Type-Safe Eventful Sessions in Java

Raymond Hu*, Dimitrios Kouzapas*, Olivier Pernet*
Nobuko Yoshida*, and Kohei Honda†

*Imperial College London †Queen Mary, University of London

Abstract. Event-driven programming is a major paradigm in concurrent and communication-based programming, and a widely adopted approach to building scalable high-concurrency servers. However, traditional event-driven programs are more difficult to read, write and verify than their multi-threaded counterparts due to low-level APIs and fragmentation of control flow across disjoint event handlers. This paper presents a Java language extension and a novel type discipline for type-safe *event-driven session programming* that counters the problems of traditional event-based programming with abstractions and safety guarantees based on *session types*, while retaining the expressiveness and performance characteristics of events. The type discipline extends session types and their primitives with *asynchronous input*, *session typecase* and *session set types*, ensuring *event-handling safety* and *event progress* in addition to the standard type soundness and communication safety. The advantages, expressiveness and performance of event-driven session programming are demonstrated through a range of examples and benchmarks, including a session-typed SMTP server.

1 Introduction

Asynchronous event-driven programming is characterised by a reactive flow of control driven by the occurrence of computation events. It is one of the major paradigms in concurrent and communication-based programming, where events are typically detected by the arrival of messages on asynchronous channels. Primary motivations for asynchronous event programming include performance and scalability, particularly for high-concurrency applications such as Web servers [24, 36]. Unfortunately, the flexibility and performance of traditional event-driven programming comes at the cost of more complex programs: low-level APIs and the obfuscation of event-driven control flow [3, 34] make programs difficult to read, write and verify, and hence potentially unsafe to execute. Consequently, several recent works [25, 27, 35] have proposed simpler thread-based programming interfaces that hide event-driven runtimes. In contrast to these approaches, our aim in this paper is to develop a high-level, structured and safe programming discipline for event-driven programming based on, and extending, *session types* [19, 33]. We generalise the existing session types for asynchronous event-driven programming, using which we obtain both formal safety guarantees and programmatic benefits to overcome the problems of traditional event-driven programming.

Session types [19, 33] are one of the well-studied type-based methods for structuring a series of distributed interactions. Previous works have studied the theory and practice of session types in object-oriented languages [10, 12, 16, 21] that ensure the

so-called *communication safety*, meaning that communicating programs correctly interact following the associated session type structures. However, typing asynchronous event-driven programming is *not* possible so far, for the reasons outlined below.

A general mechanism underlying all event-based systems is the asynchronous detection of heterogeneous events (i.e. of varied types) from a dynamic collection of channels. This idea is embodied by e.g. the Unix `select` system call and the Java NIO `Selector` API. In the context of session programming, this means we need a framework where we collect multiple channels of *different* session types, *asynchronously check* for the arrival of messages on these channels, and later retrieve and use the "ready" channels as directed by their session types. The preceding session type disciplines cannot support these ideas because the lack of *non*-blocking input prohibits core event idioms such as event loops, and because statically determined channel types make it impossible to treat a collection of channels with heterogeneous types.

Contributions. This paper develops a framework for type-safe *event-driven session programming* that integrates session types and asynchronous event programming in Java. The key concepts of event-driven session programming are introduced through initial motivating examples in § **2**. The rest of the paper presents the following contributions.

- **(§ 3)** We explore a theoretical basis of event-driven session programming using a small process calculus based on [18, 19, 33]. The formalism captures the semantics of asynchronous, event-driven sessions through two new constructs, the *message arrival predicate* and *session typecase*. Combining the latter with the new *session set types* allows us to treat dynamic collections of heterogeneously typed channels.
- **(§ 4)** A type theory based on session set types for the extended session constructs. In addition to *type soundness and communication safety in the presence of dynamically registered sessions*, we prove a novel progress property which we call *event progress*. The theory captures a wide range of event-driven programming idioms (e.g *event selectors*, *event loops* and *join patterns*) via encoding, from which we can derive sound typing rules for such complex constructs.
- **(§ 5)** Building on the new theory, we present the design and implementation of a practical language, compiler and runtime support for event-driven session programming as an extension to Java, on the basis of SJ (Session Java)[21]. The resulting *Eventful SJ* (ESJ) enjoys the formal properties and safety guarantees established by the theory. We discuss an ESJ implementation of an SMTP server and show that ESJ preserves the performance characteristics of traditional event-driven programming by benchmarking the ESJ server against a multithreaded equivalent.

Our implementation of ESJ exploits the modular architecture of the SJ runtime [30] for *transport independence*: the ESJ selector enables the event-driven execution of sessions not only of different types but also over different transports, under a single programming abstraction (§ 5.2). Implementing ESJ reveals further applications of the eventful session theory. For example, runtime session type monitoring enables applications like the SMTP server to execute sessions with *non*-SJ parties while protecting communication safety (§ 5.3). Related work and future topics are discussed in § **6**. Omitted definitions, proofs, ESJ source code and benchmark results can be found at [30].

2 Event-driven Session Programming

This section introduces the key concepts of *event-driven session programming* through examples, presented in the syntax of Eventful SJ (ESJ for short), illustrating both the safety guarantees and practical programming benefits of our new event-driven programming framework compared to the traditional one.

ESJ is built on SJ, an extension of Java for type-safe concurrent and distributed session programming [21]. Session programming in SJ, as detailed in [21, § 2], starts with the declaration of the intended communication protocols as session types. The communication actions comprising a session, such as message passing, branching and recursion, are implemented as operations on session-typed channel endpoints called *session sockets*, objects of type `SJSocket`. The SJ compiler statically checks session implementations against the declared protocols, ensuring correct communication behaviour. ESJ extends SJ with facilities for session-typed asynchronous event handling.

Event loops. The core of any event-based system is the *event loop* [26], which waits for event occurrences (i.e. messages) and dispatches them by invoking an appropriate handler. The performance and scalability of event-based systems come from the asynchronous decoupling of event handlers from the event source (e.g. the network interface) through the event loop, which enables many concurrent sessions to be serviced as a fine-grain sequential interleaving of actions within a single thread or a thread pool.

Our first example is a basic event loop that handles sessions of type

```
?(Data).?(Data).!<Result>                                    (1)
```

which says: *at this side of the session, we first receive (*`?`*) a message, a Java object of type* `Data`*, then receive another* `Data` *message, and finish by sending (*`!`*) a* `Result`. The other side will conform to the *dual* protocol, `!<Data>.!<Data>.?(Result)`.

We implement an event-driven server for handling multiple, concurrent sessions of the above type. The standard event loop pattern is adapted to SJ session programming as follows. The server *registers* the initialised session sockets with a *session event selector* (a session typed version of e.g. the Java NIO `Selector` [22]) to monitor them for event occurrences. By session typing, the first event on a new session will be the receipt of a `Data` message. After handling this event, the session socket is returned to the selector to await the second `Data` message. The functionality of the selector combines that of a dynamic collection for session sockets with asynchronous event detection. The key point, with respect to session typing, is that the selector is required to store not only sessions of the initial type (1), but also the intermediary type (2) `?(Data).!<Result>`.

Figure 1 outlines an ESJ program for the above server. Lines 1–2 declare a *session set type* containing session types (1) and (2). Lines 3–4 then declare and initialise the session event selector `sel`, an object of type `SJSelector{pSelector}`. This means `sel` can store and monitor session sockets of the two types in the `pSelector` set type. In SJ, the `using` statement has two main purposes. As in C#, the declared resources (`sel`) are cleaned up after we leave the scope of the statement. In addition, session type checking requires the session type to be completed within the `using`, as we describe below.

The initial sessions can be registered with the selector as on Line 5. The main event loop then starts on Line 7. The first action in the loop is the `select` operation on Line 8,

```
1  protocol pSelector // A session set type containing the two event types.
2       { ?(Data).?(Data).!<Result>, ?(Data).!<Result> }
3  using(SJSelector{pSelector} sel // Create a selector of type pSelector.
4       = SJSelector.create(params)) {
5    sel.register(source); // Register event source session(s) with the selector.
6    ...
7    while(run) { // Main event loop.
8      using(SJSocket{pSelector} s = sel.select()) { // Select a session event.
9        typecase(s) { // Identify the type of the occurred event.
10         when(SJSocket{?(Data).?(Data).!<Result>} s1) {
11           Data d1 = s1.receive(); // Handle the first Data event and..
12           sel.register(s1); //..re-register the session with the selector.
13         }
14         when(SJSocket{?(Data).!<Result>} s2) {
15           Data d2 = s2.receive(); // Handle the second Data event, then..
16           s2.send(new Result(...)); // ..send the Result; session completed.
17         }
18 } } } }
```

Fig. 1. Combining a session typed event selector with session typecase in a basic ESJ event loop.

which blocks until the selector detects an event occurrence on one of the registered session sockets. The returned session socket has type `SJSocket{pSelector}` — we know only that the session is of either of the two `pSelector` types — and is assigned to the variable `s` (enclosed by another `using` statement). To determine which event has occurred, i.e. whether we have received the first or second `Data` message on `s`, we use the *session typecase* starting on Line 9. The `typecase` selects the first `when` case for which the specified type matches the current *runtime session type* of the session. *Static* type checking ensures that the `typecase` covers *all* the cases of the `pSelector` set type, so at least one case is guaranteed to match at runtime. The `when` case on Lines 10–13 handles the first `Data` event, i.e. for session type (1). The session socket is rebound[1] to the variable `s1` of type `SJSocket{!<Data>.!<Data>.?(Result)}`, and `s1` is used to receive the `Data` message on in Line 11. Since the second `Data` may not yet have arrived, Line 12 re-registers `s1` with the selector (and then we loop to handle the next event). Similarly, the other `when` case on Lines 14–17 handles the second `Data` when the runtime session type of `s` is (2): following the session type, we receive the `Data` message and then send a `Result` via `s2`. The session is now finished; we do not re-register the session socket.

Event streams. In this example, we implement a prevalent pattern where an event handling party consumes streams of mixed events. This example extends the basic event loop to support session initiation and branching events, and uses session recursion to represent unbounded streams. We specify a simple event stream as `sbegin.pStream` (`sbegin` represents the server-side session initiation action), with `pStream` declared as

```
protocol pStream rec X [ ?{NEXT: ?(Data).#X, QUIT: } ]
```

where `rec X [...]` binds the recursion type variable `X` within the scope of the brackets. Inside the recursion, the branch type `?{...}` allows the opposing session party to select

[1] `typecase` variable rebinding comes from dynamic typing in the λ-calculus [1], and ensures type soundness. The following "event streams" example further demonstrates this point.

```
1  ... // Create a selector 'sel' and register a session server socket.
2  while(run) { // An event loop for recursive pStream event stream sessions.
3    using(SJChannel{pSelector2} c = sel.select()) {
4      typecase(c) {
5        when(SJServerSocket{sbegin.pStream} ss) { // Session initiation event.
6          using (SJSocket{pStream} s0 = ss.accept()) { // Accept the session.
7            s0.recursion(X) { // Unfold the recursion..
8              sel.register(s0); // ..and register the new session.
9            }
10           sel.register(ss); // Re-register the server socket.
11       } }
12       when(SJSocket{?{NEXT: ?(Data).pStream, QUIT: }} s1) {
13         s1.inbranch() { // Handle the branch event.
14           case NEXT: { sel.register(s1); } // Re-register the session.
15           case QUIT: { } // End of stream: no re-registration.
16       } }
17       when(SJSocket{?(Data).pStream} s2) {
18         Data d = s2.receive(); // Handle the arrived Data message.
19         s2.recursion(X) { // Unfold another recursion..
20           sel.register(s2); // ..and re-register the session.
21       } }
22 } } }
```

Fig. 2. Handling a stream of mixed event types, including session initiation and branching events.

one of the two paths, labelled `NEXT` and `QUIT`. If `NEXT` is selected, we receive a `Data` and the recursion is enacted (denoted by `#X`). `QUIT` ends the stream, and the session is completed. For our ESJ implementation, we declare the session set type `pSelector2`

```
protocol pSelector2 { sbegin.pStream, // Session initiation event.
                      ?{NEXT: ?(Data).pStream, QUIT: }, // Branch event
                      ?(Data).pStream } // Data message event.
```

which specifies three event types: the session initiation event that creates the stream, the branch event when a branch label is received, and the message event when a `Data` arrives. Figure 2 lists an event loop for the above event stream. The new features in this program are as follows. The session initiation event described by `sbegin.pStream` involves calling `accept` on the `SJServerSocket` (a session-typed server endpoint for accepting client requests) [30, § 2] registered with the selector. Hence, the `select` operation returns an object of class `SJChannel`, the common superclass of `SJSocket` and `SJServerSocket`. Again, the type of the occurred event is determined using `typecase`, which rebinds (i.e. casts) the `SJChannel` to the appropriate subclass: `SJServerSocket` in the first `when` case, which handles the initiation event, and `SJSocket` in the other two cases. The recursive session is unfolded using the `recursion` construct, e.g. on Line 7, `pStream` is unfolded to `?{NEXT: ?(Data).pStream, QUIT: }`. The branch type is implemented by the `inbranch` construct (Lines 13–16), which receives a label and selects the corresponding case; static typing ensures all specified cases are covered.

Benefits due to session types. The main source of difficulty in traditional event-driven programming is the fragmentation of control flow across disjoint event handlers [3, 34]. Below we summarise how session types counteract this key issue.

(1) Delineation of control flow. A session type is an abstraction of control flow (sequencing, branching and recursion) for interactions. Using session types, events are precisely defined by both the immediate action (e.g. the first `?(Data)` in the first example) and by *the remaining session flow* (`?(Data).!<Result>`). Traditional events lack the latter information, often requiring burdensome manual state management [3] to distinguish ambiguous events (e.g. the first and second `Data` messages). The combination of session set types and session typecase promotes clear structuring of event-driven programs, elucidating their communication and event-handling behaviour.

(2) Event-handling safety and progress. In addition to the above programming benefits, session types provide formal safety guarantees which traditional event-driven programming lacks. Static session type checking ensures *event-handling safety* (Theorem 4.3): each event is correctly handled as directed by the governing session type. The type safety also entails that each session is either completed or re-registered with a selector by the handler, ensuring that *each session flow is faithfully preserved across separate event handlers*. Combined with the standard linearity of session channels (which prevents interference of events by other threads), we obtain a strong progress property for asynchronous event programming, *event progress* (Theorem 4.6).

3 A Process Model for Eventful Sessions

We formalise the key programming ideas introduced in § 2 as a small process calculus. The calculus, which we call ESP (Eventful Session Pi-calculus), is the π-calculus with session primitives [19, 28] based on asynchronous communication semantics [18], to which we add minimal extensions for event-driven session programming: the *message arrival predicate*, *session typecase*, and *session set types*.

3.1 Syntax of the Eventful Session π-Calculus

Types. The type syntax of ESP extends the standard binary session types [19] with *session set types*. This simple extension allows us to treat type-safe event handing for an arbitrary collection of differently typed communication channels.

$$\begin{array}{llll} \text{(Shared)} & U & ::= & \mathsf{bool} \mid \langle S \rangle \mid \mathsf{X} \mid \mu \mathsf{X}.U \qquad \text{(Value)} \quad T ::= U \mid \{S_i\}_{i \in I} \\ \text{(Session)} & S & ::= & !(T);S \mid ?(T);S \mid \oplus\{l_i : S_i\}_{i \in I} \mid \&\{l_i : S_i\}_{i \in I} \mid \mu \mathsf{X}.S \mid \mathsf{X} \mid \mathtt{end} \end{array}$$

The shared types $(U, U', \ldots)$ are the booleans bool (we also use nat in examples), shared channel types $\langle S \rangle$ (shared channels of this type are used to establish sessions whose accepting side acts as S), type variables $(\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots)$ and recursive types. The session types $(S, S, \ldots)$ are standard [19, 28]. The output type $!(T);S$ represents the output of a value of type T followed by the behaviour represented by S; similarly for the dual input type $?(T);S$. The select type $\oplus\{l_i : S_i\}_{i \in I}$ describes a behaviour that selects one of the labels l_i followed by S_i. The branch type $\&\{l_i : S_i\}_{i \in I}$ waits for a select decision with I options, and behaves as the S_i corresponding to the chosen l_i. We assume recursive

(Identifiers) $u ::= a,b,c \mid x,y,z \quad k ::= s,\overline{s} \mid x,y,z$ (Values) $v ::= \texttt{tt},\texttt{ff} \mid a,b,c \mid s,\overline{s}$

(Expressions) $e ::= v \mid x,y,z \mid \texttt{arrived}\,u \mid \texttt{arrived}\,k \mid \texttt{arrived}\,k\,h$

(Processes) $P,Q ::= u(x{:}S).P \mid \overline{u}(x{:}S);P \mid k!\langle e\rangle;P \mid k?(x).P \mid k \triangleleft l;P \mid k \triangleright \{l_i{:}P_i\}_{i\in I}$
$\mid \texttt{if}\,e\,\texttt{then}\,P\,\texttt{else}\,Q \mid (\nu\,u{:}\langle S\rangle)P \mid P \mid Q \mid \mathbf{0} \mid \texttt{def}\,D\,\texttt{in}\,P \mid X\langle\vec{e}\rangle$
$\mid \texttt{typecase}\,k\,\texttt{of}\,\{(x_i{:}T_i)\,P_i\}_{i\in I} \mid a[\vec{s}] \mid \overline{a}\langle s\rangle \mid (\nu\,s)P \mid k[S,\texttt{i}{:}\vec{h},\texttt{o}{:}\vec{h'}]$

(Agents) $D ::= X_1(\vec{x}_1) = P_1\ \texttt{and}\ \ldots\ \texttt{and}\ X_n(\vec{x}_n) = P_n$ (Messages) $h ::= v \mid l$

Fig. 3. The syntax of ESP processes.

types $\mu\,\mathsf{X}.S$ are contractive, i.e. that type variables are guarded in the standard way. `end` represents session completion and is often omitted.

