We report on two tools that extend Java with support for static typechecking of communication protocols. Our Mungo tool extends Java with *typestate* definitions, which allow classes to be associated with state machines defining permitted sequences of method calls. A complementary tool, StMungo, takes a communication protocol specified in the Scribble protocol description language, and generates a typestate specification for each endpoint, capturing the permitted sequences of messages along that channel. Endpoint implementations can be validated by Mungo against their typestate definitions and then compiled as usual with javac. We formalise Mungo’s typestate inference system and demonstrate the Scribble, Mungo and StMungo toolchain via a typechecked SMTP client that connects Java to the broader setting of communication protocols specified in the Scribble protocol language [41]. Given a Scribble protocol projected to a particular endpoint (a so-called *local protocol*), StMungo will generate a typestate specification capturing the sequences of sends and receives permitted along that endpoint. Each endpoint implementation can be validated separately by Mungo against its typestate definition and then compiled as usual with javac.

The separate typechecking of each endpoint is integral to our approach, and is justified by the theory of *multiparty session types* [25], the formal foundation of Scribble. Multiparty session types provide an important safety guarantee: once each endpoint implementation is known to conform to its local protocol, the various implementations can be composed into a system free of communication errors.

Our work contributes to a line of research applying session types to real-world programming languages [9, 15, 17, 16, 22, 28, 34, 36, 33, 39]. In particular, our work builds on that of Gay et al. [23], which first connected session types to the object-oriented notion of typestate. They observed that the valid sequences of messages for a given endpoint could be captured by a typestate definition for a class, allowing the channel endpoint to be modelled as an object. While an important idea, this earlier work lacked a practical implementation and relied on typestate declaration on parameters and return types.

Mungo improves on this earlier work by employing an inference system, removing the need for typestate declarations on parameters and return types. The Mungo/StMungo toolchain offers other practical advances over previous efforts to combine session types with objects. For example, SJ [28] only supports binary session types, whereas StMungo generates Mungo specifications from multiparty session types. Furthermore, Mungo permits non-local use of objects with typestates. Using the @Typestate annotation means we avoid any need for language extensions.

Tracking object typestates requires a mechanism for managing object aliasing. For Mungo, we require objects that declare a typestate to be used linearly. While this is probably too restrictive for general-purpose programming, it is a standard technique for enabling typed communication along channels; most session type systems impose similar constraints on channel usage. Objects that lack typestate definitions can be used unrestrictedly alongside linear objects. In future work (§8) we will investigate more flexible alias control mechanisms, drawing on the substantial existing literature.

### 1.1 Contributions

The main contributions of the paper are as follows:

**Mungo.** We describe the Mungo typestate checker for Java. Mungo currently supports a subset of Java; support for the full language is discussed in §8.
StMungo. We describe StMungo (§ 3), which translates Scribble local protocols into Mungo typestate specifications. StMungo also generates Java method stubs for each endpoint.

SMTP case study. A substantial example, a statically type-checked SMTP client (§ 4), illustrates the end-to-end toolchain provided by Scribble, StMungo and Mungo.

Typestate inference system. We formalise the essential features of Mungo as a core object-oriented calculus (§ 5). We define a typestate inference system for that language and prove its type-safety (§ 6).

2. MUNGO

Mungo\(^2\) extends Java with an optional typestate system. The tool is implemented in Java using the JastAdd framework [24], a meta-compiler based on reference attribute grammars. Source files are typechecked in two phases: first according to the regular Java type system, and then according to our typestate extension. The source files can then be compiled using javac and executed in the standard Java 1.8 runtime environment.

The main extension provided by Mungo is the ability to attach a typestate definition to a Java class. A typestate defines an object protocol in the form of a state machine. Each state offers a set of methods that must be a subset of the methods defined by the class; each method specifies a transition to a successor state. Typestate definitions are defined in separate files, using the Java-like syntax shown in Example 1 below. A typestate definition is attached to a class using the annotation @Typestate("ProtocolName"), where "ProtocolName" names the file where the typestate is defined. The typestate inference algorithm, presented in § 6, constructs the sequences of methods called on all objects associated with a typestate, and then checks if the inferred typestate is a subtype of the object’s declared typestate. An object without a declared typestate is typechecked as normal.

Some Java features are not yet supported. Some we anticipate to be relatively straightforward extensions (synchronised statements, the conditional operator ?::, inner and anonymous classes, and static initialisers). Generics, inheritance and exceptions are non-trivial and are discussed in future work (§ 8). Currently, generics are not supported; inheritance is supported for classes without typestate definitions; and exceptions are supported syntactically but are typechecked under the (unsound) assumption that no exceptions are thrown. (A try-catch statement is typechecked by typechecking the try body; if an exception is thrown a typestate violation may result.)

Example 1. We introduce Mungo through the example of a stack data structure that follows a typestate specification. Given the following enumerated type:

```
enum Check { EMPTY, NONEMPTY }
```

then one possible typestate protocol for a stack is as follows:

```
typestate StackProtocol {
  Empty = { void push(int): NonEmpty,
      void deallocate(): end }
  NonEmpty = { void push(int): NonEmpty,
      int pop(): Unknown }
  Unknown = { void push(int): NonEmpty,
      Check isEmpty():
        EMPTY: Empty,
        NONEMPTY: NonEmpty }
}
```

This definition specifies that a stack is initially Empty. The Empty state declares two methods: push(int) pushes an integer onto the stack and proceeds to the NonEmpty state; deallocate() frees any resources used by the stack and terminates its usage. The deallocate() method is not available in any other state, requiring a client to empty the stack before it is done using it. In the NonEmpty state a client can either push() an element onto the stack and remain in the same state, or pop() an element from the stack and transition to Unknown.

Unlike push(), pop() must leave the stack in the Unknown state because the number of elements on the stack are not tracked by the protocol. From the Unknown state, one can either push() and proceed to NonEmpty, or call isEmpty() to explicitly test whether the stack is empty. Calling isEmpty() returns a member of the enumeration Check defined earlier. This idiom, based on Java enumerations, is the mechanism for communicating a choice made by the callee synchronously back to the client, and is explained in more detail below. Here, a stack implementation can choose between returning EMPTY and transitioning to Empty, or returning NONEMPTY and transitioning to NonEmpty.

We can now define a stack implementation Stack that conforms to the StackProtocol specification, using an integer array to store the elements. The annotation @Typestate("StackProtocol") is used to associate the typestate definition with the class:

```
@Typestate("StackProtocol")
class Stack {
  private int[] stack; private int head;
  Stack() { stack = new int[MAX]; head = 0; }
  void push(int d) { stack[head++] = d; }
  int pop() { return stack[head--]; }
  Check isEmpty() { if(head == 0) return Check.EMPTY; return Check.NONEMPTY; }
  void deallocate() {} }
```

Finally, having implemented StackProtocol, we can define a stack client that makes use of the Stack implementation, with Mungo verifying that Stack instances are used correctly.

```
class StackUser {
  Stack pushN(Stack s, int n)
  { do { s.push(n-1); } while(n>0); return s; }
  Stack popAll(Stack s)
  { loop : do {
    System.out.println(s.pop());
    switch(s.isEmpty()) {
      case EMPTY: break loop;
      case NONEMPTY: continue loop;
    } while(true);
    return s; }
  public static void main(String[] args)
  { StackUser su = new StackUser();
    Stack s = new Stack(); Stack s2;
    s = su.pushN(s,16); s2 = su.popAll(s);
    s = su.pushN(s2,64); s = su.popAll(s);
    s.deallocate(); } }
```

For illustrative purposes, the client defines two helper methods: pushN(Stack s, int n), for any \( n > 0 \), pushes the integers \( n \ldots 1 \) onto the stack \( s \), and popAll(Stack s) pops all the elements of \( s \). We now discuss some details of the programming model, drawing on this example where appropriate.

Local variables, parameters, and return values. The main() method above creates a single Stack instance, stores it in a local variable \( s \), and then passes it to various invocations of pushN and popAll, from which it is also returned as a result. We also make

\(^{2}\) The tool is developed and maintained by the first author and can be downloaded from our web page [1].
use of the additional local variable s2. When returned from a method, the stack has a potentially different typestate than it did as an argument. No explicit typestate definitions are required for the parameter or return types of pushN and popAll, since Mungo can infer them. An alternative to this “continuation-passing” style, using fields, is discussed below.

Recursion and internal choice. Method pushN() illustrates the consumption of a recursive typestate offering a choice. The loop of the form do-while(exp) requires s to initially be either Empty or NonEmpty; at each iteration the client decides which of the available methods to call. In this case it chooses to push another value onto the stack. This leaves the stack in state NonEmpty, allowing another choice to be made on the next iteration. This is compatible with the recursive structure of the NonEmpty state, which permits an unbounded number of push() operations, looping back to NonEmpty each time.

Recursion and external choice. Method popAll(Stack s) also illustrates the consumption of a recursive typestate, but here the stack rather than the client makes the choice. (In session type terminology, the client offers an external choice.) This takes the form of a labelled do-while(true) in conjunction with a switch. The switch statement inspects the Check enumeration returned by isNotEmpty: in the NOTEMPTY case, the loop continues, and in the EMPTY case the loop terminates. Due to their particular control flow, loops of the form

    label: do switch block while(true)

are a suitable pattern for consuming a recursive typestate when the condition on the recursion is an external choice (i.e. based on an enumeration label).

Linear objects. Mungo ensures linear usage of objects that follow a typestate protocol; aliasing on objects allows for different method calls on an object that might lead to an inconsistent typestate. Notice that in line 15 of the StackUser example:

```java
ds = su.pushN(s,16); s2 = su.popAll(s);
```

the return value of popAll() is assigned to s2. Now, suppose line 16 were replaced with the following:

```java
s = su.pushN(s,64); s = su.popAll(s);
```

In this case Mungo would report a linearity error on argument s in su.pushN(s, 64) informing the programmer that variable s is used uninitialized, because the usage of variable s in line 15 as an argument consumed its linear value.

