Typechecking Protocols with Mungo and StMungo

Citation for published version:

Digital Object Identifier (DOI):
10.1145/2967973.2968595

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
18th International Symposium on Principles and Practice of Declarative Programming PPDP 2016

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Typechecking Protocols with Mungo and StMungo

Dimitrios Kouzapas  Ornella Dardha  Roly Perera  Simon J. Gay
School of Computing Science, University of Glasgow, UK

Abstract
We report on two tools which 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 can communicate with a real-world SMTP server.

1. Introduction
In this paper we present two tools which extend the Java development process with support for static typechecking of communication protocols. Mungo1 extends Java with typestate definitions, which associate classes with state machines defining permitted sequences of method calls [42]. To associate a typestate definition with a class, the programmer adds a @Typestate annotation to the class telling Mungo where to find the typestate definition file. Mungo will then ensure that instances of the class are used in a manner consistent with the declared typestate.

StMungo (Scribble-to-Mungo) uses this typestate feature to connect Java to the broader setting of communication protocols specified in the Scribble protocol language [40]. 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, 22, 28, 32, 33, 35, 38]. 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 which 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 which 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 typechecked 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
Mungo1 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, which must be a subset of the methods defined by the class; each method specifies a transition to a successor state. Typestate definitions are provided in separate files, using the Java-like syntax shown in Example 2.1 below. A typestate definition is attached to a class using the annotation @Typestate(""ProtocolName""), where "ProtocolName" names the typestate definition file. The typestate inference algorithm, presented in §6 below, 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 2.1. We introduce Mungo through the example of a stack data structure which follows a typestate specification. Given the following enumerated type:

```java
enum Check { EMPTY, NONEMPTY }
```
	hen one possible typestate protocol for a stack is as follows:

```java
@Typestate("StackProtocol")
class Stack {
    @Typestate("Empty")
    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() {} }
```

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:

```java
class Stack {
    Stack User {
        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.

```java
class StackUser {
    Stack pushN(Stack s, int n)
    { do { s.push(n--); } while(n>0); return s; }
    Stack popAll(Stack s)
    { do { s.push(n--); } while(n>0); return s; }
    public static void main(String[] args)
    { System.out.println(s.pop());
        switch(s.isEmpty()) {
            case EMPTY: break loop;
            case NONEMPTY: continue loop;
        } while(true);
    }
```
of the form \( \text{label: do } \{ \text{switch } \{ \text{block } \} \} \text{while(true) } \) 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
s = 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 uninitialised, 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 finish() {
    s.deallocate();
  }
  public static void main(String[] args)
  { loop : do {
    System.out.println(s.pop());
    switch(s.isEmpty()) {
      case EMPTY: break loop;
      case NONEMPTY: continue loop;
    } while(true); }
  void finish() { s.deallocate(); }
  public static void main(String[] args)
  { StackUser2 su = new StackUser2();
    if(su.pushN(15) || su.pushN(32))
      su.pushH(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
propTypes StackUserProtocol {
  Init = { boolean pushN(int n): 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) tool, which translates Scribble [40, 45] 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. 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. StMungo is the first tool to provide a practical embedding of Scribble multiparty 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 App. B. 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:

```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(Price) from F to R,A;
    ticket(String) from A to R;
    invoice(Price) from A to F;
    payment(Price) from F to A; }
  or { refuse(Price) from F to R,A; }}
```

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.

```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(Price) to F;
    ticket(String) from R; }
  or { refuse(Price) 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

---

1The tool is developed and maintained by the second author and can be downloaded from our web page [1].
the request, quote and ticket messages. StMungo converts the
BuyTicket_R local protocol into the RProtocol typestate protocol:

```java
1       local protocol SMTP_C(role S, self C) {
2          _220(String) from S; ...
3          rec X1 {
4              choice at S {  
5              _250dash(String) from S; continue X1; }  
6              or {  
7              _250(String) from S; ...
8              rec Z1 {  
9              ... data(String) to S; ...
10             rec Z3 {  
11             choice at C {  
12             subject(String) to S;  
13             continue Z3; }  
14             or {  
15             dataline(String) to S;  
16             continue Z3; }  
17             or {  
18             atad(String) to S;  
19             _250(String) from S;  
20             continue Z1; }  
21             } } ... } }
```

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 which 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 {
2          State0={String receive_220StringFromS():State1}  
3          ...  
4          State3={Choice1 receive_Choice1LabelFromS():  
5              < _250DASH: State4, _250: State5 >}  
6          State4={  
7              String receive_250dashStringFromS():State3}  
8          State5={String receive_250StringFromS():State6}  
9          ...  
10         State27={void send_dataStringToS(String):State28}  
11         ...  
12         State29={void send_SUBJECTToS():State30,  
13             void send_DATAFILEToS(): State31,  
14             void send_ATADToS():State32} ... }
```

The receive and send messages in the SMTP_C local protocol are interpreted as typestate methods in the CProtocol typestate specification. For example, the message _220(String) received from S given in line 2 in SMTP_C becomes a method with signature:

```java
String receive_220StringFromS()
```

given in line 2 in CProtocol.

Similarly, the message data(String) sent to S and given in line 9 of SMTP_C becomes a method with the following signature:

```java
void send_dataStringToS(String)
```

given in line 10 in CProtocol.

Let us now comment on choice. The external choice made at role S different from self is given in lines 4-18 of SMTP_C. For every external choice in the local protocol there is an enumerated type in the typestate, such as the following:

```java
tenum Choice1 { _250DASH, _250; }
```
The values of Choice1 are determined by the first interaction of every branch in the choice. The external choice itself is translated as a receive method returning the enumerated type Choice1 and given in lines 4-5 of CProtocol:

Choice1 receive_CHOICE1LabelFromS():
   <_250DASH: State4, _250: State5>

After choosing one of the branches, _250DASH or _250, the payload of type String is received via another method call, following the choice: receive_250dashStringFromS() in line 7 and receive_250StringFromS() in line 8, respectively for the two available choices.

The internal choice made at self, namely role C (lines 11-17 of SMTP_C), is translated into a set of send methods, one for each branch of the choice (lines 12-14 of CProtocol). When running the program, only one of these methods will be called, thus performing a single message selection corresponding to it.

CRole implements all the methods in CProtocol. In this implementation, since communication occurs on Java sockets, we declare and create a new socket to connect to the gmail server. This is given in lines 2 and 4 in CRole, respectively.