Value types for message values are the shared types and the session set types $\{S_i\}_{i\in I}$, where I is finite (can be empty) and all S_i are closed (i.e. do not contain free type variables). A session set type $\{S_i\}_{i\in I}$ represents a behaviour capable of safely interacting as any one of S_i. For example, a session with type $\{!(\mathsf{bool}), ?(\mathsf{nat})\}$ can safely interact with a session of both $?(\mathsf{bool})$ and $!(\mathsf{nat})$ types. Session set types are used to type the typecase, and are so called for their set-like properties (derived from subtyping, § 4.1), e.g. $\{S_1,S_2\} = \{S_2,S_1\}$ and $\{S,S\} = \{S\}$. A singleton $\{S\}$ is often written S. We write $\mathscr{T}$ for the set of all closed types, and $\mathscr{S}$ for the set of closed session types.

Processes. Figure 3 gives the syntax for ESP processes. Terms that only appear at runtime are shaded; the other terms are *user syntax*. The new primitives are the *message arrival predicate* `arrived` for non-blocking inspection of messages buffers, and the session *typecase* `typecase` k `of` $\{...\}$ for dynamically inspecting the runtime type of a session. We also introduce asynchronous session initiation (cf. [19]).

Values $v,v',\ldots$ include the constants, shared channels $a,b,c,\ldots$, and session channels $s,s',\ldots$. A session channel s designates one endpoint of a session, and $\overline{s}$ the opposing end of the same session, with $\overline{\overline{s}} = s$.[2] Branch labels range over $l,l',\ldots$, variables over x,y,z, and process variables over X,Y,Z. Shared channel identifiers u,u' are shared channels and variables; session identifiers k,k' are session channels and variables. A session message h is a value or a label. Expressions e are values, variables, and the *message arrival predicate*: `arrived` u for session initiation requests, `arrived` k for session messages, and `arrived` $k\,h$, which checks for the arrival of the specific message h at k. We write $\vec{s}$ and $\vec{h}$ for their respective vectors, and ε for the empty vector.

The session initiation actions on shared channels are the request $\overline{u}(x{:}S);P$ and the accept $u(x{:}S).P$. On an established session channel, output $k!\langle e\rangle;P$ sends the value of e through channel k, input $k?(x).P$ receives a value through k, selection $k \triangleleft l;P$ chooses and sends the label l through k, and branching $k \triangleright \{l_i{:}P_i\}_{i\in I}$ follows the branch with the label received through k. The $(\nu\,u{:}\langle S\rangle)P$ binds a shared channel u of type $\langle S\rangle$ to the scope of P. The *session typecase* `typecase` k `of` $\{(x_i{:}T_i)\,P_i\}_{i\in I}$ takes a session channel k and a list of cases $(x_i{:}T_i)$, each binding a free variable x_i of type pattern T_i in P_i.[3]

[2] We simply say "session channel" rather than "session channel endpoints" (i.e. the programming entities used to perform session actions) for brevity; similarly for shared channels.

[3] The full typecase construct that supports typecase of general expressions e, matching variables in type patterns, and the default case is given in the long version available from [30].

$$
\begin{array}{lrcll}
\text{[Request1]} & \overline{a}(x{:}S);P & \longrightarrow & (\nu\, s)\,(P\{\overline{s}/x\} \mid \overline{s}\,[S, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \mid \overline{a}\,\langle s\rangle) & (s \notin \mathrm{fn}(P)) \\
\text{[Request2]} & a\,[\vec{s}] \mid \overline{a}\,\langle s\rangle & \longrightarrow & a\,[\vec{s}\cdot s] & \\
\text{[Accept]} & a(x{:}S).P \mid a\,[s\cdot\vec{s}] & \longrightarrow & P\{s/x\} \mid s\,[S, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \mid a\,[\vec{s}] & \\
\text{[Send]} & s!\langle v\rangle;P \mid s\,[!(T);S, \mathtt{o}:\vec{h}] & \longrightarrow & P \mid s\,[S, \mathtt{o}:\vec{h}\cdot v] & \\
\text{[Receive]} & s?(x).P \mid s\,[?(T);S, \mathtt{i}:v\cdot\vec{h}] & \longrightarrow & P\{v/x\} \mid s\,[S, \mathtt{i}:\vec{h}] & \\
\text{[Sel], [Bra]} & s \triangleleft l_i;P \mid s\,[\oplus\{l_i:S_i\}_{i\in I}, \mathtt{o}:\vec{h}] & \longrightarrow & P_i \mid s\,[S_i, \mathtt{o}:\vec{h}\cdot l_i] & (i \in I) \\
& s \triangleright \{l_j:P_j\}_{j\in J} \mid s\,[\&\{l_i:S_i\}_{i\in I}, \mathtt{i}:l_i\cdot\vec{h}] & \longrightarrow & P_i \mid s\,[S_i, \mathtt{i}:\vec{h}] & (i \in I \cap J) \\
\text{[Comm]} & s\,[\mathtt{o}:v\cdot\vec{h}] \mid \overline{s}\,[\mathtt{i}:\vec{h}'] & \longrightarrow & s\,[\mathtt{o}:\vec{h}] \mid \overline{s}\,[\mathtt{i}:\vec{h}'\cdot v] & \\
\text{[Instance]} & \mathtt{def}\ D\ \mathtt{in}\ (X\langle\vec{v}\rangle \mid Q) & \longrightarrow & \mathtt{def}\ D\ \mathtt{in}\ P\{\vec{v}/\vec{x}\} \mid Q & X(\vec{x}) = P \in D \\
\text{[Arriv-req]} & E[\mathtt{arrived}\ a] \mid a\,[\vec{s}] & \longrightarrow & E[b] \mid a\,[\vec{s}] & (|\vec{s}| \geq 1) \downarrow b \\
\text{[Arriv-msg]} & E[\mathtt{arrived}\ s\ h] \mid s\,[\mathtt{i}:\vec{h}] & \longrightarrow & E[b] \mid s\,[\mathtt{i}:\vec{h}] & (\vec{h} = h\cdot\vec{h}') \downarrow b \\
\text{[Typecase]} & & & & \\
& \mathtt{typecase}\ s\ \mathtt{of}\ \{(x_i:T_i)\,P_i\}_{i\in I} \mid s\,[S] & \longrightarrow & P_i\{s/x_i\} \mid s\,[S] & \exists i \in I.(\forall j < i.T_j \not\leq S \wedge T_i \leq S)
\end{array}
$$

Fig. 4. Selected reduction rules.

Our calculus incorporates two forms of asynchronous communication, *asynchronous session initiation* [23] and *asynchronous session communication* (over an established session). The former models the *unordered* transport of session request messages to acceptors listening on a shared channel. We use $\overline{a}\,\langle s\rangle$ to represent a request message in transit on shared channel a, carrying a fresh session channel s of type S. In real network communications, messages are buffered for reading on arrival at the destination. This mechanism is formalised by introducing a *shared input queue* $a\,[\vec{s}]$, which represents an acceptor's input buffer at a containing the pending requests for sessions $\vec{s}$.

Communication in an established session is asynchronous but *order-preserving*, as in TCP. For this purpose, each session channel s is associated with an *endpoint configuration* (or simply *configuration*) $s\,[S, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h}']$, which encapsulates both input (`i`) and output (`o`) buffers. Sending a message first enqueues it at the source `o`-buffer before it is eventually transferred to the destination `i`-buffer, signifying the arrival of that message. For both unordered session requests and ordered session messages, decoupling message transmission and arrival captures the intuitive semantics for `arrived`: only messages that are present in the local input buffer can be detected, and *not* those still in transit. The S in $s\,[S, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h}']$ is called the *active type*, and represents the remaining session actions to be performed at this endpoint (representing a runtime session monitoring mechanism). For brevity, one or more components may be omitted from a configuration when they are irrelevant, written as e.g. $s\,[S]$ or $s\,[\mathtt{i}:\vec{h}]$.

$(\nu\, s)\,P$ binds *both* session endpoints, s and $\overline{s}$, making them private within P. The remaining constructs, conditional, parallel composition, agent definition and instantiation, and inaction, are standard. Type annotations and $\mathbf{0}$ are often omitted. The notions of free variables and channels are standard [28]; we write $\mathrm{fn}(P)$ for the set of free channels in P. Terms in closed user syntax are called *programs*.

3.2 Operational Semantics

The reduction relation on closed terms $\longrightarrow$ captures the communication and event handling dynamics of ESP processes, and updates active types as session interactions progress. Figure 4 lists the key rules. We use the standard evaluation contexts $E[\_]$ defined as $E ::= \ - \ \mid \ s!\langle E\rangle;P \ \mid \ \texttt{if}\ E\ \texttt{then}\ P\ \texttt{else}\ Q \ \mid \ X\langle \vec{v}E\vec{e}\rangle$. Structural congruence $\equiv$ and the omitted reduction rules are standard; the full definitions are found at [30].

[Request1] issues a new request for a session of type S via shared channel a. A fresh (i.e. ν-bound) session with endpoints s (acceptor-side) and $\overline{s}$ (requestor-side) and the initial configuration at the requestor are generated, dispatching the session request message $\overline{a}\langle s\rangle$. [Request2] enqueues the request in the shared input queue at a. [Accept] dequeues the earliest session request, instantiates the session to the s in the request message, and creates the acceptor-side configuration: the new session is now established.

[Send] enqueues a value in the o-buffer of the *local* configuration and removes the prefix from the current active type, signifying the completion of this action. [Receive] dequeues the first value from the i-buffer of the local configuration and updates the active type accordingly. [Sel] and [Bra] similarly enqueue and dequeue a label, using the label to select the appropriate case in the current active type. Note these rules manipulate only the local configurations, and output actions are always non-blocking. The actual transmission of a session message is embodied by [Comm], which removes the first message from the o-buffer of the source configuration and enqueues it in the i-buffer at the opposing configuration. [Instance] is a standard recursion rule for processes.

Although input actions block if no message is available in the corresponding input buffer, blocking can be avoided using the message arrival predicates. [Arriv-req] evaluates `arrived` a to `tt` if the queue is non-empty; similarly for `arrived` k (rule omitted). [Arriv-msg] evaluates `arrived` $s\ h$ to `tt` if the queue is non-empty and the first message matches h. The notation $e \downarrow b$ means e evaluates to the boolean value b. Lastly, [Typecase] is the key rule which enables *dynamic* inspection of the active type of a session. The process continues the session s along the first P_i for which T_i can be successfully matched against the current active type S up to subtyping (defined in § 4.1).

3.3 Representing High-level Event Constructs in ESP

Example 3.1 (Selector and Event Loops). The core functionality of `SJSelector` (illustrated in § 2) can be distilled to three operations: *create* a new selector, *register* a channel with the selector, and *select* (i.e. retrieve from the selector) a channel on which a message has arrived. Session typecase is then used to type the selected channel. We extend ESP with these operations, with the following reduction semantics.[4] We omit type annotations for selectors, which we shall discuss in § 4.

$$\texttt{new}\ \mathsf{sel}\ r\ \texttt{in}\ P \longrightarrow (\nu\, r)(P|\mathsf{sel}\langle r, \varepsilon\rangle) \qquad \texttt{reg}\ s\ \texttt{to}\ r\ \texttt{in}\ P|\mathsf{sel}\langle r, \vec{s}\rangle \longrightarrow P|\mathsf{sel}\langle r, \vec{s}\cdot s\rangle$$

$$\texttt{select}(r)\{(x_i:T_i) : P_i\}_{i\in I}|\mathsf{sel}\langle r, s\cdot\vec{s}\rangle|s[S, \texttt{i}:\vec{h}] \longrightarrow P_i\{s/x_i\}|\mathsf{sel}\langle r, \vec{s}\rangle|s[S, \texttt{i}:\vec{h}] \qquad (\vec{h}\neq\varepsilon)$$

$$\texttt{select}(r)\{(x_i:T_i) : P_i\}_{i\in I}|\mathsf{sel}\langle r, s\cdot\vec{s}\rangle|s[\texttt{i}:\varepsilon] \longrightarrow \texttt{select}(r)\{(x_i:T_i) : P_i\}_{i\in I}|\mathsf{sel}\langle r, \vec{s}\cdot s\rangle|s[\texttt{i}:\varepsilon]$$

[4] The selector semantics presented here is based on polling, which is suitable for our current purpose (i.e. giving a semantic basis for selectors). For further discussion on the behaviour and implementation of selectors, see § 5.2.

In the second line, S and T_i satisfy the condition for [Typecase] in Figure 4. We also add structural rules, e.g. $(\nu\, r)\mathsf{sel}\langle r, \varepsilon\rangle \equiv \mathbf{0}$. Operator $\texttt{new}\ \mathsf{sel}\ r\ \texttt{in}\ P$ (binding r in P) creates a new selector $\mathsf{sel}\langle r, \varepsilon\rangle$, named r, with the empty queue ε. $\texttt{reg}\ s\ \texttt{to}\ r\ \texttt{in}\ P$ registers the session channel with r, adding s to the queue $\vec{s}$. $\texttt{select}(r)\{(x_i : T_i) : P_i\}_{i\in I}$ checks whether a message is available (i.e. an event has occurred) on the first session in the queue, s. If so, we select the first P_i for which the type of s matches T_i (condition omitted); otherwise, s is re-enqueued and the next session is tested.

We now show this behaviour can be easily encoded by combining the *message arrival predicate* and *session typecase*. We again omit type annotations (until § 4).

$$[\![\texttt{new}\ \mathsf{sel}\ r\ \texttt{in}\ P]\!] \stackrel{\text{def}}{=} (\nu\, b)(\overline{b}(\overline{r}); b(r).[\![P]\!] \mid b : [\varepsilon]) \qquad [\![\texttt{reg}\ s\ \texttt{to}\ r\ \texttt{in}\ P]\!] \stackrel{\text{def}}{=} \overline{r}!\langle s\rangle; [\![P]\!]$$

$$[\![\texttt{select}(r)\{(x_i : S_i) : P_i\}_{i\in I}]\!] \stackrel{\text{def}}{=} \qquad [\![\mathsf{sel}\langle r, \vec{s}\cdot\vec{s}'\rangle]\!] \stackrel{\text{def}}{=} \overline{r}[\mathtt{o} : \vec{s}'] \mid r[\mathtt{i} : \vec{s}]$$

$$\texttt{def}\ \mathsf{Select}(x\overline{x}) = x?(y); \texttt{if}\ \texttt{arrived}\ y\ \texttt{then}\ \texttt{typecase}\ y\ \texttt{of}\ \{(x_i : T_i) : [\![P_i]\!]\}_{i\in I}$$
$$\texttt{else}\ \overline{x}!\langle y\rangle; \mathsf{Select}\langle x\overline{x}\rangle \qquad \texttt{in}\ \mathsf{Select}\langle r\overline{r}\rangle$$

The use of `arrived` is the key to avoiding blocked inputs, allowing the selector to proceed asynchronously while handling any available messages. The operations on the collection queue (via r and $\overline{r}$) exchange session channels, hence session delegation [19] is essential. We can easily check that this encoding is operationally faithful to the native selector (i.e. the direct ESP extension) using a suitable bisimulation. Using the above selector, a basic event loop similar to Figure 1 (§ 2) can be represented as:

$$\texttt{new}\ \mathsf{sel}\ r\ \texttt{in}\ \texttt{reg}\ s_1\ \texttt{to}\ r\ \texttt{in}\ \texttt{reg}\ s_2\ \texttt{to}\ r\ \texttt{in}\ ...\texttt{reg}\ s_n\ \texttt{to}\ r\ \texttt{in}$$
$$\texttt{def}\ \mathsf{Loop} = \texttt{select}(r)\ \{(x_1?(U_1); ?(U_1); !(U_2)) : x_1?(y_1).\texttt{reg}\ x_1\ \texttt{to}\ r\ \texttt{in}\ \mathsf{Loop}$$
$$(x_2?(U_1); !(U_2)) : \qquad x_2?(y_2).x_2!\langle v\rangle; \mathsf{Loop}\} \qquad \texttt{in}\ \mathsf{Loop}$$

In § 4.4, we prove *event progress* for processes that use selectors, such as event loops.

