Type-Safe Eventful Sessions in Java

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

*Imperial College London †Queen Mary, University of London

Abstract. Event-driven programming is a widely accepted approach to building robust, scalable servers. However, it forces application programmers to manually maintain client-specific state, leading to unsafe programs with obfuscated control flows. This paper introduces a Java language extension and its type system for structured event-driven programming, based on *session types*. Built on a simple extension of the standard session types, the proposed language enables safe, session type-directed event handling, supporting both traditional and novel event programming idioms. In addition to type soundness and communication safety, a theoretical framework of the language ensures event-handling safety and event progress, which have not been guaranteed by previous session type systems. We demonstrate applicability of the language through a conversion of existing event-based distributed applications.

1 Introduction

Asynchronous event programming is a well-known method for concurrent programming. It is characterised by the use of reactive flow of control driven by the detection of computation events, typically message arrivals through communication channels. Event-driven programming contributes to performance and scalability in highly concurrent applications. At the same time, the extant framework for this programming style forces programmers to work with low-level, untyped interface and obfuscated control flow [2, 32]. This leads to programs difficult to write, read and verify and unsafe to execute. Consequently, several recent works [23, 25, 33] propose a combination of thread-based programming interfaces with event-driven runtime, but they do not offer a language stratum to directly harness this programming style as a part of a safe and structured programming discipline.

This paper presents an integration of *session types* [18, 31] and *asynchronous event handling* in Java. Our aim is to make the most of the flexibility and resource management of the latter, while maintaining the merits of principled communications programming in the former. Session types offer one of the well-studied type-based methods for structuring a series of concurrent, distributed interactions [9, 15, 20]. In the preceding incorporation of session types into object-oriented programming, a thread of control will carry out communications through (possibly interleaving) sessions, following the associated session type structures. The description of communication events is tightly coupled with the flow of interactions specified by session types: in particular, a thread blocks when it is expecting an input. In contrast, the standard event-based idioms such as event loops demand the detection of events from a dynamic collection of channels (through e.g. *selector* in Unix and Java NIO [26]), which has not been possible in the previous works on theory and practice of the session typed object-oriented languages [9, 11, 15, 20].

A dynamic channel collection used in event handling idioms such as event-loops often need to treat channels of *diverse* types, leading to the need for untyped operations on such collections. This is necessary because, even limited to a single conversation, two parties usually exchange different message types at different points of their conversation.

In the context of session typed programming, this means we need a framework where we can *store* channels with different session types into a collection, *asynchronously check* messages have arrived at these channels, and finally *retrieve* and *use* them, all under the full control of a type discipline so that channels are used following their session types.

We explore a theoretical basis of such an integration through a small process calculus based on [17, 18, 31], augmenting session primitives with two constructs, *message arrival predicate* and *runtime session type inspection*. We show that its type theory ensures not only the standard safety but a novel progress property which we call *event progress*. Building on the theory and extending the runtime architecture of SJ (Session-based Java [20]), we next present the design and implementation of the associated language facilities, with consideration for efficiency and abstraction. The resulting language, *eventful SJ*, ensures key properties such as type soundness, communication safety and event progress, in the presence of asynchronous event handling. We experiment our extension through the development of a SMTP server [29] interoperable with existing clients.

Contributions. We present motivating programming examples in § **2**. The rest of the paper presents the following technical contributions.

- **(§ 3)** A formalism which accurately captures the semantics of asynchronous sessions integrated with event handling through two constructs, *message arrival predicate* and *session typecase*. New session types, *session set types*, enable the treatment of a dynamic collection of differently typed channels.
- **(§ 4)** A type theory based on set session types for the extended session constructs, establishing key properties including type soundness, communication-safety and progress in the presence of dynamically registered sessions. The proofs demand a delicate formulation of the runtime typing system that tracks dynamic session type information. The theory captures a wide range of event-driven programming patterns including *event loops* and *join patterns* [3, 11] through encoding, from which we can derive sound typing rules for these constructs.
- **(§ 5)** A design and implementation of practical language constructs for integration of events and sessions, including a type checker and a modular runtime for event-handling, both built on the basis of SJ [20], guaranteeing type-safety and event progress. An open-source application, a SMTP server [29], is converted into the extended SJ, guaranteeing type, communication and event-handling safety through session types.

The proposed extension of SJ [20] is non-trivial not only because of new primitives and their type disciplines using session set types, but also because of the need to handle session type information at runtime, demanding more modular runtime architecture than in the previous version [20]. This modular runtime in turn is exploited to expand the range of transports and data formats sessions can handle, resulting in broader applicability of session typed programming, as we show through the communication-safe scalable SMTP server. The paper discusses related work (§ **6**) and concludes with future work (§ **7**). The detailed definitions, proofs, implementation details, source code and the omitted benchmark results are found in [28].

2 Session-based Event Programming

Session Programming in SJ As a basis of the integration of session types and event-driven programming, we use SJ [20], an extension of Java for type-safe concurrent and distributed session programming. Session programming in SJ, as detailed in [20, § 2],

```
1  protocol pSelector { ?(Event1).?(Event2).!<Result>, ?(Event2).!<Result> }
2  using(SJSelector{pSelector} sel // Create a selector of type pSelector.
3        = SJSelector.create(params)) {
4    ...
5    sel.register(source); // Register source session(s) with the selector.
6    ...
7    while(run) { // Main event loop.
8      using(SJSocket{pSelector} s = sel.select()) { // Select an event.
9        typecase(s) { // Identify and handle the event type.
10         when(SJSocket{?(Event1).?(Event2).!<Result>} s1) {
11           Event1 e1 = s1.receive(); // Receive the Event1 event and..
12           sel.register(s1); //..re-register the session with the selector.
13         }
14         when(SJSocket{?(Event2).!<Result>} s2) {
15           Event2 e2 = s2.receive(); // Receive the Event2 event and..
16           s2.send(new Result(...)); // ..send the Result, and we're done.
17         }
18       }
19 } } }
```

Fig. 1. A Basic Event Loop in SJ.

starts with the declaration of the intended communication protocols as session types. The communication actions during a session, such as message passing, branching and dynamic channel passing, are implemented as operations on typed session channel endpoints initiated with session types, whose conformance to the declared protocol is statically checked by the SJ compiler. This assures the correct communication behaviour. The eventful SJ generalises this framework with the capability to treat dynamic event occurrences.

A Simple Event Loop We illustrate the key elements of the event-based session programming in SJ using a simplified *event loop* [24], listed in Figure 1. The program uses a session channel endpoint (called `session socket` in SJ) with the following session type:

```
?(Event1).?(Event2).!<Result>                                    (1)
```

which says: *a program should first expect to receive (`?`) a message, a Java object of type `Event1`, then to receive (again `?`) an `Event2`, and finish by sending (`!`) a `Result`.* The other side of the session should have the *dual* protocol, `!<Event1>.!<Event2>.?(Result)`. The program would store a collection of session sockets of this type, picks up one of them at which a message arrives, receives that message, stores the session socket back in the same collection and move to the next iteration. Note that, when back in the collection, the session socket now has, rather than (1), the different session type, `?(Event2).!<Result>`. Thus the collection should also store session sockets of this type: as is typical in event-loops [23], the program should treat a collection of differently typed channels.

