Typechecking Protocols with Mungo and StMungo

Dimitrios Kouzapas Ornela Dardha Roly Perera Simon J. Gay
School of Computing Science, University of Glasgow, UK
{Dimitrios.Kouzapas, Ornela.Dardha, Roly.Perera, Simon.Gay}glasgow.ac.uk

ABSTRACT
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 can communicate with a real-world SMTP server.

Categories and Subject Descriptors
D.3.2 [**Language Classifications**]: Object-oriented languages; D.3.1 [**Formal Definitions and Theory**]; F.3.2 [**Semantics of Programming Languages**]: Operational semantics; F.3.3 [**Studies of Program Constructs**]: Type structure

Keywords
Session types, object-oriented programming, typestate, protocols, typestate inference.

1. INTRODUCTION
In this paper we present two tools that extend the Java development process with support for static typechecking of communication protocols. Mungo[1] extends Java with *typestate* definitions, which associate classes with state machines defining permitted sequences of method calls [43]. 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.

[1]Saint Mungo is the founder and patron saint of the city of Glasgow.

StMungo (Scribble-to-Mungo) uses this typestate feature to connect 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], that 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 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

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 provided 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>}}
```

[2]The tool is developed and maintained by the first author and can be downloaded from our web page [1].

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--); } 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 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 `isEmpty`: in the `NONEMPTY` 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:

```
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:

```
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.

```
1  class StackUser2 {
2    private Stack s;
3    StackUser2() { s = new Stack(); }
4    boolean pushN(int n)
5    { do{ s.push(n--); }while(n>0);
6      return true; }
7    void popAll()
8    { loop : do {
9        System.out.println(s.pop());
10       switch(s.isEmpty()) {
11         case EMPTY: break loop;
12         case NONEMPTY: continue loop;
13     } while(true); }
14   void finish() { s.deallocate(); }
15   public static void main(String[] args)
16   { StackUser2 su = new StackUser2();
17     if(su.pushN(15)||su.pushN(32))
18       su.pushN(32);
19       su.popAll(); su.finish();
20   }
21 }
```

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`:

```
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) tool[3], 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.

[3]The 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 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 [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.

```
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 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.

```
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. StMungo converts the `BuyTicket_R` local protocol into the `RProtocol` typestate protocol:

```
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}}
```

StMungo generates an API for this role, class `RRole` given in [30], which provides an implementation of `RProtocol`. When instantiated, it connects to the other role objects in the session (`ARole` and `FRole`). The method calls, describing the messages exchanged with the other roles, follow the interleaving specified by the `RProtocol` typestate. Alternatively, the developer may choose to ignore this API (and the Mungo socket library that it depends on), and use only the generated typestate protocols to develop his/her own implementation. He/she also has the ability to further refine the generated state machine, e.g., give appropriate names to states, or use anonymous states to have a coarser state refinement.

4. CASE STUDY: TYPECHECKING SMTP

In order to show the practicality and robustness of our StMungo and Mungo toolchain, we have developed a substantial case study in which we statically typecheck an SMTP client. We use this client to communicate with the `gmail` server. The full source code of the SMTP client can be found in [1].

SMTP (Simple Mail Transfer Protocol) is an Internet standard electronic mail transfer protocol, which typically runs over a TCP (Transmission Control Protocol) connection. We consider the version defined in RFC 5321 [42]. An SMTP interaction consists of an exchange of text-based commands between the client and the server. For example, the client sends the `EHLO` command to identify itself and open the connection with the server. The commands `MAIL FROM` : <address> and `RCPT TO` : <address> specify the e-mail address of the sender and the receiver of the e-mail, respectively. The `DATA` command allows the client to specify the text of the e-mail. The `QUIT` command is used to terminate the session and close the connection. The responses from the server have the following format: three digits followed by an optional dash "-", such as `250-`, and then some text, like OK. The server might reply to `EHLO` with `250` <text> or to `MAIL FROM` or `RCPT TO` with `250` OK.