Example 3.2 (Switch-receive). *Join patterns* [4, 13] are one mechanism for correlating multiple event occurrences [11]. Programmers use join patterns as guards to handle particular combinations (i.e. conjunctions) of message arrivals on one or more sessions. The `switch receive` construct in Sing# [12] implements this mechanism for *channel contracts* (a version of session types), which we can formalise as an ESP extension.

$$\texttt{switch-receive}\{J_1 : P_1, ..., J_m : P_m\} \qquad J_j \ ::= \ s_{j1}.l_{j1}(x_{j1} : U_{j1}) \wedge ... \wedge s_{jn_j}.l_{jn_j}(x_{jn_j} : U_{jn_j})$$

Above we set $m, n, j \geq 1$ and all $s_{j1}, .., s_{jn_j}$ should be pairwise distinct for each j. Each J_j denotes a join pattern, a conjunction of expressions of the form $s_{ji}.l_{ji}(x_{ji} : U_{ji})$, where l_{ji} is a branch label expected at s_{ji}, and $(x_{ji} : U_{ji})$ is a formal parameter for a message of type U_{ji} following l_{ji}. The formal semantics of `switch-receive` can be found at [30]; here, we informally illustrate its behaviour through a simple example. Let R be:

$$R \quad \stackrel{\text{def}}{=} \quad \texttt{switch-receive}\{s.l_1(x_1) : P,\ s.l_2(x_2) \wedge s'.l_3(x_3) : Q\}$$

R listens on s (where l_1 or l_2 is expected) and s' (for l_3); by the above design, a message is expected to follow each label. Suppose l_1 and then v_1 arrive: R will become $P\{v_1/x_1\}$. On the other hand, if l_2 and v_2 arrive at s, and v_2 and l_3 and v_3 at s', then R becomes $Q\{v_2v_3/x_2x_3\}$. In all other cases, R continues to wait for further messages on s and s'.

An inductive ESP encoding of `switch-receive` can be formulated using `arrived`. We illustrate the idea of the encoding using R from above. Assuming that the communication of a branch label is always followed by a message: $\texttt{def}\ \mathsf{SRLoop} =$

$$\begin{array}{lll} \texttt{if} & (\texttt{arrived}\ s\ l_1) & \texttt{then}\ s \triangleright l_1 : s?(x_1).[\![P]\!] \\ \texttt{else if} & (\texttt{arrived}\ s\ l_2\ \texttt{and}\ \texttt{arrived}\ s'\ l_3) & \texttt{then}\ s \triangleright l_2 : s?(x_2).s' \triangleright l_3 : s'?(x_3).[\![Q]\!] \\ & & \texttt{else}\ \mathsf{SRLoop} \qquad\qquad \texttt{in}\ \mathsf{SRLoop} \end{array}$$

Above, the sequential branch notation $s_i \triangleright l_i : P$ stands for a branch at s_i that omits the superfluous branches ruled out by the preceding `arrived`. More complex join patterns featuring predicates on received values are also encodable into ESP.

4 Typing Eventful Sessions

This section presents a type discipline for ESP and establishes its key properties: subtyping (Proposition 4.1); type safety (Theorem 4.2); communication and event-handling safety (Theorem 4.3); soundness of the ESP encoding of a high-level event primitive, selector (Proposition 4.4); and event progress (Theorem 4.6).

4.1 Subtyping from Composability

If P has a session channel s of type S, the ways in which P can use s are *at most* as S; e.g. if S is $\&\{l_1 : S_1, l_2 : S_2\}$, then P handles labels l_1 and l_2 but not any others, thus P can interact with peers that select either one of these two labels. By this intuition, for a process Q with session type S' to be safely used in place of P (i.e. $S' \leq S$), Q should be composable in the same or more ways (i.e. with more peers) than P; e.g. if S' is $\&\{l_i : S_i\}_{1 \leq i \leq 3}$, then Q can also interact with peers that select l_3. Formally, the subtyping relation is defined on the set of all closed and contractive types $\mathscr{T}$ as follows: T is a subtype of T', written $T \leq T'$, if (T, T') is in the largest fixed point of the monotone function $\mathscr{F} : \mathscr{P}(\mathscr{T} \times \mathscr{T}) \to \mathscr{P}(\mathscr{T} \times \mathscr{T})$, where $\mathscr{F}(\mathscr{R})$ is given by:

$$\begin{array}{ll} & \{(\mathsf{bool}, \mathsf{bool})\} \cup \{(\langle S \rangle, \langle S' \rangle) \mid (S, S'), (S', S) \in \mathscr{R}\} \\ \cup & \{(\mu \mathsf{X}.U, U') \mid (U\{\mu \mathsf{X}.U/\mathsf{X}\}, U') \in \mathscr{R}\} \cup \{(U, \mu \mathsf{X}.U') \mid (U, U'\{\mu \mathsf{X}.U'/\mathsf{X}\}) \in \mathscr{R}\} \\ \cup & \{(!(T_1); S_1', !(T_2); S_2') \mid (T_2, T_1), (S_1', S_2') \in \mathscr{R}\} \cup \{(?(T_1); S_1', ?(T_2); S_2') \mid (T_1, T_2), (S_1', S_2') \in \mathscr{R}\} \\ \cup & \{(\oplus\{l_i : S_i\}_{i \in I}, \oplus\{l_j : S_j'\}_{j \in J}) \mid \forall i \in I \subseteq J. (S_i, S_i') \in \mathscr{R}\} \\ \cup & \{(\&\{l_i : S_i\}_{i \in I}, \&\{l_j : S_j'\}_{j \in J}) \mid \forall j \in J \subseteq I. (S_j, S_j') \in \mathscr{R}\} \\ \cup & \{(\mu \mathsf{X}.S, S') \mid (S\{\mu \mathsf{X}.S/\mathsf{X}\}, S') \in \mathscr{R}\} \cup \{(S, \mu \mathsf{X}.S') \mid (S, S'\{\mu \mathsf{X}.S'/\mathsf{X}\}) \in \mathscr{R}\} \\ \cup & \{(\{S_i\}_{i \in I}, \{S_j'\}_{j \in J}) \mid \neg(|I| = |J| = 1), \forall j \in J, \exists i \in I. (S_i, S_j') \in \mathscr{R}\} \end{array}$$

Line 1 is standard: $\langle S \rangle$ is invariant on S since it supports both S and $\overline{S}$ (see duality below). Lines 2 and 6 are the standard rules for recursion. In Line 3, the linear output (resp. input) is contravariant (resp. covariant) on the message type following [28]. In Line 4, a select that requires support for more labels means fewer peers can be safely composed; dually for branching in Line 5. Finally, the ordering of set types says that if every element in the set type $\{S_j'\}_{j \in J}$ has a subtype in $\{S_i\}_{i \in I}$, then the latter is at least as composable as the former. The condition $\neg(|I| = |J| = 1)$ avoids the case where $\{S_i\}_{i \in I} = S$ and $\{S_j'\}_{j \in J} = S'$, which would make the relation universal.

We now clarify the semantics of $\leq$ using *duality*. The dual of S, denoted $\overline{S}$, is defined in the standard way: $\overline{!(T); S} = ?(T); \overline{S}$, $\overline{?(T); S} = !(T); \overline{S}$, $\overline{\mu \mathsf{X}.S} = \mu \mathsf{X}.\overline{S}$, $\overline{\mathsf{X}} = \mathsf{X}$, $\overline{\oplus\{l_i : S_i\}_{i \in I}} = \&\{l_i : \overline{S_i}\}_{i \in I}$, $\overline{\&\{l_i : S_i\}_{i \in I}} = \oplus\{l_i : \overline{S_i}\}_{i \in I}$ and $\overline{\mathtt{end}} = \mathtt{end}$. The set of *composable* types of $\{S_i\}_{i \in I}$ is defined as: $\mathsf{comp}(\{S_i\}_{i \in I}) = \cup_{i \in I}\{S' \mid S' \leq \overline{S_i}\}$. We observe:

Proposition 4.1 (Subtyping Properties). *(1) $\leq$ is a preorder; (2) given T and T', $T \leq T'$ is decidable; and (3) (semantics of $\leq$) $T_1 \leq T_2$ if and only if $\mathsf{comp}(T_2) \subseteq \mathsf{comp}(T_1)$.*

$$\frac{}{\Gamma\cdot u:T\vdash u:T}(\mathsf{Shared}) \quad \frac{\Gamma\vdash u:\langle S\rangle}{\Gamma\vdash \mathtt{arrived}\,u:\mathsf{bool}}(\mathsf{aReq}) \quad \frac{\Gamma,\Sigma\vdash \mathtt{arrived}\,k\,h:\mathsf{bool}}{\Gamma,\Sigma\vdash \mathtt{arrived}\,k\,:\mathsf{bool}}(\mathsf{aMsg})$$

$$\frac{\Gamma\vdash v:U}{\Gamma,\Sigma\cdot k:?(U);S\vdash \mathtt{arrived}\,k\,v:\mathsf{bool}}(\mathsf{aVal}) \quad \frac{l\in\{l_i\}_{i\in I}}{\Gamma,\Sigma\cdot k:\&\{l_i:S_i\}_{i\in I}\vdash \mathtt{arrived}\,k\,l:\mathsf{bool}}(\mathsf{aLab})$$

$$\frac{\Gamma\vdash P\triangleright\Sigma' \quad \Sigma'\leq\Sigma}{\Gamma\vdash P\triangleright\Sigma}(\mathsf{Subs}) \quad \frac{\Gamma\vdash u:\langle S\rangle \quad \Gamma\vdash P\triangleright\Sigma\cdot x:S}{\Gamma\vdash u(x:S).P\triangleright\Sigma}(\mathsf{Acc}) \quad \frac{\Gamma\vdash u:\langle S\rangle \quad \Gamma\vdash P\triangleright\Sigma\cdot x:\overline{S}}{\Gamma\vdash \overline{u}(x:\overline{S});P\triangleright\Sigma}(\mathsf{Req})$$

$$\frac{\Gamma,\Sigma\vdash e:U \quad \Gamma\vdash P\triangleright\Sigma\cdot k:S}{\Gamma\vdash k!\langle e\rangle;P\triangleright\Sigma\cdot k:!(U);S}(\mathsf{Send}) \quad \frac{\Gamma\cdot x:U\vdash P\triangleright\Sigma\cdot k:S}{\Gamma\vdash k?(x).P\triangleright\Sigma\cdot k:?(U);S}(\mathsf{Recv})$$

$$\frac{\Gamma\vdash P\triangleright\Sigma\cdot k:S}{\Gamma\vdash k!\langle k'\rangle;P\triangleright\Sigma\cdot k:!(T);S\cdot k':T}(\mathsf{SSend}) \quad \frac{\Gamma\vdash P\triangleright\Sigma\cdot k:S\cdot x:T}{\Gamma\vdash k?(x).P\triangleright\Sigma\cdot k:?(T);S}(\mathsf{SRecv})$$

$$\frac{1\leq i\leq n \quad \Gamma\vdash P\triangleright\Sigma\cdot k:S_i}{\Gamma\vdash k\triangleleft l_i;P\triangleright\Sigma\cdot k:\oplus\{l_i:S_i\}_{i\in I}}(\mathsf{Select}) \quad \frac{\forall i.1\leq i\leq n \quad \Gamma\vdash P_i\triangleright\Sigma\cdot k:S_i}{\Gamma\vdash k\triangleright\{l_i:P_i\}_n\triangleright\Sigma\cdot k:\&\{l_i:S_i\}_{i\in I}}(\mathsf{Branch})$$

$$\frac{\Gamma,\Sigma\vdash e:\mathsf{bool}\ \Gamma\vdash P\triangleright\Sigma\ \Gamma\vdash Q\triangleright\Sigma}{\Gamma\vdash \mathtt{if}\,e\,\mathtt{then}\,P\,\mathtt{else}\,Q\triangleright\Sigma}(\mathsf{If}) \quad \frac{\Gamma\cdot a:\langle S\rangle\vdash P\triangleright\Sigma\cdot a}{\Gamma\vdash(\nu\,a:\langle S\rangle)P\triangleright\Sigma}(\mathsf{Chan}) \quad \frac{\Gamma\vdash P\triangleright\Sigma \quad \Gamma\vdash Q\triangleright\Sigma'}{\Gamma\vdash P\mid Q\triangleright\Sigma\cdot\Sigma'}(\mathsf{Par})$$

$$\frac{\forall i\in I \quad \Gamma\vdash P_i\triangleright\Sigma\cdot x_i:T_i \quad \cup_{i\in I}T_i\leq T}{\Gamma\vdash \mathtt{typecase}\,k\,\mathtt{of}\,\{(x_i:T_i)\,P_i\}_{i\in I}\triangleright\Sigma\cdot k:T}(\mathsf{Typecase}) \quad \frac{\Sigma\ \mathtt{end\ only}}{\Gamma\vdash a[\varepsilon]\triangleright\Sigma\cdot a}(\mathsf{Queue}) \quad \frac{\Sigma\ \mathtt{end\ only}}{\Gamma\vdash\mathbf{0}\triangleright\Sigma}(\mathsf{Nil})$$

Fig. 5. Selected typing rules for ESP programs.

4.2 Program Typing

We first define a typing system for *programs* (§ 3.1). Program typing, which can be considered a static typing phase performed by a compiler on user-level code before execution, uses two environments:

$$\Gamma ::= \emptyset \mid \Gamma\cdot u:U \mid \Gamma\cdot X:\vec{T} \qquad \Sigma ::= \emptyset \mid \Sigma\cdot a \mid \Sigma\cdot k:\{S_i\}_{i\in I}$$

Γ is called the *shared environment*, which maps variables and shared channels to constant types and process variables to sequences of message types; Σ is called the *linear environment*, which maps session channels to set types (writing $k:S$ for $k:\{S\}$) and records the shared channels that have input queues (to ensure that each a has exactly one queue). $\Sigma\cdot a$ means $a\notin\mathrm{dom}(\Sigma)$, and similarly for others. Subtyping is extended to environments by $\Sigma\leq\Sigma'$ iff $\mathrm{dom}(\Sigma)=\mathrm{dom}(\Sigma')$ and $\forall k\in\mathrm{dom}(\Sigma).\Sigma(k)\leq\Sigma'(k)$. The typing judgements for processes and expressions are:

$$\Gamma\vdash P\triangleright\Sigma \qquad \Gamma,\Sigma\vdash e:T$$

On the left, the program typing judgement says the program P, under shared environment Γ, features the channel usage specified by linear environment Σ; similarly for the expression typing judgement, which can be shortened to $\Gamma\vdash e:T$ if Σ is not required.

Figure 5 presents selected typing rules for programs (the rules can be seen in full at [30]). (Shared) is the standard rule for shared channel expressions (the Figure omits the other standard expression typing rules, e.g. $\Gamma\vdash\mathtt{tt}:\mathsf{bool}$, $\Gamma\cdot x:\mathsf{bool}\vdash x:\mathsf{bool}$, etc.). The next four rules check that the message arrival predicate is used appropriately. (aReq) is for session request arrival on shared channels. (aVal) and (aLab) are respectively for the arrival of a specific value and branch label on a session channel; (aMsg) is for either kind and any value of session message. (Subs) is standard subsumption.

Although ESP has asynchronous session initiation and i/o-buffered communication semantics, program typing for the basic session initiation and communication actions remain standard. Rule (Acc) (resp. (Req)) says that the session implementation following an accept (resp. request) should conform to the type annotation and the shared channel type. (Send) and (Recv) check that the expected value types are communicated. For an output, we check the session type prefix is $!(U)$ where U is the message expression type; dually for input. Rules (SSend) and (SRecv) for session delegation are similar. (Select) checks that the selection action chooses and follows one of the specified branches; (Branching) checks that branching offers at least the specified branches. The (If) rule checks conditional expressions.

(Queue) records the presence of an empty shared input queue in the linear environment. (Par) disallows multiple queues for the same shared channel, and prevents the composition of processes with the same session (i.e. linear) channels. (Chan) records $a : \langle S \rangle$ in the shared environment after checking the presence of a (unique) shared queue for a. (Nil) and the omitted rules for agent definition and instantiation are standard from [19]. "Σ `end` only" means $\forall k \in \mathrm{dom}(\Sigma), \Sigma(k) = \mathtt{end}$. Finally, (Typecase) is an extension of dynamic types in the λ-calculus [1] to session types. It checks for each case that the body P_i is typable under Σ with the session channel k "rebound" to x_i as type T_i. Then the whole process is typed with k set to T, which is a supertype of all T_i.

4.3 Type Soundness and Event-Handling Safety

This subsection establishes the fundamental safety properties of ESP program typing. The proofs require additional mechanisms for runtime process typing, detailed in Appendix A. We note here that our approach extends [28] to support the new configuration active types and the finer-grained i/o-buffers. We start with type soundness.

