Math. Struct. in Comp. Science (2016), *vol.* 26, *pp.* 465–509. © Cambridge University Press 2015
doi:10.1017/S0960129514000231 First published online 23 February 2015

An extensible approach to session polymorphism[†]

MATTHEW GOTO[‡], RADHA JAGADEESAN[‡], ALAN JEFFREY[§], CORIN PITCHER[‡] and JAMES RIELY[‡]

[‡]*School of Computing, DePaul University, 243 S. Wabash Ave. Chicago, IL 60604, U.S.A.*
Emails: mattgoto@gmail.com,rjagadeesan@cs.depaul.edu, cpitcher@cs.depaul.edu,jriely@cs.depaul.edu
[§]*Alcatel-Lucent Bell Labs, 1960 Lucent Lane, Naperville, IL 60563, U.S.A.*
Email: ajeffrey@bell-labs.com

Received 12 December 2011; revised 30 January 2013

Session types describe and constrain the input/output behaviour of systems. Existing session typing systems have limited support for polymorphism. For example, existing systems cannot provide the most general type for a generic proxy process that forwards messages between two channels. We provide a polymorphic session typing system for the π calculus, and demonstrate the utility of session-type-level functions in combination with polymorphic session typing. The type system guarantees subject reduction and safety properties, but not deadlock freedom. We describe a formalization of the type system in Coq. The proofs of subject reduction and safety properties, as well as typing of example processes, have been mechanically verified.

1. Introduction

Session types for communication-centred programming (Honda *et al.* 1998; Takeuchi *et al.* 1994) specify the interaction between the sender and the receiver on a communication channel within the channel's type. For example, in the syntax of Gay and Hole (2005) the session type ?[int].?[int].![int].end describes a channel endpoint upon which two integers are received and an integer is subsequently transmitted. Modelling more complex protocols makes use of branching and recursion in session types. Session type systems were originally developed for process calculi, but have been adapted to functional (Vasconcelos *et al.* 2006) and object-oriented (Coppo *et al.* 2007; Gay *et al.* 2010) programming languages. There are implementations of session typing – providing static typechecking of I/O – for languages such as Haskell (Kiselyov *et al.* 2010; Neubauer and Thiemann 2004a,a; Pucella and Tov 2008; Sackman and Eisenbach 2008) and Java (Hu *et al.* 2008).

However, current approaches to session types provide inadequate support for session polymorphism and extensibility via user-defined session type functions. Consequently, a software component with generic I/O behaviour may have to be typechecked once for each communication protocol with which it is used. In this paper, we define a type system that supports polymorphism over session types. This is the first session type system that

[†] This material is based upon the work supported by the National Science Foundation under Grant No. 0916741.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

can assign a useful polymorphic session type to a generic buffer process that forwards messages from one channel to another. Building on ideas from dependent type theory, we integrate functions over session types into the type system. Our type system guarantees that the type of a message occurring in an output is permitted by the typing of the process. The type system, examples, and proofs of subject reduction and safety properties have been formalized in Coq. The Coq definitions and proofs are available online (Goto *et al.* 2011).

In the remainder of this section, we provide an overview of the current support for polymorphism in session typing systems, its limitations and outline our solution.

1.1. *Session polymorphism*

Consider a recursively-defined *sink* process that repeatedly reads and discards messages from a channel parameter *lft*:

$$\textbf{proc}\ \mathit{sink}(\mathit{lft}) = \mathit{lft}?x\,.\,\mathit{sink}(\mathit{lft})$$

We model session types as labelled transition systems (LTSs), where states are session types and labelled transitions represent input and output events, following the semantic subtyping approach (Castagna *et al.* 2009). Then, intuitively, the *lft* parameter in the *sink* process above should have a session type sink that permits a self transition upon input of a value of the top type Top, written as:

$$\mathsf{sink} \xrightarrow{?\mathsf{Top}} \mathsf{sink}.$$

A channel with session type sink may be read to yield a value of type Top, and the channel will have type sink in the continuation. (In the syntax of Gay and Hole (2005), sink would be recursively defined as $\mathsf{sink} = \mu X.?[\mathsf{Top}].X.$)

Session subtyping (Castagna *et al.* 2009; Gay and Hole 2005) allows *sink* to accept channels with session types that are subtypes of sink. However, session subtyping can yield types that are too weak. For example, in the following *echo* process, each message received on a channel *lft* is echoed back on the same channel.

$$\textbf{proc}\ \mathit{echo}(\mathit{lft}) = \mathit{lft}?x\,.\,\mathit{lft}!x\,.\,\mathit{echo}(\mathit{lft})$$

Since x is given the type Top, the *echo* process should permit subtypes of the session type $\mathsf{echo}_{\mathsf{Top}}$ satisfying:

$$\mathsf{echo}_{\mathsf{Top}} \xrightarrow{?\mathsf{Top}!\mathsf{Top}} \mathsf{echo}_{\mathsf{Top}}$$

where we write $s_0 \xrightarrow{m_0 m_1 \ldots m_n} s_{n+1}$ for a sequence of transitions $s_0 \xrightarrow{m_0} s_1 \xrightarrow{m_1} \ldots \xrightarrow{m_n} s_{n+1}$ when the intermediate session types are uninteresting.

Unfortunately, the session type $\mathsf{echo}_{\mathsf{int}}$ satisfying

$$\mathsf{echo}_{\mathsf{int}} \xrightarrow{?\mathsf{int}!\mathsf{int}} \mathsf{echo}_{\mathsf{int}}$$

(upon which a process is expected to read an integer and then send an integer) is not a subtype of $\mathsf{echo}_{\mathsf{Top}}$. The closest approximation

$$\mathsf{echo}_{\mathsf{intTop}} \xrightarrow{?\mathsf{int}!\mathsf{Top}} \mathsf{echo}_{\mathsf{intTop}}$$

is weaker because responses are only constrained to be a subtype of Top instead of int.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

In seeking expressiveness beyond that provided by session subtyping, (Dezani-Ciancaglini *et al.* 2007b; Gay 2008) investigate bounded polymorphism in the context of session types. Bounded polymorphism allows the *echo* process to be used with subtypes of a session type $\mathsf{echo}_\forall$ satisfying:

$$\forall a <: \mathsf{Top}, (\mathsf{echo}_\forall \xrightarrow{?a\,!a} \mathsf{echo}_\forall).$$

That is, a process reads from a channel with session type $\mathsf{echo}_\forall$ yielding a value of some type parameter a, a subtype of Top, and then sends a value of type a back on the same channel.

However, common generic processes cannot be typed adequately with bounded polymorphism. Consider the unidirectional forwarder process that copies messages from channel *lft* to channel *rht*:

$$\mathbf{proc}\ \mathit{fwd}(\mathit{lft}, \mathit{rht}) = \mathit{lft}?x\,.\,\mathit{rht}\,!\,x\,.\,\mathit{fwd}(\mathit{lft}, \mathit{rht})$$

For any session type s, it should be possible to use the *fwd* process with channel *lft* assigned session type s and *rht* assigned session type $(\mathsf{dual}\ s)$, where the dual operation replaces inputs with outputs and vice versa. The forwarder process will not produce incorrect output on *rht*, as long as it receives correct input on *lft*. We refer to this different form of polymorphism as *session polymorphism.*

To enable session polymorphism we allow processes to read on channels with abstract session types *and* incorporate deductive reasoning principles about session types into the type system. In contrast, bounded polymorphism (Dezani-Ciancaglini *et al.* 2007b; Gay 2008) prevents reads on channels with abstract session types, precluding the typing of *fwd*.

A call to the *fwd* process generates a single use channel upon which to pass the parameters *lft* and *rht*. This single use channel is given the polymorphic session type fwd satisfying:

$$\forall\ s,\ (\mathsf{fwd} \xrightarrow{?Ch(s)\ ?Ch(\mathsf{dual}\ s)} \mathsf{end}).$$

This specification of the session type fwd states that a channel with session type fwd may be used to read two channels with session types s and $(\mathsf{dual}\ s)$ respectively. For example, if a channel n has type $Ch(\mathsf{fwd})$, then we expect that *lft* and *rht* have dual session types at the point where they are read in a one-step forwarding process:

$$n?\mathit{lft}\,.\,n?\mathit{rht}\,.\,\mathit{lft}?x\,.\,\mathit{rht}\,!\,x$$

Then, the key to our type system is that the *fwd* process may read upon the channel *lft* with session type s, *and* we deduce that s permits reading a value of some type a. Moreover, we can deduce that $(\mathsf{dual}\ s)$ permits writing a value of type a, thus justifying typing of the *fwd* process.

1.2. *Session transducers*

Polymorphism allows code to place requirements on some of the structure of a session type and be parametric in the remainder. For example, consider the translation process below that forwards tokens from *lft* to *rht*, but substitutes a COLOR token for each COLOUR

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

token. We seek a polymorphic session typing for the *translate* process where the session type for *rht* is described in terms of the session type for *lft*.

$$\begin{aligned}&\textbf{proc}\ \mathit{translate}(\mathit{lft}, \mathit{rht}) = \\ &\quad \mathit{lft}?x.(\ [x{=}\textsc{colour}]\,\mathit{rht}!\textsc{color}.\mathit{translate}(\mathit{lft}, \mathit{rht}) \\ &\qquad + [x{\neq}\textsc{colour}]\,\mathit{rht}!x.\mathit{translate}(\mathit{lft}, \mathit{rht}))\end{aligned}$$

We take inspiration from dependent type theory and recent implementations of type-level functions (Jeffrey and Rathke 2011; Kiselyov *et al.* 2010) by adopting type-level functions that act upon session types, referred to as session type functions in the sequel. An open-world assumption applies to session types, and we allow users to define new session type functions. Session type functions are specified in terms of the input and output transitions of their session type parameters.

To illustrate in the context of the *translate* process, we define a unary session type function $\mathsf{translate} : \mathit{session} \Rightarrow \mathit{session}$ acting as a *transducer* on session types using the following inference rules:

$$\forall\ s\ s',\ (s \xrightarrow{?\{\textsc{colour}\}} s') \ \Rightarrow\ (\mathsf{translate}(s) \xrightarrow{?\{\textsc{color}\}} \mathsf{translate}(s')),$$
$$\forall\ s\ k\ s',\ k \neq \textsc{colour} \ \Rightarrow\ (s \xrightarrow{?\{k\}} s') \ \Rightarrow\ (\mathsf{translate}(s) \xrightarrow{?\{k\}} \mathsf{translate}(s')).$$

The first inference rule states that if the underlying session type s allows an input of singleton type $\{\textsc{colour}\}$ – in which case the value received must be the token COLOUR – then $\mathsf{translate}(s)$ allows an input of singleton type $\{\textsc{color}\}$. The second inference rule states that any other token that can be input on s can be input on $\mathsf{translate}(s)$. It is implicit that these are the only rules defining the behaviour of translate, and we make use of the associated inversion rule to type processes.

1.3. *Deduction*

In prior session type systems, the typing rule for an input command $\mathit{lft}?x.P$ requires the channel *lft* to be assigned a session type S such that an input transition $S \xrightarrow{?A} T$ follows from the semantics of S. As suggested in the *fwd* example above, the key to our solution is a relaxation of input typing: any session type associated with *lft* permits input, including session types with free session type variables. For example, in typechecking $\mathit{lft}?x.P$, the continuation process P is typed in a context extended with an input transition hypothesis $S \xrightarrow{?a} t$, for a fresh type variable a and a fresh session type variable t. Runtime violations of the protocols specified by session types are prevented by the standard session typing discipline for output commands.

Transition hypotheses serve two roles in typing. Firstly, they may be used directly to justify the typing of an output command. Secondly, they may be used in the deduction of (in)equations and other transition hypotheses. For example, from $\mathsf{translate}(S) \xrightarrow{?A} T$ and the equation $A = \{\textsc{color}\}$, we deduce that there exists a session type T' such that $S \xrightarrow{?\{\textsc{color}\}} T'$ and $T = \mathsf{translate}(T')$. This deduction arising from input transitions can be seen as type refinement (Freeman and Pfenning 1991; Gordon and Fournet 2010) arising from input commands.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

1.4. *Formalization*

We formalize our polymorphic session type system for the π calculus within Coq. The permissible deductions about transitions are given a shallow embedding in Coq for readability and convenience in our mechanized proofs. Other logical systems with support for inductive families of data types (Coquand 1992; Coquand and Paulin-Mohring 1990; Dybjer 1991) could be substituted in place of Coq.

Session typing requires a linear type system, and consequently we use a deep embedding of the process typing judgement within Coq. We represent π calculus processes using first-order abstract syntax. (See Röckl and Hirschkoff (2003) for an analysis of deep versus shallow embeddings and first-order versus higher-order abstract syntax (HOAS) in the context of the π calculus.) To facilitate reasoning with first-order syntax we adopt a locally-nameless representation of π variables and names, following the well-established use of locally-nameless representation in λ calculi (Aydemir *et al.* 2008; Charguéraud 2012).

The Coq formalization has been used to establish subject reduction, and, hence, freedom from runtime errors and communication error, where communication protocols for channels are specified via their session types. The proofs of these safety results do not include additional axioms.

The type system presented here sacrifices deadlock freedom (or progress) guarantees. Whether this trade off between deadlock freedom guarantees and session polymorphism is necessary is left to future work (we discuss this point further in Section 6).

Nevertheless, we argue that safety results are particularly useful in a polymorphic setting via a series of examples. These examples culminate in Example 13 where we validate the theory by showing that an implementation of the alternating bit protocol (ABP) admits a polymorphic session typing. The ABP provides error correction over lossy channels, and is thus a surprising candidate for a non-degenerate session typing. Session type functions are used to express the invariants of *internal* lossy channels in relationship to external channels, including a static indication of which messages have been successfully delivered. However, the *external* interface to the ABP has a straightforward typing that, in essence, indicates that the ABP acts as the identity on channels carrying tokens. The following Coq statement gives the process typing judgement `|-p` for the ABP process `abp` with external channels `lft` and `rht` that are assigned session types `SToks s` and `SDual (SToks s)` respectively:

```
abp_typing : forall s : session,
               CTX.add (Nm (Free "lft"), TChannel (SToks s))
                 (CTX.add (Nm (Free "rht"), TChannel (SDual (SToks s)))
                    CTX.empty) |-p abp (Nm (Free "lft")) (Nm (Free "rht"))
```

In this statement, `abp_typing` is the name of a constructive proof of a proposition about process typing. The proof uses the type system defined in this paper. The process typed is `abp (Nm (Free "lft")) (Nm (Free "rht"))`, where `abp` is a Coq function that returns a representation of the π process for the ABP when given two values (names in this case) to use as the external channels. The typing occurs in a process context that adds types for the values to the empty context. The session types in the channel types

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

assigned to the names refer to a universally quantified session variable `s`. Thus the proof of `abp_typing` can be instantiated at any session type.

1.5. *Organization*

In Sections 2 and 3, we present our process language and polymorphic session type system. We demonstrate the type system in a series of detailed examples in Section 4. The final example shows that our typing system is powerful enough to show partial correctness of the ABP, demonstrating the use of polymorphic session typing with lossy channels. Section 5 presents our subject reduction and safety results. Section 6 describes related work. We conclude in Section 7 with a summary of our contributions and directions for future work.

2. Operational model

In this section, we describe the process language and its operational semantics.

2.1. *Process language*

The process language is a synchronous π calculus (Milner *et al.* 1992) with matching and mismatching tests. (Mis)matching tests compare not channels but a distinguished set of values that we refer to as *tokens*. Tokens may be compared and transmitted, but cannot be generated by $\boldsymbol{\nu}$. We use tokens to represent primitive data, and as labels for an encoding of the standard label selection and branching for session types (Honda *et al.* 1998; Takeuchi *et al.* 1994).

The original π calculus does not distinguish the endpoints of a channel by polarities, and nor does the session type system described in Honda *et al.* (1998). We follow Gay and Hole (2005) in using polarized channel names in conjunction with session typing. That is, we distinguish the name n from the coname $\overline{n}$, representing the two sides of a channel. Conames are not themselves names, but are a derived set. Both names and conames are values, which may be communicated independently; thus, the $\boldsymbol{\nu}$ operator creates one name, but two values. The distinction between the name and coname is arbitrary but must be maintained consistently throughout an execution.

Formally, we distinguish three disjoint sets of identifiers: names, n; tokens, k, ℓ; and variables, x, y. We use the term *channel* to include both names and conames. Channels, tokens and variables are values. Values and processes are defined in Figure 1.

In examples, we sometimes drop final occurrences of $\mathbf{0}$. The variable x is bound by input $(u?x.P)$ with scope P. The name n and coname $\overline{n}$ are bound by restriction in $((\boldsymbol{\nu} n)P)$ with scope P. There are no binders for tokens. We identify syntax up to renaming of bound variables and names and write $P\{^{u}/_{x}\}$ for the capture-free substitution of u for x in P. Let $fv(P)$ denote the set of free variables in a process, $fn(P)$ the set of free names, and $fval(P)$ the set of values. For example, $fval((\boldsymbol{\nu} n')n?x.\overline{n}!\overline{n'}.\mathbf{0}) = \{n, \overline{n}, x\}$.

Processes are identified up to the structural equivalence, written $P \equiv Q$, as defined in Figure 2.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

$$
\begin{array}{rll}
u, v & ::= n & \text{(Channel name value)} \\
& \mid \overline{n} & \text{(Channel coname value)} \\
& \mid k & \text{(Token value)} \\
& \mid x & \text{(Variable value)} \\
P, Q, R & ::= \mathbf{0} & \text{(Zero process)} \\
& \mid u!v.P & \text{(Output process)} \\
& \mid u?x.P & \text{(Input process)} \\
& \mid [u{=}v]P & \text{(Match process)} \\
& \mid [u{\neq}v]P & \text{(Mismatch process)} \\
& \mid (\boldsymbol{\nu} n)P & \text{(New name process)} \\
& \mid P{+}Q & \text{(Choice process)} \\
& \mid P \mid Q & \text{(Parallel process)} \\
& \mid *P & \text{(Replication process)}
\end{array}
$$

Fig. 1. Values and processes.

$$
\frac{}{P \equiv P} \qquad \frac{P \equiv Q}{Q \equiv P} \qquad \frac{P \equiv Q \qquad Q \equiv R}{P \equiv R} \qquad \frac{}{*P \equiv P \mid *P}
$$

$$
\frac{}{P{+}\mathbf{0} \equiv P} \qquad \frac{}{P{+}Q \equiv Q{+}P} \qquad \frac{}{P{+}(Q{+}R) \equiv (P{+}Q){+}R} \qquad \frac{n \notin fn(Q)}{((\boldsymbol{\nu} n)P){+}Q \equiv (\boldsymbol{\nu} n)(P{+}Q)}
$$

$$
\frac{}{P \mid \mathbf{0} \equiv P} \qquad \frac{}{P \mid Q \equiv Q \mid P} \qquad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R} \qquad \frac{n \notin fn(Q)}{((\boldsymbol{\nu} n)P) \mid Q \equiv (\boldsymbol{\nu} n)(P \mid Q)}
$$

$$
\frac{}{(\boldsymbol{\nu} n)\mathbf{0} \equiv \mathbf{0}} \qquad \frac{n \notin fn(P)}{(\boldsymbol{\nu} n)P \equiv P} \qquad \frac{}{(\boldsymbol{\nu} n)(\boldsymbol{\nu} n')P \equiv (\boldsymbol{\nu} n')(\boldsymbol{\nu} n)P}
$$

$$
\frac{P \equiv P'}{P{+}Q \equiv P'{+}Q} \qquad \frac{P \equiv P'}{P \mid Q \equiv P' \mid Q} \qquad \frac{P \equiv P'}{(\boldsymbol{\nu} n)P \equiv (\boldsymbol{\nu} n)P'}
$$

Fig. 2. Structural equivalence for processes.