Inferring typestate for fields. Using fields to store objects can lead to a more idiomatic object-oriented style than explicitly passing values between methods. To show how this works, we define a second client, StackUser2, that stores a Stack as a field.

```java
class StackUser2 {  
  private Stack s;  
  StackUser2() { s = new Stack(); }  
  boolean pushN(int n)  
  { do{ s.push(n--); }while(n>0); 
    return true; }  
  void popAll()  
  { loop : do { 
    System.out.println(s.pop()); 
    switch(s.isEmpty()) {  
    case EMPTY: break loop;  
    case NONEMPTY: continue loop;
```

} while(true); }

```java
public static void main(String[] args)  
{ StackUser2 su = new StackUser2(); 
  if(su.pushN(15))|su.pushN(32)) 
  { su.pushN(32); 
    su.popAll(); su.finish(); 
  }

To track the typestate of a field we need to know the possible sequences in which methods of its containing class may be called. That, in turn, requires having a typestate for the containing class. In this case, to track the typestate of the field s, Mungo requires us to provide a typestate for StackUser2. This state machine will then drive typestate checking for those fields of StackUser2 which have their own typestate definitions. For example we could define the following StackUserProtocol for StackUser2:

```java
typestate StackUserProtocol { 
  Init = { boolean pushN(int): Cons , 
    void finish() : end } 
  Cons = { boolean pushN(int): Cons , 
    void popAll(): Init } }
```

Typechecking the field s of StackUser2 field follows the possible sequences of method calls specified by StackUserProtocol, and also takes into account the constructor body of StackUser2. Then Mungo can guarantee that if a StackUser2 instance is used according to StackUserProtocol then the Stack field of the object is also used according to StackProtocol.

Short-circuit boolean expressions. Line 16 above illustrates a final technical detail of typestate inference. The inference algorithm takes into account the fact that logical disjunction short-circuits if the first disjunct evaluates to true. Mungo will ensure that the typestate of su is consistent with there either being one, two or three successive invocations of pushN() .

3. STMUNGO: SCRIBBLE-TO-MUNGO

The integration of session types and typestate, defined by Gay et al. [23], consists of a formal translation of session types for communication channels into typestate specifications for channel objects. The main idea is that a channel object has methods for sending and receiving messages and the typestate specification defines the order in which these methods can be called; therefore it is a specification of the permitted sequences of messages, i.e. a channel protocol.

We extend this translation from binary to multiparty session types [25] and implement it as the StMungo (Scribble to Mungo) tool3, which translates Scribble [41, 46] local protocols into typestate specifications and skeleton socket-based implementation code. The resulting code is typechecked using Mungo. A Scribble local protocol describes the communication between one role and all the other participants in a multiparty scenario, including the way in which messages sent to different participants are interleaved. This interleaving is not captured by binary session types and by tools based on them, like SJ [28]. StMungo is based on the principle that each role in the multiparty communication can be abstracted as a Java class following the typestate corresponding to the role’s local protocol. The typestate specification generated from StMungo together with the Mungo typechecker can guide the user in the design and implementation of distributed multiparty communication-based programs with guarantees on communication safety and soundness.

3The tool is developed and maintained by the second author and can be downloaded from our web page [1].
StMungo is the first tool to provide a practical embedding of Scribble multipartty protocols into object-oriented languages with typestate. We illustrate StMungo on a multiparty protocol that models the process of booking flights through a university travel agent. The full details of this example are given in [30]. There are three participants involved: Researcher (abbreviated R), who intends to travel; Agent (A), who is able to make travel reservations; and Finance (F), who approves expenditure from the budget. After the request, quote and check messages requesting authorisation for a trip, Finance can choose to approve or refuse the request. The global protocol is defined as follows.

```plaintext
1 global protocol
2   BuyTicket(role R, role A, role F){
3       request(Travel) from F to R; A;
4       quote(Price) from A to R;
5       check(Price) from R to F;
6       choice at F {
7           approve(Code) from F to R,A;
8           ticket(String) from A to R;
9           invoice(Code) from A to F;
10          payment(Price) from F to A; }
11       or {
12          refuse(String) from F to R,A;
13       }
14     }
15 end}
16 State5={String receive_ticketStringFromA ():}
17     {choice at F {
18       ticket(String) from A to R;
19       invoice(Code) from A to F;
20       payment(Price) from F to A; }
21     or {
22       refuse(String) from F to R,A; }
23 end}
24 State0={void send_requestTravelToA(Travel):
25     {choice at S {
26       reply(250) from C; _250; }
27     or {
28       reply(250) from C; _250谴 dash;
29     }
30     }
31     }
32 State2={void send_checkPriceToF(Price):
33     {choice at S {
34       _220 from C; _220;
35     or {
36       _250 from C; ...
37     ...
38     }
39     }
40     }
41 State3={Choice1 receive_Choice1LabelFromF ():
42     {choice at C {
43       _220 from C; ...
44     rec X1 { choice at S {
45       _250Dash from C to S; continue X1; }
46     or {
47       _250 from C to S; ...
48     ...
49     ...
50     }
51     }
52     }
53 State4={Code receive_approveCodeFromF():
54     {choice at S {
55       _250Dash from C to S; continue X1; }
56     or {
57       _250 from C to S; ...
58     ...
59     ...
60     ...
61     }
62 }
63 State5={String receive_ticketStringFromA():
64     end}
65 State6={String receive_refuseTravelFromF():
66     end}}
67 }
68 }
```

The Scribble tool is used to check the above protocol definition for well-formedness and to derive a local version of the protocol for each role, according to the multiparty session types theory [25]. This is known as endpoint projection. Here we show the local protocol for Researcher, which describes only the messages involving that role. The self keyword indicates that R is the local endpoint.

```plaintext
1 local protocol
2   BuyTicket_R(self R, role A, role F){
3       request(Travel) to A;
4       quote(Price) from A to F;
5       check(Price) to F;
6       choice at F {
7           approve(Code) to F; A;
8           ticket(String) to A; R;
9           invoice(Code) to F;
10          payment(Price) from F to A; }
11       or {
12          refuse(String) from F to A; R;
13       }
14     }
15 end}
16 State3={Choice1 receive_Choice1LabelFromF ():
17     {choice at C {
18       _220 from C; ...
19     rec X1 { choice at S {
20       _250Dash from C to S; continue X1; }
21     or {
22       _250 from C to S; ...
23     ...
24     ...
25     ...
26     }
27     }
28     }
29 State2={void send_checkPriceToF(Price):
30     {choice at S {
31       _220 from C; _220;
32     or {
33       _250Dash from C to S; ...
34     ...
35     ...
36     ...
37     }
38     }
39     }
40 State1={String receive_quotePriceFromA ():}
41 State0={void send_requestTravelToA(Travel): State1}
CProtocol is a typestate, such as the following:

```java
enum Choice1 { _250DASH, _250; }
```

Typically the programmer would flesh out the skeletal implementation with extra logic that, for example, gets relevant input from the user or decides which choice to make when several are available, or

StMungo translates the local protocol (SMTP_C) into a typestate specification (CProtocol). In addition, it generates a skeletal implementation based on sockets, although other implementations are possible. Every interaction in the local protocol becomes a method call in the typestate specification, as we will see shortly. State definitions group methods into choices and impose sequencing.

Running the StMungo tool on SMTP_C produces the files CProtocol.protocol, CRole.java and CMain.java:

1. CProtocol.protocol, captures the interactions local to the SMTP_C role as a typestate specification.
2. CRole.java, is a class that implements CProtocol by communication over Java sockets. This is an API that can be used to implement the SMTP client endpoint.
3. CMain.java, is a skeletal implementation of the SMTP client endpoint. This runs as a Java process and provides a main() method that uses CRole to communicate with the other parties in the session, in this case the SMTP server.

The CProtocol generated by StMungo is defined in the following.

```java
typestate CProtocol {
  State0={String receive_220StringFromS():
    /*subject*/;
    State1}
  ... 
  State3=
    {Choice1 receive_Choice1LabelFromS():
     _250DASH: State4, _250: State5 }
  State4=
    {String receive_250dashStringFromS():
     State3}
  State5={String receive_250StringFromS():
     State6}
  ... 
  State27={void send_dataStringToS(String):
     State28} 
  ...
  State29={void send_SUBJECTToS(): State30,
    void sendDataLINEToS(): State31,
    void send_ATADToS():State32} ... }
```

We now describe the correspondence between the text-based commands in SMTP and the method calls in Mungo. Consider "SUBJECT: Hello World" which is an atomic command starting with the keyword SUBJECT and followed by the subject text. In our framework we use an intermediate layer to split this command into two separate method calls, as shown in lines 7-9 in CMain. The first, send_SUBJECTToS(), selects the command SUBJECT. The second, send_subjectStringToS("Hello World"), completes and sends the message "SUBJECT: Hello World". The intermediate layer is also used when receiving a command from the server, by splitting it into a choice and the corresponding text. Finally, CMain.java contains the main method where the CRole object is created and used to implement the client logic.

```java
public static void main(String[] args) {
  CRole currentC = new CRole();
  ... _Z3:
  do( ... 
    switch("/label to be sent") {
      case "SUBJECT":
        currentC.send_SUBJECTToS();
        String subject = // input subject;
        currentC.send_subjectStringToS("/subject");
        continue _Z3;
      case "DATA LINE":
      case "ATAD" :
        currentC.send_ATADToS();
        currentC.send_atADStringToS("/single dot");
        String _250msg =
          currentC.receive_250StringFromS();
        continue _Z1; } 
    } while(true); } 
```
customise $\text{CMain}$ by adding SSL connection code for authentication with the gmall server. Mungo is able to statically check $\text{CMain}$, or any code that uses a $\text{CRole}$ object, to ensure that methods of the protocol are called in a valid sequence and that all possible responses are handled. The programmer is not required to use the skeleton implementation of $\text{CMain}$, or even the $\text{CRole}$ API. It is possible to write new code that uses the API, or to use the typestate specification to guide the development of an alternative API, or to refactor the typestate specification itself.

5. A CORE CALCULUS FOR MUNGO

In this section we define the syntax and operational semantics of a core object-oriented calculus, based on [23] and used to formalise Mungo. Note that we only formalise the inference system and not the ability of Mungo to work with full Java, as this would require formalising a large subset of Java.

Syntax. The syntax of the calculus is given in Fig. 1. We use $\exists$ to denote a possibly empty set of elements that range over the subject meta-variable. A program is a set of type declarations $D$, each of which declares either a class or an enumerated type. A class declaration defines a class named $C$ with typestate specification $S$, fields $\vec{F}$ and methods $\vec{M}$. An enumeration declaration defines an enumerated type named $E$ with a non-empty set $\vec{l}$ of enum values. Our language has no support for inheritance or interfaces. We assume that a program has unique names for classes and enumerations, and a class has unique names for fields and methods. The formal treatment assumes as an implicit context a program $D$, which can be accessed by the following functions: given that $\text{class } C : S \{ \vec{F}; \vec{M} \} \in D$ we define $\text{fields}(C) = \vec{F}$, $\text{methods}(C) = \vec{M}$, $\text{typestate}(C) = S$; and $\text{enums}(E) = \vec{l}$ if enum $E \{ \vec{l} \} \in D$. A typestate definition $S$ specifies a state machine that has as actions the methods of a class. A typestate definition is either an internal choice $H$ of method signatures, or a recursive typestate $\mu X S$, which may contain the recursive typestate variable $X$. A method signature $H$ can have two forms, depending on whether the method transitions to a state $S$, or it is an external choice $E \text{ m}(T) : \{ l : S \}_{l \in E}$ with the method signature defining the transition to one of the possible states $(S)_{l \in E}$: in the latter case the return type of the method must be $E$. The empty or inactive typestate $[]$ can also be written end. Well-formedness conditions ensure that state $\mu X S$ is not well-formed and that all state definitions are closed. A type is either the name of a class or enumeration, void or boolean. A field declaration is a field name $f$ associated with a type $T$. A method declaration $T \text{ m}(T \cdot x) \{ e \}$ specifies a return type $T$, the name $m$ of the method, the type $T'$ of the parameter $x$, and the expression $e$ that comprises the method body. A path is either the atomic path $\text{this}$ denoting the current object (receiver), the composite path $r.f$ denoting the field $f$ of the object denoted by $r$, or a parameter $x$. At runtime paths are resolved to heap locations (see runtime syntax below). A constant is the special value $\text{null}$, which is assignable to any class type, a boolean or void literal, or an enum value $l$. A constant or a path is an expression.

The expression forms method call $r.m(e), field assignment r.f = e$, and object creation $r.f = \text{new } C$, have the target object of the invocation or assignment is restricted to a path $r$, rather than an arbitrary expression. The other expression forms include sequential composition $e; e'$, switch expressions, if...else expression, labelled expressions $\lambda : e$, and continue expressions that jump to the enclosing expression labelled by $\lambda$.