Theorem 4.2 (Type Soundness). *(1) If $\Gamma \vdash P \triangleright \Sigma$ and $P \equiv P'$, then we have $\Gamma \vdash P \triangleright \Sigma$. (2) If $\Gamma \vdash P \triangleright \emptyset$ and $P \longrightarrow Q$, then we have $\Gamma \vdash Q \triangleright \emptyset$.*

Next we prove communication safety. An *s-redex* is a parallel composition of two processes that has one of the following shapes:

(a) $s!\langle v \rangle; P \mid s[!(T); S, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h}']$ (b) $s \triangleleft l_j; P \mid s[\oplus\{l_i : S_i\}_{i\in I}, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h}']$ with $j \in I$
(c) $s?(x).P \mid s[?(T); S, \mathtt{i}:v\cdot\vec{h}, \mathtt{o}:\vec{h}']$ (d) $s \triangleright \{l_j : P_j\}_{j\in J} \mid s[\&\{l_i : S_i\}_{i\in I}, \mathtt{i}:l_{i'}\cdot\vec{h}]$ with $i' \in I \cap J$
(e) $s[S, \mathtt{i}:\vec{h}_1, \mathtt{o}:v\cdot\vec{h}'_1] \mid \bar{s}[S', \mathtt{i}:\vec{h}_2, \mathtt{o}:\vec{h}'_2]$
(f) $E[\mathtt{arrived}\ s\ v] \mid s[?(U); S, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h}']$ with v of type U
(g) $E[\mathtt{arrived}\ s\ l_j] \mid s[\&\{l_i : S_i\}_{i\in I}, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h}']$ with $j \in I$
(h) $\mathtt{typecase}\ s\ \mathtt{of}\ \{(x_i : T_i)\, P_i\}_{i\in I} \mid s[S]$ with $\exists i \in I.\ T_i \leq S$

All redexes require the immediate action to correspond with the active type prefix in the local configuration. (f–h) are for the new primitives for asynchronous event handing. We say a process P is an *error* if up to structural congruence (following [20, § 5]), P contains more than two s-processes which do not form an s-redex, or an expression in P contains a type error in the standard sense. Then from Theorem 4.2 we obtain:

Theorem 4.3 (Communication and Event-Handling Safety). *If P is a well-typed program, then $\Gamma \vdash P \triangleright \emptyset$, and P never reduces to an error.*

4.4 Typing Event Selectors and Event Progress

Typing selectors. Typing rules for the extended ESP selector construct defined in Example 3.1 naturally follow from the ESP-typing of the selector encoding (§ 3.3). We restore the previously omitted selector type annotations: $\texttt{new}\ \texttt{sel}\langle T\rangle\ r\ \texttt{in}\ P$ creates a selector that stores channels of type T. Then the type for a *user* of the selector is written $\overline{\texttt{sel}}(T)$, and for the selector itself $\texttt{sel}(T)$. For simplicity, we assume these types do not occur as part of other types. The linear environment Σ is extended with two additional type assignments, $r:\overline{\texttt{sel}}(T)$ and $r:\texttt{sel}(T)$, the latter only used for runtime typing for selector queues. The program typing rules for the selector operations are:

$$\frac{\Gamma \vdash P \triangleright \Sigma\cdot r:\overline{\texttt{sel}}(T)}{\Gamma \vdash \texttt{new}\ \texttt{sel}\langle T\rangle\ r\ \texttt{in}\ P \triangleright \Sigma}\ (\mathsf{Selector}) \qquad \frac{\Gamma \vdash P \triangleright \Sigma\cdot r:\overline{\texttt{sel}}(T) \quad S \leq T}{\Gamma \vdash \texttt{reg}\ s\ \texttt{to}\ r\ \texttt{in}\ P \triangleright \Sigma\cdot r:\overline{\texttt{sel}}(T)\cdot s:S}\ (\mathsf{Reg})$$

$$\frac{\forall i \in I.\Gamma \vdash P_i \triangleright \Sigma\cdot r:\overline{\texttt{sel}}(T)\cdot x_i:S_i \quad S_i \leq T}{\Gamma \vdash \texttt{select}(r)\{(x_i:S_i) : P_i\}_{i\in I} \triangleright \Sigma\cdot r:\overline{\texttt{sel}}(T)}\ (\mathsf{Select})$$

We omit the runtime typing rules. By setting $[\![\Sigma]\!]$ as the compositional mapping such that $[\![r:\overline{\texttt{sel}}(T)]\!]$ is given as $r:S_r\cdot\overline{r}:\overline{S_r}$ when $S_r = \mu \mathsf{X}.?(T);\mathsf{X}$, and otherwise identity, as well as extending the notion of error to the internal typecase of the select operation, we obtain, writing ESP^+ for the extension of ESP with selectors:

Proposition 4.4 (Soundness of Selector Typing Rules).

1. *(Type Preservation)* $\Gamma \vdash P \triangleright \Sigma$ *in* ESP^+ *if and only if* $\Gamma \vdash [\![P]\!] \triangleright [\![\Sigma]\!]$.
2. *(Soundness)* $P \equiv P'$ *implies* $[\![P]\!] \equiv [\![P']\!]$*; and* $P \longrightarrow P'$ *implies* $[\![P]\!] \longrightarrow^* [\![P']\!]$.
3. *(Safety) A typable process in* ESP^+ *never reduces to an error.*

(1, 2) are straightforward. (3) is a corollary from (1, 2) and Theorems 4.2 and 4.3. This example demonstrates how the fine-grained typing rules of ESP can suggest and justify sound typing rules for high-level event handling constructs through ESP encodings.

The typing rules for the switch-receive construct from Example 3.2 can also be derived and justified by its encoding; the details are available from [30].

Event progress. The behaviour of the ESP^+ selector implicitly features *session delegation* (session channel passing) as get and put operations on channel collections. The presence of delegation generally makes it impossible to guarantee progress in session typed processes without specialised techniques [5].[5] However, we observe that a selector does *not* use delegation in an *arbitrary* way; indeed, one of the key characteristics of event-driven programs is their non-blocking nature. In the following, we show that an ESP^+-process driven by a selector does satisfy a strong form of progress.

Definition 4.5 (Selector Usage). We say a closed ESP^+-process $\Gamma \vdash P \triangleright \Sigma$ *uses selectors well* if each select action at r is preceded by a register action at r, and each register action at r is followed by a select action at r, both up to the unfolding of recursion; moreover each $\texttt{select}(r)\{(x_i:S_i) : P_i\}_{i\in I}$ in P satisfies: (1) (consumption) each S_i starts from a branching or linear input and P_i starts from the corresponding action on x_i; and (2) (exhaustiveness) in each P_i, no input or branching actions occur other than (1).

[5] One approach is to impose an order on the channels stored via session channel passing; however, the standard ordering techniques cannot be applied to event selectors (see § 6), so progress is difficult to establish in this way.

By the above conditions, all expected events will be handled on every registered channel: as far as the environment generates the events (i.e. produces the messages), the selector will proceed to process them one by one.

We now introduce two conditions from [20]. A typable ESP^+-process P is *simple* if the session typings in the premise and conclusion of each prefix rule from Figure 5 in P's typing derivation are restricted to at most a singleton; and $\Sigma = \emptyset$ in $(\mathsf{Selector},\mathsf{Reg},\mathsf{Sel})$.[6] We also say P is *well-linked* if $P \longrightarrow^* Q$ implies whenever Q has an active prefix whose subject is a (free or bound) shared name, then the dual active prefix always occurs in Q.

We use the refined reduction $\searrow$, defined as $\longrightarrow_s^* \longrightarrow_{ns} \longrightarrow_s^*$ where $\longrightarrow_s$ is the reduction induced by the last of the selector reduction rules in § 3.3; and $\longrightarrow_{ns} = \longrightarrow \setminus \longrightarrow_s$, that is $\longrightarrow_{ns}$ is the whole reduction minus $\longrightarrow_s$. We state the event progress theorem in terms of $\searrow$ because $\longrightarrow_s$ can still occur in a deadlocked configuration: it does not constitute a "progress step".

Theorem 4.6 (Event Progress). *We say an* ESP^+*-process P is* eventful *if $\Gamma \vdash P \triangleright \emptyset$, it is simple and well-linked, and it uses selectors well in the sense of Definition 4.5. Then we have: (1) if P is eventful and $P \longrightarrow Q$ then Q is eventful; and (2) if P is eventful then either $P \equiv \mathbf{0}$ or $P \searrow Q$ for some Q.*

This result strictly extends progress for session types to support implicit, well-disciplined usage of session delegation. In addition, all session actions registered with a selector will indeed be processed in a non-blocking fashion, formally justifying the implicit assumptions and expected behaviour of the standard event-driven programming patterns found in practice.

5 Eventful SJ: Implementation and Evaluation

This section firstly discusses the design and implementation of *Eventful SJ* (ESJ). We then report our experience of programming a substantial application in ESJ, a session-typed SMTP server, and discuss benchmark results.

5.1 Event Primitives and ESJ Compilation

The theoretical inquiries in § 3 and § 4 give a firm formal basis for, and insight on, potential primitives for event-based session programming. The current design of ESJ is based on the *selector* construct, *message arrival predicate* and the session *typecase*. Our selector enhances its untyped counterpart as found in Java NIO and Unix with session-typed operations. In § 3, we have seen that such a selector is encodable using the message arrival predicate through polling: however, polling is inefficient from the performance viewpoint, favouring the introduction of this construct as a primitive.

ESJ is implemented using Polyglot [29], and currently comprises approximately 30 KLOC of Java. First, the compiler statically type checks and transforms ESJ-programs

[6] This condition precludes the explicit use of delegation and the interleaving of sessions, but can be relaxed to support nested sessions as in [5].

```
SJProtocol _sjtypecase0 = new SJProtocol(...); // sbegin.pStream
SJProtocol _sjtypecase1 = new SJProtocol(...); // ?{NEXT: ..., QUIT: }
SJProtocol _sjtypecase2 = new SJProtocol(...); // ?(Data).pStream
... // Declaration and initialisation of the selector.
while(run) {
  { // Braces delimiting the lexical scope of the using statement.
    SJChannel c = null; // The using statement variable declaration.
    try {
      c = SJRuntime.select(sel); // 'sel' is the selector.
      SJSessionType _sjtmp0 = c.remainingSessionType();
      if(_sjtmp0.isSubtype(_sjtypecase0.getType())) { // Start of typecase.
        SJServerSocket ss = (SJServerSocket) c; // "Rebind"..
        c = null; // ..the typecase variable.
        ... // Call accept on the server socket; register the new session.
        ... // Re-register the server socket.
      } else if(_sjtmp0.isSubtype(_sjtypecase1.getType())) {
        ... // Translation of the inbranch for NEXT and QUIT cases.
      } else if(_sjtmp0.isSubtype(_sjtypecase2.getType())) {
        SJSocket s2 = (SJSocket) c;
        c = null;
        Data d = (Data) SJRuntime.receive(s2); // Cast inserted.
        ... // Translation of the session recursion construct.
      } else {
        throw new SJIOException("Runtime session typecase error: " + ...);
      } // End of typecase.
    } finally {
      SJRuntime.close(c);
} } }
```

Fig. 6. Compilation of the ESJ "event streams" example from Figure 2 to Java (extract).

into standard Java. The transformation serializes and embeds session type information into the generated classes, and translates the SJ session constructs into *transport-independent* SJ Runtime (SJR) operations. Then at runtime, the SJR (implemented as a Java library) is used to perform the abstract session operations as actions on a "concrete" transport connection, such as TCP or shared memory: the transport is negotiated at session initiation according to system and user parameters. In this way, the SJ framework supports the execution of SJ programs on any compliant JVM while decoupling type-safe session abstraction from specific modes of transport.

The new ESJ extensions replace the previous compiler with a generalised design based on session set typing (singleton set types subsume the previous non-set types). The use of `SJSelector` operations and `typecase` are checked as part of static session type checking following § 4.2 to preserve the safety properties presented in § 4.3, 4.4. Here, we illustrate the compiler translation of high-level session constructs into SJR operations using the ESJ "event streams" example from § 2, focusing on the treatment of `typecase` and `SJSelector`. Figure 6 extracts from the standard Java code generated by the ESJ compiler for the main event loop. As in C#, `using` statements are translated into `try-finally` statements with appropriate resource cleanup in the `finally` block, and the lexical scope of `using` variable declarations is controlled by an outer pair of

block braces. Basic session operations like `select` (Line 9) and `receive` (Line 21) are directly translated into SJR operations, passing the target references (respectively the `SJSelector sel` and the `SJSocket s2`) as arguments; similarly for `send` and `arrived`.

We now explain the translation of the `typecase`. First, the session types guarding each case are embedded into the parent class as serialized `SJProtocol` objects. We then insert a `remainingSessionType` call to the typecase target `c` (Line 10) to determine the runtime session type of the `SJChannel` when the typecase is performed. The structure of the `typecase` is translated into an `if-else` statement. Following the formal semantics of `typecase` (§ 3.2), the `if`-cases find the first `typecase`-case, according to their original syntactic order, where the current session type of `c` is a subtype of the specified type. The "rebinding" of the typecase variable in each case is achieved by inserting an appropriate cast: to `SJServerSocket` in the first case, and to `SJSocket` in the second, as directed by their corresponding session types. Note that the rebinding also sets `c` to `null`. Although session types are embedded in their serialized forms, the first call to `getType` on an `SJProtocol` decodes the type and caches the value for future use. The final `else` case serves only as a security measure, since session typing guarantees that at least one of the main cases will match.

5.2 Eventful SJ Runtime

SJ Runtime structure and the ESJ extensions. The SJ Runtime (SJR) provides an abstract platform for executing the same high-level, typed session programs over different transports. As described above, the SJ compiler transforms SJ classes into standard java binaries, translating user-level session constructs into SJR operations. The SJR operations target specific *Interaction Service* (IS) components, which form the upper layer of the SJR, encapsulating services such as session initiation, message serialization, branch synchronisation and session delegation [30]. The IS components are implemented in terms of actions on the *Abstract Transport Interface* (ATI), which specifies the communication actions of an idealised session transport: bi-directional, order-preserving and reliable delivery of byte segments. The ATI is in turn implemented by *Transport* components, encapsulating the concrete communication mechanisms of specific transports.

The new ESJ Runtime features several SJR extensions. To support the above translation of `typecase` (and following the formal semantics), we create an IS service for *runtime session type monitoring* that tracks the progress of session execution (i.e. the active type, § 3.1). The following focuses on the ESJ extensions related to the `SJSelector` facilities. The challenge is to integrate the infrastructure for asynchronous event handling underlying the `SJSelector` API in a way that (1) fits the transport-independent SJ framework, and (2) gives the performance expected from event-based systems.

Runtime support for session event selectors. Figure 7 depicts the main IS and Transport components involved in the execution of an event-driven SJ program implemented using the `SJSelector` API. The key elements of the ESJ extensions are the *event-driven ATI extensions*, the *asynchronous message deserializers*, and the abstract *selector services*.

Event-driven ATI. The ESJ ATI extensions export a `TransportSelector` interface for detecting asynchronous communication events at the transport level. The implementa-

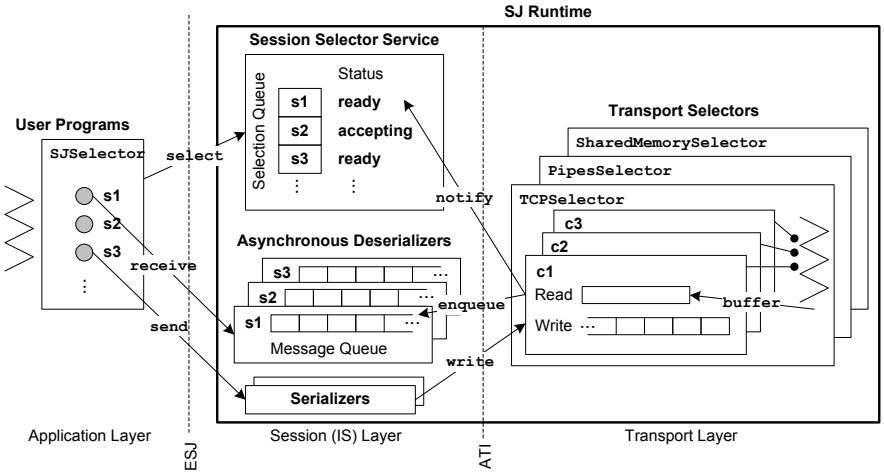

Fig. 7. The main ESJ Runtime components for executing event-driven session programs.

tion of each transport-specific selector is provided by the Transport component that implements the ATI, e.g. the `TCPSelector` provided by the TCP component uses Java NIO. On initialisation, the SJR creates and maintains one instance of the `TransportSelector` for each ESJ-compatible transport available. As Figure 7 shows, the ESJ ATI decouples the abstract session selector service (explained below) from the various transports over which the registered sessions are being conducted.

Asynchronous deserializers. Sessions initialised in asynchronous I/O mode (i.e. for event-driven execution via an `SJSelector`) specify an appropriate asynchronous deserializer component, which encapsulates the routines for converting transport-level binary data into application-level messages. When a read event occurs at the transport-level, the `TransportSelector` upcalls the deserializer to determine whether enough data for a complete application-level message has been received. If so, the message is enqueued, ready for user consumption, and any remaining data is re-buffered. Otherwise, the data is similarly re-buffered for use on the next read event.

Session selector services. Each `SJSelector` is supported by an instance of an IS-level `SJSelectorService`. This service maintains a record of the asynchronous sessions registered with the parent `SJSelector`, and in turn registers the ATI connection underlying each session with the appropriate `TransportSelector`. The `SJSelectorService` is notified by the `TransportSelector` when an application-level message has been deserialized and enqueued; the `SJSelectorService` can then report this event when `select` is called from the application-level.