To typecheck the SMTP protocol using StMungo and Mungo, we first represent the text-based commands as messages in a Scribble global protocol, based on Hu's work [27].

```
global protocol SMTP(role S, role C) {
  // Global interaction between
  // server and client.
  _220(String) from S to C;
  rec X1 {
    choice at S {
      _250dash(String) from S to C;
      continue X1; }
    or {
      250(String) from S to C; ... }
    ... } }
```

Then, we use the Scribble tool to validate and project the above global protocol into local protocols, one for each role. We focus only on the client side and describe in the following the `SMTP_C` local protocol. This fragment of code of the SMTP describes a loop (`rec X1`), in which the server `S` performs a choice between the messages `_250DASH` and `_250`. Next, other loops follow (`rec Z1` and `rec Z3`), where in the second one the client `C` chooses among the messages `SUBJECT`, to send the subject, `DATALINE`, to send a line of text, or `ATAD` to terminate the e-mail by sending a dot.

```
local protocol SMTP_C(role S, self C) {
  _220(String) from S; ...
    rec X1 {
      choice at S {
        _250dash(String) from S;
        continue X1; }
      or {
        _250(String) from S; ...
        rec Z1 {
        ... data(String) to S; ...
        rec Z3 {
          choice at C {
            subject(String) to S;
14             continue Z3; }
15           or {
16             dataline(String) to S;
17             continue Z3; }
18           or {
19             atad(String) to S;
20             _250(String) from S;
21             continue Z1; }
22       } } }    ...   } }
```

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.

```
1  typestate CProtocol {
2    State0={String receive_220StringFromS():
3              State1}
4    ...
5    State3=
6      {Choice1 receive_Choice1LabelFromS():
7         < _250DASH:State4, _250:State5 >}
8    State4=
9      {String receive_250dashStringFromS():
10        State3}
11   State5={String receive_250StringFromS():
12             State6}
13   ...
14   State27={void send_dataStringToS(String):
15             State28}
16   ...
17   State29={void send_SUBJECTToS():State30,
18            void send_DATALINEToS(): State31,
19            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:

```
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:

```
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:

```
enum 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 5 in CRole, respectively.

```
1  @Typestate("CProtocol") class CRole {
2    private Socket socketS = null; ...
3    public CRole()
4    { socketS =
5        new Socket("smtp.gmail.com", 587);
6      ...}
7     /* 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 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.

```
1  public static void main(String[] args) {
2   CRole currentC =  new CRole();
3   ... _Z3:
4   do{ ...
5    switch(/*label to be sent*/) {
6      case /*SUBJECT*/:
7        currentC.send_SUBJECTToS();
8        String subject = // input subject;
9        currentC.send_subjectStringToS
10                 (/*subject*/);
11       continue _Z3;
12     case /*DATALINE*/:
13     case /*ATAD*/:
14       currentC.send_ATADToS();
15       currentC.send_atadStringToS
16                 (/*single dot*/);
17       String _250msg =
18           currentC.receive_250StringFromS();
19       continue _Z1; }
20   } 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

$$
\begin{array}{rcl}
D & ::= & \texttt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \mid \texttt{enum}\ E\ \{\widetilde{l}\} \\
S & ::= & \widetilde{H} \mid \mu X.S \mid X \\
H & ::= & T\ m(T) : S \mid E\ m(T) : \langle S_l \rangle_{l \in E} \\
T & ::= & C \mid E \mid \texttt{bool} \mid \texttt{void} \\
F & ::= & T\ f \\
M & ::= & T\ m(T\ x)\ \{e\} \\
r & ::= & \texttt{this} \mid r.f \mid x \\
c & ::= & l \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{null} \mid * \\
e & ::= & c \mid r \mid r.m(e) \mid r.f = e \mid e; e \mid r.f = \texttt{new}\ C \\
 & \mid & \lambda : e \mid \texttt{continue}\ \lambda \\
 & \mid & \texttt{switch}\ (e)\ \{e_l\}_{l \in E} \mid \texttt{if}\ (e)\ e\ \texttt{else}\ e
\end{array}
$$

Figure 1: Top-level syntax

$$
\begin{array}{rcl@{\qquad}rcl}
o & ::= & C[\widetilde{f : o}] \mid c & r & ::= & \texttt{root} \mid r.f \\
e & ::= & \ldots \mid e@r & v & ::= & c \mid r \\
S & ::= & \ldots \mid \langle S_l \rangle_{l \in E} & s & ::= & T\ m(T) \mid E\ m(T) : l \mid l
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E} & ::= & [\,] \mid r.m(\mathcal{E}) \mid r.f = \mathcal{E} \mid \mathcal{E}; e \mid \texttt{switch}\ (\mathcal{E})\ \{e_l\}_{l \in E} \\
 & \mid & \mathcal{E}@r \mid \texttt{if}\ (\mathcal{E})\ e\ \texttt{else}\ e \\
\ell & ::= & r.f.\texttt{new}\ C \mid r.\langle l \rangle \mid r.T\ m\ T' \mid r.f = v \mid \tau \mid \texttt{if}
\end{array}
$$