Figure 1: Top-level syntax

$D ::= \text{class } C : S \{ \vec{F}; \vec{M} \} | \text{ enum } E \{ \vec{l} \}$
$S ::= H | \mu X S | X$
$H ::= T m(T) : S | E m(T) : (S)_{l \in E}$
$T ::= C | E | \text{ bool } | \text{ void}$
$F ::= T f$
$M ::= T m(T \cdot x) \{ e \}$
$r ::= \text{ this } | r.f | x$
$e ::= l | tt | ff | \text{ null } | ^*$
$e ::= c | r | r.m(e) | r.f = e | e; e | r.f = \text{ new } C$
$| \lambda : e | \text{ continue } \lambda$
$| \text{ switch } (e) \{ e_i \}_{i \in E} | \text{ if } (e) \text{ e else } e$

Figure 2: Runtime syntax

Configurations and runtime syntax. Fig. 2 extends the source syntax with additional runtime constructs used by the operational semantics. A configuration $h, e$ is the pair of a heap $h$ and runtime expression $e$. The heap $h$ is defined as an object $C[f : o]$, where $C$ is the class of the object and $f : o$ are its fields; the contents $o$ of each field is either a constant $c$ or another object. The “heap” is a tree of objects, with neither cycles nor sharing, due to the linearity of object references enforced by the type system (see §6).

The expression $e$ in a configuration $h, e$ is a runtime expression in which every (compile-time) path of the form $\text{this}, r.f$ or $x$ has been replaced by a runtime path that refers to a heap value. A runtime path $r$ in a heap $h$ is either the atomic path $\text{root}$ denoting $h$ itself or the composite path $r'.f$ denoting the field $f$ of the object denoted by $r'$, where $r'$ is also a path in $h$. Runtime expressions also include the form $e@r$, which is an expression $e$ that has been tagged with $r$ to track the active receiver. A value $v$ is either a constant $c$ or runtime path $r$. Every runtime expression is either a value, or uniquely of the form $E[e]$, where $E$ is an evaluation context (an expression with a hole). As usual, the notation $E[e]$ denotes the plugging of the hole in $E$ with an expression $e$.

The operational semantics is annotated with labels $\ell$ that denote the creation of a new object $(r.f = \text{new } C)$, an enum value choice $(r:l)$, method call $(r.m(T))$, assigning a field $(r.f = v)$, the conditional label $(i.f)$, and the silent label $(r)$. The definition of states is extended to the set of enum values $\ell : S \}_{l \in E}$ and we define action labels $\ell$ for labels: internal choice $T m(T)$, external choice $E m(T) : l$, and for enum values $l$.

Labelled reduction semantics. We define heap access and update functions that are used by the reduction relation in Fig. 3: $h(\text{root}) = h$; $h(r.f) = o$ and $h(r.f \mapsto o') = h[r \mapsto C[f : o : o']]$ if $h(r) = C[f : o : f : o]$. The root object is accessed via $h(\text{root})$. The access of a field $h(r.f)$ is inductively defined on the access of $h(r)$. Similarly, we use the heap access function to update object fields as in $h[r.f \mapsto o]$, Fig. 3 defines the labelled reduction seman-
which must be an object
with

C

is deterministic. Assume a heap consisting of an instance of class
evaluation context. It is easy to show that the operational semantics
for every occurrence of
Rule
the
expression; rule
R-S
R-V

label when the value is fully evaluated and it is not an enum label,
receiver. The active receiver tag @
expression is tagged with @
r
C

for the formal parameter. In addition, the resulting runtime

τ

the

being assigned is a constant or an object path. Both forms return
object is constructed at a location within an already existing object

in
/init
C

c

and simply updates the heap to store

(\text{fields}(C) = \overline{Tf})

h, r.f = \text{new } C \rightarrow h[r.f \mapsto C[f:\text{init}(T)]]

R-NEW

\begin{align*}
R-VALUE \quad (v \neq l) & \quad h, v @ r \rightarrow h, v \\
R-ASGNR \quad (h' = h(r' \mapsto \text{null})) & \quad h, r.f = r' \mapsto v \rightarrow h[r.f \mapsto h(r')]
\end{align*}

R-SWITCH \quad (l' \in E) & \quad h, \text{switch } (l' @ r) \{e_1\}_{i \in E} \rightarrow h, e_v

R-LABEL \quad h, \lambda : e \rightarrow h, e[e/\text{continue } \lambda]

R-CTX \quad h, e \rightarrow h', e'

\text{Figure 3: Operational semantics}

tics; hereafter by “expression” we shall mean runtime expression,
and by “path” runtime path, unless otherwise indicated. Rule R-SEQ
discards the value \(v\) in a \(r\) label and proceeds with the evaluation of
\(e\). Rules R-TRUE and R-FALSE are the usual rules for the if \(\ldots\) else
expression and are annotated with label if. Rule R-New is labelled
with \(r.f\text{.new } C\) and overwrites the contents of the field \(r.f\) by a new
object \(C[f = \text{init}(T)]\) whose fields are all initialised to the value
init(\(T\)), where \(T\) is the type of the field, defined as: in\(it(C) = \text{null}\); in\(it(E) = E_{eq}\); in\(it(\text{bool}) = \text{ff};\) and in\(it(\text{void}) = \ast\), where for every
equenced type \(E\) we require there to be a distinguished element
\(E_{eq} \in \text{enums}(E)\). The result of R-New is the void value \ast. The

\begin{align*}
R-SEQ & \quad h, (v; e) \rightarrow h, e \\
R-TRUE & \quad h, \text{if}(tt) e_1\text{ else } e_2 \rightarrow h, e_t \\
R-FALSE & \quad h, \text{if}(ff) e_1\text{ else } e_2 \rightarrow h, e_f
\end{align*}

\text{corresponding value } \text{init}(T).\) Execution can then be initiated using
a top-level expression that substitutes path this with path root.

\section{6. Typestate Inference}

In this section we formalise a typestate inference system and
prove its safety properties. The system presented here infers a
\textit{typestate specification} for a class definition. The typestate imposes
an order on how the methods of the class should be called. To
this end, the system checks how each instance of the class statically
behaves. Finally, the inferred typestate is checked against the
declared typestate of the class. The inference system is the basis of
the implementation of Mungo (§ 2). Proving the soundness of the
inference system requires to prove that the trace of the execution of
a well-typed program is included in the trace of the inferred type
for that program. A sound inference system should be able to guarantee
the progress property requiring that a program either reduces or is
a value. The syntax of the inferred types, ranged over by \(U\), and the
typing context, ranged over by \(\Delta\), are defined below:

\begin{align*}
U & := C[S] | E | \text{bool} | \text{void} | \text{bot} \\
\Delta & := \emptyset | \Delta, r : U \rightarrow \Delta, \lambda : X
\end{align*}

The inferred types \(U\) differ from top-level types \(T\); every class type
\(C\) is refined with a typestate specification \(S\). There is a distinguished
bottom type \text{bot}. Typing context \(\Delta\) is a partial function from runtime
paths \(r\) to types \(U\), and expression labels \(\lambda\) to recursive type variables
\(X\). A type \(U\) that is not a class type is referred to as \text{constant type}.

The inference system uses a subtyping relation \(\leq_{\text{st}}\) and a binary
operator \(\text{join}(\cdot, \cdot)\).

\textbf{Definition 1.} (\(\leq_{\text{st}}, =_{\text{st}}, \text{join}\)) The following relations are defined
on typestates, inferred types and typing contexts.

\begin{itemize}
  \item The subtyping relation \(\leq_{\text{st}}\) is defined by the rules in Fig. 4.
  \item The equivalence relation is defined as \(\equiv_{\text{st}} = \leq_{\text{st}} \cap \leq_{\text{st}}^{-1}\).
  \item The join operator \(\text{join}(\cdot, \cdot)\) is defined by the rules in Fig. 5.
\end{itemize}

Subtyping on typestates is essentially a simulation relation and is
given in an algorithmic style. It coinductively constructs a set \(R\)
of pairs of typestates using rules S-Rec1 and S-Rec2. The algorithm
terminates either when end matches end (rule S-End) or when a
pair of typestates has been revisited (rule S-Terminate). Rule S-
Method checks for prefix matching. Rule S-Set requires covariance
A disjoint union of the rest of the methods is performed. Join on inferred types and typing contexts. Finally, we define a transition for the methods in common, the continuation typestates are joined.

**Definition 2.**
\[
\begin{align*}
\text{S-START} & \quad \emptyset \vdash S \triangleleft S' \\
\text{S-END} & \quad \mathcal{R} \vdash \text{end} \triangleleft \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{S-Rec} & \quad \mathcal{R} \cup \{(S, \mu X.S')\} \vdash S \triangleleft S' \leftarrow \mu X.S' \\
\end{align*}
\]

\[
\begin{align*}
\text{S-Enum} & \quad \forall I \in E. \mathcal{R} \vdash S_I \triangleleft S_I' \\
\text{S-Class} & \quad S \triangleleft S' \\
\text{S-Delta} & \quad \Delta \triangleleft \Delta' \quad U \triangleleft U' \\
\\text{S-End} & \quad \Delta, r : U \triangleleft \Delta, r : U'
\end{align*}
\]

**Fig. 4:** Subtyping relation (symmetric rule S-Rec2 omitted)

The first two rules state that a method prefixed typestate reduces to its continuation under a label denoting the prefix method signature itself. The next two rules state that reduction can occur in a set of typestates and under recursion, respectively. The last rule defines a reduction on a runtime typestate, as defined in Fig. 2. It states that a branching typestate reduces to one of its components by using the corresponding enumerated value.

### 6.1 Typestate Inference Rules

**Fig. 5:** Join operator (symmetric recursion rule omitted)

on subtyping with the empty set being treated as a special case. Rule S-Enum matches the external choice prefix. It requires subtyping on typestates for every value of the enumerated type. The subtyping relation generalises to inferred types and typing contexts. It is easy to show that $\triangleleft$ is a preorder.

The join operator is the least upper bound with respect to subtyping. It is used in typing rules that combine multiple execution paths, in order to compute a common final typestate. The most interesting case of join on typestates is the join of method signatures. For the methods in common, the continuation typestates are joined. A disjoint union of the rest of the methods is performed. Join on recursive typestates is done up to unfolding. The relation generalises to inferred types and typing contexts. Finally, we define a transition relation on typestates as follows.

**Definition 2.** (Transition on typestates) The transition relation $S \rightarrow S'$ is defined by the following rules:

\[
\begin{align*}
T m(T') : S & \rightarrow S \\
E m(T) : (S)_r \rightarrow S & \rightarrow S \\
H \in \bar{H} & \rightarrow S \\
\end{align*}
\]

The typestate inference rules for expressions are given in Fig. 6. We illustrate the most important rules using examples. The full typestate derivation of the example code can be found in [30]. The type inference is syntax-driven, meaning that at any point of the derivation there is only one rule that can be applied. Rules **Void**, **Bool**, **Enum** and **Null** type the constants with their corresponding types under any typing context without producing any effect on it, namely the left and right typing contexts are the same. Rules **Strengthen** and **WeakEn** allow arbitrary removal and addition, respectively, of inactive typestate assumptions.

