{\tau} \to 0 \quad \Gamma(\tilde{z}) = \tilde{\tau} \quad \Theta = \tilde{y} : \tilde{\gamma}}{\Gamma|\Theta \vdash x[\tilde{y}]\ \tilde{z}}$$

$$\frac{\Theta = \Theta' + [s : [\overline{l_i} : \gamma_i]_{i=1}^n] \quad \Gamma|\Theta', s : \gamma_i \vdash e_i}{\Gamma|\Theta \vdash \mathtt{RecCase}\ s\ \mathtt{of}\ [l_i \to e_i]} \qquad \frac{\Theta = \Theta_1 + \Theta_2 \quad \Gamma|\Theta_1 \vdash e_1 \quad \Gamma|\Theta_2 \vdash e_2}{\Gamma|\Theta \vdash e_1 \parallel e_2}$$

Fig. 2. Typing rules for expressions

$$\frac{\Gamma(x_i) = b}{\Gamma|\Theta \vdash x = \mathtt{Op}(x_1, \dots, x_n) \Rightarrow \Gamma, x : b|\Theta} \qquad \frac{\Gamma, f : [\tilde{\gamma}]\tilde{\tau} \to 0, \tilde{x} : \tilde{\tau}|\tilde{s} : \tilde{\gamma} \vdash e}{\Gamma|\Theta \vdash \mathtt{rec}\ f[\tilde{s}](\tilde{x}) = e \Rightarrow \Gamma, f : [\tilde{\gamma}]\tilde{\tau} \to 0|\Theta}$$

$$\Gamma|\Theta \vdash x = \mathtt{NewPort} \Rightarrow \Gamma, x : \mathtt{Port}\ \gamma|\Theta \qquad \frac{\Theta \vdash s : \varepsilon \Rightarrow \Theta'}{\Gamma|\Theta \vdash \mathtt{Close}\ (s) \Rightarrow \Gamma|\Theta'}$$

$$\frac{s \notin dom(\Theta) \quad \Gamma(x) = \mathtt{Port}\ \gamma_0 \quad \gamma \preceq \gamma_0}{\Gamma|\Theta \vdash s = \mathtt{Connect}(x) \Rightarrow \Gamma|\Theta, s : \overline{\gamma}} \qquad \frac{s \notin dom(\Theta) \quad \Gamma(x) = \mathtt{Port}\ \gamma_0 \quad \gamma_0 \preceq \gamma}{\Gamma|\Theta \vdash s = \mathtt{Listen}(x) \Rightarrow \Gamma|\Theta, s : \gamma}$$

$$\frac{\Theta = \Theta' + [s : (b, \gamma)] \quad \Gamma(x) = b}{\Gamma|\Theta \vdash \mathtt{Send}\ s(x) \Rightarrow \Gamma|\Theta', s : \gamma} \qquad \frac{\Theta = \Theta' + [s : [l_i : \gamma_i]_{i=1}^n]}{\Gamma|\Theta \vdash \mathtt{Send}\ s(l_j) \Rightarrow \Gamma|\Theta', s : \gamma_j}\ 1 \le j \le n$$

$$\frac{\Theta = \Theta' + [s : (\overline{b}, \gamma)]}{\Gamma|\Theta \vdash x = \mathtt{Receive}(s) \Rightarrow \Gamma, x : b|\Theta', s : \gamma}$$

Fig. 3. Typing rules for definitions

state s of the channel according to the branch taken. Concurrent execution of two expressions splits the linear typing environment in two disjoint parts.

Figure 3 contains the rules for definitions. Primitive operations require base type values as arguments and yield base type results. A recursive function cannot refer to channels defined in the environment. Otherwise, it would be much harder to guarantee that channels are only used linearly. `NewPort` generates a fresh port name with an arbitrary session type. `Connect` creates the client end of a channel. It assigns the channel the complement of a type acceptable to the port's session type. `Listen` creates the server end of a channel.