Figure 2: Runtime syntax

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 $\widetilde{\cdot}$ to denote a possibly empty set of elements that range over the subject meta-variable. A program is a set of *type declarations* $\widetilde{D}$, each of which declares either a class or an enumerated type. A class declaration defines a *class* named C with typestate specification S, fields $\widetilde{F}$ and methods $\widetilde{M}$. An enumeration declaration defines an *enumerated type* named E with a non-empty set $\widetilde{l}$ 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 $\widetilde{D}$, which can be accessed by the following functions: given that $\texttt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \in \widetilde{D}$ we define $\mathsf{fields}(C) = \widetilde{F}$, $\mathsf{methods}(C) = \widetilde{M}$, $\mathsf{typestate}(C) = S$; and $\mathsf{enums}(E) = \widetilde{l}$ if $\texttt{enum}\ E\ \{\widetilde{l}\} \in \widetilde{D}$. 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* $\widetilde{H}$ 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* $E\ m(T) : \langle l : S_l \rangle_{l \in E}$ with the method signature defining the transition to one of the possible states $\langle S_l \rangle_{l \in E}$; 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 $\mu X.X$ 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 f associated with a type T. A method declaration $T\ m(T'\ x)\ \{e\}$ 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 `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 `null`, which is assignable to any class type, a `bool` 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 $r.f = \texttt{new}\ C$, 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 λ.

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* $C[\widetilde{f : o}]$, where C is the class of the object and $\widetilde{f : o}$ 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 `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 `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 $\mathcal{E}[e]$, 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 $\mathcal{E}$ with an expression e.

The operational semantics is annotated with labels ℓ that denote the creation of a new object ($r.f.\texttt{new}\ C$), an enum value choice ($r.\langle l \rangle$), method call ($r.T\ m\ T'$), assigning a field ($r.f = v$), the conditional label (`if`), and the silent label (τ). The definition of states is extended to the set of enum values $\langle l : S_l \rangle_{l \in E}$ and we define action labels s 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: $h(\texttt{root}) = h$; $h(r.f) = o$ and $h\{r.f \mapsto o'\} = h\{r \mapsto C[\widetilde{f : o}, f : o']\}$ if $h(r) = C[\widetilde{f : o}, f : o]$. The root object is accessed via $h(\texttt{root})$. 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-

$$\textsc{R-Seq}\ \frac{}{h, (v; e) \xrightarrow{\tau} h, e} \qquad \textsc{R-True}\ \frac{}{h, \texttt{if}\,(\texttt{tt})\ e_1\ \texttt{else}\ e_2 \xrightarrow{\texttt{if}} h, e_1} \qquad \textsc{R-False}\ \frac{}{h, \texttt{if}\,(\texttt{ff})\ e_1\ \texttt{else}\ e_2 \xrightarrow{\texttt{if}} h, e_2}$$

$$\textsc{R-New}\ \frac{(\mathsf{fields}(C) = \widetilde{T\ f})}{h, r.f = \texttt{new}\ C \xrightarrow{r.f.\texttt{new}\ C} h\{r.f \mapsto C[\widetilde{f : \mathsf{init}(T)}]\}, *} \qquad \textsc{R-AsgnC}\ \frac{}{h, r.f = c \xrightarrow{\tau} h\{r.f \mapsto c\}, *}$$

$$\textsc{R-Value}\ \frac{(v \neq l)}{h, v@r \xrightarrow{\tau} h, v} \qquad \textsc{R-AsgnR}\ \frac{(h' = h\{r' \mapsto \texttt{null}\})}{h, r.f = r' \xrightarrow{r.f=r'} h'\{r.f \mapsto h(r')\}, *} \qquad \textsc{R-Call}\ \frac{(h(r) = C[\widetilde{f : o}]\ \wedge\ T\ m(T'\ x)\ \{e\} \in \mathsf{methods}(C))}{h, r.m(v) \xrightarrow{r.T\ m\ T'} h, e\{v/x\}\{r/\texttt{this}\}@r}$$