**Typestate Linearity.** In the typestate inference system we adopt linearity in order to forbid aliasing. We use the following example to explain rules **Seq**, **PathR**, **PathC**, **AsgnR**, **AsgnC**, and **New** that require treatment of linearity. Consider the following code that uses the implementation of class `Stack` in Section §1:

\[
s = \texttt{new} \text{Stack}; \ k = s
\]

The code expression matches rule **Seq**. We assume

\[
\Delta_0 = s : \text{Stack[end]}, \ k : \text{Stack[S]}
\]
as an input typing context and $S \sqsubseteq_{inf} StackProtocol$. Rule $\text{Seq}$ requires an inference for the second expression before the first, because the output typing context of the second expression is the input typing context of the first. In order to type the second expression by $\text{AsgnR}$ we need to infer a typestate for $s$. The derivation is:

$$\text{AsgnR}$$

$\frac{U \neq C[S]}{\Delta \vdash e : U + \Delta'}$

where $\Delta_1 = s : Stack[end], k : Stack[end]$. The output typing context for $\text{ParnR}$ is

$$\Delta_2 = s : Stack[S], k : Stack[end]$$

meaning that $k$ has an inactive typestate before assignment. Rule $\text{ParnR}$ "guesses" a type for a path expression. However, the combination of $\text{ParnR}$ and $\text{AsgnR}$ is the key to this inference since it enforces a match on the type of $s$ in the output typing context $\Delta_2$ and the type of $k$ in the input typing context $\Delta_0$. For the first expression in (1) we use rule $\text{New}$. By assumption we simplify its premise: we have $S \sqsubseteq_{inf} StackProtocol$, meaning path $s$ is used according to the $StackProtocol$ typestate (this is shown in $\Delta_0$). Rule $\text{New}$ infers a type void for the first expression. Since it is not a class type it satisfies the premise of rule $\text{Snew}$ which requires the type of the first expression not to be a class type, so it can be discarded without violating linearity. It also requires that the first expression is not of type $\text{bot}$ to forbid dead code after a $\text{continue}$ $\Delta$ expression (see rule $\text{Continue}$). The type of the sequential expression is the type of the latter expression, void. We summarize the derivations described so far in the following:

$\text{Seq}$

$$\frac{\text{Snew StackProtocol}}{\Delta_3 = s : Stack[end], k : Stack[end]}$$

$\text{AsgnR}$

$$\frac{\Delta_3 \vdash s = \text{new Stack} : \text{void} + \Delta_2}{\Delta_2 \vdash k = \text{void} + \Delta_0}$$

To preserve linearity, $s$ and $k$ exchange their typestates before and after assignment, as expected. If the type of $s$ in $\Delta_0$ is not inactive, it means that path $s$ can be used after its assignment, thus violating linearity, as in the following code:

$$s = \text{new Stack}; k = s; s.push(s)$$

(3)

To conclude, in rules $\text{AsgnR}$ and $\text{New}$ the path $\text{this}$ is not assignable. In rule $\text{ParnR}$ the path $\text{this}$ is not inferable.

The other rules for paths and assignments are as follows. Rule $\text{ParnR}$ infers a constant type $U$ for a path $r$ and has no effect in the input typing context, if $r$ is mapped to $U$ in the input typing context. Rule $\text{AsgnR}$ follows the same line as $\text{AsgnR}$, the difference being the type of $e$ which is a constant type $U$ that is left unchanged in the input and output typing contexts.

Recursion and Choice. We now explain recursion and choice by using an example of a recursive loop. The example is used to explain rules $\text{LExpr}$, $\text{Continue}$, $\text{Switch}$, and $\text{If}$. Consider the following class StackUser that defines methods that use a Stack object:

```java
1 class StackUser:
2 {{Stack pushN( Stack):}
3 {Stack popAll( Stack): end}}
4 {{Stack pushN( Stack x) { x.push(2) ; x }]
5 Stack popAll( Stack x)
6 {loop: switch( x.isEmpty() )
```
{case EMPTY:x,
case NOTEMPTY:x.pop();
    continue loop})

and the input typing context:

\[ \Delta_0 = x : \text{Stack[end]}, \text{this} : \text{StackUser[end]} \]

The body of method popAll in line 4 is a labelled expression, and so rule LEStr applies. The premise requires an inference for the \text{switch} expression by using in input \Delta_0 augmented with the assumption \text{loop} : X, where X is fresh. Let \Delta_1 = \Delta_0, \text{loop} : X. LEStr closes all occurrences of X in the output typing context. For the \text{switch} expression rule Swtchr is used, which requires a typestate inference for all the switch branches. The input typing context for every branch is the same as the one for \text{switch}, namely \Delta_1. The inferred output contexts of the branches are then joined and used in input to infer a typestate for the method call expression in the condition of the \text{switch}. The condition should have an enumeration type that matches the type of the \text{switch} definition. Finally, the type of \text{switch} is the join of the types of its branches. For the \text{TRUE} branch we use rule PnR:

\[
P_{\text{PnR}} \quad \Delta_2 = x : \text{Stack}[S], \text{this} : \text{StackUser[end]}, \text{loop} : X
\]

\[ \Delta_2 \vdash x : \text{Stack}[S] + \Delta_1 \]

For the FALSE branch we first use rule S00 and then rule \text{continue} to infer the typestate of the \text{continue loop} expression. \text{continue loop} requires \text{loop} to be mapped to a recursive variable X in the input typing context. It then outputs a typing context where all paths mapped to a typestate are updated to the typestate X, as in:

\[ \text{continue} \quad \Delta_3 = x : \text{Stack}[X], \text{this} : \text{StackUser}[X] \]

\[ \Delta_1 \vdash \text{continue loop} : \text{bot} + \Delta_1 \]

The type of the \text{continue expression} is \text{bot}, since we want \text{join}(\cdot, \cdot) to be defined (cf. Fig. 5). To complete the typing of the FALSE branch, we apply rule \text{call} for \text{x.pop()} and conclude with rule S00. The output typing context is:

\[ \Delta_4 = x : \text{Stack}[[\text{int pop}() : X]], \text{this} : \text{StackUser}[X] \]

We join the output typing contexts \Delta_2 and \Delta_4 of the \text{TRUE} and FALSE branch, respectively and use the result as an input typing context for the method call \text{x.isEmpty()}, as stated by the premises of Swtchr. The output typing context of Swtchr is:

\[ \Delta_5 = x : \text{Stack}[[\text{Choice isEmpty()} : \text{join}(S, \text{int pop}() : X)],
\text{this} : \text{StackUser}[\text{join}(\text{end}, X)] \]

To complete the inference of \text{LEStr} we close the recursive variable X in \Delta_5 and obtain the output typing context for the labelled expression in lines 4-5, which is:

\[ \Delta_6 = x : \text{Stack}[[\mu X. \text{Choice isEmpty()} : \text{join}(S, \text{int pop}() : X)],
\text{this} : \text{StackUser}[\mu X. \text{join}(\text{end}, X)] \]

Notice the equivalence of the type \( \mu X. \text{join}(\text{end}, X) \), that appears in the mapping of path this, and the type end, meaning that rule \text{Eqv} can be applied.

Rule \text{for} types the conditional expression in a similar way as rule Swtchr. Both conditional branches are individually inferred and then joined to obtain the output typing context of the if ... else expression. We further require that the condition has type \text{bool}.

\textbf{Method Call.} Rule \text{call} records the method call trace of paths in a program, to respect the principle that the trace of the execution of an object follows its inferred typestate. It uses the function \text{initT}, defined by \( T \neq C \implies \text{initT}(T) = T \) and \( \text{initT}(C) = C[\text{end}] \).

Rule \text{call} requires typechecking the method body every time a method is called. This is a simplification for presentational purposes. It means that if an algorithm is directly extracted from the rules, it is unable to construct a type in the case of a recursive method call. However, the rules can be used to derive typings if suitable pre- and post-conditions are put into the derivation by hand. The implementation of Mungo's type inference system uses a more complex notion of partial typestate so that method bodies do not need to be checked at every call site; recursive methods are also supported. As an example of the rule \text{call}, consider the following code that uses class \text{StackUser}:

\[
s = c.\text{pushN}(s) \]

and the input typing context:

\[ \Delta_0 = s : \text{Stack}[S], c : \text{StackUser}[[\text{Stack popAll(Stack) : end}]] \]

By applying rule \text{AsoR} on the above assignment with input \Delta_0, the output typing context in the premise of the rule is:

\[ \Delta_1 = s : \text{Stack[end}],
\text{c} : \text{StackUser}[[\text{Stack popAll(Stack) : end}]] \]

At this point we can apply rule \text{call} on \text{c.pushN(s)} and have the following derivation:

\[
\text{Stack pushN(Stack x) \{ x.push(2): x \}
\text{c.methods(\text{StackUser})} \quad (1)
\Delta_2 = (x.\text{push}(2): x)[\text{this} : \text{Stack}[S] + \Delta_1 \quad (2)
\Delta = s : \text{Stack}[[\text{void push(int) : S}] + \Delta_2 \quad (3)
\text{Method Call.} \]

The premise of \text{call}, given in (1), performs a lookup in the methods of the class of the receiver, ListCons, to obtain the definition of method \text{prod(Stack)}. Next, in (2), the premise infers a typestate for the body of the method in which \text{c} has been substituted for the keyword this. Both the method call and its body use the same input typing context. The output typing context of the body of the method should contain a typestate assumption for the method parameter and the receiver, as follows:

\[
\Delta_2 = \Delta_1, x : \text{Stack}[[\text{void push(int) : S}]] \]

Then, in (3) \text{call} requires a typestate inference in order to match the typestate of the method parameter with the type of the method call argument. For this rule \text{PnR} is used where \Delta_3 also updates the type of the receiver:

\[
\Delta_3 = s : \text{Stack[end]},
\text{c} : \text{StackUser}[[\text{Stack pushN(Stack) : end}]]
\Delta = \text{Stack}[[\text{void push(int) : S}]],
\text{c} : \text{StackUser}[[\text{Stack pushN(Stack) : end}]] \quad (4)
\]

Rule \text{call} requires that the types of the receiver \text{c} in the input and output typing contexts for the body of the method are equivalent, according to the relation \( =_{\text{astr}} \). This is to respect the abstraction principle: the client would know how a method uses its receiver. For example, assume method \text{Stack pushN(Stack)} is defined as:

\[
\text{Stack pushN(} \text{Stack x) \{ x.push(2); x = \text{this}.popAll(x); x \}}
\]

If we infer a typestate for the body of \text{Stack pushN(} \text{Stack)} with input context \( \Delta_4 \), we get: an output typing context, \( \Delta' \), such that:

\[
\Delta'(c) = \text{StackUser}[[\text{Stack popAll(Stack) : end}]]
\]
Given that $\Delta_t(c) = \text{StackUser}([[\text{Stack popAll(Stack) : end}]])$, it is revealed that the body of Stack pushAll(Stack) calls method Stack popAll(Stack) on its receiver object, thus violating the abstraction principle.

### Classes and Programs
The rules for classes and programs are given in Fig. 7. They make use of inference rules for the fields of a class, which we explain first. The typestates of the fields of a class are inferred when method calls of that class take place. This procedure is described by the inference rules for typestates. Rule $\text{Set-St}$ requires the inference and join of the typestates of all branches in an internal choice. Rule $\text{Method-St}$ relies on the $\text{infer}(T)$ definition that maps a type $T$ to the corresponding inferred type $U$ as: $T \neq C \Rightarrow \text{infer}(T) = T$ and $\text{infer}(C) = C[S]$, for some $S$.