The two `Send` operations differ mainly in their typing. Sending a base type value strips an outbound base type message from the session type, whereas sending a label selects a branch from an alternative type. `Receive` strips an inbound message from the session type. `Close` removes an empty channel from the environment.

The typing rules for creating channels rely on the acceptance relation $\preceq$. This relation specifies our notion of asynchronous communication and is one of the key contributions of the paper. It is defined in the next section.

5 Asynchronous acceptance

The definition of the acceptance relation $\preceq$ is based on a trace semantics. We lift the relation from words to languages, and finally we define a syntactic relation on types that abstracts from the correponding notion on languages.

Definition 1. *The relation* $\preceq$ *on* $(\Sigma \cup \overline{\Sigma})^*$ *is the least relation such that*

$$\varepsilon \preceq \varepsilon \qquad \frac{w \preceq w' \qquad a \in \Sigma}{aw \preceq aw'} \qquad \frac{w \preceq v'w' \qquad a \in \Sigma \qquad v' \in \Sigma^*}{\overline{a}w \preceq v'\overline{a}w'}$$

The intuition behind $w \preceq w'$ is that w is the *swapped* trace of a client and w' is the trace of a server. If both traces are empty, then client and server are compatible. If the client wants to receive an a, then the server should send an a immediately. If the client wants to send an a, the server need not receive it immediately, instead it can send some $v' \in \Sigma^*$ first. However, the server cannot attempt to receive some other message first. The same holds with roles exchanged as part two of the following lemma shows.

Lemma 1. *1. The relation* $\preceq$ *is a partial ordering on* $(\Sigma \cup \overline{\Sigma})^*$*.*
2. $w \preceq w'$ *iff* $\overline{w'} \preceq \overline{w}$*.*

Definition 2. *The relation* $\preceq$ *extends to languages* $L, L' \subseteq (\Sigma \cup \overline{\Sigma})^*$ *by*

$$L \preceq L' \quad \text{iff} \quad ((\forall w \in L)\ (\exists w' \in L')\ w \preceq w') \wedge ((\forall w' \in L')\ (\exists w \in L)\ w \preceq w')$$

Define the *trace language* $L(\gamma) = L_{\{\}}(\gamma) \subseteq (\Sigma \cup \overline{\Sigma})^*$ of a session type γ by

$$\begin{array}{llll} L_\varsigma(\varepsilon) &= \{\varepsilon\} & L_\varsigma([\ell_i : \gamma_i]_{i=1}^n) &= \bigcup_{i=1}^n L_\varsigma(\gamma_i) \\ L_\varsigma(\iota, \gamma) &= \iota . L_\varsigma(\gamma) & L_\varsigma(\mu\beta.\gamma) &= \mu X. L_{\varsigma[\beta \mapsto X]}(\gamma) \end{array}$$

where $\mu X \ldots$ is the least fixpoint operator. This language is always regular.

Next, we approximate $\preceq$ by a syntactic relation on types. The latter induces an algorithm, $\Delta \vdash \gamma \preceq \gamma'$, to check if γ and γ' are related. The set Δ contains assumptions of the form $\beta \preceq \gamma''$. For the definition, the language of types is extended with the empty type $\emptyset$ where $L(\emptyset) = \emptyset$.

$$\Delta \vdash \emptyset \preceq \emptyset \qquad \frac{\Delta \vdash \gamma \preceq \gamma'' \qquad \gamma' \approx (b, \gamma'')}{\Delta \vdash (b, \gamma) \preceq \gamma'} \qquad \frac{\Delta \vdash \gamma \preceq \overline{b} \setminus \gamma'}{\Delta \vdash (\overline{b}, \gamma) \preceq \gamma'}$$

$$\Delta \vdash \varepsilon \preceq \varepsilon \qquad \frac{(\forall i)\ \Delta \vdash \gamma_i \preceq \gamma_i' \qquad \gamma' \approx [l_i : \gamma_i']}{\Delta \vdash [l_i : \gamma_i] \preceq \gamma'} \qquad \frac{(\forall i)\ \Delta\gamma_i \preceq \overline{l_i} \setminus \gamma'}{\Delta \vdash [\overline{l_i} : \gamma_i] \preceq \gamma'}$$

$$\frac{\Delta, \beta \preceq \gamma' \vdash \gamma \preceq \gamma'}{\Delta \vdash \mu\beta.\gamma \preceq \gamma'} \qquad \frac{\beta \preceq \gamma' \in \Delta \qquad \gamma \equiv \gamma'}{\Delta \vdash \beta \preceq \gamma}$$