Figure 1 shows how the proposed framework can treat such a collection of channels without losing the clear programming structure and safety assurance of session-typed programming. First, Line 1 declares a *session set type*. Unlike the preceding session types, a set type can type a channel whose type is unknown except being in a set of session types. In Line 1, a session socket (to be stored in a collection) of type `pSelector` can have either `?(Event1).?(Event2).!<Result>` or `?(Event2).!<Result>`. In Line 2 we declare a selector as in NIO [26] which encapsulates a channel collection. `SJSelector{pSelector}` declares an `SJSelector`, `sel`, initiated by a new selector instance in Line 3 (the `using` keyword, as in C#, defines the lexical scope for `sel` for resource management purposes). The type `pSelector` indicates that this selector treats a collection of channels with this set type, hence it is to

```
1  while(run) {
2    ... // Create a selector sel and register a server socket.
3    using(SJChannel{pSelect2} s = sel.select()) {
4      typecase (s) {
5        when(SJServerSocket(pStream) ss) {
6          using (SJSocket{rec X...} s0 = ss.accept()) {
7            sel.register(s0); // newly register a session socket
8            sel.register(ss); // Re-register a server socket.
9          } }
10       when(SJSocket{?{NEXT: ..., QUIT:end}} s1) {
11         s1.inbranch() {
12           case NEXT: { sel.register(s1); // Re-register. }
13           case QUIT: { ; // No re-registration. }
14         } }
15         when(SJSocket{?(Event)...} s2) {
16           Event e = s2.receive(); ...
17           s.recursion(X) {        // Unfold the recursion.
18             sel.register(s2);     // Re-register a session.
19         } }
20 } } }
```

Fig. 2. A Dynamic Stream Processing in SJ.

monitor sessions of these types. Line 5 shows how we can register a session socket `source` in the selector, to be monitored for event occurrences.

The type-safe event loop starts from Line 7. The first action in the loop, in Line 8, is the `select` operation, enclosed in a `using` clause discussed already. As in `selector` in Java NIO, this operation blocks until the selector detects an event occurrence on one of the registered session sockets, in which case an `SJSocket` `s` with the session set type `pSelect`, is returned. This set-session-typed `s` is processed in a type-safe manner by the `typecase` in Line 9 and its subsequent two `when` blocks, from Line 9 and from Line 18 respectively. The `typecase` command uses the *runtime session type* of `s` to decide which one of the `when` blocks to use. Note that, by `s` being typed by `pSelect`, these two exhaust all possible session types of `s`. Line 10 is when the session type is `?(Event1).?(Event2).!<Result>`, in which case the program rebinds the channel to `s1` of this session type (this rebinding comes from the dynamic typing of the λ-calculus in [1] and ensures type soundness) and, in Line 11, receives an `Event1` via `s1`. Since the next action is again an input, Line 12 re-registers the session with the selector. Similarly, the runtime session type of `s` is `?(Event2).!<Result>` in the second `when` block, to which `s2` is bound. The program now receives, in Line 15, an `Event2` via `s2`, and sends the `Result`, exhausting all interactions in this session (hence no re-registration is necessary).

Event Streams. As another example, we treat a common pattern where an event handling party consumes a stream of events. Combining with a recursive session type, we can specify a simple event stream as:

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

where `X` is a recursion type variable. In this expected event sequence, a process may receive one of two possible branch labels, `NEXT` and `QUIT` (where `?{NEXT: ..., QUIT: ...}` denotes a branch in a session). If `NEXT` comes, it receives an `Event` from the stream and repeats the initial part of the session (where `rec X [...#X...]` is a recursive session with `#X` denoting a recursion). The corresponding session set type is given as:

```
protocol pSelect2 {
    @pStream,
    ?{NEXT: ?(Event).rec X [ ?{NEXT:?(Event).#X, QUIT:end} ], QUIT:end},
    ?(Event).rec X [ ?{NEXT:?(Event).#X, QUIT:end} ]
}
```

The session set type `pSelect2` includes `pStream` (the prefix @ indicates a reference to an already declared type), which is for a shared channel endpoint for session initialisation (called `SJServerSocket` in SJ) via which a process can repeatedly receive session requests. Figure 2 lists another event loop that consumes each event, treating a session request from a client and registers to the selector a new session channel endpoint. This event-loop treats not only events at session channels but also those at shared channels, ensured to process all potential events expected at the registered channels because of explicit declaration of the session set type. Further programming examples are discussed in § 5.

Summary. Through two simple examples, we have seen how the combination of session set types and `typecase` leads to a well-structured description of interaction flows in event loops. The preservation and manifestation of conversation flows as types in program texts are in marked contrast to the loss of types and flows in the traditional event-driven programming.[1] By the linearity of session channels ensured by typing, communication events are processed without interference by other threads. As we shall prove later (Theorem 4.6), this leads to a strong static assurance that all possible events at the registered sessions are handled in the presence of dynamically stored sessions, called *event progress*.

3 A Process Model for Eventful Sessions

We formalise the key programming ideas illustrated in the previous section as a small process calculus. The calculus, which we call ESP (for Eventful Session Pi-calculus), is the π-calculus with session primitives [18, 27] based on asynchronous communication semantics [17],[2] with a minimal extension for events, *message arrival predicate* and *typecase*. The formalism can accurately capture the behaviour of a wide range of event-handling idioms including the selector and various join patterns through encodings, offering foundations for their sound treatment in our language design.

3.1 Syntax of ESP

Types. The type syntax of ESP extends the standard binary session types [18] with session *set types*. This simple extension allows us to treat type-safe event handing for an arbitrary collection of differently typed communication channels.

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

The shared types U include booleans bool (we also use nat in examples); shared channel types $\langle S \rangle$ for shared channels through which a session of type S is established; type variables $(\mathsf{X}, \mathsf{Y}, \mathsf{Z}, ..)$; and recursive types. The session types S are standard [18, 27]: output type $!(T);S$ represents outputting values/channels of type T, then performing the action represented by S. Dually for input type $?(T);S$. Selection type $\oplus\{l_i : S_i\}_{i \in I}$ describes the behaviour which selects one of the labels say l_i then behaves as S_i. Branching type

[1] The flow information is maintained even when a program dispatches event handling to separate event handlers since SJ's method invocation preserves session types [20].

[2] A model based on the π-calculus and asynchronous communication is a concise but expressive medium to investigate the theory of eventful sessions which focus on communications and asynchronous event handling.

(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 processes.

$\&\{l_i : S_i\}_{i\in I}$ waits with I options, and behaves as type S_i if i-th label is chosen. End type `end` represents the session completion and is often omitted. In recursive type $\mu \mathsf{X}.S$ we assume type variables are guarded in the standard way, i.e. type variables only appear under prefixes (hence contractive).

Value types (for message values) include the constants and the session set types $\{S_i\}_{i\in I}$, where I is finite (can be empty) and S_i is closed, i.e. does not contain free type variables. A *session set type* $\{S_i\}_{i\in I}$ represents the capability to interact safely as any one of S_i. For example, if a session has type $\{!(\texttt{bool}), ?(\texttt{nat})\}$, then it can interact safely with a process with a session typed by $?(\texttt{bool})$ and one with a session typed by $!(\texttt{nat})$. It will be used for typing the type-case construct. We write $\mathscr{T}$ for the set of all closed types (i.e. without free type variables), $\mathscr{S}$ for the set of closed session types. $\{S_i\}_{i\in I}$ is treated as a set, satisfying $\{S_1,\ldots,S_m\} \cup \{S'_1,\ldots,S'_n\} = \{S_1,\ldots,S_m,S'_1,\ldots,S'_n\}$. A singleton $\{S\}$ is often written S.

Processes. In the syntax of processes in Figure 3, terms that only appear at runtime are shaded; the other terms are *user syntax*. Two event-based primitives are introduced: the *message arrival predicate* `arrived` for non-blocking inspection of messages buffers, and the session *typecase* `typecase` k `of` $\{...\}$ for inspecting the runtime session type of k. The syntax also adds asynchronous session initiation (cf. [18]).

Values $v, v', ...$ include the constants, *shared channels* a,b,c, and *session channels* s,s'. A session channel designates one endpoint of a session,[3] where s and $\overline{s}$ denote two ends of a single session, with $\overline{\overline{s}} = s$. Labels for branching and selection range over $l, l', ...$, variables over x,y,z, and process variables over X,Y,Z. Shared channel identifiers u,u' include shared channels and variables; session identifiers k,k' are session endpoints and variables. Expressions e are values, variables and the message arrival predicates (`arrived` u, `arrived` k and `arrived` $k\,h$: the last one checks for the arrival of the specific value h at k). We write $\vec{s}$ and $\vec{h}$ for their vectors, writing ε for the empty vector.