```
@Typestate("CProtocol") class CRole {
  private Socket socketS = null; ...
  public CRole() {
    socketS = new Socket("smtp.gmail.com", 587); ...
    /* CProtocol method definitions */
  }
}
```

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 the above 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.

```
public static void main(String[] args) {
  CRole currentC = new CRole();
  ...
  switch(…)
  {
    case /*label to be sent*/:
      case /*SUBJECT*/:
        currentC.send_SUBJECTToS();
        String subject = // input subject;
        currentC.send_subjectStringToS("subject");
      case /*DATACLINES*/:
      case /*ATAD*/:
        currentC.send_ATADToS();
        currentC.send_subjectStringToS("single dot");
        String _250msg =
          currentC.receive_250StringFromS();
        continue _23;
    }
    while(true); }
```

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 customise CMain by adding SSL connection code for authentication with the gmail server. Mungo is able to statically check CMain, or any code that uses a 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 CMain, or even the 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 "~" 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 F and methods M. An enumeration declaration defines an enumerated type named E with a non-empty set of enum values. For simplicity, 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 may contain the recursive typestate variable ~H, or it is an ~T of the form ~C. Fields are denoted with the ~operator. A class declaration is either an internal choice ~H of method signatures, or a recursive typestate ~T~E, which may contain the recursive typestate variable TX. A field signature H can have two forms, depending on whether the method transitions to a state S, or it is an external choice E m(T): (l : S)~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 ~X~ is not well-formed and that all state definitions are closed. A type is either the name of a class or enumeration, void or bool. A field declaration is a field name associated with a type T. A method declaration T m(T'): (e) specifies a return type T, the parameter e of the method, the type T' of the parameter x, and the expression e which comprises the method body. A path is either the atomic path 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 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. In the expression forms method call r.m(e), field assignment r.f = e, and object creation r = new C, have the target object of the invocation or assignment is restricted to a path

```
D ::= class C : S { F; M } | enum E { l }  
S ::= H | μX | X  
H ::= T m(T) : S | E m(T) : (S)~E~  
T ::= C | E | bool | void  
F ::= T f  
M ::= T m(T x) { e }  
r ::= this | r.f | x  
c ::= l | tt | ff | null | *  
e ::= c | r | r.m(e) | r.f = e | e | e | r.f = new C  
λ : e | continue λ  
| switch(e) { } else e  
```

**Figure 1.** Top-level syntax
\[ o := C[f \rightarrow a] \mid c \quad r := \text{root} \mid r.f \]
\[ e := \ldots \mid e \uparrow r \quad v := e \mid r \]
\[ S ::= \ldots \mid (S)_{\text{BE}} \quad s ::= T(m(T)) \mid E(m(T)) : l \mid l \]
\[ E ::= \{ \} \mid r.m(E) \mid e = \hat{E} \mid E; e \mid \text{switch}(E) \{ e \}_E \]
\[ E \uparrow r \mid \text{if}(E) e \quad \text{else} e \]
\[ \ell ::= r.f.rnew C \mid r.\ell l \mid r.T m T' \mid r.f = v \mid \tau \mid \text{if} \]

\[ \text{Figure 2. Runtime syntax} \]

\[ r, \text{rather than an arbitrary expression. The other expression forms include sequential composition } e; e', \text{ switch expressions, if } \ldots \text{ else expression, labelled expressions } \lambda : e, \text{ and continue expressions which jump to the enclosing expression labelled by } \lambda. \]

\textbf{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 (§6).

The expression \( e \) in a configuration \( h, e \) is a runtime expression in which every (compile-time) path of the form \( r, f \) or \( x \) has been replaced by a runtime path which refers to a heap value. A runtime path \( r \) in a heap \( h \) is either the atomic path 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 \( e \uparrow r \), which is an expression \( e \) which has been tagged with \( \uparrow r \) 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 \( \mathcal{E}[e] \), where \( \mathcal{E} \) is an evaluation context (an expression with a hole). As usual, the notation \( \mathcal{E}[e] \) denotes the plugging of the hole in \( h \) with an expression \( e \).

The operational semantics is annotated with labels \( \ell \) that denote the creation of a new object \( r.f.rnew C \), an enum value choice \( (r.l)_i \), method call \( (r.f.m T') \), assigning a field \( (r.f = v) \), the conditional label \( (l.f) \), and the silent label \( (r) \). The definition of states is extended to the set of enum values \( (l : S)_{\text{BE}} \) and we define action labels \( s \) for labels: internal choice \( T(m(T)) \), external choice \( E(m(T)) : l \), and for enum values \( l \).

\textbf{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 \( h[r.f \mapsto o'] = h[r \mapsto C[f : o.f : o']] \) if \( h(r) = C[f : o] \). 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 semantics; 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 \( \tau \) label and proceeds with the evaluation of \( e \). Rules R-TRUE and R-FALSE are the usual rules for the \( \uparrow \) \ldots \ else expression and are annotated with label \( i \). Rule R-NEW is labelled with \( r.f.rnew C \) and overwrites the contents of the field \( r.f \) by a new object \( C[f \rightarrow \text{init}(T)] \) whose fields are all initialised to the value \( \text{init}(T) \), where \( T \) is the type of the field, defined as: \( \text{init}(C) = \text{null} \); \( \text{init}(E) = E_{\text{EQ}}; \text{init}(\text{bool}) = \text{false} \); \( \text{init}(\text{void}) = s \), where for every enumerated type \( E \) we require there to be a distinguished element \( E_{\text{ EQ}} \in \text{enum}(E) \). The result of R-NEW is the void value \( * \). There is no allocation of a fresh location; instead the object is constructed at an existing location \( r.f \). There are two assignment rules, depending on whether the value being assigned is a constant or an object path. Both forms return the void value \( * \). A constant \( c \) has no associated typestate and may be used unrestrictedly; therefore the R-AssgnC rule is labelled with \( \tau \) and simply updates the heap to store \( c \) in \( r.f \). A path \( r' \), on the other hand, refers to an object and must be used linearly. Therefore the effect of the R-AssgnR rule is to relocate the object from \( r' \) to \( r \), leaving null at its old location. The annotation label for R-AssgnR is \( r.f = v \). The R-CALL rule is labelled with \( r.T m T' \) and resolves the method \( m \) by first looking up the receiver \( r \) in the heap, which must be an object \( C[f : o] \), and then selecting the method \( m \) from the definition of \( C \). Prior to executing the selected method, we convert its body \( e \), which is a source-level expression, into a runtime expression by substituting the runtime path \( r \) for this and also \( v \) for the formal parameter. In addition, the resulting runtime expression is tagged with \( \uparrow r \), recording the fact that \( r \) is the active receiver. The active receiver tag \( \uparrow r \) on a value is removed using a \( \tau \) label when the value is fully evaluated and it is not an enum label, as defined by rule R-VALUE. If the value returned by the method is an enum label \( l' \), then it must occur as the scrutinee of a switch expression; rule R-Switch defines the reduction via action \( r.(l') \), of the switch expression to the branch indicated by \( l' \). The \( r \) is used in the reduction label to indicate which object made the choice. Rule R-Label is labelled with \( \tau \), and says that a labelled expression \( \lambda : e \) labels the active receiver tag \( \uparrow r \) and substitutes a copy of the labelled expression for every occurrence of continue \( \lambda \) that occurs in the loop body \( e \). Rule R-Ctx lifts these rules to an arbitrary expression using an evaluation context. It is easy to show that the operational semantics is deterministic.

Assume a heap consisting of an instance of class \( C \), where given \( \text{fields}(C) = \overline{T(f)} \), each field of \( C \) is initialised with the corresponding value \( \text{init}(T) \). Execution can then be initiated using a top-level expression that substitutes path this with path root.

\textbf{6. Typestate Inference}. In this section we formalise a typestate inference system and prove its safety properties. The system presented here infers a typestate
Figure 4. Subtyping relation (Symmetric rule S-Rec2 omitted).
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**, **PamR**, **PamC**, **AsgnR**, **AsgnC**, and **New** that require
the treatment of linearity. Consider the following code that uses the
implementation of class **Stack** in section §1:

```java
s = new Stack(); k = s;
```

(1)

The code expression matches rule **Seq**. We assume \(\Delta_0 = s : Stack[end], k : Stack[s]\) as an input typing context and \(S \preceq_{st} StackProtocol\). Rule **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 expression. In order to type the second expression by **AsgnR** we need to infer a typestate for \(s\). To respect linearity we take \(\Delta_1 = s : Stack[end], k : Stack[end]\) as input. The derivation is as:

\[
\begin{align*}
\Delta_1 + k & = s : void + \Delta_0 = s : Stack[S], k : Stack[S] & \text{AsgnR} \\
\Delta + r & = e : void + \Delta', r : U & \text{PamR}
\end{align*}
\]

The output typing context for **PamR** is \(\Delta_2 = s : Stack[S], k : Stack[end]\), meaning that \(k\) has an inactive typestate before assignment. Rule **PamR** on its own “guesses” a type for a path expression. However, the combination of **PamR** and **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 **New**. By assumption we satisfy its premise: we have \(S \preceq_{st} StackProtocol\), meaning path \(s\) is used according to the StackProtocol typestate (this is shown in \(\Delta_2\)). Rule **New** infers a type void for the first expression. Since it is not a class type it satisfies the premise of rule **Seq** 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 type of the first expression is not of type void to disallow dead code after a continue expression (see rule **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:

```java
s = new Stack(); k = s;
```

(2)
every branch is the same as the one for \texttt{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 \texttt{switch}. The condition should have an enumeration type that matches the type of the \texttt{switch} definition. Finally, the type of \texttt{switch} is the join of the types of its branches. For the \texttt{true} branch we use rule \texttt{PanR}:

\[
\text{PANR} \quad \Delta_2 = x: \text{Stack}[S].\text{this} : \text{StackUser[end]}, \text{loop} : \text{X} \quad \Delta_3 = x: \text{Stack}[S] + \Delta_1
\]

For the \texttt{false} branch we first use rule \texttt{Seq0} and then continue \texttt{continue} to infer the typestate of the \texttt{continue} loop expression. \texttt{continue} requires loop to be mapped to a recursive variable \texttt{X} in the input typing context. It then outputs a typing context where all paths mapped to a typestate are updated to the typestate \texttt{X}, as in:

\[
\text{CONTINUE} \quad \Delta_1 = x: \text{Stack}[X].\text{this} : \text{StackUser[X]} \quad \Delta_3 = x: \text{Stack}[X] + \Delta_1 \quad \text{loop} : \text{bot} + \Delta_1
\]

The type of the continue expression is \texttt{bot}, since we want \texttt{join}ing to be defined (cf. Fig. 5). To complete the typing of the \texttt{false} branch, we apply rule \texttt{Call} for \texttt{x.pop()} and conclude with rule \texttt{Seq0}. The output typing context is:

\[
\Delta_3 = x: \text{Stack}[X].\text{this} : \text{StackUser[X]} \quad \Delta_1 = x: \text{Stack}[X].\text{this} : \text{StackUser[X]} \quad \text{loop} : \text{bot} + \Delta_1
\]

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

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

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

Rule \texttt{If} types the conditional expression in a similar way as rule \texttt{Switch}. 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 \texttt{bool}.

\textbf{Method Call.} Rule \texttt{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 \texttt{initT}, defined by $T \neq C \Rightarrow \text{initT}(T) = T$ and \texttt{initT}(C) = C[end].

Rule \texttt{Call} requires typechecking the method body every time a method is called. This is a simplification for presentation 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 \texttt{Call}, consider the following code that uses class \texttt{StackUser}:

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

and $\Delta_0 = s: \text{Stack}[S]$, for some $\text{StackUser[Stack popAll(Stack) : end]}$, the input typing context. By applying rule \texttt{AssignR} on the above assignment with input $\Delta_0$, the output typing context in the premise of the rule is $\Delta_1 = s: \text{Stack[end]}, \text{c: StackUser[Stack popAll(Stack) : end]}$. At this point we can apply rule \texttt{Call} on $\text{c.push(s)}$ and have the following derivation:

\[
\text{Stack.push(StackSize x} \{ \text{x.push2(); x} \in \text{methods(StackSize) (1)} \quad \Delta_2 = \text{x.push2(); x} \in \text{methods(StackSize)} \}
\]

\[
\Delta_1 = s: \text{Stack[end]}, \text{c: StackUser[Stack popAll(Stack) : end]} \quad \text{Stack.push(StackSize x} \{ \text{x.push2(); x} \in \text{methods(Stack) : (2)} \quad \Delta_1 = s: \text{Stack[end]}, \text{c: StackUser[Stack popAll(Stack) : end]} \}
\]

\[
\text{c: StackUser[Stack pushAll(Stack) : Stack.push(StackSize x} \{ \text{x.push2(); x} \in \text{methods(Stack) : (3)} \quad \text{rule PanR is used where $\Delta_3$ also updates the type of the receiver:}
\]

\[
\Delta_3 = s: \text{Stack[end]}, \text{c: StackUser[Stack pushAll(Stack) : [Stack.popAll(Stack) : end]] : (4)} \quad \Delta = \text{Stack[void.push(int) : S] : (5)}
\]

The premise of \texttt{Call}, given in (1), performs a lookup in the methods of the class of the receiver, \texttt{ListCons}, to obtain the definition of method \texttt{Stack pushAll(Stack)}. Next, in (2), the premise infers a typestate for the body of the method in which $c$ has been substituted for the keyword \texttt{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:

\[
\Delta_2 = \text{x : Stack[void.push(int) : S]} : (6)
\]

Then, in (3) \texttt{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 \texttt{PanR} is used where $\Delta_3$ also updates the type of the receiver:

\[
\Delta_3 = s: \text{Stack[end]}, \text{c: StackUser[Stack pushAll(Stack) : Stack.popAll(Stack) : end]] : (7)} \quad \Delta = \text{Stack[void.push(int) : S] : (8)}
\]

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

\[
\text{Stack.pushAll(Stack x) : x.push2(); x = this.popAll(x) : x : (9)}
\]

If we infer a typestate for the body of \texttt{Stack pushAll(Stack) with input context $\Delta_1$, we get: an output typing context, $\Delta'$, such that:

\[
\Delta'(c) = \text{StackUser[Stack popAll(Stack) : [(Stack.popAll(Stack) : end)] : (10)}
\]

Given that $\Delta_1(c) = \text{StackUser[Stack popAll(Stack) : end]}$, it is revealed that the body of \texttt{Stack pushAll(Stack) calls method \texttt{Stack popAll(Stack) on its receiver object, thus violating the abstraction principle.}

\textbf{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 \texttt{Set-Sr} requires the inference and join of the typestates of all branches in an internal choice. Rule \texttt{Method-Sr} relies on the \texttt{infer(T)} definition that maps a type $T$ to the corresponding inferred type $U$ as:

\[
\text{U : (C \Rightarrow \text{infer}(T)) \Rightarrow \text{infer}(T) = T \quad \text{and \text{infer}(C) = C[end]}}
\]

Rule \texttt{Method-Sr} infers a method-prefixed typestate, whereas 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 \texttt{infer(T)} is used to check that the return and parameter types of the method match the types of the inferred ones. As in \texttt{Call}, a self-call should preserve the typestate of the receiver up to type equivalence. Rule \texttt{End-Sr} is similar to rule \texttt{Method-Sr}. It requires the inference and join of the typestates of all the external choices and then infers the method prefix. Rule \texttt{End-Sr} requires all fields of the class to finish in the inactive typestate. Rules \texttt{Rec-Sr} and \texttt{Var-Sr} are similar to rules \texttt{LEsOp} and \texttt{Continue}, where they bind and use a recursive variable, respectively. Rule \texttt{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 \texttt{Class}. A program is well-typed if all of its classes are well-typed, as stated by rule \texttt{Program}. To illustrate the rules, we show a typestate inference for \texttt{StackUser} in App. C.

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 \texttt{Switch-Ar} is similar to \texttt{Switch}, the difference being the condition of the switch, which is evaluated to an active receiver.