The relation $\equiv$ is equivalence of type expressions (*i.e.*, finite automata). The relation $\approx$ unrolls recursion **at the top-level**. It is the least reflexive and transitive relation such that $\mu\beta.\gamma \approx \gamma[\beta \mapsto \mu\beta.\gamma]$. The functions $\overline{b} \setminus \gamma$ and $\overline{l} \setminus \gamma$ are defined in Fig. 4. They are closely related to derivatives of regular expressions[2].

Lemma 2. *There is an algorithm that decides the judgment* $\Delta \vdash \gamma \preceq \gamma'$*.*

Proof. A suitable termination ordering is the lexicographic ordering on pairs of 1. the size of γ and 2. the nesting depth of $\mu\beta$'s at the toplevel of γ' (that is, n if $\gamma' = \mu\beta_1 \ldots \mu\beta_n.\gamma''$ and γ'' does not start with μ). All cases involve only terminating functions; with $\equiv$ being equivalence of finite automata. Top-level unrolling $\approx$ terminates since γ' is expansive.

The above algorithm is sound.

Lemma 3. *If* $\emptyset \vdash \gamma \preceq \gamma'$ *then* $L(\gamma) \preceq L(\gamma')$*.*

$$
\begin{array}{ll}
\overline{b} \setminus \emptyset = \emptyset & \overline{l} \setminus \emptyset = \emptyset \\
\overline{b} \setminus \varepsilon = \emptyset & \overline{l} \setminus \varepsilon = \emptyset \\
\overline{b} \setminus (b', \gamma) = (b', \overline{b} \setminus \gamma) & \overline{l} \setminus (b', \gamma) = (b', \overline{l} \setminus \gamma) \\
\overline{b} \setminus (\overline{c}, \gamma) = \begin{cases} \gamma & b = c \\ \emptyset & b \neq c \end{cases} & \overline{l} \setminus (\overline{b}, \gamma) = \emptyset \\
& \overline{l} \setminus [l_i : \gamma_i] = [l_i : \overline{l} \setminus \gamma_i] \\
\overline{b} \setminus [l_i : \gamma_i] = [l_i : \overline{b} \setminus \gamma_i] & \overline{l} \setminus [\overline{l_i} : \gamma_i] = \begin{cases} \emptyset & l \notin \{l_i\} \\ \gamma_j & l = l_j \end{cases} \\
\overline{b} \setminus [\overline{l_i} : \gamma_i] = \emptyset & \\
\overline{b} \setminus \mu\beta.\gamma = (\overline{b} \setminus \gamma)[\beta \mapsto \mu\beta.\gamma] & \overline{l} \setminus \mu\beta.\gamma = (\overline{l} \setminus \gamma)[\beta \mapsto \mu\beta.\gamma] \\
\overline{b} \setminus \beta = \emptyset & \overline{l} \setminus \beta = \emptyset
\end{array}
$$