Rule $\text{Method-St}$ infers a method-prefixed typestate, where first it requires an inference of the continuation typestate, and then uses the output typing context to infer the method prefix; it infers a typestate for a method definition by first inferring a typestate for its body. The auxiliary function $\text{infer}(T)$ is used to check that the return and parameter types of the method match the types of the inferred ones. As in $\text{Call}$, a self-call should preserve the typestate of the receiver up to type equivalence. Rule $\text{Enum-St}$ is similar to rule $\text{Method-St}$. It requires the inference and join of the typestates of all the external choices and then infers the method prefix. Rule $\text{End-St}$ requires all fields of the class to finish in the inactive typestate. Rules $\text{Rec-St}$ and $\text{Var-St}$ are similar to rules $\text{LEXP}$ and $\text{CONTINUE}$, where they bind and use a recursive variable, respectively. Rule $\text{Class}$ initiates the inference of the typestate of the class. It states that a class declaration is well-typed if every field of the class has an inactive typestate and this is assumed in the typing context in the premise of $\text{Class}$. A program is well-typed if all of its classes are well-typed, as stated by rule $\text{Program}$. To illustrate the rules, we show a typestate inference for StackUser in [30].

In Fig. 8 we give the inference rules for runtime expressions. We show only the ones that are different with respect to the rules in Fig. 6. Rule $\text{Switch-ArR}$ is similar to $\text{Switch}$, the difference being the condition of the switch, which is evaluated to an active receiver rather than a method call. Rule $\text{ArR}$ infers a typestate for $e@c$, by first inferring a typestate for $e$. The other rules are used to type runtime configurations. Rule $\text{Heap}$ uses rule $\text{Object}$ to check whether a typing context is consistent with all the objects in the heap.

### 6.2 Properties of the Typestate Inference System
Progress and subject reduction require that the output typing context of an expression mimics the reductions of the expression itself. To this end, we define a labelled reduction relation on the typing context in Fig. 9 which use the same labels as the reductions on expressions. Rule $\text{Ty-If}$ states that $\Delta$ remains unchanged under a $r$-reduction. Rule $\text{Ty-IfNew}$ states that a path in $\Delta$ mapped to an inactive typestate reduces under $r,f$ new $C$ and its typestate is updated accordingly. Rules $\text{Ty-AsmC}$ and $\text{Ty-AsmC}$ label the reduction with an assignment of a path and a constant, respectively. The former reduction ensures linearity conditions when an assignment takes place. The latter leaves the typing context unchanged. Rule $\text{Ty-Call}$ performs a reduction of a method-prefixed typestate with the method prefix itself being the label. Similarly, rule $\text{Ty-Label}$ reduces with an enumerated value for paths that have a runtime switch typestate. The behaviour of the $\text{If}$ label is captured by rule $\text{Ty-If}$. In both the
 builds on earlier work \cite{15, 14, 17} to add binary inference to remove the need for typestate declarations on methods.

Mungo improves on Bica by using type of their type system as a language called Bica, which is not currently maintained and is unusable. Mungo lifts these restrictions by allowing the abstraction of multiparty session of a program is included in the declared typestate of the program.

The work in \cite{26} extends Session Java with runtime type inspection and asynchronous communication semantics to enable an event-driven framework based on binary session types. As a use case they implement a binary session-typed SMTP server that uses a reactive structure to handle multiple clients concurrently. In our work we implement an SMTP client by using StMungo, which automatically generates code from a global protocol. Extending Mungo with runtime typestate inspection would enable us to investigate event-driven programming with multiparty session types.

Capecci et al. \cite{9} proposed that a class defines sessions instead of methods. A session generalises a method to an extended session typed dialogue over a communication channel. As far as we know, this new paradigm has not yet been implemented.

The work in \cite{37} typechecks the operations of a library that implements multiparty session types using a restricted set of MPI \cite{31} primitives. In contrast, our framework typechecks Java statements and expressions, instead of higher-level operations. The work in \cite{36} uses Scribble to automatically generate MPI code based on user-defined kernels that produce and consume data. The generated code does not require typechecking. On the other hand, the StMungo translation can be used together with the Mungo typechecker to develop more flexible multiparty session type implementations.

**Monitoring based on Scribble definitions.** Neykova et al. \cite{35} have used Scribble protocol definitions to achieve dynamic monitoring in Python, by translating local protocols into finite state machines that intercept communication and check the validity of runtime messages. Subsequently, \cite{34} implements a session-based Actor framework that uses runtime monitoring to integrate multiparty session types. A hybrid approach has been used by Hu \cite{27} to analyse an SMTP client in Java. Hu’s SMTP API implements multiparty session types using a pattern in which each communication method returns the receiver object with a new type that determines which communication methods are available at the next step. If the pattern is used properly then standard Java typechecking can verify correctness of communication, but runtime monitoring is needed to check linearity constraints. In contrast, our analysis of SMTP is able to statically check all aspects of the protocol implementation.

The receiver-returning pattern is at the basis of functional programming with session types \cite{22} and has been used to achieve protocol checking in Idris \cite{29} and as a replacement for explicit typestate in Rust \cite{40}.

**Typestate.** There have been many efforts to add typestate to practical languages, since their introduction in \cite{43}. Vault \cite{12, 19} is an extension of C, and Fugue \cite{13} applies similar ideas to C#. Plural \cite{6} is based on Java and has been used to study access control systems \cite{5} and transactional memory \cite{4}, and to evaluate the effectiveness of typestate in Java APIs \cite{6}. In contrast Mungo follows Gay et al. which is inspired by session types; the possible sequences of method calls are explicitly defined, rather than being consequences of pre- and post-conditions. Like Plural, a typestate in Mungo can depend on the return value of a method call.

**Sing#** \cite{18} is an extension of C# which was used to implement Singularity, an operating system based on message-passing. It incorporates typestate-like contracts, which are a form of session type, to specify protocols. Bono et al. \cite{8} have formalised a core calculus based on Sing# and proved type safety.

Aldrich et al. \cite{2, 44} proposed a new paradigm of typestate-oriented programming, implemented in the Plaid language. Instead of class definitions, a program consists of state definitions containing methods that cause transitions to other states. Transitions are specified in a similar way to Plural’s pre- and post-conditions. Like
Another aim is to support generics and inheritance. Inheritance be-
tween state classes, states are organised into an inheritance hierarchy. The most
recent work [20, 45] uses gradual typing to integrate static and dy-
namic typestate checking. We focus on the object-oriented paradigm
in order to be able to apply our results to Java.

Bodden and Hendren [7] developed the Clara framework, which
combines static typestate analysis with runtime monitoring. The
monitoring is based on the tracematches approach [3], using regular
expressions to define allowed sequences of method calls. The static
analysis attempts to remove the need for runtime monitoring, but if
this is not possible, the runtime monitor is optimised. Mungo uses a
purely static analysis, and can allow the state after a method call to
depend on the method’s (enumerated type) result.

Typestate systems must control aliasing, otherwise method calls
via aliases can cause inconsistent state changes. Literature in-
cludes the “adaptation and focus” approach of Vault and Fugue, the
permission-based approaches of Plural and Plaid, and an expres-
sive fine-grained system by Militão et al. [32]. Also relevant is
recent work by Crafa and Padovani [11] which applies the chemical
approach to concurrent typestate oriented programming, allowing
objects to be accessed and modified concurrently by several pro-
cesses, each potentially changing only part of their state. We expect
that many of these systems can be applied to Mungo. However,
linear typing has not been a limiting factor for the applications
described in the present paper.

8. CONCLUSION AND FUTURE WORK

Concluding Remarks. We have presented two tools, Mungo and
StMungo, which extend the Java development process with support
for static typechecking of communication protocols. Mungo extends
Java with typestate definitions, which associate classes with state
machines defining permitted sequences of method calls. StMungo
uses the typestate feature to connect Java to Scribble, the latter being
a language used to specify communication protocols. In order to
illustrate the practicality and robustness of Mungo and StMungo, we
have implemented a substantial use case, an SMTP client, which we
were able to statically typecheck. We use this client to communicate
with the Gmail server. Finally, we have formalised the essential
features of Mungo by defining a typestate inference system for a
core object-oriented language. We proved safety and progress
properties (Theorem 1). These properties guarantee the coherence of
the typestate inference system with respect to the declared typestate
in a program (Corollary 1).

Future Work. The combination of Mungo and StMungo is effec-
tive for statically checking the correct implementation of communi-
cation protocols. We intend to extend Mungo to increase its power
for general-purpose programming with typestate. Our first aim is
to generalise the use of linear typing as a mechanism for the alias
control required by typestate systems. Candidates include the “adap-
tion and focus” technique of Vault and Fugue, the permission-based
approaches of Plural and Plaid, and the system by Militão et al. [32].
Another aim is to support generics and inheritance. Inheritance be-
tween typestate classes requires a subtyping relation between their
typestate specifications, based on standard definitions of subtyping
for session types [21]. Method calls on an object whose type is a
generic parameter must be typechecked against the typestate spec-
ification of the parameter’s upper bound. To extend typechecking
to exception handlers, we need to allow typestate specifications to
define the state transitions corresponding to exceptions, and check
that these transitions are consistent with the states of fields at the
point where an exception is thrown. Existing work on exceptions in
session types [10] provides inspiration, but doesn’t address the
complexities of Java’s exception mechanism. Using these Mungo
extensions with StMungo for more sophisticated protocol verifica-
tion will also require extensions to Scribble to support generic
protocols, inheritance between protocols, and more general handling
of exceptions.

Acknowledgements.
This research was supported by UK EPSRC grant EP/K034413/1
From Data Types to Session Types: A Basis for Concurrency and
Distribution. We thank Laura Voinea for her contribution to Mungo
and StMungo. We thank Garrett Morris, Raymond Hu and Nobuko
Yoshida for useful comments and discussion.

9. REFERENCES

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and
Zachary Sparks. Typestate-oriented programming. In
[3] Chris Allan, Pavel Avgustinov, Aske Simon Christensen,
Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege
de Moor, Damien Sereni, Ganesh Sittampalam, and Julian
Tibble. Adding trace matching with free variables to AspectJ.
Verifying correct usage of atomic blocks and typestate. In
checking of aliased objects. In OOPSLA ’07, pages 301–320.
Practical API protocol checking with access permissions. In
ECOOP ’09, volume 5635 of Springer LNCS, pages 195–219,
2009.
hybrid typestate analysis. Software Tools for Technology
copyless message passing. In ESOP ’11, volume 6602 of
Springer LNCS, pages 57–76, 2011.
[9] Sara Capecchi, Mario Coppo, Mariangiola
Dezani-Ciancaglini, Sophia Drossopoulou, and Elena
Giachino. Amalgamating sessions and methods in
object-oriented languages with generics. Theoret. Comp. Sci.,
Structured interactional exceptions in session types. In
CONCUR’08, volume 5201 of Springer LNCS, pages
typestate-oriented programming. In OOPSLA ’15, pages
In ECOOP ’04, volume 3086 of Springer LNCS, pages
[14] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou,
Dimitris Mostrous, and Nobuko Yoshida. Objects and session
[15] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia
Drossopoulou, and Nobuko Yoshida. Bounded session types
APPENDIX

A. PROGRESS AND SUBJECT REDUCTION

A.1 Auxiliary Results

In the following we use $\Delta[v : U]$ to denote $\Delta$ where the value $v$ is updated to the type $U$.

Lemma 1. (Typability of Heap Update) Let $h$ be a heap and $r$ a runtime path such that $\Delta \vdash h$ and $r : U \in \Delta$.

1. If $U \neq C[S]$ and $\Delta \vdash o : U + \Delta$, then $\Delta \vdash h[r \mapsto o]$.
2. If $U = C[S]$ and $\Delta' \vdash o : C[S'] + \Delta'$, then $\Delta' \vdash h[r \mapsto o]$, where $\Delta' = \Delta[r : C[S']]$.