Requester $\overline{u}(x:S);P$ requests a session initiation, while acceptor $u(x:S).P$ accepts this initiation. Through an established session, output $k!\langle e\rangle;P$ sends e through channel k asynchronously, input $k?(x).P$ receives a value or channel through k, selection $k \triangleleft l;P$ chooses the branch with label l, and branching $k \triangleright \{l_i : P_i\}_{i\in I}$ offers branches. The $(\nu\, u : \langle S\rangle)\,P$ binds a channel u of type $\langle S\rangle$ to its scope P. `typecase` k `of` $\{(x_i:T_i)\,P_i\}_{i\in I}$ takes a session endpoint k and a list of cases $(x_i:T_i)$, each binding the free variable x_i of type pattern T_i in P_i. The conditional, parallel composition, agent definition/instantiation, and inaction are standard. Type annotations and $\mathbf{0}$ are often omitted.

We model two kinds of asynchronous communication, *asynchronous session initiation* [21] and *asynchronous session communication* (over an established session). The former

[3] In this sense, one may call $s,s',..$ "session channel endpoints" rather than "session channels", though for brevity we usually use the latter. Similarly for shared channels.

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

Fig. 4. Selected reduction rules.

involves the *unordered* delivery of a *session request message* $\overline{a}\,\langle s\rangle$, where $\overline{a}\,\langle s\rangle$ represents an asynchronous message in transit towards an acceptor at a, carrying a fresh session channel s [17]. In actual network, a request message will first move through the network and eventually get buffered at a receiver's end. Note a message arrival can only be detected at the time of a message arrival, *not* at the time of a message sending. This aspect is formalised by the introduction of a *shared input queue* $a\,[\vec{s}]$, which denotes an acceptor's local buffer at a with pending session requests for $\vec{s}$.

Communications during a session are asynchronous and order-preserving, as in TCP. For capturing this semantics with precision in the presence of message arrival predicates, we use, at each session endpoint, an *endpoint configuration* (often simply *configuration*) $s\,[S, \mathtt{i}{:}\vec{h}, \mathtt{o}{:}\vec{h}']$, which encapsulates both input (`i`) and output (`o`) buffers. A message is first enqueued by a sender at its output queue, which intuitively represents a communication pipe extending from the sender's locality to the receiver's. The message will eventually move from this queue to the receiver's input queue, signifying the arrival of that message in the receiver's local buffer. The S in $s\,[S, \mathtt{i}{:}\vec{h}, \mathtt{o}{:}\vec{h}']$ is called *active type* and designates the remaining session actions to be performed at this endpoint.

$(\nu\, s)\,P$ binds the two endpoints, s and $\overline{s}$, making them private within P. For brevity, one or more components may be omitted from a configuration when they are irrelevant, writing e.g. $s\,[S]$ or $s\,[\mathtt{i}{:}\vec{h}]$. The notions of free variables and channels are standard [27]; we write $\mathrm{fn}(P)$ for the set of free channels in P. A closed user syntax is called *program*.

3.2 Operational Semantics

The reduction $\longrightarrow$ on closed terms captures the dynamics of processes including communication and event handling, tracking active types as session interactions progress. Figure 4 lists the key rules. We use the standard evaluation contexts $E[\_]$ defined as $E ::= \ \_ \mid s!\langle E\rangle;P \mid \mathtt{if}\ E\ \mathtt{then}\ P\ \mathtt{else}\ Q \mid X\langle\vec{v}E\vec{e}\rangle$. The structural congruence $\equiv$ and the remaining rules for reduction are standard; the full definitions are found in [28].

In [Request1], the client requests a server for a fresh session via shared channel a. A fresh (i.e. ν-bound) session channel, with two ends s (server-side) and $\overline{s}$ (client-side) and the empty configuration at the client side are generated and the session request message $\overline{a}\,\langle s\rangle$ is dispatched. [Request2] enqueues the request in the shared input queue at a. A server

accepts a session request from the queue using [Accept], instantiating its variable with s in the request message; the new session is now established.

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

An input action can block if no message is available at the corresponding local input buffer. The use of the message arrivals can avoid this blocking: [Arrive-req] evaluates `arrived` a to `tt` iff the queue is non-empty; similarly for `arrived` k. [Arrive-msg] evaluates `arrived` $s\ h$ to `tt` iff the queue is nonempty and its next message matches h. The notation $e \downarrow b$ means e evaluates to the boolean value b.

[Typecase] is the key rule which permits *dynamic* inspection of the active type of a session: after the type is inspected, a process continues the session s along the first P_i for which T_i can be successfully matched against the active type S up to subtyping (the subtyping relation is defined in § 4.1).

3.3 Examples

We show how the semantics of two high-level event primitives/idioms can be encoded into ESP. The next section (§4) shall show these encodings are type and semantic preserving, and that we can derive a sound type discipline for these constructs.

Example 3.1 (Selector). The behaviour of `SJSelector` used in the event-loop in § 2 uses at least three operations: to *create* a new selector, to *register* a channel in a selector, and *retrieve* (or *select*) a channel at which a message has arrived from a selector. After selecting, we use the *typecase* to type the selected channel. We can extend ESP with these operations, with the following reduction semantics.[4] Below we omit type annotations for selectors, on which we shall discuss in §4.

$$\texttt{new selector}\ r\ \texttt{in}\ P \longrightarrow (\nu\, r)(P \mid \mathsf{sel}\langle r, \varepsilon\rangle) \qquad \texttt{register}\ s'\ \texttt{to}\ r\ \texttt{in}\ P \mid \mathsf{sel}\langle r, \vec{s}\rangle \longrightarrow P \mid \mathsf{sel}\langle r, \vec{s}\cdot s'\rangle$$
$$\texttt{select}(r)\{(x_i:T_i) : P_i\}_{i\in I} \mid \mathsf{sel}\langle r, s'\cdot\vec{s}\rangle \mid s'\,[S, \texttt{i}:\vec{h}] \longrightarrow P_i\{s'/x_i\} \mid \mathsf{sel}\langle r, \vec{s}\rangle \mid s'\,[S, \texttt{i}:\vec{h}] \quad (\vec{h} \neq \varepsilon)$$
$$\texttt{select}(r)\{(x_i:T_i) : P_i\}_{i\in I} \mid \mathsf{sel}\langle r, s'\cdot\vec{s}\rangle \mid s'\,[\texttt{i}:\varepsilon] \longrightarrow \texttt{select}(r)\{(x_i:T_i) : P_i\}_{i\in I} \mid \mathsf{sel}\langle r, \vec{s}\cdot s'\rangle \mid s'\,[\texttt{i}:\varepsilon]$$

where in the second line S and T_i satisfies the condition for `typecase` in Figure 4. We also include the structural rules with garbage collection rules for queues as $(\nu\, r)\mathsf{sel}\langle r, \varepsilon\rangle \equiv \mathbf{0}$. Operator `new selector` r `in` P (binding r in P) creates a new selector $\mathsf{sel}\langle r, \varepsilon\rangle$, named r and with the empty queue ε. Operator `register` s' `to` r `in` P registers a session channel s to r, adding s' to the original queue $\vec{s}$. `select`$(r)\{(x_i:T_i) : P_i\}_{i\in I}$ retrieves a registered session and checks the availability to test if an event has been triggered. If so, find the match of the type of s' among $\{T_i\}$ and select P_i; if not, the next session is tested.

We now show this behaviour can be easily encoded by combining *arrival predicates* and *typecase*. Below we omit type annotations.

$$[\![\texttt{new selector}\ r\ \texttt{in}\ P]\!] \stackrel{\text{def}}{=} (\nu\, b)(\overline{b}(r); b(\overline{r}).[\![P]\!] \mid b : [\varepsilon]) \qquad [\![\texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P]\!] \stackrel{\text{def}}{=} \overline{r}!\langle s\rangle; [\![P]\!]$$
$$[\![\texttt{select}(r)\{(x_i:S_i) : P_i\}_{i\in I}]\!] \stackrel{\text{def}}{=} \qquad [\![\mathsf{sel}\langle r, \vec{s}\cdot\vec{s}'\rangle]\!] \stackrel{\text{def}}{=} r\,[\texttt{o}:\vec{s}'] \mid \overline{r}\,[\texttt{i}:\vec{s}]$$

[4] The presented semantics is based on polling, which serves our current purpose (i.e. obtaining semantic and type-theoretic basis of selectors). For further discussions on the behaviour of selectors, see §5.2.