Fig. 4. Generalized derivative

$$
\begin{array}{ll}
\mathit{PortName} & \text{unspecified set} \\
\mathit{StreamName} & \text{unspecified set} \\
\mathit{ThreadName} & \text{unspecified set} \\
\mathit{StreamStatus} & = \{\mathit{Open}, \mathit{Closed}\} \\
\mathit{Role} & = \{\mathit{Client}, \mathit{Server}\} \\
\mathit{StreamItem} & = \mathit{BaseValue} + \mathit{Label} \\
\mathit{StreamValue} & = \mathit{StreamName} \times \mathit{Role} \\
\mathit{Env} & = \mathit{Var} \rightarrow \mathit{Value} \\
\mathit{StreamEnv} & = \mathit{Var} \rightarrow \mathit{StreamValue} \\
\mathit{Closure} & = \mathit{Env} \times \mathit{FunDef} \\
\mathit{Value} & = \mathit{BaseValue} + \mathit{Closure} + \mathit{PortName} \\
\mathit{State} & = \mathcal{P}(\mathit{PortName}) \times \mathit{StreamStore} \times \mathit{Thread} \\
\mathit{Thread} & = \mathit{Env} \times \mathit{StreamEnv} \times \mathit{Exp} \\
\mathit{StreamStore} & = \mathit{StreamValue} \rightarrow \mathit{PortName} \times \mathit{StreamStatus} \times \mathit{StreamItem}^* \\
\mathit{FullState} & = \mathcal{P}(\mathit{ThreadName}) \times \mathcal{P}(\mathit{PortName}) \times \mathit{StreamStore} \times (\mathit{ThreadName} \rightarrow \mathit{Thread})
\end{array}
$$

$\mathit{rev} : \mathit{Role} \rightarrow \mathit{Role}$ swaps roles: $\mathit{rev}(\mathit{Client}) = \mathit{Server}$ and $\mathit{rev}(\mathit{Server}) = \mathit{Client}$.

Fig. 5. Semantic values and auxiliary functions

6 Dynamic semantics

The dynamic semantics is defined by a small-step transition system. Figure 5 defines the components of the system's state. A *FullState* describes the entire state of the system. The main components are *StreamStore*, a mapping from stream names and roles to the current state of the stream, and a mapping from thread names to a thread description. Client and server communicate via a pair of streams that share a single stream name with the two directions distinguished by the role. Each peer writes to its own end of the stream and reads from the head of the other peer's stream. A thread consists of a standard environment, a linear environment, and an expression.

Figure 6 specifies the evaluation rules on $(T, P, Q, R) \in \mathit{FullState}$. The first rule specifies spawning of a thread, the second performs one evaluation step inside a thread, and the third rule removes a thread that has reached a `Halt`.

Figure 7 specifies the rules for evaluating expressions. They relate two states of the form P, Q, ρ, σ, e. Evaluation of let expressions (evaluation of definitions is given separately), conditionals, and function calls is standard. `RecCase` requires that the input component of the stream has a label l ready for reading. Depending on the actual label, execution continues with one of the branches e_j.

The evaluation rules for definitions in Figure 8 relate a state fragment and a definition to a new state fragment. The rules for primitive operations and

$$
\begin{array}{l}
T, P, Q, R \longrightarrow \\
\quad T \cup \{t'\}, P, Q, R[t \mapsto (\rho_{\downarrow X}, \sigma_{\downarrow X}, e_1), t' \mapsto (\rho_{\downarrow Y}, \sigma_{\downarrow Y}, e_2)] \\
\text{if } t \in T \wedge R(t) = (\rho, \sigma, e_1 \parallel e_2) \wedge t' \notin T \wedge X = fv(e_1) \wedge Y = fv(e_2) \\
T, P, Q, R \longrightarrow \\
\quad T, P', Q', R[t \mapsto (\rho', \sigma', e')] \\
\qquad \text{if } t \in T \wedge R(t) = (\rho, \sigma, e) \wedge P, Q, \rho, \sigma, e \longrightarrow P', Q', \rho', \sigma', e' \\
T, P, Q, R \longrightarrow \\
\quad T \setminus \{t\}, P, Q, R \setminus \{t\} \qquad\qquad \text{if } t \in T \wedge R(t) = (\rho, \sigma, \mathtt{Halt})
\end{array}
$$