Proof. Both cases follow by using rule $\texttt{Heap}$ and for 1. typing rules for constants are used, and for 2. rule $\texttt{Object}$ is used. \hfill \Box

Lemma 2. (Replacement) If

- $d$ is a derivation for $\Delta \vdash E[e] : U + \Delta'$,
- $d'$ is a subderivation of $d$ concluding $\Delta \vdash e : U_\varepsilon + \Delta_x$,
- the position of $d'$ in $d$ corresponds to the position of the hole in $E$,
- $\Delta' \vdash e' : U_\varepsilon + \Delta_\varepsilon$, such that $U_\varepsilon \trianglelefteq U_x$,

then $\Delta' \vdash E[e'] : U' + \Delta''$ such that $U' \trianglelefteq U$.

Proof. Follows [23], by replacing the derivation $d'$ in $d$ with the derivation for $\Delta' \vdash e' : U_\varepsilon + \Delta_\varepsilon$. \hfill \Box

Lemma 3. (Substitution) Assume $\Delta_1 = \Delta$, $x : X$ and $\Delta_2 \vdash e' : U + \Delta'$.

Then, $\Delta \vdash e[e'/\text{continue}\lambda] : U + \Delta'$ with

$$\Delta = \{ r : C[S[S'/X]] \mid r : C[S] \in \Delta_1 \text{ and } r : C[S'] \in \Delta_2 \}$$

$$\cup \{ r : U' \mid r : U' \in (\Delta_1 \cup \Delta_2)(U' = C[S']) \}$$

$$\cup \Delta_1 \Delta_2 \cup \Delta_2 \Delta_1$$

Proof. The proof proceeds by induction on the last typing rule used for the assumed judgement. The second case uses Lemma 2. \hfill \Box

Lemma 4. (Subtyping and join) The following relate subtyping and join on inferred types $U$ and typing contexts $\Delta$.

- Let $U, U'$ be inferred types such that $\text{join}(U, U')$ is defined. Then, $U \trianglelefteq \text{join}(U, U')$ and $U' \trianglelefteq \text{join}(U, U')$.
- Let $\Delta, \Delta'$ such that $\text{join}(\Delta, \Delta')$ is defined. Then, $\Delta \trianglelefteq \text{join}(\Delta, \Delta')$ and $\Delta' \trianglelefteq \text{join}(\Delta, \Delta')$.

Proof. The proof follows immediately by combining the definition of subtyping in Fig. 4 and the definition of join Fig. 5. \hfill \Box

Lemma 5. (Typability of Subterms) If $d$ is a derivation for $\Delta \vdash E[e] : U + \Delta''$ then there exist $\Delta'$ and $U'$ such that $d$ has a subderivation $d'$ concluding $\Delta \vdash e : U' + \Delta'$ and the position of $d'$ in $d$ corresponds to the position of the hole in $E$.

Proof. The proof proceeds by induction on the structure of context $E$.

- $E = [];$ follows trivially by assumption.
- $E = r.m(E')$: by assumption $\Delta \vdash r.m(E'[e]) : U + \Delta''$. By inversion on rule $\texttt{Call}$ $\Delta \vdash E'[e] : U + \Delta''$, $r : C[T \text{ m}(T') : S]$, where the typing context $\Delta''$ and the type of $r$ are inferred by the premise of $\texttt{Call}$. We conclude by induction hypothesis on $E'$.
- $E = r.f = E'$: by assumption $\Delta \vdash r.f = E'[e] : U + \Delta''$. There are two rules that can be applied for assignment, rule $\texttt{AsgnC}$ and rule $\texttt{AsgnR}$. By inversion on the former we obtain $\Delta \vdash E'[e] : U + \Delta''$, $r : U$; by inversion on the latter we obtain $\Delta \vdash E'[e] : C[S] + \Delta''$, $r : C[\text{end}]$, where the typing context $\Delta''$ and the type for $r$ are inferred by the premise of the rule. We conclude by induction hypothesis on $E'$.
- $E = E'; e'$: by assumption $\Delta \vdash E'[e'] : U' + \Delta'''$. By inversion on rule $\texttt{Seq}$ $\Delta \vdash E'[e'] : U'' + \Delta'''$, where the typing context $\Delta'''$ and the type $U''$ are inferred by the premise of the rule. We conclude by induction hypothesis on $E'$. \hfill \Box

The rest of the cases follow the same idea as the above.

\hfill \Box
A.2 Progress and Subject Reduction

Proof of Theorem 1: Let $\overline{D}$ be a set of declarations such that $\vdash \overline{D}$. In a context parametrized by $\overline{D}$, let $e$ be a run time expression and suppose $\Delta \vdash h, e : U + \Delta''$.

Then, either $e$ is a value, or there exist $\ell, h'$ and $e'$ such that $h, e \overset{\ell}{\longrightarrow} h', e'$, and there exist $\Delta'$ and $U'$ such that $\Delta \overset{\ell}{\longrightarrow} \Delta'$ and $\Delta' \vdash h', e' : U' + \Delta''$ and $U' \triangleleft_{\text{stat}} U$.

Proof. The proof proceeds by induction on the structure of the expression $e$ with respect to contexts. We present first the inductive case. Let $e = e_{[1]}$ where $e_1$ is not a value and $\mathcal{E} \neq \emptyset$. By assumption and inversion on rule $\text{Cons}$ we have $\Delta \vdash \mathcal{E}[e_1] : U + \Delta''$. By Lemma 5 there exist $\Delta_1$ and $U_1$ such that $\Delta \vdash e_1 : U_1 + \Delta_1$. By induction hypothesis there exist $h', \ell$ such that $h, e_1 \overset{\ell}{\longrightarrow} h', e_2$. By induction hypothesis we also have $\Delta \overset{\ell}{\longrightarrow} \Delta'$ and $\Delta' \vdash h, e_2 : U_2 + \Delta_1$, which by inversion on $\text{Cons}$ means that $\Delta' \vdash h$ and $\Delta' \vdash e_2 : U_2 + \Delta_1$, where $U_2 \triangleleft_{\text{stat}} U_1$. By rule $\text{RCrX}$ we have $h, \mathcal{E}[e_1] \overset{\ell}{\longrightarrow} h', \mathcal{E}[e_2]$. By Lemma 2 we obtain $\Delta' \vdash \mathcal{E}[e_2] : U' + \Delta''$ with $U' \triangleleft_{\text{stat}} U$. We conclude by rule $\text{Cons}$.

The base cases when $e$ is of the form $\mathcal{E}[\ell]$ with $\mathcal{E}$ elementary, not being of the form $\mathcal{E}[\ell']$ with $\mathcal{E}' \neq \emptyset$, and not of the form $\mathcal{E}[e_1]$ are in the following. If $e$ is a value, then there is nothing to prove. If $e$ is not a value, by the operational semantics rules, we have the following cases for $e$ with respect to contexts.

- $e = v ; e'$. By hypothesis and reduction rule $\text{R-Seq}$

$$h, (v; e') \overset{\ell}{\longrightarrow} h, e'$$

By hypothesis and typing rule $\text{Conj}$ $\Delta \vdash h$ and $\Delta \vdash v ; e' : U + \Delta''$. By inversion and typing rule $\text{Seq}$

$$\frac{\Delta \vdash v : U' + \Delta_1 \quad U' \neq C[S]}{\Delta \vdash e' : U + \Delta''}$$

By reduction rule $\text{TvLo}$ $\Delta \overset{\ell}{\longrightarrow} \Delta$. Since $U' \neq C[S]$ and value $v$ is of type $U'$ it means $v$ is some constant $c$. Hence, the judgement $\Delta \vdash v : U' + \Delta_1$ is obtained by applying one of the following typing rules: $\text{Void}$, $\text{Enum}$, or $\text{Bool}$. By inversion this implies $\Delta_1 = \Delta$. Then, by rewriting the premise of the typing rule for $e'$ we have $\Delta \vdash e' : U + \Delta''$. We conclude by rule $\text{Conj}$.

- $e = (r . f = \text{new } C)$. By hypothesis and typing rule $\text{R-New}$

$$h, r, f = \text{new } C \overset{\text{new } C}{\longrightarrow} h[ r, f \mapsto C[ f : \overline{\text{init}}(T)]]$$

such that $\text{fields}(C) = \overline{f}$. By hypothesis and typing rule $\text{Conj}$ $\Delta \vdash h$ and $\Delta \vdash r, f = \text{new } C : U + \Delta''$. By inversion and typing rule $\text{New}$ we have:

$$\frac{\text{New}}{S \triangleleft_{\text{stat}} \text{typestate}(C) \quad \forall r, f': C'[S'] \in \Delta_1 \implies S' = \text{end}}{\Delta_1, r, f : C[\text{end}] \vdash r, f = \text{new } C : \text{void } + \Delta_1, r, f : C[S]}$$

where $\Delta = \Delta_1, r, f : C[\text{end}], U = \text{void}$ and $\Delta'' = \Delta_1, r, f : C[S]$. By rule $\text{TyNew}$ we have

$$\Delta_1, r, f : C[\text{end}] \overset{\text{new } C}{\longrightarrow} \Delta_1, r, f : C[S] = \Delta''$$

such that $S \triangleleft_{\text{stat}} \text{typestate}(C)$ and for all fields $r, f', f : C'[S'] \in \Delta_1$ and state $S' = \text{end}$. By applying typing rule $\text{Void}$ we have:

$$\Delta'' \vdash * : \text{void } + \Delta''$$

It remains to prove $\Delta'' \vdash h'$ namely,

$$\Delta_1, r, f : C[S] \vdash h[r, f \mapsto C[ f : \overline{\text{init}}(T)]]$$

Recall that, by hypothesis $\Delta_1, r, f : C[\text{end}] \vdash h$. Now we want to type the updated reference $r, f$ to $C[ f : \overline{\text{init}}(T)]$. By rule $\text{Object}$ and an empty set of labels $\overline{\ell}$

$$\text{typestate}(C) = S$$

$$\Delta_1, r, f : C[S] \vdash C[ f : \overline{\text{init}}(T)] : C[S] + \Delta_1, r, f : C[S]$$

We conclude by Lemma 1.

- $e = (r . f = c)$. By hypothesis and by rule $\text{R-Asoc}$

$$h, r, f = c \overset{\tau}{\longrightarrow} h[r, f \mapsto c], *$$

By hypothesis and by rule $\text{Conj}$ $\Delta \vdash h$ and $\Delta \vdash r, f = c : U + \Delta''$. By inversion and typing rule $\text{Asoc}$ we have

$$\frac{\text{Asoc}}{U' \neq C[S] \quad \Delta \vdash c : U' + \Delta'', r, f : U'}{\Delta \vdash r, f = c : \text{void } + \Delta', r, f : U'}$$

where $U = \text{void}$ and $\Delta' = \Delta', r, f : U'$. Since the value assigned to $r, f$ is a constant $c$ the judgement of the premise $\Delta \vdash c : U' + \Delta', r, f : U'$ must have been obtained by one of the following typing rules: $\text{Void}$, $\text{Enum}$, or $\text{Bool}$. This implies that $\Delta = \Delta', r, f : U'$. By
We need to prove that $\Delta \vdash h[r.f \mapsto c] \cdot \triangleright : \text{void} + \Delta', r.f : U'$. By rule Vom, $\Delta \vdash \cdot : \text{void} + \Delta', r.f : U'$. Recall that, by hypothesis and rule Heap and inversion we have

$$\begin{align*}
\text{Heap} \\
\text{h}(r.f) = c' \\
\Delta', r.f : U' + c' : U' + \Delta', r.f : U' \\
\Delta', r.f : U' + h
\end{align*}$$

By updating the heap to $h(r.f \mapsto c)$, using the typing judgement for $c$ in the premise of $\text{AssocC}$ and Lemma 1 we derive $\Delta', r.f : U' + h$. By rule Heap and inversion we have $\Delta \vdash h[r.f \mapsto c]$.