2.2. *Reduction semantics*

The standard reduction semantics (Milner 1991) for π describes the effects of interaction but not the interaction itself. Following Gay and Hole (2005), we annotate the reduction relation with observations about values communicated on channels that are not hidden, in order to reason about interactions and session types. These observations are referenced in the statement of subject reduction (Theorem 5) and conformance (Theorem 7).

The sets of observable values and observations are defined in Figure 3. The observable values represent communicated values. An observable value is either a token k, or $\star$ which represents any channel name or coname. Thus, the observations can be used to distinguish processes that communicate different tokens, but cannot be used to observe the identity of communicated channels. This is natural in the context of session typing, because session types describe the I/O capabilities of channels rather than their identity.

In the definition of the reduction relation below, the identity of channels is hidden using the function obsv, which maps a non-variable value to an observable value. It is defined by $\mathsf{obsv}(k) \triangleq k$ for any token, and otherwise $\mathsf{obsv}(u) \triangleq \star$.

Observations represent: (1) an input of observable value γ on channel n, written $n?\gamma$; (2) an output of observable value γ on visible channel n, written $n!\gamma$; or (3) a hidden operation, such as interaction on a hidden channel, written τ. The operation $(\alpha \setminus n)$ hides

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

$$
\begin{array}{lll}
\gamma ::= k & & \text{(Token observable value)} \\
\quad \mid \star & & \text{(Hidden channel observable value)} \\
\alpha ::= n?\gamma & & \text{(Input observation)} \\
\quad \mid n!\gamma & & \text{(Output observation)} \\
\quad \mid \tau & & \text{(Silent observation)}
\end{array}
$$

Fig. 3. Observable values and observations.

$$
\frac{}{((n!u.P_1)+P_2) \mid ((\overline{n}?x.Q_1)+Q_2) \xrightarrow{n!\mathsf{obsv}(u)} P_1 \mid Q_1\{^u/_x\}}
$$

$$
\frac{}{((\overline{n}!u.P_1)+P_2) \mid ((n?x.Q_1)+Q_2) \xrightarrow{n?\mathsf{obsv}(u)} P_1 \mid Q_1\{^u/_x\}}
$$

$$
\frac{P \equiv Q \quad Q \xrightarrow{\alpha} Q' \quad Q' \equiv P'}{P \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{P \xrightarrow{\alpha} P'}{(\boldsymbol{\nu} n) P \xrightarrow{\alpha \setminus n} (\boldsymbol{\nu} n) P'}
$$

$$
\frac{}{[k{=}k]P \xrightarrow{\tau} P} \qquad \frac{k \neq \ell}{[k{\neq}\ell]P \xrightarrow{\tau} P}
$$

Fig. 4. Reduction relation.

the (channel) name n in observation α.

$$
\alpha \setminus n = \begin{cases} \tau & \text{if } \alpha = n?\gamma \text{ or } \alpha = n!\gamma \\ \alpha & \text{otherwise} \end{cases}
$$

Finally, the annotated reduction relation $P \xrightarrow{\alpha} P'$ captures reduction from a process P to P', yielding observation α. It is defined in Figure 4.

The annotated reduction rules for structural equivalence and parallel composition are straightforward. Matching and mismatching tests compare tokens, but not variables or (co)names. Later, runtime safety ensures that only tokens are compared in well-typed processes with no free variables.

Communication between name n and coname $\overline{n}$ results in an observable annotation for n. For example, consider the annotated reduction sequence:

$$
(n!k.n?y.\mathbf{0}) \mid (\overline{n}?x.\overline{n}!n'.\mathbf{0}) \xrightarrow{n!k} (n?y.\mathbf{0}) \mid (\overline{n}!n'.\mathbf{0}) \xrightarrow{n?\star} \mathbf{0}
$$

The session type system in Section 3 requires that n be assigned a session type permitting an output of token k, followed by an input of a channel.

Annotated reduction is notationally similar to reduction in a LTS, but it is semantically unrelated. Our non-τ annotations indicate completed interactions, rather than potential ones as in an LTS. Our τ annotations indicate hidden interactions, rather than completed ones as in an LTS.

2.3. *Label selection and branching*

Our choice of process language differs from earlier work by the omission of branching and choice based on labels. This omission is motivated by additional expressive power arising from the introduction of (in)equational constraints in the type system; and the desire for a minimal core language. Our process language is sufficiently expressive to encode

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

branching and choice constructs via ordinary communication and matching/mismatching tests.

Recall the selection $u \lhd l.P$ and branching $u \rhd \{l_1 : P_1 \,[]\, l_2 : P_2 \,[]\, \cdots \,[]\, l_n : P_n\}$ constructs from Honda *et al.* (1998). The expected reductions for selection and branching with polarized channels are given by (we omit the reduction when n and $\overline{n}$ are swapped):

$$(n \lhd l_i.P) \mid (\overline{n} \rhd \{l_1 : P_1 \,[]\, l_2 : P_2 \,[]\, \cdots \,[]\, l_n : P_n\}) \rightarrow P \mid P_i.$$

We now define an encoding of selection and branching constructs in our process language. In Section 3 we provide derived typing rules for these encodings. We restrict attention to binary selection and branching for brevity. First, assume that the set of labels used for selection and branching is a subset of the set of tokens. We let l range over tokens used as labels. The selection construct $u \lhd l.P$ is encoded as an output:

$$u \lhd l.P \triangleq u!l\,.P$$

The encoding for branching $u \rhd \{l_1 : P_1 \,[]\, l_2 : P_2\}$ inputs a label (token) on u, and then uses choice to ensure that at most one of the branches P_1, P_2 is executed – this is necessary to support the expected derived typing rules for branching. Intuitively, we would like the following encoding.

$$u \rhd \{l_1 : P_1 \,[]\, l_2 : P_2\} = u?x\,.([x{=}l_1]P_1 + [x{=}l_2]P_2).$$

However, our semantics only reduces interactions that sit immediately inside a choice. We could modify the semantics so that (mis)matching is treated using structural equivalence rather than reduction. Here, instead, we leave reduction alone and present a slightly more complex encoding for branching.

The encoding signals the components of the choice process from matching processes that examine the label. We assume a token $\star$ for such signalling. Then we encode branching as follows (where the variable y and names n_1, n_2 for signalling channels are fresh for P_1 and P_2):

$$\begin{aligned} u \rhd \{l_1 : P_1 \,[]\, l_2 : P_2\}) \triangleq u?x\,.(\boldsymbol{\nu} n_1)(\boldsymbol{\nu} n_2)(\quad & [x{=}l_1]n_1!\star \\ \mid\ & [x{=}l_2]n_2!\star \\ \mid\ & [x{\neq}l_1]\,[x{\neq}l_2]\,error!\star \\ \mid\ & ((\overline{n}_1?y\,.\,[x{=}l_1]P_1){+}(\overline{n}_2?y\,.\,[x{=}l_2]P_2))). \end{aligned}$$

The name *error* is used to indicate when the input label matches neither l_1 nor l_2. We discuss this further below. The presence of operationally-redundant tests $[x{=}l_i]$ under choice is for typing purposes; they would be eliminated with a more sophisticated construct that performed matching and mismatching in conjunction with choice.

Define the relation $P_1 \xrightarrow{\alpha}_{\tau} P_n$ to hold iff $P_1 \xrightarrow{\alpha} P_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_{n-1} \xrightarrow{\tau} P_n$. The encodings of selection and branching have the following reductions (again, we omit the reductions when n and $\overline{n}$ are swapped):

$$(n \lhd l_1.P) \mid (\overline{n} \rhd \{l_1 : P_1 \,[]\, l_2 : P_2\}) \xrightarrow{n!l_1}_{\tau} P \mid P_1 \mid (\boldsymbol{\nu} n_1)(\boldsymbol{\nu} n_2)([l_1{=}l_2]n_2!\star \mid [l_1{\neq}l_1]\,[l_1{\neq}l_2]\,error!\star),$$

$$(n \lhd l_2.P) \mid (\overline{n} \rhd \{l_1 : P_1 \,[]\, l_2 : P_2\}) \xrightarrow{n!l_2}_{\tau} P \mid P_2 \mid (\boldsymbol{\nu} n_1)(\boldsymbol{\nu} n_2)([l_2{=}l_1]n_1!\star \mid [l_2{\neq}l_2]\,error!\star).$$

The remaining matching and mismatching subprocesses are not garbage collected, but are stuck and so do not affect subsequent computation.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

$$
\begin{array}{rll}
A,B ::= & \{k\} & \text{(Singleton type)} \\
 & \mid Ch(S) & \text{(Channel type)} \\
M ::= & \mathtt{!}A & \text{(Output message)} \\
 & \mid \mathtt{?}A & \text{(Input message)} \\
S,T ::= & \mathsf{end} & \text{(Terminal session)} \\
 & \mid M.S & \text{(Message prefix session)} \\
 & \mid S+T & \text{(Union session)} \\
 & \mid \mathsf{dual}\, S & \text{(Dual session)} \\
 & \mid \cdots &
\end{array}
$$

Fig. 5. Types, messages, and sessions.

If $l_1 = l_2$, both of the reductions above are possible, but only one of the subprocesses P_1, P_2 will be chosen as discussed above. However, it is necessary to assume $l_1 \neq l_2$ to derive the usual typing rule for (our encoding of) branching.

We now return to the case when a token $k \notin \{l_1, l_2\}$ is sent on n. In order to observe the synchronous transmission on *error* via the annotated reduction relation, we place processes in parallel with the process $\overline{error}?z$ that receives a message on $\overline{error}$ – such parallel composition does not enable additional reductions when either l_1 or l_2 is sent on n, because transmission on *error* occurs after stuck mismatching tests.

$$
\begin{array}{l}
\overline{error}?z \mid (n \lhd k.P) \mid (\overline{n} \rhd \{l_1 : P_1 \,[\!]\, l_2 : P_2\}) \xrightarrow{n!k}_{\tau} \xrightarrow{error!\star} \\
\quad P \mid (\boldsymbol{\nu} n_1)\,(\boldsymbol{\nu} n_2)((\,[k{=}l_1]\,n_1!\star \mid [k{=}l_2]\,n_2!\star) \mid (\overline{n}_1?y.\,[k{=}l_1]\,P_1 + \overline{n}_2?y.\,[k{=}l_2]\,P_2))
\end{array}
$$

This sequence of annotated reductions captures both the erroneous transmission of $k \notin \{l_1, l_2\}$ and the subsequent use of a channel *error* that is intended to have no interaction. Both transmissions are prevented in well-typed processes by conformance Theorem 7 (see Example 14).

The reader familiar with session types may wonder about the typing of $error!\star$ when *error* has a session type that permits no interaction. We address this point in Section 3.

3. Polymorphic session typing

In this section we describe our polymorphic session type system for the π calculus.

Section 3.1 provides an overview of LTS semantics for session types. Section 3.2 formalizes the semantics using logical deduction. Section 3.3 shows the treatment of binding in session polymorphism. Section 3.4 describes our approach to extensibility of session types. Section 3.5 defines type assignment for processes in terms of the LTS semantics and discusses derived rules for label selection and branching.

3.1. *Session types*

Our session types control the messages transmitted on channels. Messages indicate the direction of communication and the type of communicated data. Figure 5 defines syntactic categories for *types*, *messages* and *sessions*. We use *session* and *session type* interchangeably.

Types and messages are straightforward. A *type* may be a singleton type $\{k\}$, for a token k, or a channel type $Ch(S)$, for a session S. The only inhabitant of the singleton type

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

$\{k\}$ is the token k. A channel with channel type $Ch(S)$ must obey the protocol described by S. A *message* may be an output $!A$ or an input $?A$, for a type A.

A session may be end which indicates that no communication is possible, $M.S$ representing prefixing of session S by message M, or the union $S + T$ of sessions S and T.

Additionally, $(\mathsf{dual}\ S)$ represents the dual of another session S. The dual session $(\mathsf{dual}\ S)$ reverses the direction of messages within the session S – we make this precise below. In the sequel, the two endpoints n and $\overline{n}$ of a channel are assigned channel types where one session is the dual of the other.

The type system is designed so that the grammar of sessions is extensible, indicated by the ellipsis in the definition. One example of a new session type constructor is $\mathsf{translate}$ from Section 1.2. Further, examples are described in this section and Section 4. This raises the question of how meaning is assigned to new session type constructors, and, more generally, for all session types.

The meaning of a session type is conventionally circumscribed by its use in type assignment rules. As discussed in Section 1, our type system decouples the meaning of a session type from its use in type assignment rules by introducing an LTS to describe the I/O operations permitted in the current state of a channel. This decoupling allows reasoning about transitions independently of the syntax of a session type, and consequently supports session polymorphism. Moreover, the use of an LTS semantics permits a modular description of session type constructors.

A labelled transition $S \xrightarrow{M} T$ indicates that interaction with a message M on a channel in state S results in a channel in state T. The message M may be an input or an output. The labelled transitions for the sessions referenced above can be stated directly. The session end has no transitions. The prefix session has a single transition $M.S \xrightarrow{M} S$. The union session has a transition $S+T \xrightarrow{M} S'$ whenever $S \xrightarrow{M} S'$, and a transition $S+T \xrightarrow{M} T'$ whenever $T \xrightarrow{M} T'$. The dual session has a transition $(\mathsf{dual}\ S) \xrightarrow{?A} (\mathsf{dual}\ S')$ whenever $S \xrightarrow{!A} S'$, and a transition $(\mathsf{dual}\ S) \xrightarrow{!A} (\mathsf{dual}\ S')$ whenever $S \xrightarrow{?A} S'$. Labelled transitions can also be defined for new session types with recursive behaviour. The formal LTS semantics for these and other session types is given in Section 3.2.

We next consider a series of examples to illustrate how session types and their LTSs control the behaviour of processes. In these examples, we informally state that processes obey session types assigned to channels rather than discussing type assignment. Formal type assignment for processes is deferred to Section 3.5.

Examples 1 and 2 demonstrate branching for output and input messages respectively. Here, we refer to branching in an LTS, rather than the dual concepts of label selection and label branching found in session types.

Example 1 (output branching). Consider the session type $S = (!k_1.!k_3.\mathsf{end}) + (!k_2.!k_4.\mathsf{end})$. The corresponding LTS for S is shown in Figure 6. Suppose that a channel endpoint n is assigned type $Ch(S)$. Then a process is allowed to send either k_1 followed by k_3, or k_2 followed by k_4. Thus the processes $n!k_1.n!k_3$ and $n!k_2.n!k_4$ obey the session type S.

Unlike Honda *et al.* (1998), we allow the terminal process $\mathbf{0}$ when channels are not in a terminal state. For example, the processes $n!k_1$ and $\mathbf{0}$ also obey the session type S. This

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

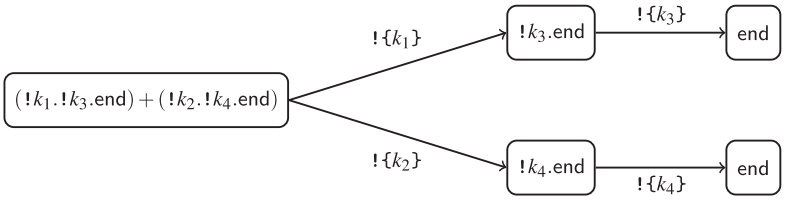


Fig. 6. Labelled transition system with output branching.

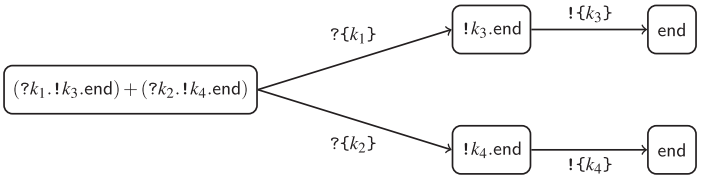


Fig. 7. Labelled transition system with input branching.

demonstrates that the output on channel n can be a prefix of the tokens output on a path from S in processes that have neither deadlock nor divergence.

The order of messages in an LTS is significant. This means that $n!k_3$ does not obey the session type if $k_3 \notin \{k_1, k_2\}$. Similarly, $n!k_2.n!k_3$ does not obey the session type S if $k_1 \neq k_2$ and $k_3 \neq k_4$. □

Session type LTSs are intended to control the output operations on a channel rather than the input operations and subsequent matching and mismatching operations. Nevertheless, transitions labelled with input messages control the evolution of session types. Example 2 illustrates this asymmetry between input and output.

Example 2 (input branching). Consider the session type $S = (?k_1.!k_3.\text{end}) + (?k_2.!k_4.\text{end})$. The corresponding LTS for S is shown in Figure 7. Suppose that a channel endpoint n is assigned type $Ch(S)$. If input is received on n, then it may be either k_1 or k_2. If k_1 is received, then an output of k_3 is allowed. Similarly, k_4 can be output if k_2 was received.

There are no requirements for processes to read an input, to use a received value, or to send output after an input. For example, the processes $\mathbf{0}$ and $n?x$ both obey the session type S.

Suppose $k_1 \neq k_2$ or $k_3 = k_4$. Since the output of k_3 is conditional upon first receiving k_1, in general the process $n?x.n!k_3$ does not obey the session type S. However, if the input is found to be k_1, then the output k_3 is enabled. In particular, the process $n?x.\,[x{=}k_1]n!k_3$ does obey the session type S.

To see why $k_1 \neq k_2$ or $k_3 = k_4$ is necessary suppose that $k_1 = k_2$ and $k_3 \neq k_4$. Then, the process $n?x.\,[x{=}k_1]n!k_3$ does not obey the session type S, because the output process $n!k_3$ obeys the first branch's session type $!k_3.\text{end}$, but does not obey the second branch's session type $!k_4.\text{end}$. This highlights the universal nature of analysis for input processes: every possible labelled transition must be considered after an input occurs. In contrast, analysis for output processes is existential: some labelled transition must exist for an output to be permitted.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

Finally, suppose that $k_3 = k_4$. In this case, the process $n?x.n!k_3$ also obeys the session type S without performing any matching to determine whether $x = k_1$ or $x = k_2$. The match is unnecessary because the output can occur in either branch of the original session type. □

Perhaps surprisingly, we do not prohibit input operations in processes. That is, an input operation is allowed at any session type, even if the session type has no input. This is for two reasons. Firstly, we believe that a framework for session polymorphism must, at least, accommodate the unidirectional forwarding process (see Section 1.1 and Example 11). It is unclear how to control the input operation in this process in a way that works for all session types. Secondly, control over output operations is sufficient because it ensures that channels only carry values permitted by a session type, yielding subject reduction, runtime safety and conformance results (see Section 5 for further discussion).