Fig. 6. Evaluation rules for full state

$$
\begin{array}{l}
P, Q, \rho, \sigma, \mathtt{Let}\ d\ \mathtt{in}\ e \longrightarrow \\
\quad P', Q', \rho', \sigma', e \qquad\qquad \text{if } P, Q, \rho, \sigma, d \longrightarrow P', Q', \rho', \sigma' \\
P, Q, \rho, \sigma, \mathtt{If}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \longrightarrow \\
\quad P, Q, \rho, \sigma, e_i \qquad \text{if } (\rho(x) = 0 \wedge i = 2) \vee (\rho(x) \neq 0 \wedge i = 1) \\
P, Q, \rho, \sigma, z[\tilde{p}]\ \tilde{w} \longrightarrow \\
\quad P, Q, \rho'[\tilde{x} \mapsto \rho(\tilde{w})][f \mapsto \rho(z)], \sigma[\tilde{y} \mapsto \rho(\tilde{p})], e \\
\qquad\qquad\qquad \text{if } \rho(z) = (\rho', \mathtt{rec}\ f[\tilde{y}](\tilde{x}) = e) \\
P, Q, \rho, \sigma, \mathtt{RecCase}\ x\ \mathtt{of}\ [l_i \to e_i] \longrightarrow \\
\quad P, Q[(sid, rev(r)) = (p, st, w)], \rho, \sigma', e_j \\
\text{if } \sigma(s) = (sid, r) \wedge Q(sid, rev(r)) = (p, st, l_j.w) \wedge Q(sid, r) = (p, Open, w') \\
\qquad\qquad\qquad \wedge\ \sigma' = \sigma[s \mapsto (sid, r)]
\end{array}
$$

Fig. 7. Evaluation rules for expressions

recursive functions are standard. `Connect` and `Listen` are asymmetric in that a client can only connect to a stream that has already been created by a server. This asymmetry could be removed at the expense of additional evaluation rules without yielding new insights. `Close` requires that no messages are waiting at the other end of the channel. However, the other peer may still have messages waiting. `Send` requires that the stream is open; `Receive` requires additionally that there is a message waiting for pickup. `Send`ing of a label performs the same action as the other `Send` operation, only its action on the types is different.

7 Typing rules for states

Working towards a type soundness proof, we need to specify when an evaluation state is type consistent. This specification is via a type system, again.

The typing rules for evaluation states in Figure 9 rely on

- $\Pi ::= \emptyset \mid \Pi, p : \pi,\ p \in PortName$, a global port environment and
- $\Psi ::= \emptyset \mid \Psi, (sid, r) : G|\gamma$ a global stream environment

where G ranges over type contexts given by $G ::= [\,] \mid (b, G) \mid [l : G, l_i : \gamma_i]$.

The majority of the rules just move information around and check that values match their specified types. We omit their description due to lack of space.

The most interesting rules concern the consistency judgment for streams. We consider only the second, which deals with a stream that is connected to a client and a server, the first works similar. Ψ contains the typing for the current state of both ends of the stream. Such a typing consists of two parts, a type