\[ \Delta' \vdash C[S] \quad T_i \ m(T_2 \ x) \{ e \} \in \text{methods}(C) \quad \forall \ell, S' \xrightarrow{\ell} S \implies S \xrightarrow{\ell} S \quad \forall H \in \overline{H}. \Delta_H \vdash C[[H]] \]

\[ \Delta, : c : [S'], x : \text{infer}(T_2) + \Delta', \text{this : } C[S], x : \text{init}(T') \]

\[ \Delta' \vdash [T \ m(T') : S] \]

**Method-ST**

**Enum-ST**

\[ \forall l \in E. \Delta, \vdash C[S'] \quad E \ m(T) \{ e \} \in \text{methods}(C) \quad \Delta, x : C[S'] \vdash E \ m(T) \{ e \} : E + \Delta'' \quad \Delta' = \text{join}(\Delta, \{ e \}) \]

\[ \Delta \vdash C[E \ m(T) : (S)] \]

**Setup-ST**

\[ \Delta \vdash C[H] \]

**Figure 7.** Typestate inference rules for methods, classes and programs

**Figure 8.** Typestate inference rules for runtime syntax

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Context</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ty-Id</strong></td>
<td>( \Delta \vdash e )</td>
<td>( e )</td>
<td>( \Delta \vdash e )</td>
</tr>
<tr>
<td><strong>Ty-Inf</strong></td>
<td>( \Delta \vdash \Delta' \leq_{stt} \Delta' )</td>
<td>( \Delta \vdash e )</td>
<td>( \Delta \vdash e )</td>
</tr>
<tr>
<td><strong>Ty-Asgn</strong></td>
<td>( \Delta \vdash e \in r : U + \Delta' )</td>
<td>( e )</td>
<td>( \Delta \vdash e )</td>
</tr>
<tr>
<td><strong>Ty-Call</strong></td>
<td>( \Delta, r : C[T \ m(T') : S] )</td>
<td>( T_i \ m(T_2 \ x) { e } \in \text{methods}(C) )</td>
<td>( \Delta' \vdash \Delta' )</td>
</tr>
<tr>
<td><strong>Ty-AsgnR</strong></td>
<td>( \Delta, r.f : C[\text{end}], r' : C[S] )</td>
<td>( \frac{r \xrightarrow{t, new c} r', \Delta, r.f : C[S], r' : C[\text{end}]}{\Delta' \leq_{stt} \Delta} )</td>
<td>( \Delta \vdash \Delta' )</td>
</tr>
<tr>
<td><strong>Ty-New</strong></td>
<td>( \Delta, r.f : C[\text{end}], r' : C[S'] \in \Delta, S' = \text{end} )</td>
<td>( \frac{(S \leq_{stt} \text{typestate}(C) \land \forall r.f, f' : C'[S'] \in \Delta, S' = \text{end}}{\Delta, r.f : C[\text{end}], r' : C[S]} )</td>
<td>( \Delta' \leq_{stt} \Delta )</td>
</tr>
<tr>
<td><strong>Ty-Label</strong></td>
<td>( \Delta, r : C[S] )</td>
<td>( \frac{r \xrightarrow{t, new c}}{\Delta' \leq_{stt} \Delta} )</td>
<td>( \Delta, r : C[S] )</td>
</tr>
</tbody>
</table>

**Figure 9.** Reduction relation on typing contexts

\[ \text{rather than a method call. Rule ArR infers a typestate for } e @ r, \text{ by first inferring a typestate for } e. \text{ The other rules are used to type runtime configurations. Rule Heap uses rule Object to check whether a typing context is consistent with all the objects in the heap. Rule Object checks that the typestate of the objects in the context match the declared typestate of their class. Finally, rule Constr infers a typestate for a runtime configuration, by first inferring a typestate for the expression and then using its output typing context to type the heap. The output typing context and the typestate of the configuration match those of the expression.} \]

6.2 Properties of the typestate inference system

6.4 (Coherence of Typestate Inference). Let \( D \) be a set of declarations that \( \vdash D \). Assuming \( D \) is the program context, let \( e \) be a run time expression and suppose \( \Delta \vdash h, e : U + \Delta' \). Then, either \( e \) is a value, or there exist \( f, h' \) and \( e' \) such that \( h, e \xrightarrow{t, new c} h', e' \), and there exist \( \Delta' \) and \( U' \) such that \( \Delta' \xrightarrow{t} \Delta' + \Delta' \) and \( U' \leq_{stt} U \).

7. Related Work

**Session types and programming languages.** The Session Java (SJ) language [28] builds on earlier work [14, 15, 17] to add binary session type channels to Java. SJ has been applied to a range of
situations including scientific computation [37] and event-driven programming [26]. SJ implements a library for binary sessions that have a pre-defined interface. The Java syntax is extended with communication statements that enable typechecking. The scope of a session is restricted to the body of a single method. Mungo lifts these restrictions by allowing the abstraction of multiparty session types as user-defined objects that can be passed and used throughout different program scopes. Gay et al. [23] outlined an implementation of their type system as a language called Bica, which is not currently maintained and is unusable. Mungo improves on Bica by using type inference to remove the need for typestate declarations on methods.

The work in [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-type 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.

Capacci et al. [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 [36] typechecks the operations of a library that implements multiparty session types using a restricted set of MPI [30] primitives. In contrast, our framework typechecks Java statements and expressions, instead of higher-level operations. The work in [35] 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. [34] 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, [33] implements a session-based Actor framework that uses runtime monitoring to integrate multiparty session types. A hybrid approach has been used by Hu [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 [22] and has been used to achieve protocol checking in Idris [29] and as a replacement for explicit typestate in Rust [39].

Typestate. There have been many efforts to add typestate to practical languages, since their introduction in [42]. Vault [12, 19] is an extension of C, and Fugue [13] applies similar ideas to C#. Plural [6] is based on Java and has been used to study access control systems [5] and transactional memory [4], and to evaluate the effectiveness of typestate in Java APIs [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# [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. [8] have formalised a core calculus based on Sing# and proved type safety.

Aldrich et al. [2, 43] 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 classes, states are organised into an inheritance hierarchy. The most recent work [20, 44] uses gradual typing to integrate static and dynamic 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 includes the “adoption and focus” approach of Vault and Fugue, the permission-based approaches of Plural and Plaid, and an expressive fine-grained system by Militão et al. [31]. 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 processes, 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. Concluding Remarks 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 6.3). These properties guarantee the coherence of the typestate inference system with respect to the declared typestate in a program (Corollary 6.4). Future Work. The combination of Mungo and StMungo is effective for statically checking the correct implementation of communication 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 “adoption and focus” technique of Vault and Fugue, the permission-based approaches of Plural and Plaid, and the system by Militão et al. [31]. Another aim is to support generics and inheritance. Inheritance between 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 specification 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 SiMungo for more sophisticated protocol verification 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 Voinela for her contribution to Mungo and SiMungo. We thank Garrett Morris, Raymond Hu and Nobuko Yoshida for useful comments and discussion.