$$\textsc{R-Switch}\ \frac{(l' \in E)}{h, \texttt{switch}\ (l'@r)\ \{e_l\}_{l \in E} \xrightarrow{r.\langle l' \rangle} h, e_{l'}} \qquad \textsc{R-Label}\ \frac{}{h, \lambda : e \xrightarrow{\tau} h, e\{e/\texttt{continue}\ \lambda\}} \qquad \textsc{R-Ctx}\ \frac{h, e \xrightarrow{\ell} h', e'}{h, \mathcal{E}[e] \xrightarrow{\ell} h', \mathcal{E}[e']}$$

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 τ label and proceeds with the evaluation of e. Rules R-True and R-False are the usual rules for the `if` ...`else` expression and are annotated with label `if`. Rule R-New is labelled with $r.f.$`new` C and overwrites the contents of the field $r.f$ by a new object $C[\widetilde{f = \mathsf{init}(T)}]$ whose fields are all initialised to the value $\mathsf{init}(T)$, where T is the type of the field, defined as: $\mathsf{init}(C) = \texttt{null}$; $\mathsf{init}(E) = E_{\mathsf{init}}$; $\mathsf{init}(\texttt{bool}) = \texttt{ff}$; and $\mathsf{init}(\texttt{void}) = *$, where for every enumerated type E we require there to be a distinguished element $E_{\mathsf{init}} \in \mathsf{enums}(E)$. The result of R-New is the `void` value $*$. The object is constructed at a location within an already existing object $r.f$. 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-AsgnC rule is labelled with τ 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-AsgnR rule is to *relocate* the object from r' to $h.r$, leaving `null` at its old location. The annotation label for R-AsgnR 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 $C[\widetilde{f : o}]$, 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 $@r$, recording the fact that r is the active receiver. The active receiver tag $@r$ on a value is removed using a τ 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 $r.\langle l' \rangle$, 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 τ, and says that a labelled expression $\lambda : e$ discards the λ and substitutes a copy of the labelled expression for every occurrence of `continue` λ 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 $\mathsf{fields}(C) = \widetilde{T\ f}$, each field of C is initialised with the corresponding value $\mathsf{init}(T)$. Execution can then be initiated using a top-level expression that substitutes path `this` with path `root`.

6. TYPESTATE INFERENCE

In this section we formalise a typestate inference system and prove its safety properties. The system presented here infers a *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 Δ, are defined below:

$$\begin{array}{lcl} U & ::= & C[S] \mid E \mid \texttt{bool} \mid \texttt{void} \mid \texttt{bot} \\ \Delta & ::= & \emptyset \mid \Delta, r : U \mid \Delta, \lambda : X \end{array}$$

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 `bot`. Typing context Δ is a partial function from runtime paths r to types U, and expression labels λ to recursive type variables X. A type U that is not a class type is referred to as *constant type*.

The inference system uses a subtyping relation $\leqslant_{\mathsf{sbt}}$ and a binary operator $\mathsf{join}(\cdot, \cdot)$.

Definition 1. ($\leqslant_{\mathsf{sbt}}$, $=_{\mathsf{sbt}}$, join) The following relations are defined on typestates, inferred types and typing contexts.

- The *subtyping* relation $\leqslant_{\mathsf{sbt}}$ is defined by the rules in Fig. 4.
- The *equivalence* relation is defined as $=_{\mathsf{sbt}} = \leqslant_{\mathsf{sbt}} \cap \leqslant_{\mathsf{sbt}}^{-1}$.
- The *join* operator $\mathsf{join}(\cdot, \cdot)$ is defined by the rules in Fig. 5.