$$
\begin{array}{ll}
P, Q, \rho, \sigma, x = \mathtt{Op}(x_1, \ldots, x_n) \longrightarrow & \\
\quad P, Q, \rho[x \mapsto \mathtt{Op}(v_1, \ldots v_n)], \sigma & \text{if } (\forall 1 \le i \le n)\ \rho(x_i) = v_i \\
P, Q, \rho, \sigma, \mathtt{rec}\ f[\tilde{y}](\tilde{x}) = e \longrightarrow & \\
\quad P, Q, \rho[f \mapsto (\rho_{\downarrow X}, \mathtt{rec}\ f[\tilde{y}](\tilde{x}) = e)], \sigma & \text{if } X = fv(\mathtt{rec}\ f[\tilde{y}](\tilde{x}) = e) \\
P, Q, \rho, \sigma, x = \mathtt{NewPort} \longrightarrow & \\
\quad P \cup \{p\}, Q, \rho[x \mapsto p], \sigma & \text{if } p \notin P \\
P, Q, \rho, \sigma, s = \mathtt{Connect}(x) \longrightarrow & \\
\quad P, Q[(sid, Client) \mapsto (port, Open, \varepsilon)], \rho, \sigma[s \mapsto (sid, Client)] & \\
\quad\quad \text{if } \rho(x) = port \wedge Q(sid, Server) = (port, st, w) \wedge (sid, Client) \notin dom(Q) & \\
P, Q, \rho, \sigma, s = \mathtt{Listen}(x) \longrightarrow & \\
\quad P, Q[(sid, Server) \mapsto (port, Open, \varepsilon)], \rho, \sigma[s \mapsto (sid, Server)] & \\
\quad\quad \text{if } \rho(x) = port \wedge (sid, Server) \notin dom(Q) & \\
P, Q, \rho, \sigma, \mathtt{Close}\ (s) \longrightarrow & \\
\quad P, Q[(sid, r) \mapsto (p, Closed, w)], \rho, \sigma \setminus \{s\} & \\
\quad\quad \text{if } \sigma(s) = (sid, r) \wedge Q(sid, r) = (p, st, w) \wedge Q(sid, rev(r)) = (p, st', \varepsilon) & \\
P, Q, \rho, \sigma, \mathtt{Send}\ s(x) \longrightarrow & \\
\quad P, Q[(sid, r) \mapsto (p, Open, w.\rho(x))], \rho, \sigma & \\
\quad\quad \text{if } \sigma(s) = (sid, r) \wedge Q(sid, r) = (p, Open, w) & \\
P, Q, \rho, \sigma, x = \mathtt{Receive}(s) \longrightarrow & \\
\quad P, Q[(sid, rev(r)) = (p, st, w)], \rho[x \mapsto bv], \sigma & \\
\text{if } \sigma(s) = (sid, r) \wedge Q(sid, rev(r)) = (p, st, bv.w) \wedge Q(sid, r) = (p, Open, w') & \\
P, Q, \rho, \sigma, \mathtt{Send}\ s(l) \longrightarrow & \\
\quad P, Q[(sid, r) = (p, Open, w.l)], \rho, \sigma & \\
\quad\quad \text{if } \sigma(s) = (sid, r) \wedge Q(sid, r) = (p, Open, w) &
\end{array}
$$

Fig. 8. Evaluation of definitions

context G, which gives the type of the symbols in the buffered word, and a type γ, which describes the future interaction on the channel. The main point is that the client end and the server end must be consistent: the current type of either stream *prefixed with the type context* must be acceptable to the other: $\overline{G'[\gamma']} \preceq G[\gamma]$, where client types are swapped as usual.

8 Type Soundness

We establish type soundness in the standard way by proving type preservation and a progress result [18]. Type preservation states that an evaluation state in a consistent state remains consistent under the evaluation relation $\longrightarrow$. The second result, progress, states that a consistent state either is stuck state or it can be further reduced. The progress result is limited due to potential blocking of `Receive`, `RecCase`, and `Connect`. Taken together, the two results imply type soundness. We just state the main results here. Full proofs may be found in the Appendix of the companion technical report.

The following propositon states type preservation: if a consistent full state can be reduced, its reduct is also consistent.

Proposition 1 (Type Preservation). *Suppose $\Pi, \Psi \vdash T, P, Q, R$. If $T, P, Q, R \longrightarrow T', P', Q', R'$ then there exists Π' and Ψ‘ such that $\Pi', \Psi' \vdash T', P', Q', R'$.*

Even a consistent full state may not contain a reducible thread. The judgements specifying consistency of full states assure that two client-server pairs

Consistency of a full state

$$\frac{\begin{array}{c} dom(R) = T \qquad dom(\Pi) = P \qquad \Pi, \Psi \vdash Q \\ (\forall q \in dom(Q))\ Q(q) = (p, st, w) \wedge st \neq Closed \Rightarrow (\exists t \in T)\ (\rho_t, \sigma_t, e_t) = R(t) \quad q \in ran(\rho_t) \\ (\forall t \in T)\ (\Gamma_t, \Theta_t, \rho_t, \sigma_t, e_t) = R(t) \quad \Pi, \Psi \vdash \rho_t, \sigma_t, e_t \quad (\forall q \in ran(\sigma_t))\ Q(q) = (p, Open, w) \end{array}}{\Pi, \Psi \vdash T, P, Q, R}$$