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

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

Proof. Both cases follow by using rule Heap and for 1. typing rules for constants are used, and for 2. rule Object is used.

Lemma A.2 (Replacement). If

• $d$ is a derivation for $\Delta + E[e] : U + \Delta'$,
• $d'$ is a subderivation of $d$ concluding $\Delta + e : U' + \Delta$,
• the position of $d'$ in $d$ corresponds to the position of the hole in $E$,
• $\Delta' + e' : U' + \Delta$, such that $U' \bowtie U$,

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

Proof. Follows [23], by replacing the derivation $d'$ in $d$ with the derivation for $\Delta' + e' : U' + \Delta$. 

Lemma A.3 (Substitution). If $\Delta, \Delta' + e : U + \Delta'$ and $\Delta \bowtie v : U' + \Delta''$, then $\Delta(v : U') + e[v/x] : U + \Delta'$.

The proof proceeds by induction on the structure of context $\Delta$.

Lemma A.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 $\Delta \bowtie U$ and $\Delta \bowtie U'$.

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

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

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

A.2 Progress and Subject Reduction

Proof of Theorem 6.3: Let $\tilde{D}$ be a set of declarations such that $\vdash \tilde{D}$. In a context parametrized by $\tilde{D}$, let $e$ be a run time expression and suppose $\Delta + h, e : U + \Delta''$.

Then, either $e$ is a value, or there exist $\ell, h'$ and $e'$ such that $h, e \xrightarrow{\ell} h', e'$, and there exist $\Delta'$ and $U'$ such that $\Delta \xrightarrow{\ell} \Delta'$ and $\Delta' + h, e' : U' + \Delta''$ and $U' \bowtie U$. Proof. The proof proceeds by induction on the structure of the expression $e$ with respect to contexts. We present first the inductive case. Let $e = E[e_1]$ where $e_1$ is not a value and $E \neq \emptyset$. By assumption and inversion on rule Config we have $\Delta + E[e_1] : U + \Delta''$. By Lemma A.5 there exist $\Delta_1$ and $U_1$ such that $\Delta + e_1 : U_1 + \Delta_1$. By induction hypothesis there exist $h', \ell$ such that $h, e_1 \xrightarrow{\ell} h', e_2$. By induction hypothesis we also have $\Delta \xrightarrow{\ell} \Delta'$ and $\Delta' + h, e_2 : U_2 + \Delta_1$, which by inversion on Config means that $\Delta' + h$ and $\Delta' + e_2 : U_2 + \Delta_1$, where $U_2 \bowtie U_1$. By rule R-Ctx we have $h, E[e_1] \xrightarrow{\ell} h', E[e_2]$. Using Lemma A.2 we obtain $\Delta' + E[e_2] : U' + \Delta''$ with $U' \bowtie U$. We conclude by rule Config.

The base cases when $e$ is of the form $E[v]$ with $E$ elementary, not being of the form $E[E']$ with $E' \neq \emptyset$, and not of the form $E[e_1]$ 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 $\text{R-Seq}$

$h, (v; e') \xrightarrow{\cdot} h, e'$

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

\[
\begin{align*}
\Delta &\vdash v : U' \vdash \Delta_1 \quad U' \not\in C[S] \\
\Delta_1 &\vdash e' : U + \Delta''
\end{align*}
\]

By reduction rule $\text{Ty-IO} \quad \Delta \xrightarrow{\cdot} \Delta$. Since $U' \not\in C[S]$ and value $v$ is of type $U'$ it means $v$ is some constant $c$. Hence, the judgement $\Delta \vdash v : U' \vdash \Delta_1$ 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 $\text{Config}$.

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

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

such that $\text{fields}(C) = \overline{T \, f}$. By hypothesis and typing rule $\text{Config} \vdash h$ and $\Delta \vdash r.f = \text{new } C : U + \Delta''$. By inversion and typing rule $\text{New}$ we have:

\[
\begin{align*}
\text{New} \quad S &\xleftarrow{\text{sat \ typestate}(C)} \forall r.f. f' : C[S'] \in \Delta_1 \implies S' = \text{end} \\
\Delta_1, r.f : C[\text{end}] \vdash r.f = \text{new } C : \text{void} + \Delta_1, r.f : C[S]
\end{align*}
\]

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

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

such that $S \xleftarrow{\text{sat \ typestate}(C)}$ 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,

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

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

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

We conclude by Lemma A.1.

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

$h, r.f = c \xrightarrow{\cdot} h[r.f \mapsto c], *$

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

\[
\begin{align*}
\text{AssnC} \quad U' \not\in C[S] \quad \Delta \vdash c : U' + \Delta', r.f : U' \\
\Delta \vdash r.f = c : \text{void} + \Delta', r.f : U'
\end{align*}
\]

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 $\text{Ty-IO}$, $\Delta \xrightarrow{\cdot} \Delta$. We have to prove that $\Delta \vdash h[r.f \mapsto c], * : \text{void} + \Delta', r.f : U'$. By rule $\text{Void}$, $\Delta \vdash * : \text{void} + \Delta', r.f : U'$. Recall that, by hypothesis and rule $\text{Heap}$ and inersion we have

\[
\text{Heap} \quad h(r.f) = c' \\
\Delta', r.f : U' + c' : U' + \Delta', r.f : U'
\]

By updating the heap to $h(r.f \mapsto c)$, using the typing judgement for $c$ in the premise of $\text{AssnC}$ and Lemma A.1 we derive $\Delta', r.f : U' + h[r.f \mapsto c]$. We conclude by rule $\text{Config}$.

• $e = (r.f = r')$. By hypothesis and by rule $\text{R-AssnR}$

$h, r.f = r \xrightarrow{\cdot} h'[r.f \mapsto h(r')], *$

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

\[
\begin{align*}
\text{AssnR} \quad \Delta \vdash r' : C[S] \vdash \Delta_1, r.f : C[\text{end}] \\
\Delta \vdash r.f = r' : \text{void} + \Delta_1, r.f : C[S]
\end{align*}
\]

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 $r' \not\in r.f$. Since $r'$ is a path typed by $C[\text{end}]$, the premise of the above derivation is obtained by applying $\text{PathR}$. 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 $\Delta_2(r') = \Delta_3(r') = C[\text{end}]$. By rule TV-AsgnR

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