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

The use of `arrived` is the key to avoid blocked inputs, allowing the system to proceed asynchronously. The operations on the collection need to carry session channels, hence the use of delegation (linear channel passing) is essential [18]. We can easily check that the embedding operationally simulates the selector given above as the extension of ESP, and that, under a suitable bisimulation, that it is semantically faithful.

Using the selector, ESP can represent a basic event loop similar to Figure 1, § 2:

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

where each process starts from an input conforming to the declared type. Later we shall show progress of interactions in such event loops.

Example 3.2 (Switch-receive). A *join* [3, 12] is a synchronisation pattern where a process waits with one or more guards, each guard (called *join pattern*) being a conjunction (*join*) of message arrivals at multiple channels. The `switch-receive` in Sing# [11] elegantly translates this notion in the context of structured conversations (using *channel contracts*, a version of session types). Its syntax can be formalised as ESP extension.

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

Above we set $m,n,j\geq 1$ and all $s_{j1},..,s_{jn_j}i$ should be pairwise distinct. Each J_j denotes a join pattern, the conjunction of expressions each of form $s_{ji}.l_{ji}(x_{ji}:U_{ji})$, where (1) l_{ji} is a branching label expected at s_{ji}; (2) $(x_{ji}:U_{ji})$ is a formal parameter for a message of type U_{ji} following l_{ji}. The formal semantics of `switch-receive` is given in Appendix B. Here we informally illustrate its behaviour taking a simple example. Let R be given by:

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

then R waits at s (where l_1 and l_2 are expected) and s' (where l_3 is expected). Suppose l_1 arrives; by convention, we assume each sender sends a label immediately followed by a message. Suppose v_1 follows l_1; then R will become $P\{v_1/x_1\}$. On the other hand, if l_2 arrives at s followed by v_2 and l_3 at s' followed by v_3, then it will be $Q\{v_2v_3/x_2x_3\}$. In all other cases, it will wait for the arrival of messages at s and s'.

The inductive encoding of `switch-receive` in ESP can be easily given using `arrived`. We illustrate the idea of the encoding using R above. Assuming that the communication of a selection label is always immediately followed by a message:

$$\texttt{def}\ \mathsf{Srloop} = \texttt{if}\ \ (\texttt{arrived}\ s\ l_1) \qquad \texttt{then}\ s\triangleright l_1 : s?(x_1).[\![P]\!] \\ \texttt{else if}\ \ (\texttt{arrived}\ s\ l_2\ \texttt{and arrived}\ s'\ l_3)\ \ \texttt{then}\ s\triangleright l_2 : s?(x_2).s'\triangleright l_3 : s'?(x_3).[\![Q]\!] \\ \texttt{else}\ \mathsf{Srloop} \\ \texttt{in}\ \mathsf{Srloop}$$

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

4 Typing Eventful Sessions

This section presents the type discipline for ESP and establishes key theoretical results of the paper: properties of subtyping (Proposition 4.1), type safety, (Theorem 4.2), communication and event-handling safety (Theorem 4.3), type safety of high-level event primitives through encoding into ESP (Proposition 4.4); and event progress (Theorem 4.6).

4.1 Subtyping

If P has a session channel s typed by S, P can interact at *s at most* as S (e.g. if S has shape $\oplus\{l_1 : S_1, l_2 : S_2, l_3 : S_3\}$ then P may send l_1 or l_3, but not a label different from $\{l_1, l_2, l_3\}$). Hence, $S \leq S'$ means that a process with a session typed by S is more composable with a peer process than one by S'. Composability also characterises the subtyping on shared channel types. Formally the subtyping relation is defined for the set $\mathscr{T}$ of all closed and contractive types 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 $\mathscr{F} : \mathscr{P}(\mathscr{T} \times \mathscr{T}) \to \mathscr{P}(\mathscr{T} \times \mathscr{T})$, such that $\mathscr{F}(\mathscr{R})$ for each $\mathscr{R} \subset \mathscr{T} \times \mathscr{T}$ is given as follows.

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

Line 1 is standard ($\langle S \rangle$ is invariant at S since it logically contains both S and $\overline{S}$). Lines 2 and 6 are the standard rules for recursion. In Line 3, the linear output (resp. input) is contravariant (resp. covariant) on its carried types following [27]. In Line 4, the selection is co-variant since if a process may send more labels, it is less composable with its peer. Dually for branching in Line 5. Finally, the ordering of the set types says that if each element of the set type $\{S_j'\}_{j \in J}$ has its subtype in $\{S_i\}_{i \in I}$, the former is less composable by the latter. The condition $\neg(|I| = |J| = 1)$ avoids the case $\{S_i\}_{i \in I} = S, \{S_j'\}_{j \in J} = S'$, which makes 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: $\overline{!(T); S} = ?(T); \overline{S}$, $\overline{?(T); S} = !(T); \overline{S}$, $\overline{\mu \mathsf{X}.S} = \mu \mathsf{X}.\overline{S}$, $\overline{\mathsf{X}} = \mathsf{X}$, $\overline{\oplus\{l_i : S_i\}_{i \in I}} = \oplus\{l_i : \overline{S_i}\}_{i \in I}$ $\overline{\&\{l_i : S_i\}_{i \in I}} = \&\{l_i : \overline{S_i}\}_{i \in I}$ 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 (Set Types). (1) $\leq$ *is a preorder;* (2) *Given* T, T', $T \leq T'$ *is decidable; and* (3) *(semantics of* $\leq$*)* $T_1 \leq T_2$ *if and only if* $\mathsf{comp}(T_2) \subseteq \mathsf{comp}(T_1)$.

4.2 Program Typing

We first define a typing system for programs (§ 3.1). Program typing can be considered a static typing phase performed by a compiler, on user-level code before execution. Program typing 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 *shared environment*, which maps shared channels and process variables to constant types and message types respectively; Σ is called *linear environment* mapping session channels to set types (writing $k : S$ for $k : \{S\}$) and recording shared channels for acceptor's input queues. $\Sigma \cdot a$ means $a \notin \mathrm{dom}(\Sigma)$ and similarly for others. Shared channel a is recorded in Σ to ensure that one and only one queue for a exists. 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 a process and an expression are given as:

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

$$\frac{}{\Gamma \vdash \mathtt{tt}, \mathtt{ff} : \mathsf{bool}}(\mathsf{Bool}) \quad \frac{}{\Gamma \cdot x : \mathsf{bool} \vdash x : \mathsf{bool}}(\mathsf{Var}) \quad \frac{}{\Gamma \cdot u : T \vdash u : T}(\mathsf{Shared})$$

$$\frac{\Gamma \vdash u : \langle S \rangle}{\Gamma \vdash \mathtt{arrived}\, u : \mathsf{bool}}(\mathsf{Areq}) \quad \frac{\Gamma, \Sigma \vdash \mathtt{arrived}\, k\, h : \mathsf{bool}}{\Gamma, \Sigma \vdash \mathtt{arrived}\, k\, : \mathsf{bool}}(\mathsf{Asess}) \quad \frac{\Gamma \vdash v : U}{\Gamma, \Sigma \cdot k : ?(U); S \vdash \mathtt{arrived}\, k\, v : \mathsf{bool}}(\mathsf{Amsg})$$

$$\frac{l \in \{l_i\}_{i \in I}}{\Gamma, \Sigma \cdot k : \&\{l_i : S_i\}_{i \in I} \vdash \mathtt{arrived}\, k\, l : \mathsf{bool}}(\mathsf{Alab}) \quad \frac{\Gamma \vdash P \triangleright \Sigma' \quad \Sigma' \leq \Sigma}{\Gamma \vdash P \triangleright \Sigma}(\mathsf{Subs})$$