Sending a message on an asynchronous session uses the serializer to directly enqueue the binary data for writing at the transport-level (i.e. in the local `o`-buffer); the send operation then immediately returns. Testing session message arrival in the upcalls from the `TransportSelector`, as opposed to e.g. polling from the IS-layer, is important for fair and efficient event detection in multi-transport contexts.

```
// Server-side SMTP.
protocol pSmtpServer {
  !<Greeting>
  .?(Ehlo)
  .!<EhloAck>
  .pBody
}
```

```
// SMTP session body.
protocol pBody {
  rec LOOP [
    ?{ // SMTP commands.
      MAIL: pMail.#LOOP,
      RCPT: pRcpt.#LOOP,
      DATA: pData.#LOOP,
      ...,
      QUIT: !<QuitAck>
} ] }
```

```
// MAIL command handler.
protocol pMail {
  ?(Address) // The sender.
  .!{ // Reply codes.
    RC250: !<AddrAck>,
    RC550: !<AddrError>,
    ...
  }
}
```

Fig. 8. Extracts from a server-side session type specification of SMTP.

5.3 An ESJ SMTP Server: Implementation Experience and Benchmarks

We implemented a session-typed event-driven SMTP server as a practical evaluation of ESJ. We use this advanced example to further examine the expressiveness and benefits of event-driven session programming and the performance of ESJ. SMTP [31] is an Internet standard for e-mail transfer, used by Mail Transfer Agents (MTA) to accept, route and deliver mails. Our implementation corresponds to a simplified MTA that does not permit message relaying, i.e. it only accepts mail addressed to the local domain. The full session type declarations and source code can be found at [30].

Session type specification. An SMTP session is a dialogue of client commands and server responses that carries out a sequence of zero or more mail transactions. We first declare a formal specification of SMTP using session types: the main structure of the server-side SMTP session type is listed in Figure 9 as a collection of SJ protocols. Following this specification, we now implement an event-driven server using ESJ.

Server implementation. The first step is to specify the SMTP server events by declaring the session set type for the selector that drives the main event loop of the server:

```
protocol pSmtpEvents { // The events to be handled by the SMTP server.
  sbegin.pSmtpServer, // SMTP session initiation event.
  ?(Ehlo).!<EhloAck>.pBody, // EHLO event.
  pBody, // Mail transaction command event: MAIL, RCPT, DATA, QUIT, ...
  pMail.pBody, // Sender address event for the MAIL command.
  pRcpt.pBody, // Recipient address event for the RCPT command.
  ?(MessageData).!<MessageDataAck>.pBody, // All message data received.
  ... }
```

Figure 9 outlines the structure of the main event loop and the handler for `MAIL` events. The `mainEventLoop` method takes the selector `sel` of the above type, and uses `typecase` to handle and dispatch the event occurrences accordingly. The first listed `when` case handles the `EHLO` event: the Server receives an `Ehlo` message, returns an `EhloAck` and re-registers the session with `sel` to wait for the first command in the main session body. The second `when` case, for the recursive type of the main session body (`pBody`), handles the Client commands for mail transactions. The `MAIL` and `RCPT` branch cases directly re-register the session for the subsequent `Address` message input. In the `DATA` branch case,

```
void mainEventLoop(SJSelector{pSmtpEvents} sel) throws ... {
  while(run) {
    using(SJChannel{pSmtpEvents} c = sel.select()) { // 'sel' is the SJSelector.
      typecase(c) {
        ... // SMTP initiation event: accept and register a new session.
        when(SJSocket{?(Ehlo).!<EhloAck>.pBody} s1) { // Received EHLO.
          Ehlo ehlo = s1.receive();
          s1.send(new EhloAck("250 Hello ..."));
          sel.register(s1); // Register SMTP session for the first command.
        }
        when(SJSocket{pBody} s2) { // The main mail transaction loop.
          s2.recursion(X) {
            s2.inbranch() { // Handle an SMTP command event.
              case MAIL: sel.register(s2); // Now expecting the Sender address.
              case RCPT: sel.register(s2); // Now expecting the Recipient addr.
              case DATA: handleData(sel, s2); // Handle the DATA command.
              ...
              case QUIT: s2.send(new QuitAck("221 ...")); // SMTP session end.
        } } }
        when(SJSocket{pMail.pBody} s3) { // Received Sender address.
          handleMail(s3); // Use the handler to perform the pMail subprotocol.
          sel.register(s3); // Register for the next command.
        }
        ... // Session typecase cases for the other SMTP command events.
} } } }

void handleMail(pMail s) throws SJIOException { // Handle MAIL command events.
  Address addr = s.receive(); // ?(Address)
  switch(checkAddress(addr)) { // !{RC250: !<AddrAck>, RC550: !<AddrError>, ...}
    case ADDR_OK:  s.outbranch(RC250) s.send(new AddrAck("OK")); break;
    case UNAVAIL:  s.outbranch(RC550) s.send(new AddrError("...")); break;
    default:       s.outbranch(...) ...; break;
} }
```

Fig. 9. The main event loop and the `MAIL` event handler from the ESJ SMTP server.

we pass the session to the omitted `handleData` method (which sends an `RC354` reply before similarly re-registering the session to await the message data). The `QUIT` branch case sends an acknowledgement, but does not re-register the session since it has now been completed. The final listed `when` case handles the arrival of the `Address` message for the `MAIL` branch by passing the session to the `handleMail` method, which receives the `Address` and returns one of the specified reply codes as appropriate. The `pMail` session type prefix of the `s3` argument at the point of the `handleMail` call corresponds to the session type of the `s` parameter of the method: this prefix is consumed by the method call and the remaining type of `s3` when it is re-registered with `sel` is again `pBody`.

Taking advantage of SJ framework modularity, we implement the ESJ SMTP server to be fully interoperable with non-SJ SMTP clients while retaining session type safety. Firstly, we provide application-specific serialization components: we read and write UTF-8, formatted according to the SMTP protocol, e.g. messages are terminated by

'CRLF'. Each message class, including those for branch labels, provides its own deserialization routine. The SJR uses the *current type* of each session (tracked by the runtime session monitor) to determine the expected message type(s) and apply the appropriate routine. Note that the operational semantics of enacting a session recursion does not involve any underlying communications (see [Instance] in Figure 4), and hence does not affect the non-SJ peer. Finally, the normal SJ peer compatibility check at session initiation is not possible with non-SJ peers: full communication safety is instead guaranteed through dynamic typing of non-SJ client actions by the session monitoring service.

ESJ programming experience. Our experience of implementing the SMTP server reinforces the following attributes of event-driven session programming.

1. Session types promote clear and safe implementation of complex, real-world protocols featuring branching and recursion. The ability to decompose protocols into subprotocols (e.g. `pBody`, `pMail`) and explicit specification of system events (e.g. `pSmtpEvents`) greatly facilitates the structuring of asynchronous event-driven code. Session types ensure precise identification of event types when handling multiple, concurrent sessions at different stages of execution.
2. Our implementation is guaranteed to conform to the session type specification of SMTP through static typing, and all events will be handled correctly until session completion. Together with session monitoring to verify client conformance (for non-SJ clients), runtime communication safety is guaranteed; we have tested our server against commercial clients such as Microsoft Outlook and Apple Mail.
3. The programmer controls the level of detail at which the protocol is represented by session types. Due to space considerations, the presented SMTP specification is relatively basic: it is fully possible to capture finer-grained details of the SMTP protocol, such as the strict order for `MAIL`, `RCPT` and `DATA` commands, for verification by static type checking. Session set type subtyping also offers a natural mechanism for refining event-driven implementations to support additional event types.
4. The ESJ implementation is inherently cross-transport: the IS services for session events and custom message serialization run above the ATI, and are hence re-usable over different transports, e.g. TCP, TLS, transports for LAN messaging, etc.

We have re-used the above components to implement a type-safe SJ STMP client that is similarly interoperable with existing SMTP servers [30]. Language facilities for the formal declaration and static verification of protocols in communications-based programs will have significant impact on engineering application-level protocols. Richer protocols can be specified with more precision and less effort, supported by automatic static type checking that ensures compatibility between protocol implementations.

ESJ performance. We evaluate the performance of the event-driven ESJ SMTP server against an equivalent multithreaded SJ implementation. The results show that event-driven session programs exhibit the same performance characteristics (i.e. better scalability under high concurrent loads) as traditional event-driven programming in comparison to their multithreaded counterparts, while maintaining session type safety.

In this macro benchmark, we measure SMTP server throughput in terms of the total number of messages handled by the server while engaged in a varied number of concurrent client sessions. The server is hosted on one machine (locked to one core), and the

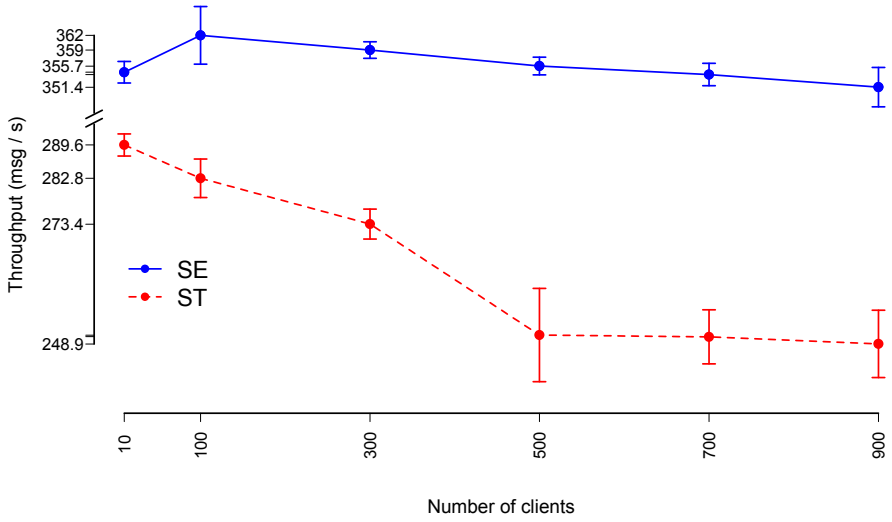

Fig. 10. Mean throughput and throughput standard deviation of multithreaded (ST) and event-driven (SE) SJ SMTP servers under increasing client loads.

remote clients are set to run continuously repeating mail transaction loops, submitting messages of certain sizes. The benchmark is conducted in a controlled cluster environment: each node is a Sun Fire x4100 with two 64-bit Opteron 275 processors at 2 GHz and 8 GB of RAM, running 64-bit Mandrakelinux 10.2 (kernel 2.6.11) and connected via gigabit Ethernet. Latency between each node is measured to be 0.5 ms on average (ping 64 bytes). The benchmark applications are compiled and executed using the Sun Java SE compiler and Runtime versions 1.6.0.

After server performance in the running benchmark configuration has stabilised, the throughput of the server is measured across a series of 50 consecutive 15 second windows; this process is repeated 10 times for each parameter combination, for each the two server implementations. Figure 10 gives the mean throughput and the throughput standard deviation (denoted by the vertical bars) for 10, 100, 300, 500, 700 and 900 concurrent clients and message size 1 KB: the multithreaded SJ server (ST) is the red line, and the event-driven ESJ server (SE) is the blue line. As expected, the results show that the performance of the multithreaded server degrades as the number of clients increases, i.e. mean throughput decreases while throughput variance increases, whereas the throughput of the event-driven SJ server is consistently higher and more stable across all client loads.

The full source code and raw results of this benchmark can be obtained from the SJ homepage [30]. The SJ homepage also presents further micro and macro ESJ benchmark results, including comparisons with multithreaded and event-driven programs implemented in standard Java: the results show that ESJ performs competitively against "untyped event-programming" using NIO. Earlier benchmark results comparing multithreaded SJ against standard TCP sockets and RMI were presented in [21].

6 Related Work

Event-driven and event-based programming. Traditional event-driven programming is known to attain performance and scalability at the cost of complex control flow and manual stack management [3, 34]. Consequently, many works have sought to make event programming easier by adding language features that raise the level of abstraction and/or facilitate code verification. Tame [25] introduces language features, similar to the synchronisation mechanisms of futures, that allow control flow to be returned from a blocked C++ function to the caller. The interface to the libeel library [9] is designed to clarify the relationships between event registrations and callbacks to support the accompanying tool suite for call graph analyses. The nesC language [14] promotes an event-based component design, based on interfaces that specify callback as well as event registration functions, to meet the requirements of sensor network programming. EventJava [11] integrates advanced event correlation techniques with O-O programming, providing high-level syntax for expressing complex patterns of predicated events and a modular framework for implementing alternative semantics for event matching. Combining these advanced event correlation facilities with session typed event programming is an interesting future topic, particularly for an extension to multiparty session types [20] which enable both multicasting and correlation over multiple sessions. Whilst these works address many of the difficulties of event programming, none offer a characterisation of communication events as enabled by structured sessions nor the associated static type safety by session types. A session type describes not only the pending event type, but also delineates the interaction flow in which the event has occurred. This enables well-structured programming, for which strong type-based properties for communications, such as communication-safety and progress, can be ensured.

Lauer and Needham presented the first study of the relationship between multithreaded and event-driven systems [26], arguing that (state-based) threads and (message-based) events are dual to each other. Some works approach this duality by combining multithreaded programming interfaces and event-based runtimes to obtain benefits from both categories. A hybrid threads-events system has been embedded into Haskell [27] where both multithreaded and event-driven components are implemented at the application level. A trace over blocking system calls is inferred from the threaded code and the scheduler invokes the user-supplied event handler when event points are reached in the trace. The Scala actors library [17] offers both thread-blocking receive operations and actor-based event handlers, decoupled from threads as closures, that "piggy-back" event handling on the source thread that triggers the event. The Capriccio system [35] uses compiler transformations of user-level thread code, replacing blocking I/O with non-blocking equivalents, coupled with efficient runtime stack management to minimise thread overheads. Although these works offer much improved runtime support for user-level threads, event-driven programming remains a fundamental programming paradigm in terms of design and achieving performance and scalability in highly concurrent communications-based applications such as Web servers [24, 36]. In contrast to the above works, the present paper aims not to circumvent, but rather to facilitate event-driven programming through a structured and type-safe programming methodology developed from a formal basis of session types.

Dynamic types. Dynamic typing with the typecase construct in the λ-calculus is studied in [1, 2] where (1) `typecase` is applied for general expression e; (2) the type can be matched against the type patterns with free type variables; and (3) the default case can be selected if there is no matching (motivated by the use of untyped IO). Our work differs in that we treat the typecase for types for communication flows, that we impose a stronger constraint on the typecase through session set types dispensing with the default case, and that we use non-trivial subtyping on session set types to control the typecase. Below we outline how the type matching in (2) and the default case (3) can be easily incorporated into our framework, using ESP (the full theory is found in [30]).

First we extend the syntax of the typecase to $\texttt{typecase}\ e\ \texttt{of}\ \{(\vec{\mathsf{X}}_i)(x_i : T_i)P_i\}_{i\in I}$ where X_i binds free type variables in T_i. We first introduce the typing system for expressions similar to [1]. For type matching, we introduce a matching function from type variables from closed types and uses the similar typing system from [1]. These are simple additions, which do not change the basic nature of the type discipline.

For (3), the default case, we include a small, but important additional rule, $\texttt{end} \leq S$ for any $S \in \mathscr{S}$, to the construction of the subtyping relation in §4.1: this rule means that under the asynchronous communication semantics, doing nothing (`end`) never leads to a lack of composability (the process sends nothing at that channel, and a message from its peer is just buffered). By encoding the type for "default" $\bot$ as $\{\texttt{end}\}$, we can type the default case, since $\bot$ can be raised to an arbitrary session type by (Subs). For example, we can type $\texttt{typecase}\ k\ \texttt{of}\ \{T_1 : P_1,\ T_2 : P_2,\ \bot : P_3\}$ where the third branch is the default case and the type $\bot$ in the default case indicates that the type of k is unknown, hence P_3 is never allowed to use k except as a value of a message it may send through another channel. Eventful SJ can treat mixed events and objects in the typecase, as seen in Figure 2 in § 2. While the extensions in the theory are straightforward, our practical choice is not to include either (2) or (3). This is because (2) may lead to relatively inefficient type matching algorithm for `typecase` [1], while (3) breaks the progress property (note **0** has an arbitrary session type). We believe that the default case is better handled as a session exception [21] with clarity and flexibility.

Session typed programming languages and formalisms. Sing# [12], a systems-level language with session types for inter-component interaction, features join constructs for handling the arrival of various message patterns, on which we already discussed in §3.3. A recent work [16] has studied a fine-grained integration of session programming and object-orientation: one of the advantages is the ability to store session endpoints as object fields. Their work does not treat either event-driven programming, progress or implementation with session end-point passing (delegations). A few process typing systems that guarantee advanced progress properties have been studied recently in the context of Web services [5–8]. The present paper is the first to include the facility for the type-safe detection of message arrival combined with dynamic inspection of session types at runtime, using them to guarantee an advanced progress property that applies to our extension of Java for communications-based event programming. We formalise and prove a new progress property, *event progress* stated in Theorem 4.6, which in effect includes delegations hence which cannot be proved using the typing systems in [5–8]. In our selectors, the order of session channels do not form a partial order, as they are re-stored in the session queue (i.e. re-registered), depending on message arrival and

the type of the session: this complex causality with session delegations is not typable in [5, 6]. The asynchronous communication semantics and recursive types are the key features of event programming (as found in § 2 and SMTP servers in § 5.3), which are not fully treated in [7, 8]. The key properties for event programming are ensured by static checking (for safety properties) and simple usage rules (for progress) in our integration of sessions and events in Java.