Subtyping on typestates is essentially a simulation relation and is given in an algorithmic style. It coinductively constructs a set $\mathcal{R}$ 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

$$\textsc{S-Start}\ \frac{\emptyset \vdash S \leqslant_{\mathsf{sbt}} S'}{S \leqslant_{\mathsf{sbt}} S'} \qquad \textsc{S-End}\ \frac{}{\mathcal{R} \vdash \mathsf{end} \leqslant_{\mathsf{sbt}} \mathsf{end}} \qquad \overset{\textsc{S-Terminate}}{\frac{(S, S') \in \mathcal{R}}{\mathcal{R} \vdash S \leqslant_{\mathsf{sbt}} S'}} \qquad \overset{\textsc{S-Method}}{\frac{\mathcal{R} \vdash S \leqslant_{\mathsf{sbt}} S'}{\mathcal{R} \vdash T'\ m(T) : S \leqslant_{\mathsf{sbt}} T'\ m(T) : S'}}$$

$$\textsc{S-Rec1}\ \frac{\mathcal{R} \cup \{(S, \mu X.S')\} \vdash S \leqslant_{\mathsf{sbt}} S'\{\mu X.S'/X\}}{\mathcal{R} \vdash S \leqslant_{\mathsf{sbt}} \mu X.S'} \qquad \textsc{S-Set}\ \frac{\widetilde{H} \neq \emptyset \qquad \forall H \in \widetilde{H}, \exists H' \in \widetilde{H'}. \mathcal{R} \vdash H \leqslant_{\mathsf{sbt}} H'}{\mathcal{R} \vdash \widetilde{H} \leqslant_{\mathsf{sbt}} \widetilde{H'}}$$

$$\overset{\textsc{S-Enum}}{\frac{\forall l \in E. \mathcal{R} \vdash S_l \leqslant_{\mathsf{sbt}} S'_l}{\mathcal{R} \vdash E\ m(T) : \langle S_l \rangle_{l \in E} \leqslant_{\mathsf{sbt}} E\ m(T) : \langle S'_l \rangle_{l \in E}}} \qquad \overset{\textsc{S-Class}}{\frac{S \leqslant_{\mathsf{sbt}} S'}{C[S] \leqslant_{\mathsf{sbt}} C[S']}} \qquad \overset{\textsc{S-Bot}}{\frac{}{\mathtt{bot} \leqslant_{\mathsf{sbt}} U}} \qquad \overset{\textsc{S-Grnd}}{\frac{U \in \{E, \mathtt{bool}, \mathtt{void}\}}{U \leqslant_{\mathsf{sbt}} U}} \qquad \overset{\textsc{S-Empty}}{\frac{}{\emptyset \leqslant_{\mathsf{sbt}} \Delta}}$$

$$\overset{\textsc{S-Delta}}{\frac{\Delta \leqslant_{\mathsf{sbt}} \Delta' \qquad U \leqslant_{\mathsf{sbt}} U'}{\Delta, r : U \leqslant_{\mathsf{sbt}} \Delta, r : U'}} \qquad \overset{\textsc{S-Lambda}}{\frac{\Delta \leqslant_{\mathsf{sbt}} \Delta'}{\Delta, \lambda : X \leqslant_{\mathsf{sbt}} \Delta', \lambda : X}}$$

Figure 4: Subtyping relation (symmetric rule S-Rec2 omitted)

$$\mathsf{join}(H, \{H'\} \cup \widetilde{H}) = \begin{cases} T\ m(T') : \mathsf{join}(S, S') \cup \widetilde{H} \\ \quad \text{if } H = T\ m(T') : S \text{and } H' = T\ m(T') : S' \\ E\ m(T) : \langle l : \mathsf{join}(S_l, S'_l) \rangle_{l \in E} \cup \widetilde{H} \\ \quad \text{if } H = E\ m(T) : \langle S_l \rangle_{l \in E} \text{and } H' = E\ m(T) : \langle S'_l \rangle_{l \in E} \\ \{H, H'\} \cup \widetilde{H} \qquad \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathsf{join}(\widetilde{H} \cup \{H\}, \widetilde{H'}) &= \mathsf{join}(\widetilde{H}, \mathsf{join}(H, \widetilde{H'})) \\ \mathsf{join}(\mathsf{end}, \mathsf{end}) &= \mathsf{end} \\ \mathsf{join}(\mu X.S_1, S_2) &= \mathsf{join}(S_1\{\mu X.S_1/X\}, S_2) \\ \mathsf{join}(C[S], C[S']) &= C[\mathsf{join}(S, S')] \\ \mathsf{join}(U, \mathtt{bot}) = \mathsf{join}(\mathtt{bot}, U) &= U \\ \mathsf{join}(U, U) &= U \qquad U \in \{E, \mathtt{bool}, \mathtt{void}\} \\ \mathsf{join}(\Delta, \Delta') &= \{r : C[\mathsf{join}(S, S')] \mid r : C[S] \in \Delta, \\ &\qquad r : C[S'] \in \Delta'\} \cup\ \Delta \backslash \Delta' \cup \Delta' \backslash \Delta \end{aligned}$$