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

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

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

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

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

$$\frac{\Sigma\ \mathtt{end\ only}}{\Gamma \vdash \mathbf{0} \triangleright \Sigma}(\mathsf{Nil}) \quad \frac{\Gamma \cdot X : \vec{U}\vec{S} \cdot \vec{x} : \vec{U} \vdash P \triangleright \vec{y} : \vec{S} \quad \Gamma \cdot X : \vec{U}\vec{S} \vdash Q \triangleright \Sigma}{\Gamma \vdash \mathtt{def}\, X(\vec{x}\vec{y}) = P\, \mathtt{in}\, Q \triangleright \Sigma}(\mathsf{Def}) \quad \frac{\Gamma, \Sigma \vdash \vec{e} \triangleright \vec{U} \quad \Sigma\ \mathtt{end\ only}}{\Gamma \cdot X : \vec{U}\vec{S} \vdash X\langle \vec{e}\vec{k} \rangle \triangleright \Sigma \cdot \vec{k} : \vec{S}}(\mathsf{Pvar})$$

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

Fig. 5. Typing rules for programs

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

Figure 5 presents the typing rules for programs. The first row is the standard expression typing (note $u : \langle S \rangle$ allows both accept and request via u by (Shared)). The next four rules are for the arrivals. Rule (Areq) is for a session request arrival. Rule (Asess) is for an in-session message arrival. Rule (Amsg) checks the message arrival and its value. Similarly (Alab) checks a branch label.

The remaining rules are standard. Rule (Subs) is the standard subsumption. Rule (Acc) (resp. Rule (Req)) says that the session following an accept (resp. request) should conform to a declared shared channel type. (Send) and (Recv) are for sending and receiving values. (Send) is standard apart from using the extended expression judgement to handle occurrences of the arrival predicates within e: the relevant type prefix $!(S)$ for the output is composed with T in the conclusion's session environment. (Recv) is a dual input rule. (SSend) and (SRecv) are the standard rules for delegation of a session and its dual [27], observing (SSend) says that $k' : T$ does not appear in P, symmetrically to (SRecv) which uses the channels in P. (Select) and (Branching), identical with [27], are the rules for selection and branching. (If) rule handles the conditional expression.

(Chan) records the $a : \langle S \rangle$ from the shared environment after checking the presence of a (in effect unique) queue at a. (Par) prevents multiple queues for the same shared channel and processes with the same session channels from being composed. (Nil), (Def)

and (PVar) are standard from [18]. (Queue) records the presence of an empty shared queue in a linear environment. "Σ end only" means Σ has the form $s_1 : \mathtt{end}, \ldots, s_n : \mathtt{end}$.

Rule (Typecase) is an extension from the dynamic typing system of the λ-calculus [1] to session types. It checks the body P_i is typed under Σ with x_i assigned to T_i. Then the whole process is typable with x assigned to T which is a supertype of all T_i.

4.3 Type Soundness and Event-Handling Safety

This subsection establishes fundamental safety properties of the type discipline for ESP we have just introduced. For the proofs, we need the typing rules for the runtime syntax, which is left to Appendix A. The runtime typing is extended from [27], in order to treat the active session types and more fine-grained buffers. We start from the 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 session 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) $s \triangleleft l_i;P \mid s[\oplus\{l_i : S_i\}_{i\in I}, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h}']$ with $i \in I$
(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) $s[S, \mathtt{i}:\vec{h}_1, \mathtt{o}:v\cdot\vec{h}'_1] \mid \overline{s}[S', \mathtt{i}:\vec{h}_2, \mathtt{o}:\vec{h}'_2]$
(f) $E[\mathtt{arrived}\ s\ v] \mid s[?(U);S, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h}']$ where v is a constant with 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 $\exists i \in I.\ T_i \leq S$.

Above we track matching between the redex and the action type. (f–h) represent the new primitives for event handing correctly interact with the queues with respect to the action types. We say a process P is an *error* if up to the structure congruence (following [19, § 5]), P contains more than two processes at s which do not form an s-redex, or expressions in P contain a type error in the standard sense. 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 High-level Event Primitives

The typing rules for the selector introduced in § 3.3 are naturally suggested from the ESP-typing of its encoding in Example 3.1. We write the type for a *user* of a selector storing channels of type T, by $\overline{\mathtt{sel}}(T)$, and the type for a selector itself by $\mathtt{sel}(T)$. For simplicity we assume these types do not occur as part of other types. The linear environment Σ now includes two new type assignments, $r : \overline{\mathtt{sel}}(T)$ and $r : \mathtt{sel}(T)$. The typing rules for the selector follow.

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

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

The typing rule for the selector queue is similar to the runtime typing for a shared input queue, cf. Appendix A. By setting $[\![\Sigma]\!]$ as the compositional mapping such that $[\![r:\overline{\mathtt{sel}}(T)]\!]$ is given as $r:S_r\cdot\overline{r}:\overline{S_r}$ where $S_r = \mu\mathsf{X}.!(T);\mathsf{X}$, and otherwise identity, as well as extending the notion of error to the internal typecase of the select command, we obtain, writing ESP$^+$ for the extension of ESP with the selector:

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 the error.*

(1, 2) are straightforward. (3) is a corollary from (1,2) and Theorems 4.2 and 4.3. In this way, the fine-grained typing rules of ESP can suggest and justify the sound typing rules for high-level event constructs through the encoding into ESP.

The typing rules for the switch-receive from Example 3.2 can also be derived and justified by its encoding, which is relegated to Appendix B.

4.5 Event Progress

In the selector in Example 3.1, the operations on a collection of channels necessarily demand *delegation* (session channel passing) since we need to put and get these channels. The presence of delegation generally makes it impossible to guarantee progress in session typed processes[5], requiring additional techniques [4]. However we observe that the selector does *not* use session channel passing in an *arbitrary* way. In fact, one of the key properties of event-driven programs is their non-blocking nature.

In the following we show that a ESP^+-process which uses a selector under natural conditions does satisfy a strong form of progress. Let P be a ESP^+-process. $\Gamma \vdash P \triangleright \Sigma$ is *delegation free* if its typing never uses (SSend) and (SRecv) in Figure 5.

Definition 4.5 (selector usage). An ESP^+ closed process $\Gamma \vdash P \triangleright \Sigma$ *uses selectors well* if each select action at r should be preceded by a register action at r, and similarly each register action at r should be followed by a select action at r, both up to the unfolding of recursion. Further for each $\mathtt{select}(r)\{(x_i : S_i : P_i\}_{i \in I}$ in P satisfies: (1) (consumption) Each S_i starts from a branching or linear input and P_i starts from the corresponding action at x_i; and (2) (exhaustiveness) In each P_i, no input/branching actions occur other than (1).

The usage of a selector above demands all possible events to be processed for registered channels, thus demanding, as far as the environment provides messages, that the seletor processes them one by one.

We now introduce two conditions from [19]. A typable ESP^+-process P is *simple* if the session typing in the premise and the conclusion of each prefix rule in Figure 5 for its derivation is restricted to at most a singleton; and $\Sigma = \emptyset$ in (Selector,Reg,Select) (simplicity in effect precludes an explicit use of delegation: the simplicity also precludes interleaving of sessions, which can be relaxed to include nested sessions as in [4]). 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 its dual active prefix always occur in Q.

We also use the refined reduction $\searrow$, defined as $\longrightarrow_s^* \longrightarrow_{ns} \longrightarrow_s^*$ where $\longrightarrow_s$ is the reduction induced by the third line (the last rule) in Example 3.1 of the reduction rules for the selector and $\longrightarrow_{ns} = \longrightarrow \setminus \longrightarrow_s$, that is $\longrightarrow_{ns}$ is the whole reduction minus $\longrightarrow_s$. We state the event progress theorem using $\searrow$ because $\longrightarrow_s$ can occur even in a deadlocked configuration: it does not constitute a "progress".