Future work. The theoretical basis and practical framework for event-driven session programming presented in this paper opens up several directions for further research. One is the extension of both the theory and implementation to support event handling for multiparty sessions [20]: combining the advanced event correlation facilities of [11] with multiparty session types would enable type-safe multicasting and correlation of multiple, heterogeneous sessions. Our mechanisms for the type-safe storage and retrieval of sessions may also serve as a basis for other high-level facilities such as session hibernation and process migration [32]. We are currently designing SJ Runtime extensions for session event selector thread pools, where we assign particular event types (i.e. session types) to specific threads. Session types can be exploited in this role for performance gains (e.g. thread locality for event handling routines) and to facilitate system profiling and load balancing. We are also continuing the practical evaluation of ESJ through the implementation of event-driven applications and further benchmarks [30]. These investigations will assist future developments in the design of safe, high-level language support for managing complex, asynchronous control flow in communication-centred programming.

Acknowledgements. We thank the ECOOP referees for their comments. This work is partially supported by EPSRC EP/F003757, EP/F002114, EP/G015635 and EP/G015481.

References

1. M. Abadi, L. Cardelli, B. C. Pierce, and G. D. Plotkin. Dynamic typing in a statically typed language. *TOPLAS*, 13(2):237–268, 1991.
2. M. Abadi, L. Cardelli, B. C. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *J. Funct. Program.*, 5(1):111–130, 1995.
3. A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *USENIX ATC 2002*, pages 289–302. USENIX Association, 2002.
4. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *TOPLAS*, 26(5):769–804, 2004.
5. L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR 2008*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
6. L. Caires and H. T. Vieira. Conversation types. In *ESOP 2009*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.
7. G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. Foundations of session types. In *PPDP 2009*, pages 219–230. ACM, 2009.
8. G. Castagna and L. Padovani. Contracts for mobile processes. In *CONCUR 2009*, volume 5710 of *LNCS*, pages 211–228. Springer, 2009.

9. R. Cunningham and E. Kohler. Making events less slippery with eel. In *HOTOS 2005*, pages 3–3. USENIX Association, 2005.
10. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *ECOOP 2006*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.
11. P. Eugster and K. R. Jayaram. Eventjava: An extension of java for event correlation. In *ECOOP*, volume 5653 of *LNCS*, pages 570–594. Springer, 2009.
12. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *EuroSys 2006*, ACM SIGOPS, pages 177–190. ACM, 2006.
13. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. *JLAP*, 57(1-2):23–69, 2003.
14. D. Gay et al. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *PLDI*, pages 1–11, 2003.
15. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
16. S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL 2010*, volume 45, pages 299–312. ACM, 2010.
17. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
18. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP 1991*, volume 512 of *LNCS*, pages 133–147. Springer-Verlag, 1991.
19. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP 1998*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
20. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL 2008*, pages 273–284. ACM, 2008.
21. R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In *ECOOP 2008*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
22. Java New I/O APIs. `http://java.sun.com/j2se/1.4.2/docs/guide/nio/`.
23. D. Kouzapas. A session type discipline for event driven programming models. Master's thesis, Imperial College London, 2009. http://www.doc.ic.ac.uk/teaching/distinguished-projects/2010/d.kouzapas.pdf.
24. M. Krohn. Building secure high-performance web services with okws. In *USENIX ATC 2004*, pages 185–198. USENIX Association, 2004.
25. M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *USENIX ATC 2007*, pages 1–14. USENIX Association, 2007.
26. H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Operating Systems Review*, 13(2):3–19, 1979.
27. P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *SIGPLAN Not.*, 42(6):189–199, 2007.
28. D. Mostrous and N. Yoshida. Session-based communication optimisation for higher-order mobile processes. In *TLCA 2009*, volume 5608 of *LNCS*, pages 203–218. Springer, 2009.
29. Polyglot homepage. `http://www.cs.cornell.edu/Projects/polyglot/`.
30. SJ homepage. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`.
31. The simple mail transfer protocol. `http://tools.ietf.org/html/rfc5321`.
32. A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MOBICOM 2000*, pages 155–166. ACM, 2000.

33. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE 1994*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
34. R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HOTOS 2003*, pages 4–4. USENIX Association, 2003.
35. R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP 2003*, pages 268–281. ACM, 2003.
36. M. Welsh, D. E. Culler, and E. A. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP 2001*, pages 230–243. ACM, 2001.

A Typing Runtime Processes

This section gives the typing system for runtime processes. Our approach follows that in [28]. The process typing judgement is extended as

$$\Gamma \vdash P \triangleright \Delta \quad \text{with} \quad \Delta ::= \Sigma \mid \Delta \cdot s:[S, \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}'] \mid \Delta \cdot s:(S,[S', \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}'])$$

where the configuration element $[S, \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}']$ records the active type S of the configuration and types of values enqueued in the buffers. $(S,[S', \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}'])$ pairs a type S for the session s and the type information $[S, \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}']$ for the associated configuration at s. We identify $(S,[\ldots])$ with $([\ldots],S)$. Expression typing judgements are as for program typing, but modified to use Δ-environments where needed.

The composition of Δ_1 and Δ_2, denoted by $\Delta_1 \odot \Delta_2$ [28], is defined as:

$$\Delta_1 \odot \Delta_2 = \{s:(\Delta_1(s) \odot \Delta_2(s)) \mid s \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)\} \cup \Delta_1 \setminus \mathrm{dom}(\Delta_2) \cup \Delta_2 \setminus \mathrm{dom}(\Delta_1)$$

where $S \odot [S', \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}'] = [S', \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}'] \odot S = s:(S,[S', \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}'])$ if $S \leq S'$; otherwise undefined. Next, we define *the session type remainder* S' obtained by subtracting a vector of message types $\vec{T}$ from a session type S, denoted by $S - \vec{T} = S'$, by: (1) $S - \varepsilon = S$; (2) $?(T);S - T \cdot \vec{T}$ if $S - \vec{T} = S'$; and (3) $\&\{l_i : S_i\}_{i \in I} - l_i \cdot \vec{T} = S'$ if for all $i \in I$, $S_i - \vec{T} = S'$. Our session type remainder differs from that in [28] because we require the stronger condition that a well-formed configuration with an output prefixed active type cannot have a non-empty local `i`-buffer or be composed with an configuration with a non-empty `o`-buffer. We say a Δ-evnvironment is *well-configured with respect to a session* s, written $\mathrm{wc}(\Delta, s)$, if the following is satisfied:

$$\Delta(s) = [S_1, \mathtt{i}:\vec{T}_1, \mathtt{o}:\vec{T}_2'],\ \Delta(\bar{s}) = [S_2, \mathtt{i}:\vec{T}_2, \mathtt{o}:\vec{T}_1'] \ \text{ implies } \ S_i' = S_i - (\vec{T}_i \cdot \vec{T}_i')\ (i \in \{1,2\}),\ S_1' \leq \overline{S_2'}$$

These conditions say that, at any stage of execution of an established session, the types of the remaining session implementations on each endpoint, modulo any messages buffered (i.e. to be consumed by pending input actions) at the local `i`-buffer and opposing `o`-buffer, should be subtypes of the active types in their respective configurations; and that the dual of the active type at one endpoint should be a supertype of the active type at the other. We say that Δ is *well-configured*, $\mathrm{wc}(\Delta)$, iff $\forall s \in \mathrm{dom}(\Delta), \mathrm{wc}(\Delta, s)$.

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \odot \Delta'}\ (\mathsf{Par'}) \qquad \frac{}{\Gamma, \{s : S\} \vdash s : S}\ (\mathsf{Endpt})$$

$$\frac{\Gamma \vdash a : \langle S \rangle}{\Gamma \vdash a[\langle S \rangle : \vec{s}] \triangleright \cup_{s \in \vec{s}} \{s : (S, [S, \mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon])\}, a}\ (\mathsf{Queue'})$$

$$\frac{\Gamma \vdash a : \langle S \rangle}{\Gamma \vdash \overline{a}\langle s \rangle \triangleright s : (S, [S, \mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon])}\ (\mathsf{AsynchReq}) \qquad \frac{\Gamma \vdash P \triangleright \Delta \quad s, \overline{s} \in \mathsf{dom}(\Delta) \quad \mathrm{wc}(\Delta, s)}{\Gamma \vdash (\nu\, s) P \triangleright \Delta \setminus \{s, \overline{s}\}}\ (\mathsf{Sess})$$

$$\frac{\forall i \leq m. \Gamma, \Sigma_i \vdash h_i : T_i \quad \forall j \leq n. \Gamma, \Sigma'_j \vdash h'_j : T'_j \quad \Sigma = \Sigma_0 \cdot \Sigma_1 \cdots \Sigma_m \cdot \Sigma'_1 \cdots \Sigma'_n \quad \mathrm{ct}(\Sigma_0)}{\Gamma \vdash s[S, \mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}] \triangleright \Sigma \odot s : [S, \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T'}]}\ (\mathsf{Config})$$

Fig. 11. Selected typing rules for runtime processes.

We now introduce Δ-ordering, which represents how session environments are updated as typable processes are reduced. We define:

1.	$s : !(T); S$	$\odot$	$s : [!(T); S, \mathtt{o} : \vec{\tau}]$	$\sqsubset$	$s : S$	$\odot$ $s : [S, \mathtt{o} : \vec{\tau} \cdot T]$
2.	$s : ?(T); S$	$\odot$	$s : [?(T); S, \mathtt{i} : T \cdot \vec{\tau}]$	$\sqsubset$	$s : S$	$\odot$ $s : [S, \mathtt{i} : \vec{\tau}]$
3.	$s : [\mathtt{o} : \tau \cdot \vec{\tau}]$	$\odot$	$\overline{s} : [\mathtt{i} : \vec{\tau}']$	$\sqsubset$	$s : [\mathtt{o} : \vec{\tau}]$	$\odot$ $\overline{s} : [\mathtt{i} : \vec{\tau}' \cdot \tau]$
4.	$s : \oplus\{l_i : S_i\}_{i \in I}$	$\odot$	$s : [\oplus\{l_i : S_i\}_{i \in I}, \mathtt{o} : \vec{\tau}]$	$\sqsubset$	$s : S_i$	$\odot$ $s : [S_i, \mathtt{o} : \vec{\tau} \cdot l_i]$
5.	$s : \&\{l_i : S_i\}_{i \in I}$	$\odot$	$s : [\&\{l_i : S_i\}_{i \in I}, \mathtt{i} : l_i \cdot \vec{\tau}]$	$\sqsubset$	$s : S_i$	$\odot$ $s : [S_i, \mathtt{i} : \vec{\tau}]$
6.	$s : \{S_i\}_{i \in I}$	$\odot$	$s : [S_i]$	$\sqsubset$	$s : S_i$	$\odot$ $s : [S_i]$
7.	$s : \mu \mathsf{X}.S$	$\odot$	$s' : [\mu \mathsf{X}.S, \mathtt{i} : \vec{\tau}_1, \mathtt{o} : \vec{\tau}_2]$	$\sqsubset$	$s : S'$	$\odot$ $s' : [S', \mathtt{i} : \vec{\tau}'_1, \mathtt{o} : \vec{\tau}'_2]$
	if $s : S\{\mu \mathsf{X}.S/\mathsf{X}\}$	$\odot$	$s' : [S\{\mu \mathsf{X}.S/\mathsf{X}\}, \mathtt{i} : \vec{\tau}_1, \mathtt{o} : \vec{\tau}_2]$	$\sqsubset$	$s : S'$	$\odot$ $s' : [S', \mathtt{i} : \vec{\tau}'_1, \mathtt{o} : \vec{\tau}'_2]$

In (7), s' is s or $\overline{s}$. If $\Delta_1 \sqsubset \Delta_2$ and $\Delta \odot \Delta_1$ are defined, then $\Delta \odot \Delta_1 \sqsubset \Delta \odot \Delta_2$. We also make use of several environment properties such as if $\Delta_1 \sqsubset \Delta_2$ and $\Delta \odot \Delta_1$ is defined, then $\Delta \odot \Delta_2$ is defined; and if $\mathrm{wc}(\Delta)$ and $\Delta \sqsubset \Delta'$, then $\mathrm{wc}(\Delta')$.

Figure 11 lists the rules for typing runtime processes. All, except for (Par'), are for typing runtime entities that do not occur in programs. We omit only (Subs') and (Chan'), which are just the original rules modified for Δ-environments. Expressions and terms not covered by these rules are typed using the existing program typing rules.

The composition by (Par') pairs up the types of each session endpoint with the associated configurations to form $(S, [S', \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T'}])$. Rule (Queue') creates a server-side type for each of the buffered session request messages, as well as recording the presence of the a-queue in the Δ-environment like (Queue). Buffered session endpoints (i.e. endpoints being delegated) are typed using (Endpt), which creates a Σ-context holding the type of the endpoint; by typing each buffered message under a separate Σ-context, the concatenation of these contexts ensures the linearity of buffered endpoints. (AsyncReq) types the request at a as a freshly initiated server $(S, [S, \mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon])$; although the precise implementation of the server-side is unknown, it is sufficient to derive a surrogate type from the $\langle S \rangle$ of the shared channel a. This is because program typing (the (Accept) rule) guarantees the eventual implementation to be a subtype of the server-side annotation, and that the annotation directly corresponds to $\langle S \rangle$.

(Sess) checks that the body of the session restriction is well-configured with respect to the session, i.e. that the session implementations on each endpoint and the two ses-

sion configurations together constitute a valid runtime state in the consistent execution of the session. Rule (Config) types all messages enqueued within the `i`- and `o`-buffers of the endpoint configuration to construct the $[S, \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}']$ representation of the configuration. Using the properties of the ordering between environments, $\Delta \sqsubset \Delta'$, we obtain:

Theorem 1.1 (Type Soundness) *If $\Gamma \vdash P \triangleright \Delta$, $wc(\Delta)$ and $P \longrightarrow Q$, then $\Gamma \vdash Q \triangleright \Delta'$ and either $\Delta = \Delta'$ or $\Delta \sqsubset \Delta'$.*

B Switch-receive

Inductive encoding of the switch-receive We define the encoding as follows.

$$((s_1.l_1(x_1) \wedge ... \wedge s_n.l_n(x_n))) \stackrel{\text{def}}{=} \mathtt{arrived}\, s_1\, l_1 \wedge ... \wedge \mathtt{arrived}\, s_n\, l_n$$

$$\langle\langle s_1.l_1(x_1) \wedge ... \wedge s_n.l_n(x_n) : P\rangle\rangle \stackrel{\text{def}}{=} s_1 \triangleright l_1 : s_1?(x_1); ...; s_n \triangleright l_n : s_n?(x_n); P$$

$$[\![\mathtt{switch\text{-}receive}\{J_1 : P_1, ..., J_n : P_n\}]\!] \stackrel{\text{def}}{=}$$
$$\mathtt{def\ loop} = \text{if } ((J_1)) \text{ then } \langle\langle J_1 : [\![P_1]\!]\rangle\rangle \text{ else if ... else if } ((J_n)) \text{ then } \langle\langle J_n : [\![P_n]\!]\rangle\rangle \text{ else } \mathtt{loop}$$

The $\wedge$ operant can be encoded as follows:

$$\mathtt{if}\ e_1 \wedge e_2\ \mathtt{then}\ P\ \mathtt{else}\ Q \stackrel{\text{def}}{=} \mathtt{if}\ e_1\ \mathtt{then}\ \mathtt{if}\ e_2\ \mathtt{then}\ P\ \mathtt{else}\ Q\ \mathtt{else}\ Q$$

The *branch operator* continuation is encoded as:

$$s_1 \triangleright l_1 : s_1?(x_1); s_2 \triangleright l_2 : s_2?(x_2); P \stackrel{\text{def}}{=} s_1 \triangleright \{\, l_1 : s_1?(x_1); s_2 \triangleright \{l_2 : s_2?(x_2); P, l_1 : \mathit{Dummy}\},\ l_2 : \mathit{Dummy}'\}$$

Dummy, *Dummy*$'$ processes denote the correct corresponding type for each label in the branch by convention, and they are not activated.
The rest of the encoding is homomorphic.