We conclude by rule $\text{Config}$.

- $e = (r.f = r')$. By hypothesis and by rule R-AssoR

$$h, r.f = h' \mapsto h[r.f \mapsto h(r')] \cdot \triangleright$$

where $h' = h[r' \mapsto \text{nul}]$. By hypothesis and by rule $\text{Config}$ $\Delta \vdash h$ and $\Delta \vdash r.f = r' : U + \Delta''$. By inversion and typing rule $\text{AssoR}$

$$\Delta \vdash r' : C[S] + \Delta_1, r.f : C[\text{end}]$$

where $U = \text{void}$ and $\Delta'' = \Delta_1, r.f : C[S]$ and for readability let $\Delta_2 = \Delta_1, r.f : C[\text{end}]$. Let $r' \neq r.f$. Since $r'$ is a path typed by $C[\text{end}]$, the premise of the above derivation is obtained by applying $\text{PanR}$. This implies that contexts $\Delta$ and $\Delta_2$ differ only in the typing of $r'$. By inversion, $\Delta(r.f) = \Delta_2(r.f) = C[\text{end}]$ and $\Delta(r') = C[S]$ and $\Delta_2(r') = C[\text{end}]$. By rule $\text{Ty-AssoR}$

$$\Delta_3, r' : C[S], r.f : C[\text{end}] \mapsto h_3, r.f \mapsto h(r')$$

where $\Delta = \Delta_3, r' : C[S], r.f : C[\text{end}]$ and $\Delta' = \Delta_3, r.f : C[S], r' : C[\text{end}]$. Since $\Delta'' = \Delta'$, by applying rule Vom we conclude $\Delta \vdash \cdot : \text{void} + \Delta''$. It remains to prove that

$$\Delta_3, r.f : C[S], r' : C[\text{end}] \vdash h[r.f \mapsto h(r')]$$

where $h' = h[r' \mapsto \text{nul}]$. Recall that

$$\Delta_3, r' : C[S], r.f : C[\text{end}] \vdash h$$

The result follows immediately by applying twice Lemma 1 for $r.f$ and $r'$. We conclude by $\text{Config}$. Let $r' = r.f$. By rewriting $\text{AssoR}$ with $r.f$ instead of $r'$ we notice that the derivation holds if $S = \text{end}$. Then the proof proceeds trivially.

- $e = r.m(v)$. By hypothesis and by rule R-Call

$$h, r.m(v) \mapsto h, e[v/x][r/\text{this}] \cdot \triangleright$$

such that $h(r) = C[\text{f} \mapsto \text{a}]$ and $T(m(T') \cdot x) \cdot \{e\} \in \text{methods}(C)$. By hypothesis and by rule $\text{Config}$ $\Delta \vdash h$ and $\Delta \vdash r.m(v) : U + \Delta''$. By inversion and typing rule Call

$$T(m(T') \cdot x) \cdot \{e\} \in \text{methods}(C) \quad S' \equiv_{\text{set}} S$$

$$\Delta_2, r : C[S'], x : U' + e[r/\text{this}] : U + \Delta_2, r : C[S]$$

$$\Delta \vdash v : U' + \Delta_2, r : C[[T(m(T') \cdot S)]]$$

$$\Delta \vdash r.m(v) : U + \Delta_2, r : C[S]$$

where $\Delta'' = \Delta_1, r : C[S]$ and for readability let $\Delta''' = \Delta_2, r : C[[T(m(T') \cdot S)]]$. Notice that $v \neq r$, otherwise the method call $r.m(v)$ would not be well-typed. Then, $\Delta(r) = \Delta''(r) = C[[T(m(T') \cdot S)]]$. Let $\Delta = \Delta_3, r : C[[T(m(T') \cdot S)]]$ By $\text{Ty-Call}$

$$\Delta_3, r : C[[T(m(T') \cdot S)]] \mapsto h_3, r \mapsto h$$

We need to prove that $\Delta_3, r : C[S] \vdash h$ and also $\Delta_3, r : C[S] \vdash e[v/x][r/\text{this}] : U + \Delta''$. By $\text{Arr}$ it suffices to show $\Delta_3, r : C[S] \vdash e[v/x][r/\text{this}] : U + \Delta''$. By the premise of Call,

$$\Delta_2, r : C[S'], x : U' + e[r/\text{this}] : U + \Delta_1, r : C[S]$$

$$\Delta_3, r : C[[T(m(T') \cdot S)]] \vdash v : U' + \Delta_2, r : C[[T(m(T') \cdot S)]]$$

we can notice that $\Delta_2$ and $\Delta_3$ are such that either $\Delta_2 = \Delta_3$ with $\Delta_2(v) = U'$ or $\Delta_2(v) = U''$ and $\Delta_3 = \Delta_2(v : U'/v : U'')$ By Lemma 3 we have

$$\Delta_3, r : C[S'] \vdash e[v/x][r/\text{this}] : U + \Delta_1, r : C[S]$$

Since $S \equiv_{\text{set}} S'$, we conclude by rule $\text{Eqv}$. Recall that $\Delta_3, r : C[[T(m(T') \cdot S)]] \vdash h$. By Heap we have

$$h(r) = C[\text{f} : \text{a}] \quad \Delta \vdash C[\text{f} : \text{a}] : C[[T(m(T') \cdot S)]] + \Delta$$

$$\Delta_3, r : C[[T(m(T') \cdot S)]] \vdash h$$
We conclude by rules and by rule Heap. We conclude by rule Config.

\[ e = v@r. \text{ By hypothesis and by rule R-Value} \]
\[ h, v@r \longrightarrow h, v \]

for \( v \neq l \). By hypothesis and by rule Config \( \Delta \vdash h \) and \( \Delta \vdash v@r : U + \Delta'' \). By inversion and typing rule ArR

\[ \Delta \vdash v : U + \Delta'' \]

By reduction rule Ty-Id \( \Delta \longrightarrow \Delta \). The thesis follows trivially.

\[ e = \text{switch}(l@r) \{ e_i \}_{i \in E}. \text{ By hypothesis and by rule R-Switch} \]
\[ h, \text{switch}(l@r) \{ e_i \}_{i \in E} \longrightarrow^{(\Delta)} h, e_r \]

for some \( l' \in E \). By hypothesis and by rule Config \( \Delta \vdash h \) and \( \Delta \vdash \text{switch}(l@r) \{ e_i \}_{i \in E} : U + \Delta'' \). By inversion and typing rule Switch-ArR

\[ \forall l \in E \quad \Delta_i, r : C[S_i] \vdash e_i : U_i + \Delta'' \quad \Delta \vdash l' : E + \Delta_i, r : C[(l : S_i)_{i \in E}] \quad \Delta_i = \bigcup_{i \in E} \Delta_i \]

we have that \( \Delta = \Delta_i, r : C[(l : S_i)_{i \in E}] \). By TV LABEL

\[ \Delta_i, r : C[(S_i)_{i \in E}] \longrightarrow^{(l')} \Delta_r, r : C[S_r] \]

where \( l' \in E \) and \( \Delta' \equiv_{\text{abs}} \Delta \). By Lemma 4 we have that \( U_r \equiv_{\text{abs}} \text{join}(U_{i \in E}) \). The judgement \( \Delta_r, r : C[S_r] \vdash e_r : U_r + \Delta'' \) holds by the premise of Switch for \( l' \in E \). We need to prove that \( \Delta_r, r : C[S_r] \vdash h \). Recall that, by hypothesis \( \Delta \vdash h \). By Heap and Object it means that there exist \( \bar{S} \), such that typestate(C) = \( \bar{S} \) and \( \bar{S} \longrightarrow (l : S)_{i \in E} \) and

\[ \Delta \vdash \text{typestate}(C) = \bar{S} \]

By Definition 2, we have \( (l : S)_{i \in E} \bar{r} S_r \). By applying rule Object on this reduction, we have

\[ \Delta_r, r : C[S_r] \vdash \text{typestate}(C) + \Delta_r, r : C[S_r] \]

We conclude by rules Heap and Config.

\[ e = \text{if}(\text{tt}) e_1 \text{ else } e_2. \text{ The case for } \text{if}(\text{ff}) e_1 \text{ else } e_2 \text{ is completely analogous. By hypothesis and by rule R-True} \]
\[ h, \text{if}(tt) e_1 \text{ else } e_2 \longrightarrow^{(if)} h, e_1 \]

By hypothesis and by rule Config \( \Delta \vdash h \) and \( \Delta \vdash \text{if}(\text{tt}) e_1 \text{ else } e_2 : U + \Delta'' \). By inversion and typing rule If

\[ \Delta_1 + e_1 : U_1 + \Delta'' \quad \Delta_2 + e_2 : U_2 + \Delta'' \]
\[ \Delta_1 = \text{join}(\Delta_1, \Delta_2) \quad \Delta \vdash \text{if}(\text{tt}) e_1 \text{ else } e_2 : \text{bool} + \Delta_1 \]

where \( U = \text{join}(U_1, U_2) \). Rule Bool implies that \( \Delta = \Delta_3 \). By TV-If \( \Delta \longrightarrow \Delta' \) and \( \Delta' \equiv_{\text{abs}} \Delta \). By Lemma 4 we have that \( U_1 \equiv_{\text{abs}} \text{join}(U_1, U_2) \). Then, \( \Delta_1 + e_1 : U_1 + \Delta'' \) follows directly by the premise of If and by letting \( \Delta' = \Delta_1 \), since \( \Delta_1 \equiv_{\text{abs}} \text{join}(\Delta_1, \Delta_2) = \Delta \). It remains to prove that \( \Delta_1 \vdash h \). Since \( \Delta_1 \equiv_{\text{abs}} \Delta \), the thesis follows trivially by applying Heap.

\[ e = (\lambda : e'). \text{ By hypothesis and by rule R-LABEL} \]
\[ h, (\lambda : e') \longrightarrow h, e'(\lambda : e'/\text{continue} \lambda) \]
By hypothesis and by rule $\text{Confg} \vdash h$ and $\Delta \vdash \lambda : e' : U + \Delta'$. By inversion and rule $\text{LExp}$ we get

$$\Delta'' \vdash e' : U + \Delta', \lambda : X$$

$$\Delta = \{ r : C[X,S] \mid r : C[S] \in \Delta'' \} \cup \{ r : U' \mid r : U' \in \Delta'' \text{ and } U' \neq C[S]\}$$

$$\Delta \vdash \lambda : e' : U + \Delta'$$

By rule $\text{Tv-to} \Delta \xrightarrow{\lambda} \Delta$. Since $\Delta \vdash h$, it remains to prove that $\Delta \vdash e' [\lambda : e' / \text{continue } \lambda] : U + \Delta'$. From the second case of the Substitution Lemma 3 we get:

$$\Delta'' \vdash e' [\lambda : e' / \text{continue } \lambda] : U + \Delta'$$

with

$$\Delta'' = \{ r : C[X,S] \mid r : C[S] \in \Delta'' \} \cup \{ r : U' \mid r : U' \in \Delta'' \text{ and } U' \neq C[S]\}$$

From the definition of $\Delta''$ we can obtain that $\Delta'' = \Delta$ as required. We conclude by rule $\text{Confg}$. 

\[ \square \]

## B. STMUNGO FOR MULTIPARTY SESSION TYPES

In this section we illustrate StMungo on a multiparty protocol that models the process of booking flights through a university travel agent.