[5] Note session channel passing enables the representation of stored session channels: since we cannot use the standard ordering methods (see § 6) for such stored channels, the progress becomes hard to establish.

Theorem 4.6 (Event Progress). *We say an* ESP^+*-process* P *is* eventful *if* $\Gamma \vdash P \triangleright \emptyset$*, it is well-linked and simple, 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, which strengthens a similar result in [21, §6.5], not only strictly extends a notion of progress in the session typing discipline to the case when an implicit, but well-disciplined usage of delegations is made, but also, when we consider a selector's interactions with its environment, that all session actions registered in a selector will indeed be processes in a non-blocking fashion, formally justifying the "good properties" of the standard event programming pattern under the present type discipline in a general semantic context.

5 Eventful SJ: Language Design and Implementation

This section presents highlights of the language and runtime design of eventful SJ, an integration of session-based programming and asynchronous event handling built on the basis of SJ [20]. We then report our experience of programming with eventful SJ, through an implementation of a substantial event-driven application, an open-source SMTP server [29].

5.1 Language and Runtime Design

Event primitives. The purpose of Eventful SJ is to experiment with a *practical* integration of asynchronous event handling (which allows the standard event-driven programming patterns) and session-typed programming (which provides flow-aware, type-safe structured communications programming), in terms of both efficiency and expressiveness. The theoretical inquiries reported in the previous section give a firm basis for its type discipline: they also offer insight on the nature of different event-based primitives. The current design of Eventful SJ is based on the three primitives: (1) *Message arrival predicate* on shared and session channels; (2) *Selector* which encapsulates a dynamic (shared and session) channel collection and event notification; and (3) *Typecase* for both shared and session channels.

(1) offers a light-weight facility for programming events and its combination with (2,3) can represent a diverse form of event primitives as discussed in § 3 and 4. However, as seen from the idle reductions $\longrightarrow_s$ in Example 3.1, there is an inherent architectural limitation in efficient implementation of event primitives using polling: we need a language-level encapsulation of channel collection and an event notification mechanism, which reacts quickly and which does not waste cycles in user's threads. The same concern motivated the event-facility part of Java NIO [26], following which we introduce the familiar *selector* for sessions. The challenge is to have an efficient architecture for realising the selector which can maintain the typed session semantics and which is extensible to different communication transports.

Compilation. The SJ programmer uses session types and session programming constructs to program communications. The SJ compiler statically type checks this program including session types and transforms the high-level session primitives into transport-independent SJ runtime operations, producing standard Java classes. The SJ Runtime then realises the semantics of high-level session communications through interactions with an underlying transport, including event-related facilities. This is done through an interface which encapsulates concrete transport characteristics as abstract operations.

```
1  SJProtocol _sjtypecase0 = new SJProtocol(...); // rec X [...]
2  SJProtocol _sjtypecase1 = new SJProtocol(...); // ?{NEXT: ..., QUIT: end}
3  SJProtocol _sjtypecase2 = new SJProtocol(...); // ?(Event)....
4  ...
5  while(run) {
6    ... // Declaration and initialisation of the selector.
7    { // Braces for the lexical scope of the using statement.
8      SJChannel c = null; // The using statement variable declarations.
9      try {
10       c = SJRuntime.select(sel); // 'sel' is the selector.
11       SJSessionType _sjtmp0 = c.remainingSessionType();
12       if(_sjtmp0.isSubtype(_sjtypecase0.getType())) { // Start of typecase.
13         SJServerSocket ss = (SJServerSocket) c; // ''Rebind''..
14         t = null; // ..the typecase variable.
15         ... // Call accept on the server socket and then..
16         ... // ..(re)register the server and new session sockets with sel
17       } else if(_sjtmp0.isSubtype(_sjtypecase1.getType())) {
18         ... // Translation of inbranch for NEXT and QUIT cases.
19       } else if(_sjtmp0.isSubtype(_sjtypecase2.getType())) {
20         SJSocket s2 = (SJSocket) c;
21         t = null;
22         Event e = (Event) SJRuntime.receive(s2); // Cast inserted.
23         ... // Translation of recursion scope.
24       } else {
25         throw new SJIOException("Runtime session type error: typecase.");
26       } // End of typecase.
27     } finally {
28       SJRuntime.close(c);
29     } } }
```

Fig. 6. Compilation of Event streams in Figure 2 into Java (extract).

We illustrate how the compiler translates high-level session operations into SJ Runtime operations through an example, focusing on eventful operations, `arrived`, `SJSelector` and `typecase`. Recall the "Event streams" example from § 2. Figure 6 lists an abridged extract of the standard Java code generated by the SJ compiler. As in C#, `using` statements are translated into `try-finally` statements with appropriate resource management in the `finally` block, typically `close` operations, and the lexical scope of `using` variable declarations is controlled by an outer pair of block braces (Lines 7 and 29). Basic session operations like `select` (Line 10) and `receive` (Line 22) are directly translated into SJ Runtime operations, passing the target references (respectively the `SJSelector sel` and the `SJSocket s2`) as arguments; similarly for `send` and `arrived`.

Next we show how the `typecase` is translated into the corresponding Java code and session operations. First, the session types in the guards are serialized and embedded into the parent class as `SJProtocol` objects. We then insert a `remainingSessionType` call to the typecase target `c` (Line 11) 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. The rest follows the formal semantics of `typecase` presented in § 3.2. 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.

5.2 Eventful SJ Runtime

General structure of runtime. The Eventful SJ Runtime (SJR) offers an abstract platform for session execution, so that the same high-level typed session programs can be

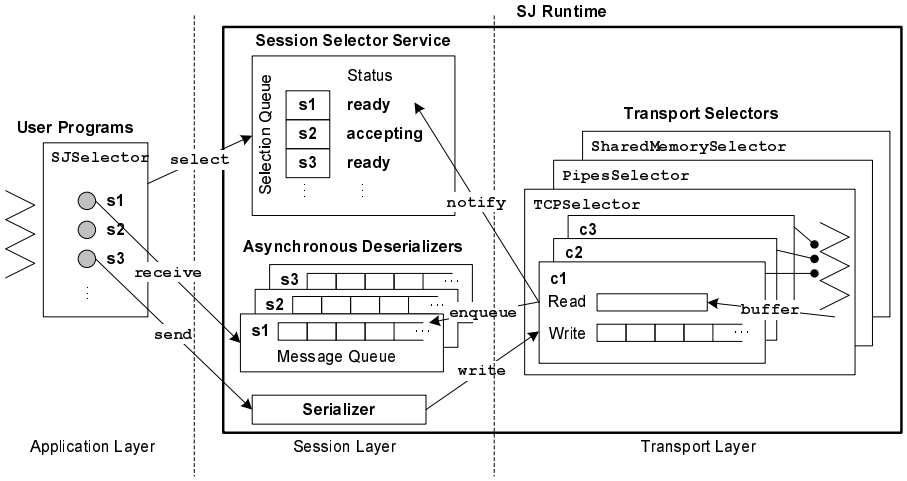

Fig. 7. The SJ Runtime components for executing an event-driven session program.

seamlessly executed under different transports, including event-based operations. As far as we implement appropriate abstract operations based on concrete transport, all high-level services are uniformly available to diverse transports.

As described above, the SJ compiler transforms SJ classes into standard java binaries, translating user-level session operations into *SJR operations*, i.e. SJ runtime operations. The SJR operations called by the session threads target specific *Interaction Service* (IS) components, which form an upper layer of the runtime, encapsulating services such as session initiation, message serialization and wire format, branch synchronisation, session delegation, and runtime session type monitoring, as well as event-based operations. The IS services are in turn realised by the SJR in terms of communication actions on an underlying transport connection. These communication actions are performed through 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, extended with event-based interface as we discuss below.