Below we let $I = \{1, 2, .., m\}$ with $m \geq 1$.

$$\mathtt{switch\text{-}receive}\{J_i : P_i\}_{i \in I} \longrightarrow \mathtt{sr}\{J_i : P_i\}^1_{i \in I}$$

$$\mathtt{sr}\{J_i : P_i\}^j_{i \in I} \mid s_{j1}\,[\mathtt{i} : l_{j1} \cdot v_1 \cdot \vec{h}_1] \mid \cdots \mid s_{jn_j}\,[\mathtt{i} : l_{jn_j} \cdot v_n \cdot \vec{h}_{n_j}] \longrightarrow P_j\{\vec{v}/\vec{x}\} \mid s_{j1}\,[\mathtt{i} : \vec{h}_1] \mid \cdots \mid s_{jn_j}\,[\mathtt{i} : \vec{h}_{n_j}]$$

$$\mathtt{sr}\{J_i : P_i\}^j_{i \in I} \mid s_{j1}\,[\mathtt{i} : \vec{h}_1] \mid \cdots \mid s_{jn_j}\,[\mathtt{i} : \vec{h}_{n_j}] \longrightarrow \mathtt{sr}\{J_i : P_i\}^{n'}_{i \in I} \mid s_{j1}\,[\mathtt{i} : \vec{h}_1] \mid \cdots \mid s_{jn_j}\,[\mathtt{i} : \vec{h}_{n_j}]$$

where $j \in I, J_j = s_{j1}.l_{j1}(x_{j1}) \wedge ... \wedge s_{jn_j}.l_{jn_j}(x_{n_j})$, and $n' = j + 1 \bmod m$. The third reduction is matched if the condition in the second one is not satisfied.

Typing the switch-receive. The typing rules for the switch-receive from Example 3.2 are also directly suggested by its encoding, which we introduce below.

The fact that the switch-receive construct may leave sessions unimplemented or unprocessed presents difficulties in the correct typing of the switvc-receive process. This suggests the introduction of a special type called *Switch* type. Every unimplemented session has this type in the Join pattern implementation. So for example if:

$$J = s_1.l_1(x_1) \wedge ... \wedge s_n.l_n(x_n) : P, k > n$$

then s_k has type *Switch*.

To complete the the theory we also define that

$$Switch \leq T$$

Meaning that *Switch* is a subtype of every type. This way we can use subtyping to type the switch-receive construct.

We use an auxiliary sequent of the form $\Gamma; J \vdash P \triangleright \Sigma$, which says that under Γ and assuming a join pattern J, a process P has a session typing Σ. When J is empty, we identify this sequent with the original $\Gamma \vdash P \triangleright \Sigma$. The rules follow.

$$\frac{\Gamma, x : U_j; J \vdash P \triangleright \Sigma, s : S_j \quad j \in I}{\Gamma; s.l_j(x : U_j) \wedge J \vdash P \triangleright \Sigma, s : \&[l_i : ?(U_i); S_i]_{i \in I}} (\mathsf{Join})$$

$$\frac{\forall i \in \{1,..,n\}.\ \Gamma, J_i \vdash P_i \triangleright \Sigma \quad \{J_1,..,J_n\} \text{ is sound w.r.t. } \Sigma.}{\Gamma \vdash \texttt{switch-receive}\{J_1 : P_1, ..., J_n : P_n\} \triangleright \Sigma} (\mathsf{Switch\text{-}Receive})$$

The condition "$\{J_1,..,J_n\}$ is sound w.r.t. Σ" says that, at the channels occurring in $\{J_1,..,J_n\}$, these join patterns together should contain all branch labels of these channels so that when all messages have arrived at these channels, at least one of the guards becomes satisfiable.

The two typing rules are an organised version of step-by-step typing of the encoded join patterns. It is straightforward to prove that the clauses (Type Preservation), (Soundness) and (Safety) in Proposition 4.4 hold.

C Additional Appendix for Section 3

We give the definitions that were omitted from § 3.

C.1 Structural Congruence

The notion of bound and free identifiers is extended to cover the subject and objects of `arrived` u, `arrived` k, `arrived` $k\ h$, `typecase` k `of` $\{(x_i : T_i)\, P_i\}_{i \in I}$, $\overline{a}\langle s \rangle$, $a[\vec{s}]$, and $s[S, \mathtt{i} : \vec{h}, \mathtt{o} : \vec{h}']$ in the expected way. We write $\mathrm{fn}(P)$ for the set of names that have a free occurrence in P; $\mathrm{fpv}(P)$ for the set of free process variables in P; and $\mathrm{dpv}(D)$ for the set of process variables declared in an agent definition scope, given by

$$\mathrm{dpv}(X_1(\vec{x}_1) = P_1 \texttt{ and } ... \texttt{ and } X_n(\vec{x}_n) = P_n) = \{X_1, ..., X_n\}$$

Then structural congruence is the smallest congruence on processes generated by the following rules in Figure 12.

C.2 Reduction

Reduction relation. The binary single-step reduction relation, $\longrightarrow$ is the smallest relation on closed terms generated by the rules in Figure 4 together with those in Figure 13.

$$
\begin{array}{rll}
P \equiv Q \quad \text{if } P =_\alpha Q & & (\alpha\text{-renaming}) \\
P \mid \mathbf{0} \equiv P & & (\text{Idempotence}) \\
P \mid Q \equiv Q \mid P & & (\text{Commutativity}) \\
(P \mid P') \mid P'' \equiv P \mid (P' \mid P'') & & (\text{Associativity}) \\
(\nu\, a{:}\langle S\rangle)\, a[\varepsilon] \equiv \mathbf{0} & & (\text{Shared channels}) \\
(\nu\, a{:}\langle S\rangle)\, P \mid Q \equiv (\nu\, a{:}\langle S\rangle)\,(P \mid Q) \quad (a \notin \mathrm{fn}(Q)) & & \\
(\nu\, a{:}\langle S\rangle)\, \mathtt{def}\ D\ \mathtt{in}\ P \equiv \mathtt{def}\ D\ \mathtt{in}\ (\nu\, a{:}\langle S\rangle)\, P \quad (a \notin \mathrm{fn}(D)) & & \\
(\nu\, s)\, \mathbf{0} \equiv \mathbf{0} & & (\text{Session channels}) \\
(\nu\, s)\,(s : [\varepsilon] \mid \overline{s} : [\varepsilon]) \equiv \mathbf{0} & & (\text{Session queues}) \\
(\nu\, s)\, P \mid Q \equiv (\nu\, s)\,(P \mid Q) \quad (s \notin \mathrm{fn}(Q)) & & \\
(\nu\, s)\, \mathtt{def}\ D\ \mathtt{in}\ P \equiv \mathtt{def}\ D\ \mathtt{in}\ (\nu\, s)\, P \quad (s \notin \mathrm{fn}(D)) & & \\
\mathtt{def}\ D\ \mathtt{in}\ \mathbf{0} \equiv \mathbf{0} & & (\text{Def scopes}) \\
(\mathtt{def}\ D\ \mathtt{in}\ P) \mid Q \equiv \mathtt{def}\ D\ \mathtt{in}\ P \mid Q \quad (\mathrm{dpv}(D) \cap \mathrm{fpv}(Q) = \emptyset) & & \\
\mathtt{def}\ D\ \mathtt{in}\ (\mathtt{def}\ D'\ \mathtt{in}\ Q) \mid Q \equiv \mathtt{def}\ D\ \mathtt{and}\ D'\ \mathtt{in}\ P \quad (\mathrm{dpv}(D) \cap \mathrm{dpv}(D') = \emptyset) & &
\end{array}
$$

Fig. 12. Structural congruence.

D Appendix: Proofs

D.1 Proof of Proposition 4.1

(1) is mechanical; (2) the decidability of subtyping is proved in [15] except the set type. Since I is finite and all elements are closed, checking two set types are in the subtyping or not is decidable, by constructing the decidable algorithm along the line of [15].

D.2 Proofs of Basic Lemmas

The following lemmas and Subject Reduction Theorem are fully proved in **[23, § 6]** (for the extended typecase) for all of the key cases. We list only the important cases, referring the corresponding subsections in [23, § 6].

Lemma D.1 (Weakening Lemma). *Let $\Gamma \vdash P \triangleright \Sigma$.*

(i) *If $X \notin dom(\Gamma)$, then $\Gamma \cdot X : \vec{T} \vdash P \triangleright \Sigma$.*
(ii) *If $u \notin dom(\Gamma)$, then $\Gamma \cdot u : U \vdash P \triangleright \Sigma$.*
(iii) *If $k \notin dom(\Sigma)$ then $\Gamma \vdash P \triangleright \Sigma \cdot k : \mathtt{end}$.*

Similarly for the runtime system by replacing Σ to Δ.

Proof. See [23, § 6.2.3, § 6.2.4]. □

Lemma D.2 (Strengthening Lemma).

(i) *If $X \notin fpv(P)$, then $\Gamma \cdot X : \vec{T} \vdash P \triangleright \Sigma$ implies $\Gamma \vdash P \triangleright \Sigma$.*
(ii) *If $u \notin fn(P) \cup fv(P)$, then $\Gamma \cdot u : U \vdash P \triangleright \Sigma$ implies $\Gamma \vdash P \triangleright \Sigma$.*
(iii) *If $k \notin fn(P) \cup fv(P)$ then $\Gamma \vdash P \triangleright \Sigma \cdot k : \mathtt{end}$ implies $\Gamma \vdash P \triangleright \Sigma$.*

$$
\begin{array}{rcrcl}
e \longrightarrow e' & \Longrightarrow & E[e] & \longrightarrow & E[e'] \quad \text{(Eval)} \\
P \longrightarrow P' & \Longrightarrow & (\nu\, a{:}\langle S\rangle)\, P & \longrightarrow & (\nu\, a{:}\langle S\rangle)\, P' \quad \text{(Chan)} \\
P \longrightarrow P' & \Longrightarrow & (\nu\, s)\, P & \longrightarrow & (\nu\, s)\, P' \quad \text{(Sess)} \\
 & & \texttt{if tt then } P \texttt{ else } Q & \longrightarrow & P \quad \text{(If-true)} \\
 & & \texttt{if ff then } P \texttt{ else } Q & \longrightarrow & Q \quad \text{(If-false)} \\
P \longrightarrow P' & \Longrightarrow & P \mid Q & \longrightarrow & P' \mid Q \quad \text{(Par)} \\
P \longrightarrow P' & \Longrightarrow & \texttt{def } D \texttt{ in } P & \longrightarrow & \texttt{def } D \texttt{ in } P' \quad \text{(Def)} \\
P \equiv P' \longrightarrow Q' \equiv Q & \Longrightarrow & P & \longrightarrow & Q \quad \text{(Struct)}
\end{array}
$$

$$
\frac{P \mid s[S\{\mu\, \mathsf{X}.S/\mathsf{X}\}, \mathtt{i}{:}\vec{h}_1, \mathtt{o}{:}\vec{h}'_1] \longrightarrow P' \mid s[S', \mathtt{i}{:}\vec{h}_2, \mathtt{o}{:}\vec{h}'_2]}{P \mid s[\mu\, \mathsf{X}.S, \mathtt{i}{:}\vec{h}_1, \mathtt{o}{:}\vec{h}'_1] \longrightarrow P' \mid s[S', \mathtt{i}{:}\vec{h}_2, \mathtt{o}{:}\vec{h}'_2]} \text{ (Unfold)}
$$

$$
\frac{X(\vec{x}) = P \in D}{\texttt{def } D \texttt{ in } (X\langle\vec{v}\rangle \mid Q) \longrightarrow \texttt{def } D \texttt{ in } P\{\vec{v}/\vec{x}\} \mid Q} \text{ (Instance)}
$$

Fig. 13. The reduction rules omitted from § 3.2.

Similarly for the runtime system by replacing Σ to Δ.

Proof. See [23, § 6.2.1, § 6.2.2]. □

Lemma D.3 (Substitution Lemma).

(i) *If $\Gamma \cdot x{:}U, \Sigma \vdash e{:}U'$ and $\Gamma \vdash v \triangleright U$, then $\Gamma, \Sigma \vdash e\{v/x\}{:}U'$.*
(ii) *If $\Gamma, \Sigma \cdot x{:}S \vdash e{:}U$ and s fresh, then $\Gamma, \Sigma \cdot s{:}S \vdash e\{s/x\}{:}U$.*
(iii) *If $\Gamma \cdot x{:}U \vdash P \triangleright \Sigma$ and $\Gamma \vdash v \triangleright U$, then $\Gamma \vdash P\{v/x\} \triangleright \Sigma$.*
(iv) *If $\Gamma \vdash P \triangleright \Sigma \cdot x{:}S$, then $\Gamma \vdash P\{s/x\} \triangleright \Sigma \cdot s{:}S$.*

Similarly for the runtime system by replacing Σ to Δ. □

Proof. The full proof is given in [23, § 6.2.5, § 6.2.6]. We select the two important cases.

Case arrived The most interesting case is $e = \texttt{arrived}\ x\ v$ or $e = \texttt{arrived}\ x\ l$ for (ii). We prove the former. Suppose $\Gamma, \Sigma \cdot x{:}S \vdash \texttt{arrived}\ x\ v{:}\mathsf{bool}$. Then we can let $S = x!(U); S'$ for some U and S' such that $\Gamma \vdash v : U$ by (Amsg). By the same rule, we can derive $\Gamma, \Sigma \cdot s{:}S \vdash \texttt{arrived}\ s\ v{:}\mathsf{bool}$, as required.

Case typecase There are two cases: (1) $x \neq k$ or (2) $x = k$. The case (1) is easy by the inductive hypothesis. Thus we prove the case (2) $x = k$ for the clause (iv) above.

Suppose $\Gamma \vdash \texttt{typecase}\ x\ \texttt{of}\ \{(x_i{:}T_i)\, P_i\}_{i\in I} \triangleright \Sigma \cdot x{:}T$. This is derived by $\Gamma \vdash P_i \triangleright \Sigma \cdot x_i{:}T_i$ with $\cup_{i\in I} T_i \leq T$. Note that $x \notin \mathsf{dom}(\Sigma)$ so that we can derive $\Gamma \vdash \texttt{typecase}\ s\ \texttt{of}\ \{(x_i{:}T_i)\, P_i\}_{i\in I} \triangleright \Sigma \cdot s{:}T$ from the exactly same premises by applying (Typecase). □

Lemma D.4 (Subject Congruence). *If $\Gamma \vdash P \triangleright \Sigma$ and $P \equiv Q$, then $\Gamma \vdash Q \triangleright \Sigma$.*

Proof. See [23, § 6.2.7]. □

Lemma D.5 (Environment Properties).

(i) $\Delta_1 \odot \Delta_2 = \Delta_2 \odot \Delta_1$
(ii) $(\Delta_1 \odot \Delta_2) \odot \Delta_3 = \Delta_1 \odot (\Delta_2 \odot \Delta_3)$
(iii) $\Sigma_1 \cdot \Sigma_2$ *is defined then* $\Sigma_1 \odot \Sigma_2$ *is defined and* $\Sigma_1 \cdot \Sigma_2 = \Sigma_1 \odot \Sigma_2$.
(iv) $\Delta \odot a$ *is well configured then* Δ *is well configured.*
(v) $\Delta \odot a \sqsubset \Delta' \odot a$ *then* $\Delta \sqsubset \Delta'$.
(vi) *If* $wc(\Delta)$ *and* $\Delta \sqsubset \Delta'$, *then* $wc(\Delta')$.
(vii) *If* $\Delta_1 \odot \Delta_2$ *defined and* $\Delta_2 \sqsubset \Delta_3$, *then* $\Delta_1 \odot \Delta_3$ *defined.*

Proof. See [23, § 6.2.8]. □

Lemma D.6 (Shared Environment Lemma). *If* $\Gamma \vdash Q \mid a[\langle S \rangle : \vec{s}] \triangleright \Delta \odot a$, *then* $\Gamma \vdash Q \triangleright \Delta'$ *with* $a \notin dom(\Delta')$.

Proof. See [23, § 6.2.9]. □

D.3 Proof of Theorems 4.2 and 4.3

Theorem 4.2 *If* $\Gamma \vdash P \triangleright \Delta$, $wc(\Delta)$ *and* $P \longrightarrow Q$, *then* $\Gamma \vdash Q \triangleright \Delta'$ *and either* $\Delta = \Delta'$ *or* $\Delta \sqsubset \Delta'$.