Figure 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 $\leqslant_{\mathsf{sbt}}$ 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 $S \xrightarrow{s} S'$ is defined by the following rules:

$$T\ m(T) : S \xrightarrow{T\ m(T)} S \qquad \frac{l' \in \mathsf{enums}(E)}{E\ m(T) : \langle S_l \rangle_{l \in E} \xrightarrow{E\ m(T):l'} S_{l'}}$$

$$\frac{H \in \widetilde{H} \qquad H \xrightarrow{s} S}{\widetilde{H} \xrightarrow{s} S} \qquad \frac{S\{\mu X.S/X\} \xrightarrow{s} S'}{\mu X.S \xrightarrow{s} S'} \qquad \frac{l' \in \mathsf{enums}(E)}{\langle S_l \rangle_{l \in E} \xrightarrow{l'} S_{l'}}$$

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

Before introducing the typestate inference rules, we define the typing judgements:

$$\Delta \vdash e : U \dashv \Delta' \qquad \Delta \vdash C[S] \qquad \vdash \mathtt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \qquad \vdash \widetilde{D}$$

The first one is the typing judgement for expressions. The judgement is read from right to left. It takes as input the typing context Δ' and the expression e, and algorithmically computes the type U. The effects of the expression on Δ' are then captured in Δ. However, it is interesting to notice that the judgement can also be read from left to right in a type system fashion, where the expression "consumes" Δ in order to produce Δ'. The second judgement infers the typestates of the fields of a class when the class is used according to its declared typestate. The last two typing judgements state the well-formedness of classes and, respectively, programs.

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:

$$\mathtt{s = new\ Stack;\ k = s} \qquad (1)$$

The code expression matches rule Seq. We assume

$$\Delta_0 = \mathtt{s} : \mathtt{Stack}[\mathsf{end}], \mathtt{k} : \mathtt{Stack}[S]$$

$$\textsc{Void}\ \frac{}{\Delta \vdash * : \mathtt{void} \dashv \Delta} \qquad \textsc{Bool}\ \frac{}{\Delta \vdash \mathtt{tt}, \mathtt{ff} : \mathsf{bool} \dashv \Delta} \qquad \textsc{Enum}\ \frac{l \in \mathsf{enums}(E)}{\Delta \vdash l : E \dashv \Delta}$$

$$\textsc{Null}\ \frac{\mathtt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \in \widetilde{D}}{\Delta \vdash \mathsf{null} : C[\mathsf{end}] \dashv \Delta} \qquad \textsc{Weaken}\ \frac{\Delta \vdash e : U \dashv \Delta' \qquad r \notin dom(\Delta')}{\Delta \vdash e : U \dashv \Delta', r : C[\mathsf{end}]}$$

$$\textsc{Strengthen}\ \frac{\Delta, r : C[\mathsf{end}] \vdash e : U \dashv \Delta'}{\Delta \vdash e : U \dashv \Delta'} \qquad \textsc{PathC}\ \frac{U \neq C[S]}{\Delta, r : U \vdash r : U \dashv \Delta, r : U}$$

$$\textsc{PathR}\ \frac{r \neq \mathtt{this}}{\Delta, r : C[S] \vdash r : C[S] \dashv \Delta, r : C[\mathsf{end}]} \qquad \textsc{Equiv}\ \frac{\Delta \vdash e : U \dashv \Delta' \qquad \Delta =_{\mathsf{sbt}} \Delta''}{\Delta'' \vdash e : U \dashv \Delta'}$$

$$\textsc{AsgnC}\ \frac{U \neq C[S] \qquad \Delta \vdash e : U \dashv \Delta', r : U}{\Delta \vdash r = e : \mathtt{void} \dashv \Delta', r : U} \qquad \textsc{AsgnR}\ \frac{r \neq \mathtt{this} \qquad \Delta \vdash e : C[S] \dashv \Delta', r : C[\mathsf{end}]}{\Delta \vdash r = e : \mathtt{void} \dashv \Delta', r : C[S]}$$