The free use of input operations in processes is demonstrated in the next example.

Example 3 (terminal). The session end has no transitions. Suppose that n is assigned type $Ch(\mathsf{end})$. The process $n!k$ does not obey the session type end, because output is controlled and end has no output transitions.

In contrast, the process $n?x$ does obey the session type end, despite the lack of input transitions from end, because input is not controlled.

However, the continuation of an input process is checked under the assumption that an input occurs. To see the consequences, consider a separate channel *error* that also has type $Ch(\mathsf{end})$. In this case, the process $n?x.error!k$ obeys the session types for n and *error* because no input can occur on n (since it has type $Ch(\mathsf{end})$), and so the output on *error* cannot occur. In other words, $error!k$ is identified as unreachable code. Technically, the assumption of an input transition from end, in conjunction with the inversion principle that there are no input transitions from end, yields a contradiction that allows typing of $error!k$. This is verified by the type system in Section 3.5. □

Unlike the session types of Honda *et al.* (1998), our session types may have labelled transitions for both input and output from the same state. Example 4 illustrates the effect on processes.

Example 4 (input and output branching). Consider $S = (?k_1.!k_3.\mathsf{end}) + (!k_2.!k_4.\mathsf{end})$. The corresponding LTSs for S and $(\mathsf{dual}\ S)$ are shown in Figure 8. Suppose that the channels n and $\overline{n}$ are assigned types $Ch(S)$ and $Ch(\mathsf{dual}\ S)$ respectively. There are two possible interactions that can occur on n. Either the token k_1 is received on n or the token k_2 is sent on n. Synchronous interaction ensures that k_2 cannot be sent on n whilst k_1 is sent on $\overline{n}$. The processes $n?x.n!k_3$ and $n!k_2.n!k_4$ both obey the session type S. Hence, the composite process $(n?x.n!k_3)+(n!k_2.n!k_4)$ also obeys the session type S. □

3.2. *Formalization*

We now turn to the formalization of the labelled transition relation for session types in Coq.

The sets of types, messages and session types seen previously are defined in a straightforward way by mutual induction in Coq (Bertot and Castéran 2004), following

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

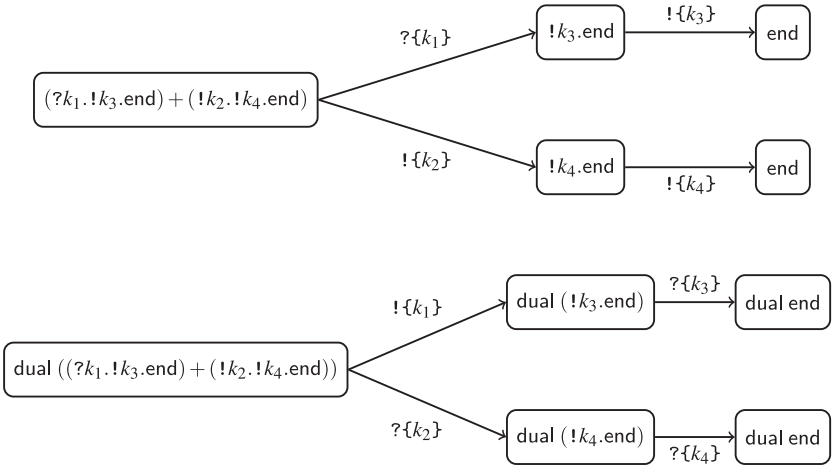


Fig. 8. Labelled transition system with input and output branching.

$$
\begin{array}{l}
\texttt{Inductive}\ \mathit{transition} : \mathit{session} \Rightarrow \mathit{message} \Rightarrow \mathit{session} \Rightarrow \texttt{Prop} := \\
\texttt{(* Part 1 *)} \\
\mid \mathit{TRPrefix} : \forall\, s\ m,\ (m.s \xrightarrow{m} s) \\
\mid \mathit{TRUnion}_1 : \forall\, s_1\ s_2\ t_1\ m,\ (s_1 \xrightarrow{m} t_1) \Rightarrow (s_1 + s_2 \xrightarrow{m} t_1) \\
\mid \mathit{TRUnion}_2 : \forall\, s_1\ s_2\ t_2\ m,\ (s_2 \xrightarrow{m} t_2) \Rightarrow (s_1 + s_2 \xrightarrow{m} t_2) \\
\mid \mathit{TRDual}_1 : \forall\, s\ t\ a,\ (s \xrightarrow{!a} t) \Rightarrow (\mathsf{dual}\ s \xrightarrow{?a} \mathsf{dual}\ t) \\
\mid \mathit{TRDual}_2 : \forall\, s\ t\ a,\ (s \xrightarrow{?a} t) \Rightarrow (\mathsf{dual}\ s \xrightarrow{!a} \mathsf{dual}\ t) \\
\texttt{(* Part 2 *)} \\
\mid \mathit{TRFwd} : \forall\, s,\ (\mathsf{fwd} \xrightarrow{?Ch(s)\ ?Ch(\mathsf{dual}\ s)} \mathsf{end}) \\
\mid \mathit{TRSink} : \forall\, s,\ (\mathsf{sink} \xrightarrow{?Ch(s)} \mathsf{sink}) \\
\mid \mathit{TRFun} : \forall\, a,\ (\mathsf{fun}\ a \xrightarrow{?a} \mathsf{fun}\ a) \\
\texttt{(* Part 3 *)} \\
\mid \mathit{TRFwdPrefix} : \forall\, s\ k,\ (\mathsf{fwdprefix}\ k \xrightarrow{?Ch(s)?Ch(!\{k\}.(\mathsf{dual}\ s))} \mathsf{end}) \\
\mid \mathit{TRTranslate}_1 : \forall\, s\ s',\ s \xrightarrow{?\{\mathrm{COLOUR}\}} s' \Rightarrow \mathsf{translate}(s) \xrightarrow{?\{\mathrm{COLOR}\}} \mathsf{translate}(s') \\
\mid \mathit{TRTranslate}_2 : \forall\, s\ k\ s',\ k \neq \mathrm{COLOUR} \Rightarrow s \xrightarrow{?\{k\}} s' \Rightarrow \mathsf{translate}(s) \xrightarrow{?\{k\}} \mathsf{translate}(s') \\
\mid \mathit{TRTranslate}_3 : \forall\, s\ s',\ s \xrightarrow{!\{\mathrm{COLOUR}\}} s' \Rightarrow \mathsf{translate}(s) \xrightarrow{!\{\mathrm{COLOR}\}} \mathsf{translate}(s') \\
\mid \mathit{TRTranslate}_4 : \forall\, s\ k\ s',\ k \neq \mathrm{COLOR} \Rightarrow s \xrightarrow{!\{k\}} s' \Rightarrow \mathsf{translate}(s) \xrightarrow{!\{k\}} \mathsf{translate}(s') \\
\ldots \\
\texttt{where}\ \text{“}s \xrightarrow{m} t\text{”} := (\mathit{transition}\ s\ m\ t).
\end{array}
$$

Fig. 9. Coq encoding of the labelled transition relation.

the grammar of Figure 5. Variables for session types are encoded as Coq variables of type *session*. Similarly for type and message variables. We use the following naming convention when we bind types, messages, and sessions explicitly: a for types, m for messages and s for sessions. In the typing rules of Section 3.5, this serves to highlight the binding that occurs in the treatment of input. However, this is merely a convention, since both s and S denote Coq variables of type *session*.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

$$\texttt{Lemma}\ \mathit{stop_inv}\ :\ \forall\, m\ t, (\mathsf{end} \xrightarrow{m} t) \Rightarrow \texttt{False}.$$

$$\texttt{Lemma}\ \mathit{prefix_inv}\ :\ \forall\, m\ s\ m'\ s'\ (\Phi\ :\ \texttt{Prop}),\ ((m = m'\ \wedge\ s = s') \Rightarrow \Phi) \Rightarrow ((m.s \xrightarrow{m'} s') \Rightarrow \Phi).$$

$$\texttt{Lemma}\ \mathit{union_inv}\ :\ \forall\, s_1\ s_2\ m\ t\ (\Phi\ :\ \texttt{Prop}),\ ((s_1 \xrightarrow{m} t) \Rightarrow \Phi) \Rightarrow ((s_2 \xrightarrow{m} t) \Rightarrow \Phi) \Rightarrow ((s_1 + s_2 \xrightarrow{m} t) \Rightarrow \Phi).$$

$$\begin{aligned}&\texttt{Lemma}\ \mathit{dual_inv}\ :\ \forall\, s\ m\ t\ (\Phi\ :\ \texttt{Prop}),\\ &\quad(\forall\, a\ t',\ (m = ?a\ \wedge\ t = \mathsf{dual}\ t'\ \wedge\ (s \xrightarrow{!a} t')) \Rightarrow \Phi)\\ &\Rightarrow (\forall\, a\ t',\ (m = !a\ \wedge\ t = \mathsf{dual}\ t'\ \wedge\ (s \xrightarrow{?a} t')) \Rightarrow \Phi)\\ &\Rightarrow ((\mathsf{dual}\ s \xrightarrow{m} t) \Rightarrow \Phi).\end{aligned}$$

Fig. 10. Inversion principles for labelled transitions.

The inductive definition of the labelled transition relation is shown in Figure 9. We use '⇒' for the Coq function space (implication) to avoid confusion with other notations. The labelled transition relation is encoded as the *transition* function, which maps into the space of propositions `Prop`.

The declared functions, such as *TRPrefix*, may be used to construct proofs that a labelled transition exists. For example, applying *TRPrefix* to the session end and the message $!\{k\}$ establishes the existence of the labelled transition $!\{k\}.\mathsf{end} \xrightarrow{!\{k\}} \mathsf{end}$. There are no functions declared for the terminal session end because it has no transitions.

The $\mathit{TRUnion}_1$ and $\mathit{TRUnion}_2$ function declarations describe the transitions from a session $S + T$ in terms of transitions from S and T respectively. Similarly, the two cases for transitions from a session (dual S) reverse the direction of messages, transforming outputs to inputs and vice versa. The dual (dual (dual S)) of a dual session type is a distinct session type from S, but they have the same labelled transitions (see the discussion of subtyping and similarity in Section 6).

We defer the discussion of transitions in parts 2 and 3 of Figure 9 to Sections 3.3 and 3.4.

The definition in Figure 9 yields an induction principle for the labelled transition relation. From this induction principle, we derive an inversion principle for each session type constructor. The inversion principle for a session type constructor θ permits case analysis upon labelled transitions of the form $\theta(\vec{S}) \xrightarrow{M} T$. The Coq statements of the derived inversion principles are shown in Figure 10.

The lemma *stop_inv* states that the false proposition `False` can be derived from the existence of a labelled transition from end. The lemma *prefix_inv* states that a proposition Φ can be deduced from the existence of a labelled transition $m.s \xrightarrow{m'} s'$, if Φ can be proven when $m = m'$ and $s = s'$. Next, *union_inv* allows deduction of Φ from a transition $s_1 + s_2 \xrightarrow{m} t$ if Φ holds both when the transition comes from s_1 and when it comes from s_2. Similarly, *dual_inv* allows case analysis upon a transition of the form $\mathsf{dual}\ s \xrightarrow{m} t$ by reasoning about the transitions of s after replacing input with output and vice versa.

The polymorphic component of our type system relies upon deducing output-labelled transitions from input-labelled transitions. Example 5 illustrates how deductions can be made from an input-labelled transition.

Example 5. Consider sessions s_1, t and a type B. Suppose $\mathsf{dual}\ s_1 \xrightarrow{?B} t$. Using the inversion lemma *dual_inv*, we establish the proposition Φ defined as $(\exists t', t = \mathsf{dual}\ t' \wedge s_1 \xrightarrow{!B} t')$. The

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

hypotheses for *dual_inv* require Φ to be proven when $?B = ?a$ and $?B = !a$. In the former case, we find that $B = a$, and Φ follows trivially. In the latter case, Φ holds because $?B = !a$ cannot be unified.

Other deductions may be made from Φ. For example, we could further deduce that $(\exists t', \forall s_2, t = \mathsf{dual}\ t' \wedge s_1 + s_2 \xrightarrow{!B} t')$ using $TRUnion_1$ and Φ. This deduction could be used to justify the correctness of a process that reads a message of type B on a channel of type $Ch(\mathsf{dual}\ s_1)$ and then writes the same message to a channel of type $Ch(s_1 + s_2)$. □

3.3. *Binding in session polymorphism*

The session type constructors defined in part 1 of Figure 9 are not exhaustive. Additional session types are necessary to describe the relationship between the types of distinct objects. Our approach to session polymorphism depends upon the use of quantification in the definition of the labelled transition relation. We now illustrate this using the transitions defined in part 2 of Figure 9. We omit the straightforward inversion principles for these session types.

The first additional session type fwd demonstrates the use of binding in session types to capture relationships between the types of different values. The transition *TRFwd* in Figure 9 is labelled with two input messages. This abbreviates multiple labelled transitions shown below. The intermediate session type constructor fwd' has a session type parameter s that is identical to the session type in the message of the first labelled transition. The second labelled transition constrains the session type of the second message to be the dual $(\mathsf{dual}\ s)$ of the session type of the first message s.

$$\begin{aligned} &| \ \mathit{TRFwd} \ : \ \forall\, s : \mathit{session},\ (\mathsf{fwd} \xrightarrow{?Ch(s)} \mathsf{fwd}'(s)) \\ &| \ \mathit{TRFwd'} \ : \ \forall\, s : \mathit{session},\ (\mathsf{fwd}'(s) \xrightarrow{?Ch(\mathsf{dual}\ s)} \mathsf{end}) \end{aligned}$$

The session type fwd is polymorphic in the sense that a channel of type $Ch(\mathsf{fwd})$ can be used to receive two channels with dual session types. Example 6 demonstrates the use of the polymorphism of fwd in a generic process that safely forwards a message from one channel to another.

Example 6 (polymorphic single-step forwarding process). Consider channels $\overline{n}_1$ and n_2 that are assigned types $Ch(T)$ and $Ch(\mathsf{dual}\ T)$ respectively, for any session T. If a third channel n_3 is assigned the type $Ch((\mathsf{dual}\ \mathsf{fwd}))$, then the sequence of outputs in the process P defined by:

$$P \triangleq n_3\,!\,\overline{n}_1\,.\,n_3\,!\,n_2$$

follows the sequence of output-labelled transitions $\mathsf{dual}\ \mathsf{fwd} \xrightarrow{!Ch(T)\,!Ch(\mathsf{dual}\ T)} \mathsf{dual}\ \mathsf{end}$, where s has been instantiated with T.

If the channel endpoint $\overline{n}_3$ is assigned type $Ch(\mathsf{fwd})$, then it is expected that, for some unknown session type s, channels with dual session types s and $(\mathsf{dual}\ s)$ can be read from $\overline{n}_3$. Now, whenever there is an input-labelled transition $s \xrightarrow{?A} s'$, then $TRDual_2$ ensures that there is a corresponding output-labelled transition $\mathsf{dual}\ s \xrightarrow{!A} \mathsf{dual}\ s'$. Then the generic single-step forwarding process Q, defined as follows, obeys the session type $(\mathsf{dual}\ s)$ for

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

channel y.

$$Q \triangleq \overline{n}_3?x.\overline{n}_3?y.x?z.y!z$$

This is because any input $x?z$ is associated with some input-labelled transition $s \xrightarrow{?A} s'$, where z has type A, and so the corresponding output-labelled transition $\mathsf{dual}\ s \xrightarrow{!A} \mathsf{dual}\ s'$ justifies the output $y!z$.

The single-step forwarding process Q can then be used to forward from $\overline{n}_1$ to n_2 by using process P to provide the channels $\overline{n}_1$ and n_2. In this case, the s session parameter in the transitions of fwd is instantiated with T. To illustrate, suppose there exists a token k such that $T \xrightarrow{?\{k\}} T'$, so token k may be received on $\overline{n}_1$, and k may be sent on n_1. The reduction sequence below shows the initialization of Q via P, the reception of k on $\overline{n}_1$ by Q, and the transmission of k on n_2 by Q. Moreover, the annotations can be used to verify that the process obeyed the session types for n_1 and n_2.

$$\begin{array}{ll} & P \mid Q \mid n_1!k \mid \overline{n}_2?x \\ \xrightarrow{n_3!\star} & n_3!n_2 \mid (\overline{n}_3?y.\overline{n}_1?z.y!z) \mid n_1!k \mid \overline{n}_2?x \\ \xrightarrow{n_3!\star} & (\overline{n}_1?z.n_2!z) \mid n_1!k \mid \overline{n}_2?x \\ \xrightarrow{n_1!k} & n_2!k \mid \overline{n}_2?x \\ \xrightarrow{n_2!k} & \mathbf{0} \end{array}$$

It is natural to ask whether the single-step forwarding process Q can be coerced into sending messages that are not permitted by the session type for the channel y. For example, suppose $T = \mathsf{end}$. Then there are no transitions from T or $(\mathsf{dual}\ T)$. In the reduction sequence above, the output $n_2!k$ is not permitted by the session type $(\mathsf{dual}\ \mathsf{end})$ for n_2. However, the blame lies with the earlier output $n_1!k$ that is also not permitted by the session type $(\mathsf{dual}\ \mathsf{end})$ for n_1. Thus the single-step forwarding process Q may send messages not permitted by session types when other subprocesses send messages not permitted by session types. Nevertheless, in typing Q we shall see that the output $y!z$ obeys session types using the argument from Example 3. The results of Section 5 show that this suffices for well-typed processes.

Although the session types of channels may be related, they evolve independently. For example, the type of n_2 is unaffected by changes to the type of n_1. As another example, any type assigned to Q in our system can also be assigned to the following Q', which inverts the order of the second and third inputs.

$$Q \triangleq \overline{n}_3?x.x?z.\overline{n}_3?y.y!z$$

Finally, the channel variables x and y in Q have dual session types s' and $(\mathsf{dual}\ s')$, for some session type s', whenever Q terminates. Thus x and y are candidates for sending on another channel of type fwd. This forms the core of a multi-step forwarding process in Example 11. □

The other session types from part 2 of Figure 9 are sink and $(\mathsf{fun}\ (A))$, for a type A. A channel with type $Ch(\mathsf{sink})$ can be used to repeatedly read messages of type $Ch(s)$, for some session type s. A channel with type $Ch(\mathsf{fun}\ (A))$ can be used to repeatedly read messages of type A. We say that the session types sink and $(\mathsf{fun}\ (A))$ – and their dual sessions $(\mathsf{dual}\ \mathsf{sink})$ and $(\mathsf{dual}\ (\mathsf{fun}\ (A)))$ – are stateless, because every transition is a self

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

loop. Channels with stateless types can be freely copied without causing confusion about their current state.