There are three participants involved: Researcher (abbreviated $R$), who intends to travel; Agent ($A$), who is able to make travel reservations; and Finance ($F$), who approves expenditure from the budget. In the Scribble language, we first define the global protocol among three roles, which are abstract representations of the participants. The protocol consists of sequences of interactions. Every message (e.g. request) can be associated with a payload type (e.g. Travel), a sender, and one or more receivers. Typically payload types are structured data types defined separately from the protocol specification.

In the following global protocol, after the quote and the check message requesting authorisation for a trip, Finance can choose to approve or refuse the request:

```java
global protocol BuyTicket(role R, role A, role F){
    request(Travel) from R to A;
    quote(Price) from A to R;
    check(Price) from R to F;
    choice at F {
        approve(Code) from F to R,A;
        ticket(String) from A to R;
        invoice(Code) from A to F;
        payment(Price) from F to A;
        } or {
        refuse(String) from F to R,A; }
}
```

The Scribble toolchain can be used to check the protocol definition for well-formedness and to derive a local version of the protocol for each role, according to the theory of multiparty session types [25]. This is known as endpoint projection. Here we show the projection for Researcher, which describes only the messages involving that role. The $\text{self}$ keyword indicates that $R$ is the local endpoint.

```java
local protocol BuyTicket_R(self R, role A, role F){
    request(Travel) to A;
    quote(Price) from A;
    check(Price) to F;
    choice at F {
        approve(Code) from F to R;
        ticket(String) from R;
        } or {
        refuse(String) from F; }
}
```

Notice that the exchange of invoice and payment between Agent and Finance is not included. Similarly, the local projection for Agent omits the check message and the local projection for Finance omits the request, quote and ticket messages.

For the $R$ role, StMungo converts the $\text{BuyTicket}_R$ local projection into the following .mungo files:

1. $\text{RProtocol}$, capturing the interactions local to the $R$ role as a typestate specification.
2. $\text{RRole}$, a class that implements $\text{RProtocol}$ by communication over Java sockets. This is an API that can be used to implement the Researcher endpoint.
3. $\text{RMain}$, a skeletal implementation of the Researcher endpoint. This runs as a Java process, and provides a $\text{main()}$ method which uses $\text{RRole}$ to communicate with the other parties in the session.

The $\text{RProtocol}$ definition generated by StMungo is as follows:

```java
typestate RProtocol {
    State0 = {
        send_requestTravelToA(Travel): Statel }
    Statel = {
```
Price receive_quotePriceFromA(): State2 }  
State2 = {  
void send_checkPriceToF(Price): State3 }  
State3 = {  
Choice1 receive_Choice1LabelFromF():<APPROVE: State4, REFUSE: State6> }  
State4 = {  
Code receive_approveCodeFromF(): State5 }  
State5 = {  
String receive_ticketStringFromA(): end }  
State6 = {  
String receive_refuseTravelFromF(): end }  

class RRole typestate RProtocol { /* Constructor and method definitions. */ }  
The RRole class provides an implementation of RProtocol based on Java sockets. When instantiated, it connects to the other role objects in the session (ARole and FRole); we omit the details here.  
Finally, RMain provides skeletal implementation of the Researcher endpoint, using the RRole class to communicate with the other roles in the system:  

```java
public static void main(String[] args) {
    RRole r = new RRole();
    Travel t = // input travel;
    r.send_requestTravelToA(t);
    Price p = r.receive_quotePriceFromA();
    r.send_checkPriceToF(p);
    switch(r.receive_Choice1LabelFromF().getEnum()) {
      case APPROVE:
        Code c = r.receive_approveCodeFromF();
        println(r.receive_ticketStringFromA());
        break;
      case REFUSE:
        println(r.receive_refuseStringFromF());
        break;
    }
}
```

As we already stated for SMTP, typically the programmer would flesh out the skeletal implementation with extra business logic. Mungo is able to statically check RMain, or any client of the RRole class, to ensure that methods of the protocol are called in a valid sequence and that all possible responses are handled.

C. TYPE INFERENCE EXAMPLES

C.1 Typestate Linearity

Consider the following code that uses the implementation of class Stack in section § 1:

```java
s = new Stack;
```

Also assume input typing context $\Delta_0 = \{s: Stack[end], k: Stack[S]\}$. The inference tree for the above code is:

$$
\text{PamR}
\begin{array}{c}
\Delta_1 = s: \text{Stack}[S], k: \text{Stack[end]} \\
\Delta_2 + s: \text{Stack}[S] + \Delta_1 \\
\Delta_1 = s: \text{Stack[end]}, k: \text{Stack[end]} \\
\Delta_2 + k = s: \text{void} + \Delta_0 \\
\Delta_3 + s = \text{new Stack}; k = s: \text{void} + \Delta_0
\end{array}
$$

C.2 Recursion and Choice

Consider a class StackUser that defines methods that use a Stack object:

```java
class StackUser : {{Stack pushN(Stack) : {Stack popAll(Stack):end}}} {
    Stack pushN(Stack x) {
        x.push(2); x
    }
    Stack popAll(0
```
case FALSE: x.pop(); continue loop
}

Method Stack popAll(Stack): Consider the input typing context \( \Delta_0 = x : \text{Stack[end]}, \text{this} : \text{StackUser[end]} \) and e is the switch expression in the body of method Stack popAll(Stack). The inference tree for the body of method Stack popAll(Stack) is:

\[
\begin{align*}
&\text{Call} \\
&\quad \Delta_1 = x : \text{Stack[int pop() : X]}, \text{this} : \text{StackUser[X]} \\
&\quad \Delta_1 \vdash x.\text{pop() : int + } \Delta'_1 \\
&\quad \text{Continue} \\
&\quad \Delta'_1 = x : \text{Stack[X], this} : \text{StackUser[X]} \\
&\quad \Delta'_1 \vdash \text{continue loop : bot + } \Delta_0, \text{loop : X} \\
&\quad \text{Seq} \\
&\quad \Delta_1 \vdash x.\text{pop(); continue loop : bot + } \Delta_0, \text{loop : X} \\
&\quad \text{PathR} \\
&\quad \Delta_2 = x : \text{Stack[S], this} : \text{StackUser[end], loop : X} \\
&\quad \Delta_2 \vdash x : \text{Stack[S] + } \Delta_0, \text{loop : X} \\
&\quad \text{Call} \\
&\quad \Delta_3 = x : \text{Stack[Choice isEmpty() : join(S, int pop() : X)]}, \\
&\quad \text{this} : \text{StackUser[join(end, X)]} \\
&\quad \Delta_3 \vdash x.\text{isEmpty() : Choice + join}(\Delta_1, \Delta_2) \\
&\quad \text{Switch} \\
&\quad \Delta_4 = x : \text{Stack[\mu X. Choice isEmpty() : join(S, int pop() : X)]}, \\
&\quad \text{this} : \text{StackUser[\mu X. join(end, X)]} \\
&\quad \Delta_4 \vdash \text{loop : e : Stack[S] + } \Delta_0 \\
&\quad \text{LEXP} \\
&\quad \text{end } = \text{\mu X. join(end, X)} \\
&\quad \Delta = x : \text{Stack[\mu X. Choice isEmpty() : join(S, int pop() : X)]}, \\
&\quad \text{this} : \text{StackUser[end]} \\
&\quad \Delta \vdash \text{loop : e : Stack[S] + } \Delta_0 \\
&\quad \text{Equiv}
\end{align*}
\]

Method Stack pushN(Stack): Assume a typing context \( \Delta_0 = x : \text{Stack[end], this} : \text{StackUser[Stack popAll(Stack) : end]} \) The inference tree for method Stack pushN(Stack) is:

\[
\begin{align*}
&\text{PathR} \\
&\quad \Delta_1 = x : \text{Stack[S], this} : \text{StackUser[Stack popAll(Stack) : end]} \\
&\quad \Delta_1 \vdash x : \text{Stack[S] + } \Delta_0 \\
&\quad \text{Call} \\
&\quad \Delta = x : \text{Stack[void pushN() : S], this} : \text{StackUser[Stack popAll(Stack) : end]} \\
&\quad \Delta \vdash x.\text{push(2) : void + } \Delta_1 \\
&\quad \text{Seq} \\
&\quad \Delta \vdash x.\text{push(2) : x : Stack[S] + } \Delta_0
\end{align*}
\]

C.3 Method Call

Consider the code

\[
s = c.\text{pushN}(s)
\]

with input context \( \Delta_0 = s : \text{Stack[S], c} : \text{StackUser[Stack popAll(Stack) : end]} \). The derivation tree for the above code is:

\[
\begin{align*}
&\text{Seq} \\
&\quad c.\text{pushN(Stack x) \{x.push(2); x\} } \in \text{methods(StackUser)} \\
&\quad \text{Stack pushN(Stack) method body inference} \\
&\quad \Delta_2 = \Delta_1, x : \text{Stack[void push(int) : S]} \\
&\quad \Delta_2 \vdash (x.\text{push(2); x}) \{c/this\} : \text{Stack[S] + } \Delta_1 \\
&\quad \text{PathR} \\
&\quad \Delta_3 = s : \text{Stack[void push(int) : S]}, \\
&\quad c : \text{StackUser[Stack popAll(Stack) : end]} \\
&\quad \Delta_3 \vdash s : \text{Stack[void push(int) : S] + } \Delta_2 \\
&\quad \Delta = s : \text{Stack[void push(int) : S]}, \\
&\quad c : \text{StackUser[Stack pushN(Stack) : Stack popAll(Stack) : end]} \\
&\quad \Delta \vdash c.\text{pushN(s) : Stack[S] + } \Delta_1 \\
&\quad \Delta_1 = s : \text{Stack[end], c} : \text{StackUser[Stack popAll(Stack) : end], x} : \text{Stack[end]} \\
&\quad \Delta \vdash s = c.\text{pushN(s) : void + } \Delta_0 \\
&\quad \text{ASGNR}
\end{align*}
\]
C.4  Class inference

The inference tree for class StackUser is:

\[
\begin{align*}
\text{CLASS} & \quad \text{Set-St} \\
\text{METHOD-St} & \quad \text{LEAPR} \\
\text{Stack popAll(St)} & \text{method body inference} \\
\Delta_1 & = x : \text{Stack}[\mu X.\{S, \text{int pop():} X\}], \text{this : StackUser[end]} \\
\Delta_1 & \vdash \text{Stack popAll(St x) \{loop : e\} : U \xrightarrow{} \text{this : StackUser[end]} } \\
\text{END-St} & \\

\vdash \text{StackUser[end]} \\
\text{METHOD-St} & \quad \text{Set-St} \\

\vdash \text{StackUser[Stack popAll(St) : end]} \\
\vdash \text{StackUser[[Stack popAll(St) : end]]} \\
\text{Seq} & \\
\text{Stack pushN(St) method body inference} \\
\Delta_2 & = x : \text{Stack}[[\text{void push(int):} S]], \text{this : StackUser[[Stack popAll(St) : end]]} \\
\Delta_2 & \vdash \text{Stack pushN(St x) \{x.push(2); x\} : U \xrightarrow{} \text{StackUser[[Stack popAll(St) : end]]} \\
\vdash \text{StackUser[Stack pushN(St) : [Stack popAll(St) : end]]} \\
\vdash \text{StackUser[[Stack pushN(St) : [Stack popAll(St) : end]]} \\
\end{align*}
\]