$$\textsc{New}\ \frac{r \neq \mathtt{this} \qquad S \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C) \qquad \forall r.f : C'[S'] \in \Delta \implies S' = \mathsf{end}}{\Delta, r : C[\mathsf{end}] \vdash r = \mathtt{new}\ C : \mathtt{void} \dashv \Delta, r : C[S]}$$

$$\textsc{Seq}\ \frac{\Delta \vdash e_1 : U' \dashv \Delta'' \qquad \Delta'' \vdash e_2 : U \dashv \Delta' \qquad U' \neq C[S] \qquad U' \neq \mathsf{bot}}{\Delta \vdash e_1; e_2 : U \dashv \Delta'}$$

$$\textsc{Call}\ \frac{T\ m(T'\ x)\ \{e'\} \in \mathsf{methods}(C) \quad S' =_{\mathsf{sbt}} S \quad \Delta'', r : C[S'], x : U' \vdash e'\{r/\mathtt{this}\} : U \dashv \Delta', r : C[S], x : \mathsf{initT}(T') \quad \Delta \vdash e : U' \dashv \Delta'', r : C[\{T\ m(T') : S\}]}{\Delta \vdash r.m(e) : U \dashv \Delta', r : C[S]}$$

$$\textsc{Switch}\ \frac{\forall l \in E.\ \Delta_l, r : C[S_l] \vdash e_l : U_l \dashv \Delta' \qquad \Delta \vdash r.m(e) : E \dashv \Delta'' \qquad \Delta'' = \mathsf{join}(\{\Delta_l\}_{l \in E})}{\Delta \vdash \mathtt{switch}\ (r.m(e))\ \{e_l\}_{l \in E} : \mathsf{join}(\{U_l\}_{l \in E}) \dashv \Delta'}$$

$$\textsc{If}\ \frac{\Delta_1 \vdash e_1 : U_1 \dashv \Delta' \qquad \Delta_2 \vdash e_2 : U_2 \dashv \Delta' \qquad \Delta'' = \mathsf{join}(\Delta_1, \Delta_2) \qquad \Delta \vdash e : \mathsf{bool} \dashv \Delta''}{\Delta \vdash \mathtt{if}\ (e)\ e_1\ \mathtt{else}\ e_2 : \mathsf{join}(U_1, U_2) \dashv \Delta'}$$

$$\textsc{LExpr}\ \frac{\Delta'' \vdash e : U \dashv \Delta', \lambda : X \qquad X \text{ fresh} \qquad \Delta = \{r : C[\mu X.S] \mid r : C[S] \in \Delta''\} \cup \{r : U' \mid r : U' \in \Delta'' \text{ and } U' \neq C'[S']\}}{\Delta \vdash \lambda : e : U \dashv \Delta'}$$

$$\textsc{Continue}\ \frac{\Delta = \{r : C[X] \mid r : C[S] \in \Delta'\} \cup \{r : U \mid r : U \in \Delta' \text{ and } U \neq C'[S']\}}{\Delta \vdash \mathtt{continue}\ \lambda : \mathsf{bot} \dashv \Delta', \lambda : X}$$

Figure 6: Typestate inference rules for expressions

as an input typing context and $S \preccurlyeq_{\mathsf{sbt}}$ `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. In order to type the second expression by AsgnR we need to infer a typestate for `s`. The derivation is:

$$\frac{\textsc{PathR}\ \dfrac{}{\Delta_2 = \mathtt{s} : \mathtt{Stack}[S], \mathtt{k} : \mathtt{Stack}[\mathsf{end}] \vdash \mathtt{s} : \mathtt{Stack}[S] \dashv \Delta_1}}{\Delta_2 \vdash \mathtt{k\ =\ s} : \mathtt{void} \dashv \Delta_0 = \mathtt{s} : \mathtt{Stack}[\mathsf{end}], \mathtt{k} : \mathtt{Stack}[S]}\ \textsc{AsgnR}$$