The (fun (A)) session type is used for communication that plays the same role as session initiation in Honda *et al.* (1998), e.g. in Example 11, a channel of type $Ch(\mathsf{fun}\ (Ch(\mathsf{fwd})))$ is used to repeatedly receive single-use channels of type $Ch(\mathsf{fwd})$.

The distinction between sink and fun lies in the fact that (fun (A)) represents a collection of session types indexed by A, whereas sink is a single session type. Consequently, a channel of type sink can be used to read a sequence of channels, but there need not be any relationship between the session types of those channels.

3.4. *Extensibility*

Similar polymorphic session types can be defined for communication of channels with more complex relationships between their session types. Example 7 gives a polymorphic session type for a generic 'prefixing' process.

Example 7 (polymorphic prefixing). The following process receives two channels as x and y, waits for an input z on x, and then sends both k and x on y. It differs from the single-step forwarding process of Example 6 by including an additional output $y!k$, for some fixed token k.

$$n?x.n?y.x?z.y!k.y!z$$

In order to define the type of the channel n, we can introduce a new session type (fwdprefix k) with the sequence of labelled transitions defined in part 3 of Figure 9. In this session type, the first value received is a channel with session type s, and the second value received is a channel that permits an initial output of k, followed by (dual s). □

This prompts the question, which session types must be defined? We adopt an open-world approach, and allow new session types to be introduced as needed. New session types are introduced with their labelled transitions. In order to avoid changing the meaning of other session types, a new session type constructor θ may only add transitions of the form $\theta(\vec{S}) \xrightarrow{M} T$ to the inductively-defined transition relation, i.e. the source of the transition must be a session type with θ as the outermost constructor. We prohibit reasoning via induction over the inductively-defined set of session types or the inductively-defined transition relation, because such arguments must be modified when new session types are subsequently added. There is one exception: we permit induction over the transition relation for the derivation of inversion principles for each session type constructor. Unlike the induction principle for the transition relation, the inversion principles for existing session type constructors remain unchanged when new session types are defined. The result is a modular framework for the definition of session types and their transitions.

As a second example, recall the *translate* process from Section 1.2 that replaces a COLOUR token with a COLOR token. A new session type translate(s) is introduced to describe the effects of translation. To specify the translation, we use equational and inequational constraints in the definition of the labelled transition relation. With such constraints, session type transducers can be constructed to add, remove or replace tokens from a session.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

Example 8 (translation). Part 3 of Figure 9 gives labelled transitions for session types of the form $\mathsf{translate}(s)$, for some session type s. The behaviour of $\mathsf{translate}(s)$ has been extended to permit replacement of COLOUR with COLOR for both input and output, i.e. the session type describes bidirectional translation.

Consider channels en_{gb} and $\overline{en}_{us}$ with types $Ch(s)$ and $Ch(\mathsf{dual}\ (\mathsf{translate}(s)))$, for some session type variable s, with the intention that tokens read from en_{gb} are translated and then sent on $\overline{en}_{us}$, and that tokens read from $\overline{en}_{us}$ are translated and then sent on en_{gb}. Examining the transitions of $(\mathsf{dual}\ (\mathsf{translate}(s)))$, we find that any token k read from en_{gb} is either COLOUR, in which case COLOR can be written to $\overline{en}_{us}$, or $k \neq \text{COLOUR}$ and k can be written to $\overline{en}_{us}$. Similarly, any token k read from $\overline{en}_{us}$ is either COLOR, in which case COLOUR can be written to en_{gb}, or $k \neq \text{COLOR}$ and k can be written to en_{gb}.

To see the effect of $\mathsf{translate}$, consider session $?\{\text{COLOUR}\}.((!\{\text{RED}\}.\mathsf{end}) + (!\{\text{BLUE}\}.\mathsf{end}))$, for en_{gb}, perhaps representing an incoming request followed by an outgoing RED or BLUE response. When s is instantiated with this session, the corresponding session $(\mathsf{dual}\ (\mathsf{translate}(s)))$ for $\overline{en}_{us}$ is easily seen to have the same LTS as $!\{\text{COLOR}\}.((?\{\text{RED}\}.\mathsf{end}) + (?\{\text{BLUE}\}.\mathsf{end}))$.

The difference in the inequations in $TRTranslate_2$ and $TRTranslate_4$ reveals a subtlety in the specification of polymorphic translations. The channels en_{gb} and $\overline{en}_{us}$ are able to carry pre-translated tokens, i.e. en_{gb} may carry a pre-translated COLOR, which can be copied to $\overline{en}_{us}$; and a pre-translated COLOUR may be received from $\overline{en}_{us}$, then copied to en_{gb}. An alternative is to prevent en_{gb} and $\overline{en}_{us}$ from using the other's language, i.e. en_{gb} is restricted to receiving and sending COLOUR, and similarly for $\overline{en}_{us}$ and COLOR. It is not sufficient to add the conjunction of $k \neq \text{COLOUR}$ and $k \neq \text{COLOR}$ to the hypotheses of $TRTranslate_2$ and $TRTranslate_4$, because it affects the session type of $\overline{en}_{us}$ but not the session type s of en_{gb}. Instead, we can assign en_{gb} the type $Ch(\mathsf{translate}'(s))$, where $\mathsf{translate}'$ is a new session type constructor that prevents transmission of COLOR. We omit the definition of $\mathsf{translate}'$. □

3.5. *Type assignment*

We now turn to type assignment for processes. Our type system shares many ideas with other session type systems: checking that outputs are permitted by the session type for the output channel, assigning dual session types to new channel endpoints created by restriction, and linear usage of process contexts containing values of channel type.

The most significant differences arise from the use of binding and constraints: an input process adds an input-labelled transition constraint to the context when typing the continuation process, a matching (resp. mismatching) process adds an equality (resp. inequality) constraint to the context when typing the continuation process, and an output process deduces that an output-labelled transition follows from the constraints in the context.

We represent binding and constraints via a *context* Δ. A context is a finite list of δ, where δ ranges over variables, (in)equations, and transitions:

$$\delta \ ::= \ a \mid m \mid s \mid A = A' \mid M = M' \mid S = S' \mid A \neq A' \mid M \neq M' \mid S \neq S' \mid S \xrightarrow{M} T.$$

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

We translate judgements using contexts into Coq propositions by first encoding the judgement form as an inductive type in the standard way (Bertot and Castéran 2004), then adding each (in)equation and transition as a hypothesis and each variable as a universal quantifier. For example, use of the context $(m, s, S \xrightarrow{m} s, S = s)$ with a judgement $\mathcal{J}$ is represented as:

$$\forall m : message, \forall s : session, ((S \xrightarrow{m} s) \Rightarrow ((S = s) \Rightarrow \mathcal{J})).$$

This encoding of contexts is not closely tied to Coq, and could be supported in other logics with inductive families of data types.

We write $\Delta \vdash \delta$ to indicate that a constraint δ can be deduced in Coq from the context Δ. For example, typing of output process relies upon deduction of a labelled transition $\Delta \vdash S \xrightarrow{!A} T$. We write $\Delta \vdash \bot$ to indicate that the set of constraints Δ is inconsistent, i.e. they allow the false proposition to be deduced. We allow reordering of contexts without comment when variables are not used before their binding occurrence.

In addition to the constraints in Δ, the process typing judgement $\Delta\,;\, G \vdash P$ requires a process context. A *process context*, $G = \{u_1 : A_1, \ldots, u_n : A_n\}$, is a partial map from values to types. If G_1 and G_2 have disjoint domains, then G_1, G_2 denotes their disjoint union, and $G, u : A$ denotes $G, \{u : A\}$. Suppose $G = \{u_1 : A_1, \ldots, u_n : A_n\}$. In this case, define $dom(G) \triangleq \{u_1, \ldots, u_n\}$ and $ran(G) \triangleq \{A_1, \ldots, A_n\}$. We only consider contexts where there are no tokens in $dom(G)$.

Session type systems place linearity restrictions on the usage of channels. However, values whose types do not change can be copied freely without confusion about their types (in the literature, such types are known as *shared* or *unrestricted* (Vasconcelos 2009)). We require that this property of a type is provable in the logic. That is, we say that a type is *stateless* for Δ, written $(\Delta \vdash A \text{ stateless})$, if it is:

— a singleton type $\{k\}$ for some token k; or
— a channel type $Ch(S)$ such that, for all messages m and session types s, if $\Delta \vdash S \xrightarrow{m} s$, then $\Delta \vdash S = s$.

We say that a process context G is a *partition* of G_1 and G_2, written $(\Delta\,;\, G \vdash G_1 \oplus G_2)$, if

— $G = G_1 \cup G_2$;
— G is a partial map; and
— $(\Delta \vdash A \text{ stateless})$, for every $A \in ran(G_1 \cap G_2)$.

In particular, if $(\Delta\,;\, G \vdash G_1 \oplus G_2)$, then values in $dom(G_1) \cap dom(G_2)$ have stateless types.

The rules for value type assignment $\Delta\,;\, G \vdash u : A$ and process type assignment $\Delta\,;\, G \vdash P$ are given in Figure 11. We have elided a certain amount of bookkeeping from the following process typing rules, we also require: (1) that the free type, message and session metavariables in $ran(G)$ are contained in Δ; (2) that $fval(P) \subseteq dom(G)$ in the rules for matching, mismatching and inconsistent contexts; and (3) that $fval(P) \subseteq dom(G) \cup \{u, x\}$ in the rule for input. (2) and (3) cannot be omitted without loss of the property that free names and variables are contained in the process context of a well-typed process.

The rules for stateless P-OUT-STATELESS and stateful output P-OUT-STATEFUL differ in whether the transmitted value v is stateless or not, and whether it may be subsequently

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

$$\frac{}{\Delta;\, G \vdash k : \{k\}} \quad [\text{V-TOKEN}]$$

$$\frac{}{\Delta;\, G, u : A \vdash u : A} \quad [\text{V-LOOKUP}]$$

$$\frac{}{\Delta;\, G \vdash \mathbf{0}} \quad [\text{P-NIL}]$$

$$\frac{\Delta \vdash \bot}{\Delta;\, G \vdash P} \quad [\text{P-INC}]$$

$$\frac{\Delta \vdash A \text{ stateless} \qquad \Delta;\, G, u : Ch(S) \vdash v : A \qquad \Delta \vdash S \xrightarrow{!A} T \qquad \Delta;\, G, u : Ch(T) \vdash P}{\Delta;\, G, u : Ch(S) \vdash u!v.P} \quad [\text{P-OUT-STATELESS}]$$

$$\frac{\Delta \vdash S \xrightarrow{!A} T \qquad \Delta;\, G, u : Ch(T) \vdash P}{\Delta;\, G, u : Ch(S), v : A \vdash u!v.P} \quad [\text{P-OUT-STATEFUL}]$$

$$\frac{\Delta, a, t, S \xrightarrow{?a} t;\, G, u : Ch(t), x : a \vdash P}{\Delta;\, G, u : Ch(S) \vdash u?x.P} \quad [\text{P-INP}]$$

$$\frac{\Delta;\, G \vdash u : \{k\},\, v : \{\ell\} \qquad \Delta, k = \ell;\, G \vdash P}{\Delta;\, G \vdash [u{=}v]P} \quad [\text{P-MATCH}]$$

$$\frac{\Delta;\, G \vdash u : \{k\},\, v : \{\ell\} \qquad \Delta, k \neq \ell;\, G \vdash P}{\Delta;\, G \vdash [u{\neq}v]P} \quad [\text{P-MISMATCH}]$$

$$\frac{\Delta;\, G, n : Ch(S), \overline{n} : Ch(\mathsf{dual}\, S) \vdash P}{\Delta;\, G \vdash (\boldsymbol{\nu} n)P} \quad [\text{P-NEW}]$$

$$\frac{\Delta;\, G \vdash P \qquad \Delta;\, G \vdash Q}{\Delta;\, G \vdash P{+}Q} \quad [\text{P-PLUS}]$$

$$\frac{\Delta;\, G \vdash G_1 \oplus G_2 \qquad \Delta;\, G_1 \vdash P \qquad \Delta;\, G_2 \vdash Q}{\Delta;\, G \vdash P \,|\, Q} \quad [\text{P-PAR}]$$

$$\frac{\Delta;\, G_1 \vdash P \qquad \forall u \in dom(G_1).\ \Delta \vdash G_1(u) \text{ stateless}}{\Delta;\, G_1, G_2 \vdash {*}P} \quad [\text{P-REP}]$$

Fig. 11. Value and process type assignment.

used by the sender. Both output rules allow type A and session T expressions that need not be variables.

In contrast, the input rule P-INP introduces fresh type a and session type t variables for the unknown type of the communicated value and the session type of channel u after input. Both variables are added to the context. In addition, a labelled-transition constraint $S \xrightarrow{?a} t$ is added. Different uses of such transition hypotheses are given in examples in the remainder of the paper. Input typing requires that P be typable for any input transition of S – in the extreme case, the judgment $\varnothing\,;\, n : Ch(\mathsf{end}) \vdash n?x.P$ holds for any P (subject to restrictions on the free values of P) since end has no transitions. This freedom in input typing, in conjunction with deduction, is key to our polymorphic session typing.

The matching P-MATCH and mismatching P-MISMATCH rules require that P be typed under an additional (in)equation constraint.

The P-NEW rule creates the two endpoints of a channel, with dual sessions.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

In the P-PAR rule, only values with stateless types may occur on both sides of parallel composition, other values may appear on at most one side. Similarly, only values with stateless types are permitted under replication by P-REP.

The rules P-NIL and P-REP admit weakening of the process context. We discuss the effect of weakening in Section 6.

The rules in Figure 11 do not describe the deductions that can occur from Δ, because these deductions use Coq's logic. For example, we do not include rules to instantiate variables in the context Δ. Instead, variable instantiation relies upon Coq's application rule for the dependent function space (representing universal quantification). The only rule that we include explicitly, for illustrative purposes, is P-INC. This rule allows any process to be typed (subject to the conditions on free values described above) when the context Δ is inconsistent. Through the shallow encoding of constraints in Coq, this rule corresponds to the Coq elimination rule for falsity. We see the use of P-INC to detect dead code in Example 9 below.

As an example of the use of deduction, we observe that the inversion principles from Figure 10 can be used when the proposition Φ contains a process typing judgement. This permits case analysis upon session types when typing processes. For example, consider a process of the form $u?x.P$. If the value u has channel type of the form $Ch(S_1 + S_2)$, then the typing of the subprocess P occurs in a context where an input-labelled transition from $S_1 + S_2$ is assumed. The typing derivation for P may then branch to a typing derivation of P under the assumption that the transition is from S_1, and to a second typing derivation of P under the assumption that the transition is from S_2. This branching arises from the inversion principle for the union session type *union_inv*. We expand on this idea in the next example.

Example 9 (negation process). The behaviour of a channel that implements negation can be described by the following session type, where we assume distinct tokens tt and ff representing boolean values.

$$(?\{\mathsf{tt}\}.!\{\mathsf{ff}\}.\mathsf{end}) + (?\{\mathsf{ff}\}.!\{\mathsf{tt}\}.\mathsf{end})$$

For simplicity, we consider a partial implementation of the negation process that responds to receipt of token tt on channel n by sending the token ff. The partial implementation, in composition with a process to send tt, is:

$$(n?x.\,[x{=}\mathsf{tt}]\,n!\mathsf{ff}.\mathbf{0}) \mid \overline{n}!\mathsf{tt}.\mathbf{0}$$

Typing this process with the negation session type given above demonstrates the importance of matching to justify the output of ff after receiving tt. We use the following abbreviations for session types S_{ff} and S_{tt}, and a process P.

$$\begin{aligned} S_{\mathsf{ff}} &\triangleq !\{\mathsf{ff}\}.\mathsf{end} \\ S_{\mathsf{tt}} &\triangleq !\{\mathsf{tt}\}.\mathsf{end} \\ P &\triangleq [x{=}\mathsf{tt}]\,n!\mathsf{ff}.\mathbf{0} \end{aligned}$$

Figure 12 contains a typing derivation for the left-hand subprocess $(n?x.\,[x{=}\mathsf{tt}]\,n!\mathsf{ff}.\mathbf{0})$. The labels *union_inv* and *prefix_inv* represent case analysis via inversion principles for

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{tt}=\mathsf{tt}\vdash S_{\mathsf{ff}}\xrightarrow{!\{\mathsf{ff}\}}\mathsf{end}}\,TRPrefix \quad \dfrac{}{\mathsf{tt}=\mathsf{tt};n:Ch(\mathsf{end}),x:\{\mathsf{tt}\}\vdash\mathbf{0}}\,\text{P-NIL}}{\mathsf{tt}=\mathsf{tt};n:Ch(S_{\mathsf{ff}}),x:\{\mathsf{tt}\}\vdash n!\mathsf{ff}.\mathbf{0}}\,\text{P-OUT-STATELESS}}{\emptyset;n:Ch(S_{\mathsf{ff}}),x:\{\mathsf{tt}\}\vdash P}\,\text{P-MATCH}}{a,t,(?\{\mathsf{tt}\}.S_{\mathsf{ff}}\xrightarrow{?a}t);n:Ch(t),x:a\vdash P}\,prefix_inv \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{tt}=\mathsf{ff};n:Ch(S_{\mathsf{tt}}),x:\{\mathsf{ff}\}\vdash n!\mathsf{ff}.\mathbf{0}}\,\text{P-INC}}{\emptyset;n:Ch(S_{\mathsf{tt}}),x:\{\mathsf{ff}\}\vdash P}\,\text{P-MATCH}}{a,t,(?\{\mathsf{ff}\}.S_{\mathsf{tt}}\xrightarrow{?a}t);n:Ch(t),x:a\vdash P}\,prefix_inv}{a,t,((?\{\mathsf{tt}\}.S_{\mathsf{ff}})+(?\{\mathsf{ff}\}.S_{\mathsf{tt}}))\xrightarrow{?a}t);n:Ch(t),x:a\vdash P}\,union_inv$$
$$\dfrac{}{\emptyset;n:Ch((?\{\mathsf{tt}\}.S_{\mathsf{ff}})+(?\{\mathsf{ff}\}.S_{\mathsf{tt}}))\vdash n?x.P}\,\text{P-INP}$$

Fig. 12. Type assignment derivation for $(n?x.\,[x{=}\mathsf{tt}]n!\mathsf{ff}.\mathbf{0})$.

$$\dfrac{\dfrac{}{\emptyset\vdash \mathsf{dual}\,((?\{\mathsf{tt}\}.S_{\mathsf{ff}})+(?\{\mathsf{ff}\}.S_{\mathsf{tt}}))\xrightarrow{!\{\mathsf{tt}\}}\mathsf{dual}\,S_{\mathsf{ff}}}\,\text{DEDN3} \quad \dfrac{}{\emptyset;\overline{n}:Ch(\mathsf{dual}\,S_{\mathsf{ff}})\vdash\mathbf{0}}\,\text{P-NIL}}{\emptyset;\overline{n}:Ch(\mathsf{dual}\,((?\{\mathsf{tt}\}.S_{\mathsf{ff}})+(?\{\mathsf{ff}\}.S_{\mathsf{tt}})))\vdash\overline{n}!\mathsf{tt}.\mathbf{0}}\,\text{P-OUT-STATEFUL}$$

Fig. 13. Type assignment derivation for $(\overline{n}!\mathsf{tt}.\mathbf{0})$.

the union and prefix session type constructors: the two branches arise from hypotheses of *union_inv*. We do not display the equational simplification that is performed when using these inversion principles. The label *TRPrefix* indicates use of the definition of labelled transitions for the prefix session type. P-INC demonstrates the use of an inconsistent constraint $\mathsf{tt}=\mathsf{ff}$ to typecheck an output $n!\mathsf{ff}.\mathbf{0}$ that is not permitted by the session typing $n:Ch(S_{\mathsf{tt}})$, i.e. the output process is disregarded when ff is read. The type derivation for $\overline{n}!\mathsf{tt}.\mathbf{0}$ is shown in Figure 13.