Consistency of a state

$$\frac{\Psi, \Theta \vdash \sigma \qquad \Pi, \Gamma \vdash \rho \qquad \Gamma | \Theta \vdash e}{\Pi, \Psi \vdash \rho, \sigma, e}$$

Consistency of environments

$$\frac{dom(\rho) = dom(\Gamma) \quad (\forall x)\ \Pi \vdash \rho(x) : \Gamma(x)}{\Pi, \Gamma \vdash \rho} \qquad \frac{dom(\sigma) = dom(\Theta) \quad (\forall s)\ \Psi \vdash \sigma(s) : \Theta(s)}{\Psi, \Theta \vdash \sigma}$$

Typing of stream values and values

$$\frac{\Psi(sid, r) = G|\gamma}{\Psi \vdash (sid, r) : \gamma} \qquad \frac{}{\Pi \vdash bv : b} \qquad \frac{\Pi(p) = \texttt{Port}\ \gamma}{\Pi \vdash p : \texttt{Port}\ \gamma}$$

$$\frac{\Gamma', f : [\tilde{\gamma}]\tilde{\tau} \to 0, \tilde{x} : \tilde{\tau} | [\tilde{y} : \tilde{\gamma}] \vdash e' \qquad \Pi, \Gamma' \vdash \rho'}{\Pi \vdash (\rho', \texttt{rec}\ f[\tilde{y} : \tilde{\gamma}](\tilde{x} : \tilde{\tau}) = e') : [\tilde{\gamma}]\tilde{\tau} \to 0}$$

Consistency of streams

$$\frac{dom(\Psi) = dom(Q) \qquad (\forall (sid, r) \in dom(Q))\ \Pi, \Psi \vdash Q, sid}{\Pi, \Psi \vdash Q}$$

$$\frac{\begin{array}{c} (sid, Client) \notin dom(Q) \qquad (sid, Server) \in dom(Q) \\ Q(sid, Server) = (port, st, w) \\ \Psi(sid, Server) = G|\gamma \qquad \vdash w : G \qquad \overline{\Pi(port)} \preceq G[\gamma] \end{array}}{\Pi, \Psi \vdash Q, sid}$$

$$\frac{\begin{array}{c} \{(sid, Client), (sid, Server)\} \subseteq dom(Q) \\ Q(sid, Server) = (port, st, w) \qquad Q(sid, Client) = (port, st', w') \\ \Psi(sid, Server) = G|\gamma \qquad \Psi(sid, Client) = G'|\gamma' \\ \vdash w : G \qquad \vdash w' : G' \qquad \overline{G'[\gamma']} \preceq G[\gamma] \end{array}}{\Pi, \Psi \vdash Q, sid}$$

Consistency of buffers

$$\vdash \varepsilon : [\,] \qquad \frac{\vdash w : G}{\vdash bv.w : (b, G)} \qquad \frac{\vdash w : G}{\vdash l.w : [l : G, l_i : \gamma_i]}$$

Fig. 9. Typing rules for evaluation states

belonging to a stream are processed consistently. However, there is no global property preventing mutual deadlocks of a common pool of threads. The rules in Figure 10 specify when a full state is blocked—that is, when evaluation gets stuck. For that to happen, each thread of the full state must be blocked. A thread blocks, if either the thread wants to receive a value and the sending buffer is still emty, or the thread wants to connect, but there is now suitable server available.

The progress property states that a consistent full state is either blocked or can be further reduced.

Proposition 2 (Progress). *If $\Pi, \Psi \vdash T, P, Q, R$ then either*

(i) $\vdash_{bl} T, P, Q, R$ *or*
(ii) there exists T', P', Q', R' *such that* $T, P, Q, R \longrightarrow T', P', Q', R'$.