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 Void we conclude $\Delta' + \star : \text{void} + \Delta''$. It remains to prove that

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

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

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

The result follows immediately by applying twice Lemma A.1 for $r.f$ and $r'$. We conclude by Cosnp. Let $r' = r.f$. By rewriting AsgnR 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

$$h, m(v) \xrightarrow{r.m(v)} h, e[v/x][r/\text{this}] @ r$$

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

$$\frac{T \cdot m(T^r) x \{e\} \in \text{methods}(C) \quad S' = \text{addr} S}{\Delta_2, r : C[S'], x : U' + e[r/\text{this}] : U + \Delta_1, r : C[S]}$$

$$\frac{\Delta + \star : U' + \Delta_1, r : C[T \cdot m(T') : S]}{\Delta \vdash r.m(v) : U + \Delta_1, r : C[S]}$$

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

$$\Delta_3, r : C[T \cdot m(T') : S] \xrightarrow{r.m(v)} \Delta_3, r : C[S]$$

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

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

$$\Delta_1, r : C[T \cdot m(T') : S] \vdash v : U' + \Delta_2, r : C[T \cdot m(T') : S]$$

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

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

Since $S' = \text{addr} S'$, we conclude by rule Eppl. Recall that $\Delta_3, r : C[T \cdot m(T') : S] \vdash h$. By Hexp we have

$$h(r) = C[f \mapsto o] \quad \Delta \vdash C[f \mapsto o] : C[T \cdot m(T') : S] + \Delta$$

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

which by Object it means that

$$\text{typestate}(C) = S_C \quad \exists \overline{x}, S_C \xrightarrow{T \cdot m(T')} T\cdot m(T') : S$$

We perform another reduction with label $T \cdot m(T)$ and we have

$$\text{typestate}(C) = S_C \quad \exists \overline{x}, S_C \xrightarrow{T \cdot m(T')} T\cdot m(T') : S \quad \Delta \vdash C[f \mapsto o] : C[T \cdot m(T') : S] + \Delta$$

$$\Delta_3, r : C[S] \vdash C[f \mapsto o] : C[S] + \Delta_3, r : C[S]$$

We now can type $\Delta_3, r : C[S] \vdash h$ by rule Hexp. We conclude by rule Cosnp.

- $e = v \star @ r$. By hypothesis and by rule R-Value

$$h, v \star @ r \xrightarrow{} h, v$$

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

$$\Delta + \star : v : U + \Delta''$$

$$\Delta + \star : v @ r : U + \Delta''$$

By reduction rule TV-Int $\Delta \xrightarrow{r} \Delta$. The thesis follows trivially.

- $e = \text{switch}(l' @ r) \{e_i\}_{l \in E}$. By hypothesis and by rule R-Switch

$$h, \text{switch}(l' @ r) \{e_i\}_{l \in E} \xrightarrow{c(l')} h, e_l$$

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

$$\forall l \in E \quad \Delta, r : C[S] \vdash e_i : U + \Delta'' \quad \Delta \vdash l' : E + \Delta_1, r : C[l : S]_{l \in E} \quad \Delta = \bigcup_{l \in E} \Delta_l$$

$$\Delta \vdash \text{switch}(l' @ r) \{e_i\}_{l \in E} : \text{join}([U_l]_{l \in E}) + \Delta''$$
where \( U = \text{join}((U_1)_{\Delta E}) \). By inversion and typing rule ENUM we have that \( \Delta = \Delta_1, r : C[(l : S_1)_{\Delta E}] \). By Ty-LABEL

\[
\Delta_1, r : C[(S_1)_{\Delta E}] \xrightarrow{(p)} \Delta_F, r : C[S_F]
\]

where \( l' \in E \) and \( \Delta' \leq_{\text{dat}} \Delta \). By Lemma A.4 we have that \( U_F \leq_{\text{dat}} \text{join}((U_1)_{\Delta E}) \). The judgement \( \Delta_F, r : C[S_F] \vdash e_F : U_F \Downarrow \Delta'' \) holds by the premise of Switch for \( l' \in E \). We need to prove that \( \Delta_F, r : C[S_F] \vdash h \). Recall that, by hypothesis \( \Delta \vdash h \). By HEAP and OBJECT it means that there exist \( \tilde{x} \), such that typestate(\( C = S_C \) and \( S_C \xrightarrow{i} (l : S_1)_{\Delta E} \) and

\[
\Delta \vdash C[\tilde{f}[\tilde{o}] : C[(l : S_1)_{\Delta E}] + \Delta
\]

By Definition 6.2, we have \( (l : S_1)_{\Delta E} \xrightarrow{\tilde{r}} S_F \). By applying rule OBJECT on this reduction, we have

\[
\Delta_F, r : C[S_F] \vdash C[\tilde{f}[\tilde{o}] : C[S_F] + \Delta_F, r : C[S_F]
\]

We conclude by rules HEAP and CONFIG.

• \( e = \text{if} \langle \text{tt} \rangle \ e_1 \text{ else } e_2 \). The case for \( \text{if} \langle \text{ff} \rangle \ e_1 \text{ else } e_2 \) is completely analogous. By hypothesis and by rule R-True

\[
h, \text{if} \langle \text{tt} \rangle \ e_1 \text{ else } e_2 \xrightarrow{\text{if}} h, e_1
\]

By hypothesis and by rule CONFIG \( \Delta \vdash h \) and \( \Delta \vdash \text{if} \langle \text{tt} \rangle \ e_1 \text{ else } e_2 : U \Downarrow \Delta'' \). By inversion and typing rule IF

\[
\begin{align*}
\Delta_1 & \vdash e_1 : U_1 \Downarrow \Delta'' \\
\Delta_2 & \vdash e_2 : U_2 \Downarrow \Delta'' \\
\Delta_3 & = \text{join}(\Delta_1, \Delta_2) \quad \Delta \vdash \text{tt} : \text{bool} \Downarrow \Delta_1 \\
\Delta \vdash \text{if} \langle \text{tt} \rangle \ e_1 \text{ else } e_2 : \text{join}(U_1, U_2) \Downarrow \Delta''
\end{align*}
\]

Where \( U = \text{join}(U_1, U_2) \). Rule Box implies that \( \Delta = \Delta_1 \). By \( \text{if} \langle \text{tt} \rangle \Delta \xrightarrow{\text{if}} \Delta' \) and \( \Delta' \leq_{\text{dat}} \Delta \). By Lemma A.4 we have that \( U_1 \leq_{\text{dat}} \text{join}(U_1, U_2) \). Then, \( \Delta_1 \vdash e_1 : U_1 \Downarrow \Delta'' \) follows directly by the premise of \( \text{if} \rangle \) and by letting \( \Delta' = \Delta_1 \), since \( \Delta_1 \leq_{\text{dat}} \text{join}(\Delta_1, \Delta_2) = \Delta \). It remains to prove that \( \Delta_1 \vdash h \). Since \( \Delta_1 \leq_{\text{dat}} \Delta \), the thesis follows trivially by applying HEAP.

• \( e = (\lambda : e') \). By hypothesis and by rule R-LABEL

\[
h, \lambda : e' \xrightarrow{r} h, e' [\lambda : e'/\text{continue } \lambda]
\]

By hypothesis and by rule CONFIG \( \Delta \vdash h \) and \( \Delta \vdash \lambda : e' : U \Downarrow \Delta' \). By inversion and typing rule LEXPR we get

\[
\begin{align*}
\Delta'' & \vdash e' : U \Downarrow \Delta', \lambda : X \\
\Delta & = \{ r : C[\mu X.S] \mid r : C[S] \in \Delta'' \} \cup \{ r : U' \mid r : U' \in \Delta'' \text{ and } U' \neq C[S'] \} \\
\end{align*}
\]

By rule TV-HE \( \Delta \xrightarrow{r} \Delta \). Since \( \Delta \vdash h \), it remains to prove that \( \Delta \vdash e' [\lambda : e'/\text{continue } \lambda] : U \Downarrow \Delta' \). From the second case of the Substitution Lemma A.3 we get:

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

with

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

From the definition of \( \Delta''' \) we can obtain that \( \Delta''' = \Delta \) as required. We conclude by rule CONFIG.

### 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. Travel1), 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:

```plaintext
1 global protocol BuyTicket(role R, role A, role F){
2 request(Travel) from R to A;
3 quote(Price) from A to R;
4 check(Price) from R to F;
5 choice at F {
6   approve(Code) from F to R,A;
7   ticket(String) from A to R;
8   invoice(Code) from A to F;
9   payment(Price) from F to A;
10 } or {
11   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 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;
    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 BuyTicket_R local projection into the following .mungo files:

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

The RProtocol definition generated by StMungo is as follows:

```java
typestate RProtocol {
  State0 = {
    void send_requestTravelToA(Travel): State1 }
  State1 = {
    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 }
}
```

```java
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
class StackUser {

    Stack pushN(Stack x) {
        x.push(2); x
    }

    Stack popAll(Stack x) {
        loop :
            switch(x.isEmpty()) {
                case TRUE: x
                case FALSE: x.pop(); continue loop
            }
    }
}
```

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

```
PmR  \Delta_3 = s : Stack[S], k : Stack[end]                                                                                     New  \Delta_3 = s : Stack[end], k : Stack[end]                                                                  \Delta_3 + s = new Stack : void + \Delta_0
S_0
```

C.2 Recursion and Choice
Consider a class StackUser that defines methods that use a Stack object:

```java
class StackUser {

    Stack pushN(Stack x) {
        x.push(2); x
    }

    Stack popAll(Stack x) {
        loop :
            switch(x.isEmpty()) {
                case TRUE: x
                case FALSE: x.pop(); continue loop
            }
    }
}
```

**Method Stack popAll(Stack):** Consider the input typing context $\Delta_0 = x : Stack[end], this : StackUser[end]$ 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:

```
\Delta_1 = x : Stack[int pop() : X], this : StackUser[X]
\Delta_1 + x.pop() : int + \Delta_1' \quad \text{CONTINUE}
\Delta_1' = x : Stack[X], this : StackUser[X]
\Delta_1 + \text{continue loop : bot + } \Delta_0, \text{loop : } X \quad \text{SEQ}
\Delta_2 = x : Stack[S], this : StackUser[end], loop : X
\Delta_2 + x : Stack[S] + \Delta_0, \text{loop : } X \quad \text{SEQ}
\Delta_3 = x : Stack[Choice isEmpty() : join(S, int pop() : X)],
\quad \text{this : StackUser[join(end, X)]}
\Delta_3 + x.isEmpty() : Choice + \text{join}(\Delta_1, \Delta_2)
\Delta_3 + e : Stack[S] + \Delta_0, \text{loop : } X
\Delta_4 = x : Stack[\mu X. Choice is Empty() : join(S, int pop() : X)],
\quad \text{this : StackUser[\mu X.join(end, X)]}
\Delta_4 + \text{loop : e : Stack[S] + } \Delta_0 \quad \text{LEXPR}
\Delta = x : Stack[\mu X. Choice is Empty() : join(S, int pop() : X)],
\quad \text{this : StackUser[end]}
\Delta + \text{loop : e : Stack[S] + } \Delta_0 \quad \text{EQUIV}
```
Method Stack pushN(Stack): Assume a typing context \( \Delta_0 = x : \text{Stack[end]}, \mathit{this} : \text{StackUser[[Stack popAll(Stack) : end]]} \) The inference tree for method Stack pushN(Stack) is:

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

C.3 Method Call
Consider the code

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

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

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

C.4 Class inference
The inference tree for class StackUser is:

\[
\begin{align*}
\text{Class} & \quad \text{Set-St} \\
\text{Method-St} & \quad \text{LEXPR} \\
\text{Stack} & \quad \text{popAll(Stack) method body inference} \\
\Delta_1 = x : \text{Stack}[[\mu X.[S, \text{int pop()} : X]], \text{this} : \text{StackUser[end]} \\
\Delta_1 \vdash \text{Stack popAll(Stack x) } \{\text{loop : e} : U + \text{this : StackUser[end]} \\
\text{End-St} & \quad \text{Method-St} \\
& \quad \text{StackUser[end]} \\
\Delta & \vdash \text{StackUser[[Stack popAll(Stack) : end]]} \\
\Delta_1 = s : \text{Stack[end]}, c : \text{StackUser[[Stack popAll(Stack) : end]]}, x : \text{Stack[end]} \\
\Delta \vdash s = c . \text{pushN}(s) : \text{void} + \Delta_0 \\
\text{Seq} & \quad \text{Stack} pushN(Stack) method body inference \\
\Delta_2 = x : \text{Stack}[[\text{void push(int) : S}]], \text{this} : \text{StackUser[[Stack popAll(Stack) : end]]} \\
\Delta_2 \vdash \text{Stack pushN(Stack x) } \{x . \text{push(2); x} : U + \text{this : StackUser[[Stack popAll(Stack) : end]]} \\
& \quad \text{StackUser[[Stack pushN(Stack) : [Stack popAll(Stack) : end]]]} \\
& \quad \text{StackUser[[Stack pushN(Stack) : [Stack popAll(Stack) : end]]]}
\end{align*}
\]

\[\text{Box}\]

\[\text{Box}\]

\[\text{Box}\]

\[\text{Box}\]