Proof. Subsection 6.3 in [23] lists the full proofs. We only list the `typecase`. Assume the reduction

$$\texttt{typecase}\ s\ \texttt{of}\ \{(x_i : T_i)\, P_i\}_{i \in I} \mid s\,[S] \longrightarrow P_i\{s/x_i\} \mid s\,[S] \tag{3}$$

with with $i \in I$, $\forall j < i . T_j \not\leq S$ and $T_i \leq S$. Suppose also

$$\Gamma \vdash \texttt{typecase}\ s\ \texttt{of}\ \{(x_i : T_i)\, P_i\}_{i \in I} \mid s\,[S] \triangleright \Sigma \cdot s : [S] \tag{4}$$

The above (4) is derived by (Par') in Figure 11 from

$$\Gamma \vdash \texttt{typecase}\ s\ \texttt{of}\ \{(x_i : T_i)\, P_i\}_{i \in I} \triangleright \Sigma_1 \cdot s : T' \tag{5}$$

and

$$\Gamma \vdash s\,[S] \triangleright \Sigma_2 \cdot s : [S] \tag{6}$$

with $T' \odot [S]$ defined, i.e.

$$T' \leq S \tag{7}$$

The judgement (5) is derived by:

$$\Gamma \vdash P_i \triangleright \Sigma_1' \cdot x_i : T_i \tag{8}$$

with

$$\cup_{i \in I} T_i \leq T \leq T' \text{ and } \Sigma_1' \leq \Sigma_1 \tag{9}$$

by (Subs) and (Typecase). From (8) by Substitution Lemma (vi), we have

$$\Gamma \vdash P_i\{s/x_i\} \triangleright \Sigma_1' \cdot s : T_i \tag{10}$$

Note that by (7) and (9), we have $T_i \leq S$, hence $T_i \odot [S]$ is again defined. Hence applying (Subs) and (Par') to (6) and (10), we obtain the following required result:

$$\Gamma \vdash P_i\{s/x_i\} \mid s\,[S] \triangleright \Sigma \cdot s : [S]$$

□

Theorem 4.3 A more detailed error freedom (which subsumes this theorem) is stated and proved in [23, § 6.4.1]. The proof is mechanical from Theorem 4.2 using the contradiction. □

D.4 Proof of Proposition 4.4

Proposition 4.4

1. (Type Preservation) $\Gamma \vdash P \triangleright \Sigma$ in ESP^+ if and only if $\Gamma \vdash [\![P]\!] \triangleright [\![\Sigma]\!]$.
2. (Soundness) $P \equiv P'$ implies $[\![P]\!] \equiv [\![P']\!]$; and $P \longrightarrow P'$ implies $[\![P]\!] \longrightarrow^* [\![P']\!]$.
3. (Safety) A typable process in ESP^+ never reduces to the error.

(2, Soundness) is mechanical. (3, Safety) is direct from (1) and (2) with Theorem 4.3. Hence we only prove (1, Type Preservation). We show the type preservation for `new selector`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash [\![P]\!] \triangleright [\![\Sigma]\!] \cdot r : S_r \cdot \overline{r} : \overline{S_r} \quad \text{by (IH)} \quad \Gamma, b : \langle \overline{S_r} \rangle \vdash b : \langle \overline{S_r} \rangle}{\Gamma, b : \langle \overline{S_r} \rangle \vdash b(\overline{r}).[\![P]\!] \triangleright [\![\Sigma]\!] \cdot r : S_r}}{\Gamma, b : \langle \overline{S_r} \rangle \vdash \overline{b}(r); b(\overline{r}).[\![P]\!] \triangleright [\![\Sigma]\!]}}{\Gamma, b : \langle \overline{S_r} \rangle \vdash \overline{b}(r); b(\overline{r}).[\![P]\!] \mid b : [\varepsilon] \triangleright [\![\Sigma]\!] \cdot b}}{\Gamma \vdash (\nu\, b)(\overline{b}(r); b(r').[\![P]\!] \mid b : [\varepsilon]) \triangleright [\![\Sigma]\!]}$$

The last line is written as $\Gamma \vdash [\![\texttt{new selector } r \texttt{ in } P]\!] \triangleright [\![\Sigma]\!]$. Similarly for the register.

We prove `selector`. First we infer the else branch in the body.

$$\dfrac{\dfrac{[\![\Sigma]\!] \ \texttt{end only} \quad \Gamma' = \Gamma, \mathsf{Select} : S_r \overline{S_r}}{\Gamma' \vdash \mathsf{Select}\langle r\overline{r} \rangle \triangleright [\![\Sigma]\!] \cdot r : S_r \cdot \overline{r} : \overline{S_r} \quad S_x = \{S_i\}_{i \in I}}}{\Gamma' \vdash r!\langle x \rangle; \mathsf{Select}\langle r\overline{r} \rangle \triangleright [\![\Sigma]\!] \cdot r : !(S_x); S_r \cdot x : S_x \cdot \overline{r} : \overline{S_r}}$$

Secondly we infer the if branch of the body.

$$\dfrac{\dfrac{[\![\Sigma]\!] \ \texttt{end only} \quad \Gamma' = \Gamma, \mathsf{Select} : S_r \overline{S_r}}{\forall\, i \in I, \Gamma' \vdash [\![P_i]\!] \triangleright [\![\Sigma]\!] \cdot x_i : S_i \cdot r : S_r \cdot \overline{r} : S_{\overline{r}} \quad \text{by (IH)}}}{\Gamma' \vdash \texttt{typecase } x \texttt{ of } \{(x_i : S_i) : [\![P_i]\!]\}_{i \in I} \triangleright [\![\Sigma]\!] \cdot x : \{S_i\}_{i \in I} \cdot r : S_r \cdot \overline{r} : \overline{S_r}}$$

Then combining the both branching, we have:

$$\dfrac{\Gamma' \vdash \texttt{if arrived } x \texttt{ then } \ldots \texttt{ else } \ldots \triangleright [\![\Sigma]\!] \cdot x : \{S_i\}_{i \in I} \cdot r : \mu \mathsf{X}.!(\{S_i\}_{i \in I}); \mathsf{X} \cdot \overline{r} : \overline{S_r}}{\Gamma' \vdash \overline{r}?(x); \texttt{if arrived } x \texttt{ then } \ldots \triangleright [\![\Sigma]\!] \cdot x : \{S_i\}_{i \in I} \cdot r : \mu \mathsf{X}.!(\{S_i\}_{i \in I}); \mathsf{X} \cdot \overline{r} : ?(\{S_i\}_{i \in I}); \overline{S_r}}$$

From this, by applying (Def) and $S_i \leq T$, we can derive the required judgement, $\Gamma \vdash [\![\texttt{select}(r)\{(x_i : S_i) : P_i\}_{i \in I}]\!] \triangleright [\![\Sigma]\!] \cdot [\![r : \overline{\texttt{sel}}(T)]\!]$.

Register construct. We first type the main process:

$$\dfrac{\Gamma \vdash [\![P]\!] \triangleright [\![\Sigma]\!] \cdot [\![r : \overline{\texttt{sel}}(T)]\!] \quad \text{by (IH)}}{\Gamma \vdash \overline{r}!\langle s \rangle; [\![P]\!] \triangleright [\![\Sigma]\!] \cdot [\![r : \overline{\texttt{sel}}(T)]\!] \cdot s : S}$$

At this point notice that $[\![r : \overline{\texttt{sel}}(T)]\!] = r : S_r \cdot \overline{r} : S_{\overline{r}}$ where $S_r = \mu \mathsf{X}.!(T);X$ and $!(T);\mu \mathsf{X}.!(T);X = \mu \mathsf{X}.!(T);X$.

We now show the straightforward type preservation for $\texttt{switch} - \texttt{receive}$.

$$\frac{\forall i \in N, \Gamma \vdash [\![\langle\langle J_1 : [\![P_i]\!]\rangle\rangle]\!] \triangleright [\![\Sigma]\!] \cdot \quad s_1 : \&\{l_1 : ?(U_{11}); S_{11} \ldots l_m : ?(U_{1m}); S_{1m}\} \cdot \ldots \cdot s_n : \&\{l_1 : ?(U_{n1}); S_{n1} \ldots l_m : ?(U_{nm}); S_{nm}\} \quad \text{by (IH)}}{\Gamma \vdash [\![\texttt{switch-receive}\{J_1 : P_1, ..., J_n : P_n\}]\!] \triangleright [\![\Sigma]\!] \cdot s_1 : S_1 \cdot \ldots \cdot s_n : S_n}$$

Note that in the first typing judgement we used the assumption that $Switch \leq T$ to type all P_i processes. Unimplemented sessions in P_i have the *Switch* type by convention and they can be subsumed by the correct type in order to have correct overall typing.

D.5 Proof of Theorem 4.6 (event progress)

NB: We use the "well-linkedness" in the same form as in [20] (as seen in this updated version, page 15). This property can be relaxed to what is given in the original submission, with a minor change in the formulation of the resulting property. Definition 4.5 is also refined (the condition for a select action to be preceded by a register action was missing) in this new updated version.

Theorem 4.6 (Progress) An ESP^+-process P is *eventful* if $\Gamma \vdash P \triangleright \emptyset$ and it is well-linked, is simple, and uses selectors well in the sense of Definition 4.5. Then

(1) If P is eventful and $P \longrightarrow Q$ then Q is eventful; and
(2) If P is eventful then either $P \equiv \mathbf{0}$ or $P \searrow Q$ for some Q.

Proof. For (1), we observe that the well-linkedness is preserved by reduction by definition, while simplicity is by the subject reduction. Hence the only issue is selector usage. Definition 4.5 says the following: in the process, if $\texttt{reg}\ k\ \texttt{to}\ r\ \texttt{in}\ R$ occurs, then this action will be followed by zero or more register actions which should further be followed by the selector at r, in the form say $\texttt{select}(r)\{(x_i : S_i) : P_i\}_{i \in I}$, up to the unfolding of recursion, satisfying each S_i should start from input or branching and so should P_i and P_i has no other input and branching actions. By definition this property holds under unfolding of recursions which is the only change in syntactic shape, hence as required.

For (2), if P is eventful in this sense, the same reasoning as in [20] gives the required result as far as selector is not involved. Suppose P has an active selector (a selector which is in an evaluation context). There are two cases. Note that, by the condition on a register action to precede a select action, the selector queue is never empty. Below we say a session channel is *active* if its local input queue is not empty.

(a) If the selector queue has an active session channel, then we have zero or more polling reductions $\longrightarrow_s$ (the third reduction rule in Example 3.1, page 9), followed by the selector's "get" reduction (the second reduction rule in Example 3.1), in which case the message will be processed, all consecutive output/selection actions at this channel (by simplicity no actions at other channels are possible) will be done

by the typing, which do not interleave with input/branching by (exhaustiveness) in Definition 4.5 (hence no deadlock by crisscrossed actions is possible). By typing of a selector, a session channel should be registered at r in the end (if actions in P_i is to terminate), followed by a selector at r by the selector usage in Definition 4.5, hence as required.

(b) If the selector queue does not have any active channel, then it has only polling reductions $\longrightarrow_s$ (the third reduction rule in Example 3.1), without changing the selector queue nor the shape of the process. In this second case, we show one of the channels will eventually become active. Suppose no messages arrive at the registered channels. In this case, all channels are in input/branching modes. Hence corresponding processes have output/selection modes. By simplicity and well-linkedness, an output action is never suppressed except by the preceding input/branching action at the same channel, which is impossible as we have seen in **(a)** above. Hence a message will eventually arrive at one of the channels, as required.

Note the arguments assume, in **(a)** above, that we *always* register a new session channel, which, if the actions at the current session channel are terminated (i.e. the corresponding session type is `end`), then it should initiate a new session, possibly by acceptance. This comes from the fact that the current typecase does not allow registration/selection of shared channels, and does lose generality, as discussed in the remark below. □

Remark D.7. Theorem 4.6 assumes an accept (or request) action at each P_i when the "current" session needs to finish in P_i. By well-linkedness, this does not pause a problem in the present formulation. If we are to use the refined typecase (as in the current eventful SJ), however, we can simply register, from the first, a shared channel, so that the event loop is ready to always receive a new session request. In this case, (1) our encoding of a selector changes accordingly, and (2) the selector typing itself does not demand a register action to be done in each branch. Under the present restriction, the similar behaviour is realised by the current typing and encoding of a select action.

E The Full Typecase: Value Types and Type Matching

This section defines the full `typecase` discussed in **Dynamic types** in Related Work of Section 6.

Syntax and types We first extend the syntax to include $\texttt{typecase}\ e\ \texttt{of}\ \{(\vec{\mathsf{X}}_i)(x_i : T_i)\ P_i\}_{i\in I}$ takes an expression e and a list of cases, where $(\vec{\mathsf{X}}_i)$ binds the type variables in T_i and $(x_i : T_i)$ binds the free variable x_i of type pattern T_i in P_i. The runtime type of the constant or the session endpoint is matched against the type patterns $(\vec{\mathsf{X}}_i)\,T_i$. where all free type variables in T_i are bound by $\vec{\mathsf{X}}_i$. The parentheses are omitted when $\vec{\mathsf{X}}_i$ is empty.

$$\frac{}{\mathsf{match}(T,T)=\emptyset} \quad \frac{}{\mathsf{match}(T,\mathsf{X})=\{T/\mathsf{X}\}} \quad \frac{\mathsf{match}(S,S')=\sigma}{\mathsf{match}(\langle S\rangle,\langle S'\rangle)=\sigma} \quad \frac{\mathsf{match}(T,T')=\sigma}{\mathsf{match}(\mu\,\mathsf{X}.T,\mu\,\mathsf{X}.T')=\sigma}$$

$$\frac{\mathsf{match}(T,T')=\sigma \quad \mathsf{match}(S,S'\sigma)=\sigma'}{\mathsf{match}(!(T);S,!(T');S')=\sigma\cdot\sigma'} \qquad \frac{\forall i.1\le i\le n,\mathsf{match}(S_i,S_i')=\sigma_i}{\mathsf{match}(\oplus\{l_i:S_i\}_{i\in I},\oplus\{l_i:S_i'\}_n)=\sigma_1\cdots\sigma_n}$$

Fig. 14. The match function (the input and branching cases are omitted).

For types, we first extend T with $\{U_i\}_{i\in I}$. For the subtyping, we include the atomic subtyping. The ordering of the value types for session actions follow [28].

$$\begin{aligned}F(\mathscr{R}) = {} & \ldots \\ & \cup\ \{(\mathsf{nat},\mathsf{real})\} \\ & \ldots \\ & \cup\ \{(!(U);S,!(U');S') \mid (U,U'),(S,S')\in\mathscr{R}\} \\ & \cup\ \{(?(U);S,?(U');S') \mid (U',U),(S,S')\in\mathscr{R}\} \\ & \cup\ \{(\{U_i\}_{i\in I},\{U_j'\}_{j\in J}) \mid \neg(|I|=|J|=1),\forall j\in J,\exists i\in I.(U_i,U_j')\in\mathscr{R}\}\end{aligned}$$

Reduction The revised reduction rule for session with match function is defined as follows:

$$\frac{\forall j<i.\ \nexists T'.(T'\le S\wedge\mathsf{match}(T',T_j))\ \exists T'.(T'\le S\wedge\mathsf{match}(T',T_i))}{\mathtt{typecase}\ s\ \mathtt{of}\ \{(\vec{\mathsf{X}}_i)(x_i:T_i)\ P_i\}_{i\in I}\mid s\,[S]\longrightarrow P_i\{s/x_i\}\mid s\,[S]}\ (i\in I)\quad [\mathsf{Typecase\text{-}s}]$$

In [Typecase-s], a process continues the session s along the first P_i for which S_i can be successfully matched against a subtype of the active type. The match function, defined in Figure 14, takes T and a type pattern T', and returns a substitution σ where $T=T'\sigma$, if one exists; otherwise match fails. The concatenation of substitutions $\sigma\cdot\sigma'$ is undefined if $\mathsf{X}\in\mathrm{dom}(\sigma)\cap\mathrm{dom}(\sigma),\sigma(\mathsf{X})\neq\sigma'(\mathsf{X})$.

For value types, [Typecase-v] is defined following [1]. The process reduces if the type of v is successfully matched with the first T_i up to subtyping.

$$\frac{\forall j<i.\ \nexists T'.(U\le T'\wedge\mathsf{match}(T',T_j))\ \exists T'.(U\le T'\wedge\mathsf{match}(T',T_i))}{\mathtt{typecase}\ v^U\ \mathtt{of}\ \{(\vec{\mathsf{X}}_i)(x_i:T_i)\ P_i\}_{i\in I}\longrightarrow P_i\{v/x_i\}}\ (i\in I)\quad [\mathsf{Typecase\text{-}v}]$$

Typing Rules Finally the typing rules are extended as follows: First we extend Γ to include mapping from identifier to a set type for U. Then we define:

$$\frac{\forall i.i\in I,\forall T_i'\ \mathsf{match}(T_i',T_i).\ \Gamma\vdash P_i\triangleright\Sigma\cdot x_i:T_i',\quad \cup_{i\in I}T_i'\le T}{\Gamma\vdash\mathtt{typecase}\ k\ \mathtt{of}\ \{(\vec{\mathsf{X}}_i)(x_i:T_i)\ P_i\}_{i\in I}\triangleright\Sigma\cdot k:T}(\mathsf{Typecase\text{-}s})$$

The rule for the extended typecase for session checks that every possible substitution for T_i, the body P_i of each case must be typed under the environment Σ with x_i assigned by T_i'. Then the whole process is typable under T such that T is a supertype of union of T_i'. Note that the first premise is quantified over all matching substitutions, which

means that a proof of this premise requires an infinite number of separate derivations, see [1]. The corresponding typing rule for the value type are defined as follows:

$$\frac{\forall i.i \in I, \forall T_i' \ \mathsf{match}(T_i', T_i).\ \Gamma, x_i : T_i' \vdash P_i \triangleright \Sigma \quad U \leq \cup_{i \in I} T_i' \quad \Gamma \vdash e : U}{\Gamma \vdash \texttt{typecase}\ e\ \texttt{of}\ \{(\vec{\mathsf{X}}_i)(x_i : T_i)\ P_i\}_{i \in I} \triangleright \Sigma} (\mathsf{Typecase\text{-}v})$$