The derivations in Figures 12 and 13 can be combined using P-PAR to type the parallel composition as follows:

$$\varnothing\,;\,n:Ch((?\{\mathsf{tt}\}.S_{\mathsf{ff}})+(?\{\mathsf{ff}\}.S_{\mathsf{tt}})),\,\overline{n}:Ch(\mathsf{dual}\,((?\{\mathsf{tt}\}.S_{\mathsf{ff}})+(?\{\mathsf{ff}\}.S_{\mathsf{tt}})))\vdash (n?x.\,[x{=}\mathsf{tt}]n!\mathsf{ff}.\mathbf{0})\,|\,\overline{n}!\mathsf{tt}.\mathbf{0}$$

Finally, since the session types of n and $\overline{n}$ are dual, they can be hidden, with the resulting typing:

$$\varnothing\,;\vdash(\boldsymbol{\nu}n)((n?x.\,[x{=}\mathsf{tt}]n!\mathsf{ff}.\mathbf{0})\,|\,\overline{n}!\mathsf{tt}.\mathbf{0}).$$

The type assignment for union and prefixing in Example 9 can be extended to the encodings of label selection and branching from Section 2.3. For tokens l_1, l_2 and session types S_1, S_2, define the binary label branching $\&\{l_1{:}S_1,l_2{:}S_2\}$ and label selection $\oplus\{l_1{:}S_1,l_2{:}S_2\}$ session types using a tagged union encoding:

$$\&\{l_1{:}S_1,l_2{:}S_2\}\triangleq(?\{l_1\}.S_1)+(?\{l_2\}.S_2)$$
$$\oplus\{l_1{:}S_1,l_2{:}S_2\}\triangleq\mathsf{dual}\,(\&\{l_1{:}S_1,l_2{:}S_2\}).$$

The label selection session type $\oplus\{l_1{:}S_1,l_2{:}S_2\}$ has the same labelled transitions as the session type $(!\{l_1\}.S_1)+(!\{l_2\}.S_2)$. However, they are distinct session types, and not interchangeable. We discuss subtyping in Section 6.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

Typing rules can be derived for the encodings of label selection and label branching processes given in Section 2.3:

$$\frac{\Delta\,;\,G,\,u:Ch(S_i)\vdash P}{\Delta\,;\,G,\,u:Ch(\oplus\{l_1:S_1,l_2:S_2\})\vdash u\lhd l_i.P}\quad[\text{P-SELECT}_i]\quad(i\in\{1,2\})$$

$$\frac{\Delta\,;\,G,\,u:Ch(S_i)\vdash P_i\quad(i\in\{1,2\})\qquad l_1\neq l_2\qquad G(error)=Ch(\mathsf{end})}{\Delta\,;\,G,\,u:Ch(\&\{l_1:S_1,l_2:S_2\})\vdash u\rhd\{l_1:P_1\,[]\,l_2:P_2\}}\quad[\text{P-BRANCH}].$$

These derived typing rules closely resemble the label selection and branching rules in Honda *et al.* (1998), modulo notational differences.

Derivation of the label selection rules is straightforward, following the type derivation of Figure 13. We can then rewrite the conclusion of Figure 13 as:

$$\varnothing\,;\,\overline{n}:Ch(\oplus\{\mathsf{tt}:S_{\mathsf{ff}},\mathsf{ff}:S_{\mathsf{tt}}\})\vdash\overline{n}\lhd\mathsf{tt}.\mathbf{0}$$

The derived typing rule for label branching uses ingredients found in the type derivation of Figure 12, but there are several subtleties that highlight the deduction used in typing processes.

First, $l_1\neq l_2$ is required. To see why, suppose $l_1=l_2$, so that the session type $\&\{l_1:S_1,l_2:S_2\}$ has input transitions labelled with l_1 to S_1 and S_2. In this situation, matching against l_1 or l_2 provides no information. As a consequence, the extra hypotheses $\Delta\,;\,G,\,u:Ch(S_2)\vdash P_1$ and $\Delta\,;\,G,\,u:Ch(S_1)\vdash P_2$ would have to be added to P-BRANCH.

Second, the type system's focus on controlling output rather than input does not affect the derived rule P-BRANCH. However, as demonstrated in Example 9, a well-typed process need not have cases for every branch of a session type $\&\{l_1:S_1,l_2:S_2\}$. Moreover, a well-typed process may have unused additional cases, assuming a generalization from binary labelling to n-ary labelling. The reason that P-BRANCH closely resembles the label branching rule of Honda *et al.* (1998) is that the initial input has to be known in order to determine which of the continuation session types S_1 or S_2 is used, in turn because those session types may permit output (which is controlled).

Finally, the introduction of the channel *error* in the encoding of the label branching process requires *error* to be in the domain of G. The session type associated with *error* is not forced, but we choose $G(error)=Ch(\mathsf{end})$. The session type end is stateless, and so can be freely duplicated across different occurrences of label branching. In addition, the session type end does not permit any output. Thus, if an incorrect token $k\notin\{l_1,l_2\}$ can be read on u, the output on *error* seen in the reduction sequence in Section 2.3 is immediately seen to be erroneous. In particular, it is not necessary to know whether the session type of u allows receipt of k to know that an output on *error* is erroneous. To rule out the possibility of output on *error*, recall that the encoding of $(u\rhd\{l_1:P_1\,[]\,l_2:P_2\})$ uses the mismatching tests $([x{\neq}l_1]\,[x{\neq}l_2]\,error!\star)$. Thus the process $error!\star$ is typechecked in an environment where x is known not to be l_1 or l_2, but must also be one of those tokens because of the possible input transitions of $\&\{\mathsf{tt}:S_{\mathsf{ff}},\mathsf{ff}:S_{\mathsf{tt}}\}$. This contradiction allows the typing of $error!\star$.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{t \vdash \mathsf{dual}\,(\mathsf{sink}) \xrightarrow{!Ch(t)} \mathsf{dual}\,(\mathsf{sink})}\,\textsc{dedn2}}{t;\, rep{:}Ch(\mathsf{sink}),\, \overline{rep}{:}Ch(\mathsf{dual}\,(\mathsf{sink})),\, lft{:}Ch(t) \vdash \overline{rep}\,!\,lft}\,\textsc{p-out-stateful}}{s, a, t, s \xrightarrow{?a} t;\, rep{:}Ch(\mathsf{sink}),\, \overline{rep}{:}Ch(\mathsf{dual}\,(\mathsf{sink})),\, lft{:}Ch(t),\, x{:}a \vdash \overline{rep}\,!\,lft}\,\textsc{weaken}}{s;\, rep{:}Ch(\mathsf{sink}),\, \overline{rep}{:}Ch(\mathsf{dual}\,(\mathsf{sink})),\, lft{:}Ch(s) \vdash lft?x\,.\,\overline{rep}\,!\,lft}\,\textsc{p-inp}}{a, t, \mathsf{sink} \xrightarrow{?a} t;\, rep{:}Ch(t),\, \overline{rep}{:}Ch(\mathsf{dual}\,(\mathsf{sink})),\, lft{:}a \vdash lft?x\,.\,\overline{rep}\,!\,lft}\,\textsc{dedn1}}{\emptyset;\, rep{:}Ch(\mathsf{sink}),\, \overline{rep}{:}Ch(\mathsf{dual}\,(\mathsf{sink})) \vdash rep?lft\,.\,lft?x\,.\,\overline{rep}\,!\,lft}\,\textsc{p-inp}$$

Fig. 14. Type assignment derivation for polymorphic sink process.

With the derived typing rule P-BRANCH, a full implementation of the negation process is then defined and typed as:

$$\varnothing\,;\, n : Ch(\&\{\mathsf{tt}{:}S_{\mathsf{ff}}, \mathsf{ff}{:}S_{\mathsf{tt}}\}) \vdash n \rhd \{\mathsf{tt} : n\,!\,\mathsf{ff}\,.\,\mathbf{0} \,[]\, \mathsf{ff} : n\,!\,\mathsf{tt}\,.\,\mathbf{0}\}.$$

4. Examples

We present a series of examples to illustrate session polymorphism. The typechecking examples in this section have been verified in Coq and their proofs are available online (Goto *et al.* 2011).

4.1. *Recursive procedures*

We define a process with a polymorphic session type that discards data on a channel, and provide syntactic sugar for recursive procedure definitions.

Example 10 (polymorphic sink process). We return to the sink example from Section 1.1. Recall that the sink process repeatedly reads input from a channel, say u. The simplest process with the required functionality is the replicated input $(*u?x)$, but non-linear use of u cannot be typed, except when u has a stateless type. Thus, we define the sink process to be:

$$(\boldsymbol{\nu} rep)((*rep?lft\,.\,lft?x\,.\,\overline{rep}\,!\,lft) \mid \overline{rep}\,!\,u).$$

This process uses a stateless channel rep to pass the (not necessarily stateless) channel u linearly between occurrences of the replicated subprocesses. Each replicated subprocess reads one message from u and then forwards it on rep again.

To construct the type derivation we use the session type sink from Figure 9. The name rep is assigned type $Ch(\mathsf{sink})$, and can thus appear under the replication operator because sink is stateless. Similarly for $\overline{rep}$ and $Ch(\mathsf{dual\ sink})$.

It follows from P-OUT-STATEFUL and P-NIL that the right-hand subprocess may be typed as:

$$s\,;\, u : Ch(s),\, \overline{rep} : Ch(\mathsf{dual}\,(\mathsf{sink})) \vdash \overline{rep}\,!\,u.$$

The left-hand subprocess is typed as shown in Figure 14 (we omit the subderivation for the $\mathbf{0}$ process). DEDN1 indicates application of inversion principles derived from the transition definitions for fun and sink respectively (and substitution). For example, the first input yields hypothesis $\mathsf{sink} \xrightarrow{?a} t$, from which we deduce $t = \mathsf{sink}$ and $a = Ch(s)$, for some session type s. Note that session polymorphism in the sink session type is used at

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

the output transition DEDN2, to justify that $\overline{rep}$ can be sent a channel of type $Ch(t)$, which was originally received on *rep* at type $Ch(s)$. WEAKEN is a derived property of the type system that we discuss in Section 5. Using P-PAR and P-REP, the entire process is typed as:

$$s\,;\,u : Ch(s) \vdash (\boldsymbol{\nu} rep)((*rep?lft\,.\,lft?x\,.\,\overline{rep}\,!\,lft) \mid \overline{rep}\,!\,u).$$

The corresponding Coq typing statement for this `sink` process is:

```
forall s:session, (CTX.add(Nm (Free "u"), TChannel s) CTX.empty) |-p sink
```

The subprocess $(\overline{rep}\,!\,u)$ that initiates the recursive reading of u is a simple use of the service. We can instantiate the session type parameter s with a session, e.g. the session S_1 defined by:

$$S_1 = ?\{k_1\}.((?\{k_2\}.\mathsf{end}) + (!\,\{k_3\}.\mathsf{end})).$$

In our Coq formalization, session type instantiation is simply instantiation of universal quantified parameters, hence we have:

$$;\, u : Ch(S_1) \vdash (\boldsymbol{\nu} rep)((*rep?lft\,.\,lft?x\,.\,\overline{rep}\,!\,lft) \mid \overline{rep}\,!\,u).$$

Since $\overline{rep}$ has a stateless type, it may be duplicated freely, and there can be multiple transmissions on $\overline{rep}$ that do not arise from the replicated subprocess. For example, define the session S_2 by:

$$S_2 = ?\{k_4\}.?\{k_5\}.\mathsf{end}$$

Then, we can similarly establish typing of the process that consumes all of the messages that can be read upon the channels u_1 and u_2 with distinct session types S_1 and S_2 respectively:

$$;\, u_1 : Ch(S_1),\, u_2 : Ch(S_2) \vdash (\boldsymbol{\nu} rep)((*rep?lft\,.\,lft?x\,.\,\overline{rep}\,!\,lft) \mid \overline{rep}\,!\,u_1 \mid \overline{rep}\,!\,u_2).$$

Note that a single subprocess is responsible for consuming messages on u_1 and u_2, and this reuse relies upon the polymorphism in the session type (`dual sink`) of $\overline{rep}$ (see the discussion of `sink` in Section 3.2). □

The direct use of replication is tedious when passing multiple channels with related session types, so we introduce syntactic sugar for recursive procedure definitions, following Milner (1991). For example, the recursive definition of a process functionally equivalent to the process of Example 10 is:

$$s\,;\,u : Ch(s) \vdash \mathbf{proc}\ rep(lft) = lft?x\,.\,rep(lft)\ \mathbf{in}\ rep(u).$$

We extend the syntax of processes with syntactic sugar for parameterized procedure invocations and declarations with multiple parameters.

$$P, Q\ ::=\ \cdots \mid f(u_1, \ldots, u_n) \mid \mathbf{proc}\ f(x_1, \ldots, x_n) = P\ \mathbf{in}\ Q$$

These abbreviations can be expanded as follows:

$$f(u_1, \ldots, u_n) \triangleq (\boldsymbol{\nu} c)\left(\overline{f}\,!\,c\,.\ \overline{c}\,!\,u_1\ \ldots\ \overline{c}\,!\,u_n\right),$$
$$\mathbf{proc}\ f(x_1, \ldots, x_n) = P\ \mathbf{in}\ Q \triangleq (\boldsymbol{\nu} f)\left((*f?c\,.\ c?x_1\ \ldots\ c?x_n\,.\ P) \mid Q\right).$$

In these definitions, a fresh channel c is created for each procedure call. The channel c is passed to the replicated procedure body using the channel $\bar{f}$. The channel $\overline{c}$ is then used

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

to send the arguments $u_1, \ldots, u_n$ to the replicated procedure body, which reads them using c. The channel endpoints c and $\bar{c}$ are discarded after carrying the parameters for one procedure call. The session type fwd, discussed in Section 3.2, is a typical session type for the channel c. In contrast, the channel f must always be assigned a stateless session type because it appears in a replicated subprocess.

4.2. *Forwarding processes*

We now consider a sequence of simple examples with polymorphic session types in which data is generically forwarded.

The first example builds on Example 6 to type a process that continually forwards messages in one direction. The unidirectional nature of the forwarding process is captured in its interface by a session type that functions as a one-way check valve.

Example 11 (polymorphic unidirectional forwarding process). Recall that the transitions of sessions (fun a) and fwd are defined in Figure 9 as follows:

$$\begin{aligned} &| \; \mathit{TRFun} \;\; : \;\; \forall\, a : \mathit{type},\; (\mathsf{fun}\; a \xrightarrow{?a} \mathsf{fun}\; a), \\ &| \; \mathit{TRFwd} \;\; : \;\; \forall\, s : \mathit{session},\; (\mathsf{fwd} \xrightarrow{?Ch(s)\; ?Ch(\mathsf{dual}\; s)} \mathsf{end}). \end{aligned}$$

The process that forwards messages unidirectionally from channel u to v can be defined as:

$$s\,;\, u : Ch(s), v : Ch(\mathsf{dual}\; s) \vdash \mathbf{proc}\; \mathit{fwd}(\mathit{lft}, \mathit{rht}) = \mathit{lft}?x\,.\,\mathit{rht}!x\,.\,\mathit{fwd}(\mathit{lft}, \mathit{rht})\; \mathbf{in}\; \mathit{fwd}(u, v).$$

In typing, we assign $\mathit{fwd} : Ch(\mathsf{fun}\; (Ch(\mathsf{fwd})))$. The session type (fun Ch(fwd)) allows a sequence of inputs that all have the fixed type Ch(fwd). Thus, *fwd* can repeatedly transmit single-use channels of type Ch(fwd). In the encoding given above, a fresh channel c with session type fwd is allocated for each call, including the recursive calls in the body of the process definition. The session type fwd is used for single-use channels that convey the parameters *lft* and *rht*, and then stop. As in Example 6, the key to successful typing lies in the fact that the two channels are constrained by the fwd transition to have dual session types s and (dual s) for some session type s. An input on *lft* creates an input transition hypothesis for s, from this input transition we deduce an output transition for (dual s), and the output on *rht* is justified by the output transition for (dual s).

The forwarder may be instantiated multiple times at different types without duplicating the code. For example, the following process can be typed in our system, where k_i are distinct.

$$\begin{aligned} \mathbf{proc}\;\; \mathit{fwd}(\mathit{lft}, \mathit{rht}) = \mathit{lft}?x\,.\,\mathit{rht}!x\,.\,\mathit{fwd}(\mathit{lft}, \mathit{rht}) \;\; \mathbf{in} \;\;\; & (\boldsymbol{\nu} n_1, n_2)(\mathit{fwd}(n_1, n_2) \,|\, n_1!k_1\,.\,n_1!k_2\,.\,\mathbf{0} \,|\, \cdots) \\ & |\, (\boldsymbol{\nu} n_3, n_4)(\mathit{fwd}(n_3, n_4) \,|\, n_3!k_3\,.\,n_3!k_4\,.\,\mathbf{0} \,|\, \cdots) \end{aligned}$$

Note that the two uses of the forwarder have disjoint session types. As with function definitions and activations, there is only one definition, but as evaluation proceeds there will be two activations of *fwd*, each specialized to a particular session type.