Blocked full state

$$\frac{(\forall t \in T)\ (\Gamma_t, \Theta_t, \rho_t, \sigma_t, e_t) = R(t) \quad \vdash_{bl} P, Q, \Gamma_t, \Theta_t, \rho_t, \sigma_t, e_t}{\vdash_{bl} T, P, Q, R}$$

Blocked expressions

$$\frac{\vdash_{bl} P, Q, \Gamma_t, \Theta_t, \rho_t, \sigma_t, d}{\vdash_{bl} P, Q, \Gamma_t, \Theta_t, \rho_t, \sigma_t, \texttt{Let}\ d\ \texttt{in}\ e} \qquad \frac{\sigma(s) = (sid, role) \quad Q(sid, rev(role)) = (p, st, \varepsilon)}{\vdash_{bl} P, Q, \Gamma_t, \Theta_t, \rho_t, \sigma_t, \texttt{RecCase}\ x\ \texttt{of}\ [\texttt{In}_{l_i}\ s_i \to e_i]}$$

Blocked definition

$$\frac{\rho(x) = port \quad (\forall (sid, Server) \in Q)\ (Q(sid, Server) = (port, st, w) \Rightarrow (sid, Client) \in dom(Q))}{\vdash_{bl} P, Q, \Gamma_t, \Theta_t, \rho_t, \sigma_t, s = \texttt{Connect}(x)}$$

$$\frac{\sigma(s) = (sid, role) \quad Q(sid, rev(role)) = (p, st, \varepsilon)}{\vdash_{bl} P, Q, \Gamma_t, \Theta_t, \rho_t, \sigma_t, x = \texttt{Receive}(s)}$$

Fig. 10. Blocked states

The type soundness theorem states that a consistent full state either gives rise to an infinite reduction sequence, or it finally rerduces to consistent full state exhibiting a blocked state.

Theorem 1 (Type Soundness). *Suppose that* $F = \{t\}, \emptyset, \emptyset, [t \mapsto (\emptyset, \emptyset, e)]$ *with* $\emptyset, \emptyset \vdash F$. *Then, either*

(i) there exists some full state T, P, Q, R *such that* $F \longrightarrow^* T, P, Q, R, \vdash_{bl} T, P, Q, R,$ *and* $\Pi, \Psi \vdash T, P, Q, R$, *or*
(ii) for each T', P', Q', R' *with* $T, P, Q, R \longrightarrow^* T', P', Q', R'$ *there exists a* T'', P'', Q'', R'' *such that* $T', P', Q', R' \longrightarrow T'', P'', Q'', R''$.

9 Conclusion

We have extended previous work on session types to asynchronous communication channels that carry heterogenous values. The main technical innovation is the capturing of asynchronous behavior with an acceptance relation on communication traces and its decidable approximation on session types. Furthermore, we give a low-level operational semantics that includes buffering on channels and prove type soundness.

There are a few obvious extensions that we left to further work. Subtyping of session types should work analogously to work by Gay et al [5]. Channel values could be transmitted over channels by providing a further variant of the send and receive operations. Locations could be introduced as in ambient calculi [3] along with constructs for providing better control over connections.

References

1. Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96, 1992.

2. Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.
3. Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 2000.
4. Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In POPL 1996 [14], pages 372–385.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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, Lissabon, Portugal, April 1998. Springer-Verlag.
10. Robin Milner. *Communication and Concurrency.* Prentice Hall, Englewood Cliffs, NJ, 1989.
11. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I + II. *Information and Control*, 100(1):1–77, 1992.
12. Oscar Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA '93*, pages 1–15, October 1993.
13. Simon Peyton Jones, Andrew Gordon, and Sigbjørn Finne. Concurrent Haskell. In POPL 1996 [14], pages 295–308.
14. *Proceedings of the 1996 ACM SIGPLAN Symposium on Principles of Programming Languages*, St. Petersburg, FL, USA, January 1996. ACM Press.
15. John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.
16. 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.
17. Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *6th International PARLE Conference (Athens, Greece, July 1994)*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.
18. Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.