Runtime facility for session event selectors. We already discussed how the *typecase* is translated into the standard Java command, together with basic SJ operations such as runtime session type inspection. In the following we illustrate the runtime structure of session event selectors, a core facility for efficient runtime realisation of type-safe, session-safe asynchronous event programming. Figure 7 depicts the main IS and Transport layer components for an event-driven session program implemented using the `SJSelector` API.

There are three key runtime components contributing to asynchronous event processing: *ATI event extension*, *asynchronous message deserializer*, and *selector service*.

Eventful ATI. The event-extended ATI exports a transport-specific `TransportSelector` interface for detecting asynchronous communication events at the transport level. The implementation of each transport-specific selector is provided by the Transport module that implements the ATI. For example, `TCPSelector` uses Java NIO. As seen in Figure 7, this framework can realise the same reactive semantics of the selector realised over multiple transports. On initialisation, the SJR creates and maintains one instance of the `TransportSelector` for each event-capable transport.

Asynchronous deserializers. Sessions using eventful operations must specify an appropriate asynchronous deserializer component, which encapsulates conversion of

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

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

```
// Handle DATA command.
protocol pData {
  !{
    RC354:
      !<DataAck>
     .?(MessageData)
     .!<MessageDataAck>
    RC554: ...,
    ... } }
```

Fig. 8. Server-side session types (extracts)

transport-level binary data into application-level messages. When a read event occurs at the transport-level, the `TransportSelector` does an *upcall* to the deserializer to determine whether enough data for a complete session-level message is received. If so, the message is enqueued in the deserializer, and any remaining data is re-buffered. Otherwise, the data is just re-buffered for 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, ready to be received; the `SJSelectorService` can then report this event when `select` is called from the application-level.

Using an upcall initiated by a `TransportSelector` instead of polling by a user-level program, event detection can be done immediately and efficiently. Note also the eventful ATI above directly supports the semantics of message arrival predicates.

5.3 An Event-driven SMTP Server in Eventful SJ.

As a testbed for our event-based extension of SJ, we implemented an event-driven SJ SMTP server. SMTP [29] is an Internet standard for e-mail transfer, used by Mail Transfer Agents (MTA) to accept, route and deliver mails. Our discussions focus on the functionality of SMTP servers when an MTA interacts with local mail clients.

Session type specification. An SMTP session is a dialogue of commands from the Client and responses from the Server that carries out a sequence of zero or more mail transactions. We give a formal specification of SMTP as a session type. Figure 9 lists extracts from the SMTP session type, seen from the side of the Server, given as a collection of SJ protocols. Based on this specification we implement an SMTP server using SJ.