The typing for *fwd* demonstrates that unidirectional forwarding can be applied to any session type. However, it is possible to develop a more informative typing that also exhibits session polymorphism. First, define a new session type (checkvalve s) that has all of the

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

input transitions of s but none of the output transitions. Its labelled transitions are:

$$\mid \mathit{TRCheckValve} \;:\; \forall\, s\; a\; s',\; (s \xrightarrow{?a} s') \;\Rightarrow\; (\mathsf{checkvalve}\; s \xrightarrow{?a} \mathsf{checkvalve}\; s').$$

The unidirectional forwarder can then be typed with (checkvalve s) as the session type for u. In this typing, the session type for *fwd* must be modified to reflect the different session type of u.

$$s\,;\, u : \mathit{Ch}(\mathsf{checkvalve}\; s), v : \mathit{Ch}(\mathsf{dual}\; s) \vdash \mathbf{proc}\; \mathit{fwd}(\mathit{lft}, \mathit{rht}) = \mathit{lft}?x\,.\,\mathit{rht}!x\,.\,\mathit{fwd}(\mathit{lft}, \mathit{rht})\; \mathbf{in}\; \mathit{fwd}(u, v)$$

This is a useful typing because the checkvalve session type makes it easy to see that the process does not write to channel u.

One might wonder whether checkvalve can be used in the session type of v. Consider how checkvalve could be used with dual . Unsurprisingly, using $v : \mathit{Ch}(\mathsf{checkvalve}\,(\mathsf{dual}\; s))$ does not allow the unidirectional forwarder to be typed, because checkvalve blocks any output transitions for (dual s), so all writes to v are illegal (unless in an inconsistent context). The other possibility $v : \mathit{Ch}(\mathsf{dual}\; (\mathsf{checkvalve}\; s))$ uses the dual of the checkvalve session type, and so blocks all input transitions, but not output transitions. Since the unidirectional forwarder does not perform input on v, the process can be typed when $v : \mathit{Ch}(\mathsf{dual}\; s)$ is replaced with $v : \mathit{Ch}(\mathsf{dual}\; (\mathsf{checkvalve}\; s))$. □

We briefly comment on some variations of the unidirectional forwarder. First we note that there are several ways to define sessions for any given protocol. Consider a lossy forwarder that drops every other message.

$$\mathbf{proc}\; \mathit{alternate}(\mathit{lft}, \mathit{rht}) = \mathit{lft}?x\,.\,\mathit{lft}?y\,.\,\mathit{rht}!x\,.\,\mathit{alternate}(\mathit{lft}, \mathit{rht})\; \mathbf{in}\; \ldots$$

We can assign type $\mathit{alternate} : \mathit{Ch}(\mathsf{fun}\; (\mathit{Ch}(\mathsf{alternateSub})))$, using the session constructor altSub that elides inputs. Here *lft* is assigned type $\mathit{Ch}(s)$ and *rht* is assigned type $\mathit{Ch}(\mathsf{dual}\; (\mathsf{altSub}\; s))$.

$$\mid \mathit{TRAlternateSub} \;:\; \forall\, s : \mathit{session},\; (\mathsf{alternateSub} \xrightarrow{?\mathit{Ch}(s)\; ?\mathit{Ch}(\mathsf{dual}\; (\mathsf{altSub}\; s))} \mathsf{end})$$
$$\mid \mathit{TRAltSub} \;:\; \forall\, s\; a\; t\; b\; u,\; (s \xrightarrow{?a} t) \Rightarrow (t \xrightarrow{?b} u) \Rightarrow (\mathsf{altSub}\; s \xrightarrow{?a} \mathsf{altSub}\; u)$$

We can also assign type $\mathit{alternate} : \mathit{Ch}(\mathsf{fun}\; (\mathit{Ch}(\mathsf{alternateAdd})))$, using the session constructor altAdd that introduces inputs. Here *lft* is assigned type $\mathit{Ch}((\mathsf{altAdd}\; s))$ and *rht* is assigned type $\mathit{Ch}(\mathsf{dual}\; s)$.

$$\mid \mathit{TRAlternateAdd} \;:\; \forall\, s : \mathit{session},\; (\mathsf{alternateAdd} \xrightarrow{?\mathit{Ch}(\mathsf{altAdd}\; s)\; ?\mathit{Ch}(\mathsf{dual}\; s)} \mathsf{end})$$
$$\mid \mathit{TRAltAdd} \;:\; \forall\, s\; a\; b\; t,\; (s \xrightarrow{?a} t) \Rightarrow (\mathsf{altAdd}\; s \xrightarrow{?a\; ?b} \mathsf{altAdd}\; t)$$

In general, forwarders may multiplex inputs. The following is the simplest such process.

$$\mathbf{proc}\; \mathit{multiplex}(\mathit{lft}_1, \mathit{lft}_2, \mathit{rht}) = \mathit{lft}_1?x\,.\,\mathit{lft}_2?y\,.\,\mathit{rht}!x\,.\,\mathit{rht}!y\,.\,\mathit{multiplex}(\mathit{lft}_1, \mathit{lft}_2, \mathit{rht})\; \mathbf{in}\; \ldots$$

We can assign type $\mathit{multiplex} : \mathit{Ch}(\mathsf{fun}\; (\mathit{Ch}(\mathsf{zip})))$, using the session constructor plex. Here lft_1 is assigned type $\mathit{Ch}(s)$, lft_2 is assigned type $\mathit{Ch}(t)$ and *rht* is assigned type $\mathit{Ch}(\mathsf{dual}\; (\mathsf{plex}\; s\; t))$.

$$\mid \mathit{TRMultiplex} \;:\; \forall\, s\; t : \mathit{session},\; (\mathsf{zip} \xrightarrow{?\mathit{Ch}(s)\; ?\mathit{Ch}(t)\; ?\mathit{Ch}(\mathsf{dual}\; (\mathsf{plex}\; s\; t))} \mathsf{end})$$
$$\mid \mathit{TRPlex} \;:\; \forall\, s\; a\; s'\; t\; b\; t',\; (s \xrightarrow{?a} s') \Rightarrow (t \xrightarrow{?b} t') \Rightarrow (\mathsf{plex}\; s\; t \xrightarrow{?a\; ?b} \mathsf{plex}\; s'\; t')$$

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

The next example shows how bidirectional forwarding of messages affects session typing.

Example 12 (polymorphic bidirectional forwarding process). The bidirectional forwarding process can be typed with the context used for the unidirectional forwarding process (without checkvalve).

$$s\,;\,u : Ch(s), v : Ch(\mathsf{dual}\ s) \vdash$$
$$\quad \mathbf{proc}\ bi(lft, rht) = (lft?x\,.\,rht!x\,.\,bi(lft, rht)) + (rht?x\,.\,lft!x\,.\,bi(lft, rht))\ \mathbf{in}\ bi(u, v)$$

As with the unidirectional forwarder, this uses the polymorphic session type $Ch(\mathsf{fun}\ (Ch(\mathsf{fwd})))$ for the channel *bi*.

Unlike the unidirectional forwarding processes, the bidirectional forwarding process cannot be typed when $u : Ch(\mathsf{checkvalve}\ s)$, because output on u is blocked by $(\mathsf{checkvalve}\ s)$. That is:

$$s\,;\,u : Ch(\mathsf{checkvalve}\ s), v : Ch(\mathsf{dual}\ s) \not\vdash$$
$$\quad \mathbf{proc}\ bi(lft, rht) = (lft?x\,.\,rht!x\,.\,bi(lft, rht)) + (rht?x\,.\,lft!x\,.\,bi(lft, rht))\ \mathbf{in}\ bi(u, v).$$

However, there is an exception. When $u : Ch(\mathsf{checkvalve}\ s), v : Ch(\mathsf{dual}\ (\mathsf{checkvalve}\ s))$, it is possible to deduce that there are no input transitions from the session type for v, so the output subprocess $u!x\,.\,bi(u, v)$ is typed trivially. Thus, the bidirectional forwarder can be typed as follows:

$$s\,;\,u : Ch(\mathsf{checkvalve}\ s), v : Ch(\mathsf{dual}\ (\mathsf{checkvalve}\ s)) \vdash$$
$$\quad \mathbf{proc}\ bi(lft, rht) = (lft?x\,.\,rht!x\,.\,bi(lft, rht)) + (rht?x\,.\,lft!x\,.\,bi(lft, rht))\ \mathbf{in}\ bi(u, v).$$

This typing overlaps with the original typing with $u : Ch(s), v : Ch(\mathsf{dual}\ s)$. □

4.3. *The alternating bit protocol*

We next define a polymorphic session typing for interaction via the ABP (Bartlett *et al.* 1969).

The ABP provides reliable transmission over an unreliable channel by tagging messages with an alternating bit, and resending messages until an acknowledgement with the correct tag is received. The existence of a polymorphic session typing allows the ABP to be used at different session types. Moreover, the polymorphic session typing captures the following correctness criterion for the ABP: if messages 1 and 2 are sent via the ABP in that order, then the ABP will not send message 2 out before message 1. This session typing holds despite internal loss and resending of messages – each of those actions could easily lead to session types that guarantee very little about message ordering. In other words, the ABP outputs a prefix of its input, and acts as the identity on channels carrying tokens, if the unreliable channel passes enough messages. This example demonstrates the expressiveness of our session typing system (by capturing the ABP's invariant) *and* its modularity (as a type system).

Example 13 (the alternating bit protocol). The ABP is illustrated in Figure 15, and the ABP process is defined in Figure 16. Following Roscoe (1997), the unreliable channel is modelled by a lossy bidirectional forwarding process that discards some messages. Sender

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

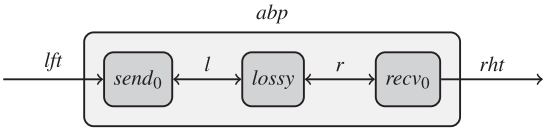


Fig. 15. The alternating bit protocol.

$$
\begin{array}{ll}
\textbf{proc}\ send_i(x,lft,l) = & \\
\quad l!(i,x).send_i(x,lft,l) & \text{Send message } x \text{ (possibly a duplicate)} \\
\quad +l?(1-i).send_i(x,lft,l) & \text{Ignore duplicate acknowledgement } 1-i \\
\quad +l?(i).lft?y.send_{1-i}(y,lft,l)\ \textbf{in} & \text{New acknowledgement } i\text{, send next message } y \text{ from } lft \\
\textbf{proc}\ lossy(l,r) = & \\
\quad l?(z,x).(lossy(l,r)+r!(z,x).lossy(l,r)) & \text{Discard or forward tagged message } (z,x) \\
\quad +r?z.(lossy(l,r)+l!z.lossy(l,r))\ \textbf{in} & \text{Discard or forward acknowledgement } z \\
\textbf{proc}\ recv_i(r,rht) = & \\
\quad r!1-i.recv_i(r,rht) & \text{Send acknowledgement } 1-i \text{ (possibly a duplicate)} \\
\quad +r?(1-i,y).recv_i(r,rht) & \text{Ignore duplicate message } y \\
\quad +r?(i,x).rht!x.recv_{1-i}(r,rht)\ \textbf{in} & \text{New message } x\text{, forward } x \text{ on } rht \text{ and acknowledge} \\
\textbf{proc}\ abp(lft,rht) = (\nu l,r)((lft?x.send_0(x,lft,l)) \mid lossy(\bar{l},r) \mid recv_0(\bar{r},rht))\ \textbf{in}\ abp(lft,rht) &
\end{array}
$$

Fig. 16. Implementation of the alternating bit protocol.

and receiver processes transmit messages from *lft* to *rht*, but can only interact with one another using channels l and r. Unreliable forwarding from l and r is provided by the lossy process. The $abp(lft,rht)$ process represents ABP transmission as the composition of sender, lossy, and receiver processes. The lossy process treats consecutive transmission of a bit and a message as an atomic transmission, so that if one component of the pair is lost, then so is the other. The sender and the receiver processes are parameterized by an alternating bit $i \in \{0,1\}$, where 0 and 1 are tokens. We have used abbreviations for multiple output, $u!(v_1,\ldots,v_n).P \triangleq u!v_1.\ldots u!v_n.P$, multiple input, $u?(x_1,\ldots,x_n).P \triangleq u?x_1.\ldots u?x_n.P$, and token-matching input, $u?(i,\vec{x}).P \triangleq u?y.u?\vec{x}.\,[y{=}i]P$, for $y \notin fv(P)$.

The full ABP also allows duplication of messages on the unreliable channel. We omit duplication for the sake of simplicity, but the session types defined below do accommodate duplication of messages by the unreliable channel – essentially because they are already insensitive to retransmissions by the sender and receiver processes.

Unlike the previous examples, the ABP protocol may resend messages internally to compensate for discarded messages. Such non-linear usage is not safe in general for channel names; we, thus, restrict attention to session types that send only tokens, using the session type $\mathsf{toks}\ s$, indexed by a session s, that is defined by:

$$\mid TRToks\ :\ \forall\ s\ k\ s',\ (s \xrightarrow{?\{k\}} s')\ \Rightarrow\ (\mathsf{toks}\ s \xrightarrow{?\{k\}} \mathsf{toks}\ s').$$

Hiding the process definitions and the channels used to implement them yields a simple *external* interface to the ABP process. The typing for the ABP with this hiding is:

$$s\ ;\ lft : Ch(\mathsf{toks}\ s),\ rht : Ch(\mathsf{dual}\ (\mathsf{toks}\ s)) \vdash abp(lft,rht).$$

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

The interesting part of the ABP typing lies in the session types for the *internal* channels l and r. The key is for the session type to: (1) record whether a message has been acknowledged or not; and (2) allow for duplicate messages until an acknowledgement occurs. The ($\mathsf{nack}_i\ s\ k\ s'$) and ($\mathsf{ack}_i\ s$) session types provide unacknowledged and acknowledged states with respect to the value of the alternating bit i. These session types have transitions defined by:

$$
\begin{array}{l}
\mid TRNack_1 \ : \ \forall\, i\ s\ k\ s',\ s \xrightarrow{?\{k\}} s' \ \Rightarrow\ \mathsf{nack}_i\ s\ k\ s' \xrightarrow{!\{i\}\ !\{k\}} \mathsf{nack}_i\ s\ k\ s' \\
\mid TRNack_2 \ : \ \forall\, i\ s\ k\ s',\ \ \mathsf{nack}_i\ s\ k\ s' \xrightarrow{?\{1-i\}} \mathsf{nack}_i\ s\ k\ s' \\
\mid TRNack_3 \ : \ \forall\, i\ s\ k\ s',\ \ \mathsf{nack}_i\ s\ k\ s' \xrightarrow{?\{i\}} \mathsf{ack}_i\ s' \\
\mid TRAck_1 \ : \ \forall\, i\ s\ k,\ \ \mathsf{ack}_i\ s \xrightarrow{!\{i\}\ !\{k\}} \mathsf{ack}_i\ s \\
\mid TRAck_2 \ : \ \forall\, i\ s,\ \ \mathsf{ack}_i\ s \xrightarrow{?\{i\}} \mathsf{ack}_i\ s \\
\mid TRAck_3 \ : \ \forall\, i\ s\ k\ s',\ s \xrightarrow{?\{k\}} s' \ \Rightarrow\ \mathsf{ack}_i\ s \xrightarrow{!\{1-i\}\ !\{k\}} \mathsf{nack}_{1-i}\ s\ k\ s'
\end{array}
$$

The first two cases for nack_i allow for resending of duplicate messages or receiving old acknowledgements without changing the session type. However, if a new acknowledgement (with the current bit i) is received, then there is a transition to an acknowledged state. In the acknowledged state ack_i, further duplicate messages can be sent, duplicate acknowledgements can be received and new messages with the alternate bit $1-i$ can be sent, resulting in a transition to an unacknowledged state again.

In typing the lossy process, the session types for l and r need not be identical, but must remain consistent. In particular, r may be in an acknowledged state when l is in an unacknowledged state. We formalize this consistency in the session types for single-use channels transmitting l (the first channel received) and r (the second channel):

$$
\begin{array}{l}
\mid TRLossy_1 \ : \ \forall\, i\ s,\ \ \mathsf{lossy} \xrightarrow{?Ch(\mathsf{dual}\ (\mathsf{ack}_i\ s))\ ?Ch(\mathsf{ack}_i\ s)} \mathsf{end} \\
\mid TRLossy_2 \ : \ \forall\, i\ s\ k\ s',\ \mathsf{lossy} \xrightarrow{?Ch(\mathsf{dual}\ (\mathsf{nack}_i\ s\ k\ s'))\ ?Ch(\mathsf{nack}_i\ s\ k\ s')} \mathsf{end} \\
\mid TRLossy_3 \ : \ \forall\, i\ s\ k\ s',\ \mathsf{lossy} \xrightarrow{?Ch(\mathsf{dual}\ (\mathsf{nack}_{1-i}\ s\ k\ s'))\ ?Ch(\mathsf{ack}_i\ s)} \mathsf{end} \\
\mid TRLossy_4 \ : \ \forall\, i\ s\ k\ s',\ \mathsf{lossy} \xrightarrow{?Ch(\mathsf{dual}\ (\mathsf{nack}_i\ s\ k\ s'))\ ?Ch(\mathsf{ack}_i\ s')} \mathsf{end}
\end{array}
$$

Similar conditions must be defined for the single-use channels used to transmit the parameters to the sending and receiving processes. These conditions provide the necessary constraints between the state of the external *lft*, *rht* and the internal l, r.

$$
\begin{array}{l}
\mid TRSend_1 \ : \ \forall\, i\ s\ k\ s',\ s \xrightarrow{?\{k\}} s' \ \Rightarrow\ \mathsf{send}_i \xrightarrow{?\{k\}\ ?Ch(\mathsf{toks}\ s')\ ?Ch(\mathsf{nack}_i\ s\ k\ s')} \mathsf{end} \\
\mid TRSend_2 \ : \ \forall\, i\ s\ k\ s',\ s \xrightarrow{?\{k\}} s' \ \Rightarrow\ \mathsf{send}_i \xrightarrow{?\{k\}\ ?Ch(\mathsf{toks}\ s')\ ?Ch(\mathsf{ack}_{1-i}\ s)} \mathsf{end} \\
\mid TRRecv_1 \ : \ \forall\, i\ s\ k\ s',\ \mathsf{recv}_i \xrightarrow{?Ch(\mathsf{dual}\ (\mathsf{nack}_{1-i}\ s\ k\ s'))\ ?Ch(\mathsf{dual}\ (\mathsf{toks}\ s'))} \mathsf{end} \\
\mid TRRecv_2 \ : \ \forall\, i\ s,\ \mathsf{recv}_i \xrightarrow{?Ch(\mathsf{dual}\ (\mathsf{ack}_{1-i}\ s))\ ?Ch(\mathsf{dual}\ (\mathsf{toks}\ s))} \mathsf{end}
\end{array}
$$

The ABP process, without hiding of internal channels, is then typed with a process context that includes: $send_i : Ch(\mathsf{fun}\ Ch(\mathsf{send}_i))$, $lossy : Ch(\mathsf{fun}\ Ch(\mathsf{lossy}))$, $recv_i : Ch(\mathsf{fun}\ Ch(\mathsf{recv}_i))$.

In this typing for the ABP, the lossy process is not implemented as the parallel composition of a process forwarding from l to r, and another forwarding from r to l. This appears to be necessary to ensure that the lossy forwarder is typed with sufficient knowledge about the acknowledgement state of a channel.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

Finally, the results of Section 5 ensure that if *lft* has session type (toks S) then output on *rht* always conforms to the session type (dual (toks S)), so internal loss or duplication of messages inside the ABP does not compromise external communication.

The Coq statement of typing for the ABP is given on page 469. □

5. Type soundness results

In this section, we state subject reduction and two safety results for the calculus. The first safety result is a runtime safety property ensuring the absence of errors. The second safety result ensures that the traces of interaction on a channel conform to the original session type for the channel. We show several applications of the conformance theorem.

The results have been verified in Coq. The Coq statements are displayed in Figure 17, and their proofs are available online (Goto *et al.* 2011).

5.1. *Auxiliary results*

The type system admits weakening for process typing judgements $\varnothing\,;\, G_1 \vdash P$. Weakening requires a second context G_2 that can be merged with the original context G_1, so that values in both contexts are assigned stateless types. Values with channel type in G_2 need not be assigned the terminal session type end, because our typing rules do not require a channel to be fully used. In contrast, the weakening lemmas of, e.g. Gay (2008), Gay and Hole (2005) and Vasconcelos *et al.* (2010), do require linear channels to have a terminal session type. We discuss this difference further in Section 6.

Lemma 1 (weakening). *If* $\varnothing\,;\, G_1 \vdash P$ *and* $\varnothing\,;\, G \vdash G_1 \oplus G_2$, *then* $\varnothing\,;\, G \vdash P$. □

Similarly, and for the same reason, the strengthening result for process typing judgements does not require channels to have terminal session types.

Lemma 2 (strengthening). *If* $\varnothing\,;\, G_2 \vdash P$, $fval(P) \subseteq dom(G_1)$, *and* $G_1 \subseteq G_2$, *then* $\varnothing\,;\, G_1 \vdash P$. □

Lemma 3 (structural equivalence preserves typing). *If* $P \equiv Q$, *then, for all* G, $\varnothing\,;\, G \vdash P$ *iff* $\varnothing\,;\, G \vdash Q$. □

The proofs of Lemmas 1–3 are routine, using induction over the derivations of typing and structural equivalence judgements. We state these and subsequent results, using the empty context $\varnothing$. The context does not appear explicitly in the corresponding Coq statements (Figure 17) because it represents the Coq hypotheses, in contrast to the process context that does appear explicitly in the Coq statements.

5.2. *Subject reduction*

To ensure that communication on both endpoints of the same channel is well behaved, subject reduction requires that processes are typed in balanced process contexts (Gay and Hole 2005; Yoshida and Vasconcelos 2007), in which dual names are assigned dual session types.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

```
weakening
     : forall (G G1 G2 : ctx) (P : proc),
         (G1 |-p P) ->
         (G |-part G1 (+) G2) ->
         G |-p P

strengthening
     : forall (G1 G2 : ctx) (P : proc),
         (G2 |-p P) ->
         CTX.Subset G1 G2 ->
         free_values_in_context G1 P ->
         G1 |-p P

struct_equiv_preserves_typed
     : forall (P Q : proc) (G : ctx),
         (P == Q) ->
         G |-p P <-> G |-p Q

ctx_preservation_preserves_balanced
     : forall (G1 G2 : ctx) (alpha : obs),
         (G1 |-wf) ->
         ctx_preservation G1 G2 alpha ->
         balanced G1 ->
         balanced G2

subject_reduction
     : forall (G1 : ctx) (P Q : proc) (alpha : obs),
         reduction P Q alpha ->
         (G1 |-p P) ->
         balanced G1 ->
         exists G2 : ctx,
           (G2 |-p Q) /\ ctx_preservation G1 G2 alpha

runtime_safety
     : forall (G : ctx) (P : proc),
         (G |-p P) ->
         balanced G ->
         ~ error P

conformance
     : forall (G : ctx) (P Q : proc) (f : free_id) (s : session) (alphas : list obs),
         reductions P Q alphas ->
         (G |-p P) ->
         balanced G ->
         CTX.In (Nm (Free f), TChannel s) G ->
         traces f (project f alphas) s
```

Fig. 17. Type soundness results verified in Coq.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

Formally, we say that a process context is balanced, written $\mathsf{bal}(G)$, whenever:

— If $G(u) = A$, then u is a name or coname and A is a channel type (i.e. not a singleton type).
— If $G(n) = Ch(S)$ and $G(\overline{n}) = Ch(T)$, then $S = \mathsf{dual}\ T$ or $T = \mathsf{dual}\ S$.

As in prior work for session types (ibid.), subject reduction states that reduction from a well-typed process P to Q, yields a process that is well-typed in another process context. As shown in Yoshida and Vasconcelos (2007), subject reduction and subsequent safety results need not depend on the existence of a relationship between the first and second process contexts. However, it is useful to know that the process context is evolving in harmony with the interaction that occurs in processes, and we make use of this property in our conformance result (Theorem 7).

The relationship between the first and second process context specifies changes to session types that reflect interaction that occurred in the reduction. To formalize this relationship, define the function $\mathsf{obst}(A)$ that maps a type to an observable value by hiding channels as follows (cf. the function $\mathsf{obsv}(u)$, used in the reduction semantics Section 2.2, that maps a value to an observable value):

$$\mathsf{obst}(\{k\}) \triangleq k,$$
$$\mathsf{obst}(Ch(S)) \triangleq \star$$

Next, we define a function $\mathsf{obsm}(u, M)$ that maps a name or coname u, and a message M, to an observation on a name. If given a name, the direction of the message is retained. If given a coname, the direction of the message is reversed. The function $\mathsf{obsm}(u, M)$ is defined by:

$$\mathsf{obsm}(n, ?A) \triangleq n?\mathsf{obst}(A)$$
$$\mathsf{obsm}(n, !A) \triangleq n!\mathsf{obst}(A)$$
$$\mathsf{obsm}(\overline{n}, ?A) \triangleq n!\mathsf{obst}(A)$$
$$\mathsf{obsm}(\overline{n}, !A) \triangleq n?\mathsf{obst}(A)$$

Finally, we define the context preservation relation $G_1 \xrightarrow{\alpha} G_2$ between process contexts before, G_1, and after, G_2, a reduction annotated with observation α. The context preservation relation requires session types for dual names to evolve according to the labelled transitions for the initial session types. The context preservation relation is defined inductively by the rules in Figure 18. This context preservation relation is close in spirit to the relationship between initial and final contexts in the subject reduction result of Gay and Hole (2005) (Theorem 1, type preservation). The difference is that our definition captures all interaction, as opposed to capturing only interaction via label selection and branching.

Since the context preservation relation advances the session types of dual channels in tandem, it preserves the property that a process context is balanced.

Lemma 4 (balanced process contexts). *If* $\mathsf{bal}(G_1)$ *and* $G_1 \xrightarrow{\alpha} G_2$, *then* $\mathsf{bal}(G_2)$. □

Note that the corresponding Coq statement (in Figure 17) for Lemma 4 requires that the process context is well-formed, written `G1 |-wf`. However, in Section 3.5, we already

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

$$\frac{}{G \xrightarrow{\tau} G}$$

$$\frac{S \xrightarrow{M} T \qquad u, v \text{ are dual names}}{G, u{:}Ch(S), v{:}Ch(\mathsf{dual}\, S) \xrightarrow{\mathsf{obsm}(u,M)} G, u{:}Ch(T), v{:}Ch(\mathsf{dual}\, T)}$$

Fig. 18. Context preservation relation.

$$\frac{u \text{ is not a channel}}{\lightning\, u!v.P} \qquad \frac{u \text{ is not a channel}}{\lightning\, u?x.P} \qquad \frac{u \text{ or } v \text{ is not a token}}{\lightning\, [u{=}v]P} \qquad \frac{u \text{ or } v \text{ is not a token}}{\lightning\, [u{\neq}v]P}$$

$$\frac{\lightning\, P}{\lightning\, (\boldsymbol{\nu} n)P} \qquad \frac{\lightning\, P}{\lightning\, P{+}Q} \qquad \frac{\lightning\, Q}{\lightning\, P{+}Q} \qquad \frac{\lightning\, P}{\lightning\, P \mid Q} \qquad \frac{\lightning\, Q}{\lightning\, P \mid Q} \qquad \frac{\lightning\, P}{\lightning\, *P}$$

Fig. 19. Error.

restricted our attention in this paper to well-formed process contexts, and so we do not include it in Lemma 4. The other Coq statements do not explicitly state that contexts are well-formed because it is implied by process typing.

The subject reduction theorem then states that reduction from a process that is well-typed in a balanced process context leads to another well-typed process. Moreover, the two process contexts, and the observation from the reduction, are related by context preservation.

Theorem 5 (subject reduction). *If* $P \xrightarrow{\alpha} Q$ *and* $\varnothing\,;\, G_1 \vdash P$*, for some context satisfying* $\mathsf{bal}(G_1)$*, then there exists a context* G_2 *such that* $\varnothing\,;\, G_2 \vdash Q$ *and* $G_1 \xrightarrow{\alpha} G_2$*.* □

5.3. *Runtime safety*

In our language, shape errors arise in programs that confuse tokens and channels. Tokens may be compared, but may not be used for input or output; channels have complementary capabilities. In Figure 19, we formally define errors as a predicate on processes (notation $\lightning\, P$). The use of session polymorphism, the subtle behaviour of input, and subsequent deduction makes the absence of shape errors less than obvious. Nevertheless, the runtime safety theorem asserts that well-typed processes do not give rise to immediate shape errors.

Theorem 6 (runtime safety). *If* $\varnothing\,;\, G \vdash P$ *for some context satisfying* $\mathsf{bal}(G)$ *then* $\neg(\lightning\, P)$.□

The proof is a straightforward induction. For the base cases, note that balanced environments do not contain free variables.

In Honda *et al.* (1998), a process of the form $(n!M.P) \mid (n!M'.Q)$ is an error, and, more generally, the parallel composition of two processes offering communication on a common channel is an error if the composition is not a redex. To support freedom from the errors of the form in Honda *et al.* (1998), it would be necessary for us to distinguish linear, stateful channels from stateless channels in processes. This is because stateless channels can be used in ways that would constitute an error for stateful channels, e.g. it is not an error for two processes in parallel to send on a common stateless channel. We leave this form of error for future work.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

5.4. *Conformance*

We now prove that the behaviour of a well-typed process conforms to the session types of its free channels. To do so, we first define the traces of a session type. We then show that any trace of a process, projected to the actions of a single name, is contained in the set of traces allowed by the name's session type. This conformance result is similar to the conformance theorem of Gay *et al.* (2010).

In the following definitions, ϵ represents the empty trace, σ ranges over traces, which is a finite list of observations, and :: denotes trace prefixing.

For a session S, consider the (prefix-closed) set of finite lists of messages obtained as paths of labelled transitions from S. The *traces of a session S at name n*, written $traces_n(S)$, is a set of traces constructed from this prefix-closed set. We define it inductively as follows:

— $\epsilon \in traces_n(S)$.
— $(\mathsf{obsm}(n, M) :: \sigma) \in traces_n(S)$ if $\varnothing \vdash S \xrightarrow{M} T$ and $\sigma \in traces_n(T)$.

For a process P, consider the (prefix-closed) set of finite lists of observations obtained from sequences of reductions from P. The *traces of a process P at name n*, written $traces_n(P)$, is the result of filtering only the observations for n from each list in this set. It is defined inductively as follows:

— $\epsilon \in traces_n(P)$.
— If $P \xrightarrow{\alpha} Q$ and $\sigma \in traces_n(Q)$ then
  – $(\alpha :: \sigma) \in traces_n(P)$ if $\alpha = n?\gamma$ or $\alpha = n!\gamma$, and
  – $\sigma \in traces_n(P)$ otherwise.

The conformance theorem states that the traces of a process for a free channel are contained in the set of traces for the session type of that channel.

Theorem 7 (conformance). *If $\varnothing ; G \vdash P$ and $G(n) = Ch(S)$, for some context satisfying* $\mathsf{bal}(G)$, *then $traces_n(P) \subseteq traces_n(S)$.* □

Conformance is a powerful tool for reasoning about the behaviour of processes, especially in conjunction with session polymorphism and session type functions. We briefly consider several applications.

Example 14 (correctness of label branching encoding). Recall the use of the *error* channel in the encoding of label selection and branching in Sections 2.3 and 3.5. The derived typing rule for label branching included the requirement that the process context assigns type $Ch(\mathsf{end})$ to *error*. Since $Ch(\mathsf{end})$ is a stateless type, uses of *error* can occur on both sides of parallel composition.

Now consider a process P that uses the encoding of label branching from Section 2.3, where output occurs on *error* when an unexpected label (simply a token) is received at a label branching process. We assume that *error* is not bound in P. The coname $\overline{error}$ can be assigned the stateless type $Ch(\mathsf{dual\ end})$, and so can be freely copied if necessary. Thus, if P is well-typed, we can always form the well-typed process $(P \mid \overline{error}?x)$. Note that it is only possible to type the input process $\overline{error}?x$ because our type system always allows input, and without the input process, the output process would be irrelevant.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

In this example, applying Theorem 7 to the process $(P \mid \overline{error}?x)$ with channel $error : Ch(\mathsf{end})$ ensures that there is never a communication on *error*, because end only has the empty trace, i.e. $traces_{error}(\mathsf{end}) = \{\epsilon\}$. We conclude that label branching processes never receive incorrect labels. □

Example 15 (correctness of the ABP). Recall the polymorphic session typing of the ABP in Example 13. We wish to deduce that the ABP does not drop, duplicate, or reorder data that it forwards. Consider any finite sequence of tokens $\vec{k} = k_1, \ldots, k_m$. Then $\vec{k}$ determines a session type consisting of a sequence of inputs:

$$S_{\vec{k}} = ?\{k_1\}.\ldots.?\{k_m\}.\mathsf{end}$$

as well as a process that performs the corresponding outputs on a channel $\overline{lft}$:

$$P_{\vec{k}} = \overline{lft}!k_1.\ldots.\overline{lft}!k_m.$$

Now the ABP typing statement is parameterized by a session type. Instantiating that session type with $S_{\vec{k}}$ yields the typing:

$$; lft : Ch(\mathsf{toks}\ S_{\vec{k}}), \overline{rht} : Ch(\mathsf{dual}\ (\mathsf{toks}\ S_{\vec{k}})) \vdash abp(lft, \overline{rht}).$$

The process $P_{\vec{k}}$ can be typed using dual (toks $S_{\vec{k}}$) as the session type for $\overline{lft}$. We may then consider the parallel composition of the sender, the ABP, and the sink process of Section 4.1 (we omit the procedure bodies of both *abp* and *rep* for brevity):

$$P_{\vec{k}} \mid abp(lft, \overline{rht}) \mid rep(rht).$$

The above process is well-typed, because the sink process has a polymorphic session type that allows it to consume data from any channel.

Applying Theorem 7 to this process and the channel $\overline{rht}$ tells us that the traces of the process for $\overline{rht}$ are contained in the traces of dual (toks $S_{\vec{k}}$). From the labelled transitions of dual and toks it can be deduced that the traces of dual (toks $S_{\vec{k}}$) are output observations for sequences of tokens that are initial prefixes of $\vec{k}$. Thus, the output of the ABP is a prefix of the input that it receives, as required. □

The ABP is a common test for mechanical verification, e.g. Divito (1981). Our proof establishes correctness via the modular route of session typing, and an application of conformance.

6. Related work

Dezani-Ciancaglini and de' Liguoro (2010) give a recent comprehensive survey of the session types literature. Also, (Vasconcelos 2009) provides a tutorial reconstruction of session types in a linear π calculus with linear and unrestricted types, where values of unrestricted types need not have linear usage, like our stateless types. We differ in using a deductive definition of statelessness that is relative to a context, rather than stating that a replication session type constructor, e.g. (fun A)), must be used to create stateless types. This allows some additional flexibility, e.g. deducing that a channel type $Ch(S + T)$ is stateless if both $Ch(S)$ and $Ch(T)$ are stateless.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

6.1. *Polymorphism*

Several forms of polymorphism have been investigated in connection with session types. The most closely-related form, discussed in the introduction, is bounded polymorphism for session types (Bono and Padovani 2011; Dezani-Ciancaglini *et al.* 2007b; Gay 2008). In that work, the syntax of a session type includes type abstractions with subtyping bounds (we also use type abstraction with session types, as discussed in Section 3.3). Our session types and types do not have subtyping bounds, but may have other logical constraints. We discuss subtyping below.

Our work differs from the prior work on bounded polymorphism in that we always allow reading from a channel, and we permit logical deduction in type derivations. The former ingredient can be seen as a form of implicit coercion, and the latter resembles a type system based upon logical deduction developed for Scheme (Tobin-Hochstadt and Felleisen 2010). We expand on the treatment of input and deduction below. Additionally, within type derivations we use abstraction over session types across multiple channels in conjunction with functions over session types.

Other research on relating session types and linear type theories (Caires and Pfenning 2010; Gay and Vasconcelos 2010) has led to work that incorporates type abstraction as second-order quantification in linear logic, e.g. to establish strong deadlock freedom guarantees (Wadler 2012), and relational parametricity results (Caires *et al.* 2013).

Dependent session types (Bonelli *et al.* 2005; Toninho *et al.* 2011) include abstraction over values. This form of abstraction is supported in our formalization in Coq.

Deniélou and Yoshida (2011) use polymorphism in connection with session types, but the polymorphism is over roles associated with processes; this kind of role polymorphism is orthogonal to the issues of session polymorphism.

6.2. *Input and deduction*

Existing session type systems only permit an input or label branching process to be typed when the corresponding session type allows the operation. In monomorphic session type systems without subtyping, e.g. Honda *et al.* (1998), session types are known in their entirety at their point of use, and so there is no reason to allow input or label branching processes that can never occur.

In session type systems with subtyping, e.g. Castagna *et al.* (2009) and Gay and Hole (2005) and work on bounded polymorphism, a channel n with a label branch session type may not be identical to the dual of the session type for its other endpoint $\overline{n}$. In this case, some of the label branches in a process may be redundant. This does not impact runtime safety because a well-typed process is forced to offer enough choices in a label branching process.

Our type system takes this idea further, by allowing inputs for any session type, and by depending on the sender to send correct data. It is this relaxation of input typing that permits typing of the forwarder examples. In some cases, the code P after an input $n?x.P$ can be identified as redundant because the input cannot occur, e.g. when n is assigned one of the types: $Ch(\mathsf{end})$, $Ch(!\{k\}.\mathsf{end})$, or $Ch(\mathsf{dual}\ (\mathsf{checkvalve}\ S))$, for a session type

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

S. However, for many generic processes with polymorphic session types, such as the forwarder examples, it is not possible to say whether an input can occur without knowing the session to which the session type variable is instantiated. Neubauer and Thiemann (2004b) investigate the elimination of redundant code in programs with session types, also using singleton types to promote values to types.