Implementation. For an event-driven SJ implementation of the SMTP server, the first step is to identify the session set type for the selector for the main Server event loop:

```
protocol pSelector {
  ?(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.
  ... // Could also handle session accept events.
}
```

```
void mainEventLoop(SJSelector{pSelector} sel) throws ... {
  while(run) {
    using (SJChannel{pSelector} c = sel.select()) {
      typecase (c) {
        ...
        when (SJSocket{?(Ehlo).!<EhloAck>.@pBody} s1) { // Received EHLO.
          Ehlo ehlo = s1.receive();
          s1.send(new EhloAck("250 Hello ..."));
          sel.register(s1); // Register for the first command.
        }
        when (SJSocket{@pbody} s2) { // The main mail transaction commands.
          s2.recursion(X) {
            s2.inbranch() {
              case MAIL: sel.register(s2); // Expecting Sender address.
              case RCPT: sel.register(s2); // Expecting Recipient address
              case DATA: handleData(s2); // Handle DATA command.
              case QUIT: s2.send(new Quit("221 ...")); // SMTP end.
        } } }
        when (SJSocket{@pMail.@pBody} s3}) { // Received Sender address.
          handleMail(s3); // Use the handler to perform @pMail.
          sel.register(s3); // Register for the next command.
        }
        ...
} } } }

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

Fig. 9. The main event loop and the `MAIL` event handler.

Figure 9 lists an extract from the implementation 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 as appropriate. 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 of the main session body. The second `when` case, for the recursive type of the main session body, handles the Client commands for mail transactions. The `MAIL` and `RCPT` branch cases immediately re-register the session for the subsequent `Address` message input. In the `DATA` branch case, we call the omitted `handleData` method to send the intermediate `RC354` acknowledgement. Following the session type, the `QUIT` branch case sends an acknowledgement but does not re-register the session: the session has 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 below, 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` method 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`.

The implemented SMTP server is fully interoperable with non-SJ SMTP clients. Its implementation framework makes the most of the modular architecture of Eventful SJ which would be generally applicable to text-based application-level protocols. Firstly, we make use of the custom serialization and de-serialization: we read and write UTF-8,

Messages (1 KB) handled per second			
Clients	100	500	900
Session Threads	393	377	371
Session Events	419	414	409

Fig. 10. Throughput for multithreaded and event-driven SJ SMTP servers.

formatted according to the SMTP protocol, e.g. messages are terminated by CRLF. Each message class, including those for branch labels, uses its own deserialization routine: the SMTP deserialization component uses the *current type of each session* (tracked by the SJR runtime session monitor) to determine the expected message type(s) and apply the appropriate routine.

Some of the notable features of our event-driven SJ SMTP server are:

1. Our server is guaranteed to conform to the session type specification of SMTP through static typing. Hence, *communication safety* is guaranteed for all sessions with compliant clients; we have tested our server against commercial clients such as Microsoft Outlook and Apple Mail.
2. Our server is inherently *cross-transport*: its custom serialization and deserialization run above ATI, hence is re-usable over different transports. This enables the best possible transports at a given time to be used for interactions between an SMTP server and a client, such as a high-speed transport such as SDP in LAN.

The ability to declare and statically verify protocols in communications-based programs will lead to various merits in engineering application-level protocols. Richer protocols may be specified with much less efforts and with more precision, and static type checking can automatically assure full compatibility among implementations, which may even be programmed in different languages as far as they are equipped with session type checking.

As a macro benchmark, we compare this event-driven SMTP server against an equivalent multithreaded SJ implementation. The machines used are Intel Core 2 Duos at 3 GHz with 4 GB RAM, running Ubuntu 9.04 (kernel 2.6.28), connected via gigabit Ethernet. Latency was measured to be 0.8 ms (64 byte ping) on average. Each server was run on one machine (for this experiment, tied to one core), to which we connected a number of concurrent clients. Each client was set to run a continuous repeating mail transaction loop, submitting messages of varying sizes, and we measured the number of messages accepted by each server within 30 s time windows. Figure 10 gives the average number of messages handled per second by each server, for 100, 500 and 900 clients and mail data size 1 KB. The results show that, whilst throughput degrades as the number of clients increases, the event-driven SJ server is able to maintain a consistently higher throughput.

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 [2, 32]. 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 [23] 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 [8] is designed to clarify the relationships between event registrations and callbacks to support the accompanying tool

suite for call graph analyses. The nesC language [13] 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 [10] 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 [19] 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 [24], 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 [25] 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 [16] 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 [33] 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 [22, 34]. 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] 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 [28]).

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 [20] with clarity and flexibility.

Session typed programming languages and formalisms. Sing# [11], 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 [15] 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 [4–7]. 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 [4–7]. 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 [4, 5]. 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 [6, 7]. 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.

7 Conclusion

We have proposed a formal theory and Java extension of the consistent integration of session types and events, offering a basis for structured asynchronous eventful communications programming. We have shown its formal semantics and typing systems, and proved their expressiveness and significant formal properties, including the *event progress*. To our best knowledge, the theory is the first to present such an extension for session types which ensure a strong progress in the presence of dynamically registered sessions.

We then materialised this theory as a concrete programming language extending Java, based on SJ [20]. Both abstraction and efficiency concerns motivated us to have three

high-level language constructs for events, distilling the key features of event-based programming. The type checker of SJ is extended to treat these constructs, based on our type theory, ensuring basic safety and progress properties. The modular eventful runtime is designed and implemented so as to allow the smooth incorporation of different transports under a uniform typed abstraction of eventful communications programming. Our implementation framework for the selector (cf. Figure 7, page 16) would be usable for these other event-handling primitives.

We have substantiated the significance of the proposed programming paradigm and runtime structure through the implementations of an event-driven SMTP server [29], which is inter-operable with existing clients whilst ensuring communication safety and progress through session type checking. As far as we know, our eventful SJ is the first extension which offers type-safe communications programming in object-oriented languages that combines the asynchronous event-handling with structured typed high-level programming, with assurance of safety properties backed up by the theory.

There are several topics which may be investigated on the basis of the eventful SJ and its theory based on ESP, in addition to performance tuning. For language design, the current three primitives offer a basis for further inquiries for primitives which are efficient in most of the significant cases and which give us good programming structures. We believe that the runtime framework for selectors discussed in § 5.2 is generally effective for high-level event primitives. The ability to treat events substantially expands the scope of session programming, enabling to treat such notions as session hibernation and process migration [30].These and other integrations (including those with state-based objects [15] and distributed events facilities [10], both discussed in § 6) can be realised by adding different services to the SJ Runtime, making the most of its modular structure.

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. 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 *Proceedings of the 2002 Usenix ATC*, 2002.
3. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *TOPLAS*, 26(5):769–804, 2004.
4. L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
5. L. Caires and H. T. Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.
6. G. Castagna et al. Foundations of session types. In *PPDP'09*, pages 219–230. ACM, 2009.
7. G. Castagna and L. Padovani. Contracts for mobile processes. In *CONCUR 2009*, number 5710 in LNCS, pages 211–228, 2009.
8. R. Cunningham and E. Kohler. Making events less slippery with eel. In *HOTOS'05*, pages 3–3, 2005.
9. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352, 2006.
10. 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.
11. M. Fähndrich et al. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys2006*, ACM SIGOPS, pages 177–190, 2006.
12. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. *J. Log. Algebr. Program.*, 57(1-2):23–69, 2003.

13. D. Gay et al. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *PLDI*, pages 1–11, 2003.
14. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
15. 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. To appear.
16. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
17. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, volume 512 of *LNCS*, pages 133–147, 1991.
18. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
19. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
20. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
21. 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.
22. M. Krohn. Building secure high-performance web services with okws. In *ATEC'04*, pages 15–15. USENIX Association, 2004.
23. M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *ATC'07*, pages 1–14. USENIX Association, 2007.
24. H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.
25. 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.
26. S. Microsystems Inc. New IO APIs. `http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html`.
27. D. Mostrous and N. Yoshida. Session-based communication optimisation for higher-order mobile processes. In *TLCA'09*, volume 5608 of *LNCS*, pages 203–218. Springer, 2009.
28. SJ homepage. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`.
29. The Simple Mail Transfer Protocol. `http://tools.ietf.org/html/rfc5321`.
30. A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MOBICOM*, pages 155–166, 2000.
31. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994.
32. R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HOTOS'03*, pages 4–4. USENIX Association, 2003.
33. R. von Behren et al. Capriccio: scalable threads for internet services. In *SOSP '03*, pages 268–281. ACM, 2003.
34. M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP'01*, pages 230–243. ACM Press, 2001.

A Typing Runtime Processes

This subsection gives typing for runtime processes following the presentation in [27]. The judgement is extended as: $\Gamma \vdash P \triangleright \Delta$ with $\Delta ::= \Sigma \mid \Delta \cdot s : [S, \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T}']$ 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

$$\frac{\Gamma \vdash P \triangleright \Delta \;\; \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \odot \Delta'} \qquad \frac{\Gamma \vdash a : \langle S \rangle}{\Gamma \vdash a\,[\langle S \rangle : \vec{s}] \triangleright \cup_{s \in \vec{s}} \{s : (S, [S, \mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon])\}, a} \qquad \frac{}{\Gamma, \{s : S\} \vdash s : S}\,(\mathsf{Par'},\mathsf{Queue'},\mathsf{ASreq'})$$

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

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

Fig. 11. Selected typing rules for runtime processes.

and the type information $[S, \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T'}]$ for the associated configuration at s. We identify $(S, [\ldots])$ with $([\ldots], S)$.

The composition of Δ_1 and Δ_2, denoted by $\Delta_1 \odot \Delta_2$ [27], 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 $S \odot [S', \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T'}] = [S', \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T'}] \odot S = [S', \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T'}]$ 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 [27] because we require the stronger condition that a well-formed configuration with an output prefixed active type cannot have a non-empty local $\mathtt{i}$-buffer or be composed with an configuration with a non-empty $\mathtt{o}$-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:

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

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 $\mathtt{i}$-buffer and opposing $\mathtt{o}$-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 $\forall s \in \mathrm{dom}(\Delta), \mathrm{wc}(\Delta, s)$.

We first 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 $\overline{s}$. When $\Delta_1 \sqsubset \Delta_2$ and $\Delta \odot \Delta_1$ defined,we define $\Delta \odot \Delta_1 \sqsubset \Delta \odot \Delta_2$. 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. 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). (ASreq') 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 $\langle S\rangle$. 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. (SRes) 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 session 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 have:

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.

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

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

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

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:

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

$\mathit{Dummy}, \mathit{Dummy}'$ 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 $j \in I, J_j = s_{j1}.l_{j1}(x_{j1}) \wedge \ldots \wedge s_{jn_j}.l_{jn_j}(x_{n_j})$, 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.

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

Fig. 12. Structural congruence.

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.

D Appendix: Proofs

D.1 Proof of Proposition 4.1

(1) is mechanical; (2) the decidability of subtyping is proved in [14] 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 [14].

D.2 Proofs of Basic Lemmas

The following lemmas and Subject Reduction Theorem are fully proved in **[21, § 6]** (for the extended typecase) for all of the key cases. We list only the important cases, referring the corresponding subsections in [21, § 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 [21, § 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 $\Gamma \vdash P \triangleright \Sigma$.*

Similarly for the runtime system by replacing Σ to Δ.

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

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

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

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

Proof. See [21, § 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 [21, § 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 [21, § 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 [21, § 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 [21, § 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 [21] lists the full proofs. We only list the `typecase`.
Assume the reduction

$$\texttt{typecase}\ s\ \texttt{of}\ \{(x_i : T_i)\, P_i\}_{i\in I} \mid s\,[S] \longrightarrow P_i\{s/x_i\} \mid s\,[S] \tag{2}$$

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

$$\Gamma \vdash \texttt{typecase}\ s\ \texttt{of}\ \{(x_i : T_i)\, P_i\}_{i\in I} \mid s\,[S] \triangleright \Sigma \cdot s : [S] \tag{3}$$

The above (3) is derived by (Par') in Figure 11 from

$$\Gamma \vdash \texttt{typecase}\ s\ \texttt{of}\ \{(x_i : T_i)\, P_i\}_{i\in I} \triangleright \Sigma_1 \cdot s : T' \tag{4}$$

and

$$\Gamma \vdash s\,[S] \triangleright \Sigma_2 \cdot s : [S] \tag{5}$$

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

$$T' \leq S \tag{6}$$

The judgement (4) is derived by:

$$\Gamma \vdash P_i \triangleright \Sigma_1' \cdot x_i : T_i \tag{7}$$

with

$$\cup_{i\in I} T_i \leq T \leq T' \text{ and } \Sigma_1' \leq \Sigma_1 \tag{8}$$

by (Subs) and (Typecase). From (7) by Substitution Lemma (vi), we have

$$\Gamma \vdash P_i\{s/x_i\} \triangleright \Sigma_1' \cdot s : T_i \tag{9}$$

Note that by (6) and (8), we have $T_i \leq S$, hence $T_i \odot [S]$ is again defined. Hence applying (Subs) and (Par') to (5) and (9), 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 [21, § 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`.

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

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 `switch` − `receive`.

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

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 [19] (as seen in this updated version, page 14). 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 `register` k `to` r `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 [19] 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 8), 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. □

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

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

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

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.

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 [27].

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

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

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

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.

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

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:

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

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:

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