6.3. *Matching*

Label branching is normally an atomic operation, e.g. Honda *et al.* (1998). Our encoding of label branching via interaction and matching/mismatching is not an atomic operation. One might wonder why our subject reduction theorem (Theorem 5) does not have a different structure to subject reduction theorems for calculi with atomic label branching that also relate process contexts according to labels from the reduction, e.g. Gay (2008) and Gay and Hole (2005). The difference lies in the process syntax and the reduction relation. In particular, our annotations occur at interaction, and our interaction does not perform the matching tests that are implicit in label branching. In contrast, the annotations of (ibid.) occur when paired label selection and branching synchronize. In other words, our annotations do not describe any matching that occurs after interaction. But they do capture any output that occurs as a result of the matching.

6.4. *Deadlock freedom and progress*

Other session type systems, starting from Honda *et al.* (1998), require that channels with linear usage have the terminal session type end when typing the terminal process $\mathbf{0}$. Consequently, weakening is restricted to channels with terminal session type also. This design decision prevents processes from prematurely terminating a conversation. However, it does not follow that conversations are completed due to the possibility of deadlock. Partly for this reason, the constraint upon premature termination is not exploited in most prior work on session types, where the focus is upon subject reduction and runtime safety.

Kobayashi (1998, 2006) provides a type system for the π calculus that ensures deadlock freedom by describing the dependencies between the order of use of different channels. Coppo *et al.* (2007) and Dezani-Ciancaglini *et al.* (2006) describe a stack discipline for the use of sessions that ensures deadlock freedom and progress. Honda *et al.* (2008) provides a progress property also in the absence of interleaved sessions, but does so in a calculus with multiparty sessions. Dezani-Ciancaglini *et al.* (2007a) instead relaxe the requirement for a stack discipline to achieve progress.

In contrast, our work seeks to extend the traditional subject reduction and safety results to session polymorphism and compositional, logical specifications of session types. We permit processes to leave a conversation incomplete, by failing to reach the terminal session end. This does not impact subject reduction, runtime safety, or conformance, but does allow some (arguably) useful generic processes with polymorphic session types, such as the sink process of Example 10.

Nevertheless, whether or not conversations may be ended prematurely appears to be orthogonal to our approach to session polymorphism. The changes to the type system

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

would involve the typing of the terminal process end, the matching and mismatching constructs (since they are terminal if they fail, using an if-then-else equality-testing construct would suffice), and the replication construct. The sink example would no longer type. On the other hand, the forwarder processes would type because they do not terminate the conversations in the *lft* and *rht* parameters. The single-use channels of type $Ch(\mathsf{fwd})$ are discarded only when they reach type $Ch(\mathsf{end})$, as expected. The ABP process would also type because it expects to receive tokens and not channels in its input, and so the lossy process can freely discard the tokens that it receives.

6.5. *Towards subtyping*

The theory of subtyping for session types was developed in Gay and Hole (2005). For example, using the syntax defined in this paper, the session type $T_1 \triangleq ?\{k_1\}.S_1$ is a subtype of $T_2 \triangleq (?\{k_1\}.S_1)+(?\{k_2\}.S_2)$, because the former has fewer inputs. Our type system does not yet incorporate subtyping. However, it is possible to use bidirectional forwarding to gain some of the expressive power of subtyping. Consider a channel n_1 with type $Ch(T_1)$. Without subtyping, it is not possible to send n_1 on a channel that expects a value of type $Ch(T_2)$. Instead, create new channel endpoints n_2, $\overline{n}_2$ with types $Ch(T_2)$ and $Ch(\mathsf{dual}\ T_2)$ respectively. The channel n_2 can be sent in place of n_1, because its type is identical to the required type. Next, a bidirectional forwarding process can be used to forward messages from n_1 to $\overline{n}_2$, and vice versa. The polymorphic session type given in Example 12 for the bidirectional forwarder does not allow n_1 and $\overline{n}_2$ to be used because their session types T_1 and $(\mathsf{dual}\ T_2)$ are not dual.

However, a more generous polymorphic session type, based upon subtyping, can be established for the bidirectional forwarder. We say that a binary relation $\mathcal{R}$ on session types is a simulation if, for every $(S_1, T_1) \in \mathcal{R}$:

— if $S_1 \xrightarrow{?a} S_2$, then there exists T_2, such that $T_1 \xrightarrow{?a} T_2$ and $(S_2, T_2) \in \mathcal{R}$; and
— if $T_1 \xrightarrow{!a} T_2$, then there exists S_2, such that $S_1 \xrightarrow{!a} S_2$ and $(S_2, T_2) \in \mathcal{R}$.

Similarity is then defined to be the greatest simulation.

Similarity provides the exact information needed to typecheck the bidirectional forwarding process. Suppose, the bidirectional forwarder is passed channels *lft* and *rht* with session types S and $(\mathsf{dual}\ T)$, where S and T are related by similarity. Then the first component of simulation guarantees that an input on *lft* justifies output on *rht*, because *rht* has session type $(\mathsf{dual}\ T)$, and so reverses the direction of interaction. The second component of simulation guarantees that an input on *rht* justifies output on *lft*.

If session types are related by a finite simulation $\mathcal{R} = \{(S_i, T_i) \mid 1 \leqslant i \leqslant n\}$, then the single-use channel that carries the arguments to the bidirectional forwarding process has session type $\mathsf{fwdsim}_{\mathcal{R}}$ defined by:

$$\begin{aligned}\mathsf{fwdsim}_{\mathcal{R}} \triangleq\ & (?Ch(S_1).?Ch(\mathsf{dual}\ T_1).\mathsf{end}) \\ & + (?Ch(S_2).?Ch(\mathsf{dual}\ T_2).\mathsf{end}) \\ & + \cdots \\ & + (?Ch(S_n).?Ch(\mathsf{dual}\ T_n).\mathsf{end})\end{aligned}$$

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

Finally, the bidirectional forwarder can be typed using $\mathsf{fwdsim}_{\mathcal{R}}$ for carrying related *lft* and *rht* channels, for all finite simulations $\mathcal{R}$. The restriction to finite simulations arises from our strategy for encoding relations. We are investigating alternative encodings for infinite simulations. We note that the polymorphic session typing for the bidirectional forwarder in Example 12 does effectively yield the (infinite) identity simulation. Moreover, it may be useful to add the identity relation to the simulations under consideration by typing the bidirectional forwarder with session type $(\mathsf{fwdsim}_{\mathcal{R}} + \mathsf{fwd})$. In this way, our original example, where $T_1 \triangleq ?\{k_1\}.S_1$ and $T_2 \triangleq (?\{k_1\}.S_1) + (?\{k_2\}.S_2)$, need only use the relation $\mathcal{R} = \{(T_1, T_2)\}$. This is not a simulation, but the union of $\mathcal{R}$ with the identity relation is a simulation.

For ground data (tokens) our definition of similarity corresponds to the behaviour of subtyping at label selection and branch session types in Gay and Hole (2005). However, the definition of similarity is invariant in the message types a, and so, for non-ground data (channels), it is weaker than subtyping in Gay and Hole (2005). We leave a more complete definition of subtyping for future work.

6.6. *Polarities*

Recent work (Dezani-Ciancaglini *et al.* 2006; Giunti *et al.* 2009; Vasconcelos *et al.* 2010; Yoshida and Vasconcelos 2007) has shown that polarities are unnecessary for type safety in a session type system for programmers, but that polarities can be useful within proofs about models with unpolarized channels. Nevertheless, we have adopted the polarity solution of Gay and Hole (2005) in an attempt to provide a more direct proof of subject reduction, and, hence, to make mechanical verification tractable.

6.7. *Formalization and mechanical verification*

Prior formalizations of the π calculus with theorem provers include the theory of strong late bisimilarity using a HOAS representation in Coq (Honsell *et al.* 2001), subject reduction for a type system of directionality of names using HOAS in Coq (Despeyroux 2000), safety of a linear type system using de Bruijn indices in Isabelle/HOL (Gay 2001) and specification and verification for processes of a π calculus extension using a HOAS representation in Coq (Affeldt and Kobayashi 2008). Malecha *et al.* (2011) describe a Coq library for verification of I/O behaviour of programs. Their I/O specifications are based on traces rather than session types (representing sets of traces).

7. Conclusion

We have shown that:

— Session type polymorphism can be applied to simple generic processes via a novel treatment of input typing in conjunction with deduction.
— Ideas from dependent type theory for certified functions on lists can be adapted to channels with bidirectional interaction

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

— Session type functions that are transducers are expressive and amenable to straightforward session typing.

These ideas appear to be robust in the sense that they may be adaptable to session typing for functional/object-oriented programming languages and may be implemented in different logical frameworks. We have developed one such implementation for the π calculus in Coq and mechanically verified subject reduction and safety results. This implementation has been used to validate polymorphic session typing, yielding partial correctness for the ABP, and demonstrating that the type discipline can be applied to create modular specifications of nontrivial protocols.

The rich logic of specifications has potential application to protocols where the primary concern lies with the complexity of the language, e.g. encoding XML schema as session types. We are exploring the application of session types to a modular analysis of generic HTTP-processing code.

We have not studied conditions under which process typing is decidable, or fully general theories of subtyping/type equivalence. These issues will be explored in future work.

Acknowledgement

We thank the anonymous referees for their detailed comments and suggestions.

References

Affeldt, R. and Kobayashi, N. (2008) A Coq library for verification of concurrent programs. *Electronic Notes in Theoretical Computer Science* **199** 17–32.

Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R. and Weirich, S. (2008) Engineering formal metatheory. In: *Symposium on Principles of programming languages* 3–15.

Bartlett, K. A., Scantlebury, R. A. and Wilkinson, P. T. (1969) A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM* **12** (5) 260–261.

Bertot, Y. and Castéran, P. (2004) *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer Verlag.

Bonelli, E., Compagnoni, A. B. and Gunter, E. L. (2005) Correspondence assertions for process synchronization in concurrent communications. *Journal of Functional Programming* **15** (2) 219–247.

Bono, V. and Padovani, L. (2011) Polymorphic endpoint types for copyless message passing. In: Proceedings of the 4th Workshop on Interaction and Concurrency Experience (ICE'11). *Electronic Proceedings in Theoretical Computer Science* **59** 53–67.

Caires, L., Pérez, J., Pfenning, F. and Toninho, B. (2013) Behavioral polymorphism and parametricity in session-based communication. In: Felleisen, M. and Gardner, P. (eds.) ESOP. *Lecture Notes in Computer Science* **7792** 330–349.

Caires, L. and Pfenning, F. (2010) Session types as intuitionistic linear propositions. In: Gastin, P. and Laroussinie, F. (eds.) CONCUR. *Springer Lecture Notes in Computer Science* **6269** 222–236.

Castagna, G., Dezani-Ciancaglini, M., Giachino, E. and Padovani, L. (2009) Foundations of session types. In: *Symposium on Principles and Practice of Declarative Programming* ACM, 219–230 .

Charguéraud, A. (2012) The locally nameless representation. *Journal of Automated Reasoning* **49** (3) 363–408.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

Coppo, M., Dezani-Ciancaglini, M. and Yoshida, N. (2007) Asynchronous session types and progress for object oriented languages. In: FMOODS. *Lecture Notes in Computer Science* **4468** 1–31.

Coquand, T. (1992) Pattern matching with dependent types. In: Nordström, B., Petersson, K. and Plotkin, G. (eds.) *Electronic Proceedings of the 3rd Annual BRA Workshop on Logical Frameworks* 66–79. Available from `http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/`.

Coquand, T. and Paulin-Mohring, C. (1990) Inductively defined types. *Lecture Notes in Computer Science* **417** 50–66.

Deniélou, P.-M. and Yoshida, N. (2011) Dynamic multirole session types. In: *Symposium on Principles of programming languages* 435–446.

Despeyroux, J. (2000) A higher-order specification of the π-calculus. In: IFIP Conference on Theoretical Computer Science. *Lecture Notes in Computer Science* **1872** 425–439.

Dezani-Ciancaglini, M. and de' Liguoro, U. (2010) Sessions and session types: An overview. In: Laneve, C. and Su, J. (eds.) WS-FM'09. *Lecture Notes in Computer Science* **6194** 1–28.

Dezani-Ciancaglini, M., de Liguoro, U. and Yoshida, N. (2007a) On progress for structured communications. In: TGC. *Springer Lecture Notes in Computer Science* 257–275.

Dezani-Ciancaglini, M., Drossopoulou, S., Giachino, E. and Yoshida, N. (2007b) Bounded session types for object-oriented languages. In: FMCO'06. *Lecture Notes in Computer Science* **4709**.

Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N. and Drossopolou, S. (2006) Session types for object-oriented languages. In: ECOOP'06. *Springer Lecture Notes in Computer Science* **4067** 328–352.

Divito, B. L. (1981) A mechanical verification of the alternating bit protocol. *ACM SIGSOFT Software Engineering Notes* **6** (3), 8–13.

Dybjer, P. (1991) Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In: Huet, G. and Plotkin, G. (eds.) *Logical Frameworks*, New York, NY, USA. Cambridge University Press 280–306.

Freeman, T. and Pfenning, F. (1991) Refinement types for ML. In: *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, ACM Press 268–277.

Gay, S. J. (2001) A framework for the formalisation of π calculus type systems in Isabelle/HOL. In: *Proceedings of 14th Theorem Proving in Higher Order Logics* 217–232.

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

Gay, S. J. and Hole, M. (2005) Subtyping for session types in the π calculus. *Acta Informatica* **42** (2-3) 191–225.

Gay, S. J. and Vasconcelos, V. T. (2010) Linear type theory for asynchronous session types. *Journal of Functional Programming* **20** (1) 19–50.

Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N. and Caldeira, A. Z. (2010) Modular session types for distributed object-oriented programming. In: *Symposium on Principles of programming languages* 299–312.

Giunti, M., Honda, K., Vasconcelos, V. T. and Yoshida, N. (2009) Session-based type discipline for pi calculus with matching. Available at `http://homepages.di.fc.ul.pt/~vv/papers/giunti.honda.etal_session-based-pi-matching.pdf`.

Gordon, A. D. and Fournet, C. (2010) Principles and applications of refinement types. In: Esparza, J., Spanfelner, B. and Grumberg, O. (eds.) *Logics and Languages for Reliability and Security*, NATO Science for Peace and Security Series - D: Information and Communication Security volume 25, IOS Press 73–104.

Goto, M., Jagadeesan, R., Jeffrey, A., Pitcher, C. and Riely, J. (2011) Coq formalization of extensible polymorphic session types. Available at `http://fpl.cs.depaul.edu/projects/xpol/`.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

Honda, K., Vasconcelos, V. T. and Kubo, M. (1998) Language primitives and type discipline for structured communication-based programming. In: ESOP. *Lecture Notes in Computer Science* **1381** 122–138.

Honda, K., Yoshida, N. and Carbone, M. (2008) Multiparty asynchronous session types. In: *Symposium on Principles of programming languages* 273–284.

Honsell, F., Miculan, M. and Scagnetto, I. (2001) π-calculus in (co)inductive type theory. *Theoretical Computer Science* **253** 239–285.

Hu, R., Yoshida, N. and Honda, K. (2008) Session-based distributed programming in Java. In: ECOOP. *Lecture Notes in Computer Science* **5142** 516–541.

Jeffrey, A. S. A. and Rathke, J. (2011) The lax braided structure of streaming I/O. In: *Proceedings Conference of Computer Science Logic* 292–306.

Kiselyov, O., Peyton Jones, S. and Shan, C.-C. (2010) Fun with type functions (version 3). Presented at Tony Hoare's 75th birthday celebration.

Kobayashi, N. (1998) A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems* **20** (2) 436–482.

Kobayashi, N. (2006) A new type system for deadlock-free processes. In: Baier, C. and Hermanns, H. (eds.) CONCUR. *Springer Lecture Notes in Computer Science* **4137** 233–247.

Malecha, G., Morrisett, G. and Wisnesky, R. (2011) Trace-based verification of imperative programs with I/O. *Journal of Symbolic Computation* **46** (2) 95–118. (Automated specification and verification of web systems.)

Milner, R. (1991) The polyadic π-calculus: A tutorial. *Technical Report* ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK. (Also in *Logic and Algebra of Specification*, F. L. Bauer, W. Brauer and H. Schwichtenberg (eds.) Springer-Verlag, 1993.)

Milner, R., Parrow, J. and Walker, D. (1992) A calculus of mobile processes, I. *Information and Computation* **100** (1) 1–40.

Neubauer, M. and Thiemann, P. (2004a) An implementation of session types. In: PADL. *Springer Lecture Notes in Computer Science* **3057** 56–70.

Neubauer, M. and Thiemann, P. (2004b) Protocol specialization. In: Chin, W.-N. (ed.) APLAS. *Springer Lecture Notes in Computer Science* **3302** 246–261.

Pucella, R. and Tov, J. A. (2008) Haskell session types with (almost) no class. In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, 25–36.

Röckl, C. and Hirschkoff, D. (2003) A fully adequate shallow embedding of the π-calculus in Isabelle/HOL with mechanized syntax analysis. *Journal of Functional Programming* **13** 415–451.

Roscoe, A. W. (1997) *The Theory and Practice of Concurrency*, Prentice Hall.

Sackman, M. and Eisenbach, S. (2008) Session types in Haskell: Updating message passing for the 21st century. Available at `http://hdl.handle.net/10044/1/5918`.

Takeuchi, K., Honda, K. and Kubo, M. (1994) An interaction-based language and its typing system. In: PARLE. *Springer Lecture Notes in Computer Science* **817** 398–413.

Tobin-Hochstadt, S. and Felleisen, M. (2010) Logical types for untyped languages. In: Hudak, P. and Weirich, S. (eds.) *International Conference on Functional Programming*, ACM 117–128.

Toninho, B., Caires, L. and Pfenning, F. (2011) Dependent session types via intuitionistic linear type theory. In: Schneider-Kamp, P. and Hanus, M. (eds.) *Symposium on Principles and Practices of Declarative Programming*, ACM 161–172.

Vasconcelos, V. T. (2009) Fundamentals of Session Types. *Springer Verlag Lecture Notes in Computer Science* **5569** 158–186.

Vasconcelos, V. T., Gay, S. J. and Ravara, A. (2006) Type checking a multithreaded functional language with session types. *Theoretical Computer Science* **368** (1–2) 64–87.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press

Vasconcelos, V. T., Giunti, M., Yoshida, N. and Honda, K. (2010) Type safety without subject reduction for session types. Available at `http://www.di.fc.ul.pt/ vv/papers/vasconcelos.giunti.etal_type-safety-session-types.pdf`.

Wadler, P. (2012) Propositions as sessions. In: Thiemann, P. and Findler, R. B. (eds.) *International Conference on Functional Programming*, ACM 273–286.

Yoshida, N. and Vasconcelos, V. T. (2007) Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In: 1st International Workshop on Security and Rewriting Techniques. *Elsevier Electronic Notes in Theoretical Computer Science* **171** (4) 73–93.

https://doi.org/10.1017/S0960129514000231 Published online by Cambridge University Press