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

$$\Delta_2 = \mathtt{s} : \mathtt{Stack}[S], \mathtt{k} : \mathtt{Stack}[\mathsf{end}]$$

meaning that `k` has an inactive typestate before assignment. Rule PathR "guesses" a type for a path expression. However, the combination of PathR and AsgnR is the key to this inference since it enforces a match on the type of `s` in the output typing context Δ_2 and the type of `k` in the input typing context Δ_0. For the first expression in (1) we use rule New. By assumption we satisfy its premise; we have $S \preccurlyeq_{\mathsf{sbt}}$ `StackProtocol`, meaning path `s` is used according to the `StackProtocol` typestate (this is shown in Δ_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 first expression is not of type `bot` to forbid dead code after a `continue` λ 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:

$$\textsc{Seq}\ \frac{\textsc{New}\ \dfrac{S \preccurlyeq_{\mathsf{sbt}} \mathtt{StackProtocol} \qquad \Delta_3 = \mathtt{s} : \mathtt{Stack}[\mathsf{end}], \mathtt{k} : \mathtt{Stack}[\mathsf{end}]}{\Delta_3 \vdash \mathtt{s\ =\ new\ Stack} : \mathtt{void} \dashv \Delta_2} \qquad \textsc{AsgnR}\ \dfrac{\dots}{\Delta_2 \vdash \mathtt{k\ =\ s} : \mathtt{void} \dashv \Delta_0}}{\Delta_3 \vdash \mathtt{s\ =\ new\ Stack;\ k\ =\ s} : \mathtt{void} \dashv \Delta_0}$$

To preserve linearity, `s` and `k` exchange their typestates before and after assignment, as expected. If the type of `s` in Δ_0 is not inactive, it means that path `s` can be used after its assignment, thus violating linearity, as in the following code:

```
s = new Stack; k = s; s.push(5)          (3)
```

To conclude, in rules AsgnR and New the path `this` is not assignable. In rule PathR the path `this` is not inferrable.

The other rules for paths and assignments are as follows. Rule PathC 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 AsgnC follows the same line as 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 LExpr, Continue, Switch, and If. Consider the following class `StackUser` that defines methods that use a `Stack` object:

```
class StackUser:
{{Stack pushN(Stack):
    {Stack popAll(Stack):end}}}{
  Stack pushN(Stack x) { x.push(2); x }
  Stack popAll(Stack x)
  {loop:switch(x.isEmpty())
```

```
          {case EMPTY:x,
            case NOTEMPTY:x.pop();
                           continue loop}}}
```

and the input typing context:

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

The body of method `popAll` in line 4 is a labelled expression, and so rule LExpr applies. The premise requires an inference for the `switch` expression by using in input Δ_0 augmented with the assumption `loop` : X, where X is fresh. Let $\Delta_1 = \Delta_0, \mathtt{loop} : X$. LExpr closes all free occurrences of X in the output typing context. For the `switch` expression rule Switch 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 `switch`, namely Δ_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 `switch`. The condition should have an enumeration type that matches the type of the `switch` definition. Finally, the type of `switch` is the join of the types of its branches. For the `TRUE` branch we use rule PathR:

$$\textsc{PathR}\ \frac{\Delta_2 = \mathtt{x} : \mathtt{Stack}[S], \mathtt{this} : \mathtt{StackUser[end]}, \mathtt{loop} : X}{\Delta_2 \vdash \mathtt{x} : \mathtt{Stack}[S] \dashv \Delta_1}$$

For the `FALSE` branch we first use rule Seq and then rule Continue to infer the typestate of the `continue loop` expression. Continue requires `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:

$$\textsc{Continue}\ \frac{\Delta_3 = \mathtt{x} : \mathtt{Stack}[X], \mathtt{this} : \mathtt{StackUser}[X]}{\Delta_3 \vdash \mathtt{continue\ loop} : \mathtt{bot} \dashv \Delta_1}$$

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

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

We join the output typing contexts Δ_2 and Δ_4 of the `TRUE` and `FALSE` branch, respectively and use the result as an input typing context for the method call `x.isEmpty()`, as stated by the premises of Switch. The output typing context of Switch is:

$$\begin{aligned}\Delta_5 \ = \ & \mathtt{x} : \mathtt{Stack}[\{\mathtt{Choice\ isEmpty}() : \mathsf{join}(S, \mathtt{int\ pop}() : X)\}],\\ & \mathtt{this} : \mathtt{StackUser}[\mathsf{join}(\mathtt{end}, X)]\end{aligned}$$

To complete the inference of LExpr we close the recursive variable X in Δ_5 and obtain the output typing context for the labelled expression in lines 4-5, which is:

$$\begin{aligned}\Delta_6 \ = \ & \mathtt{x} : \mathtt{Stack}[\mu X.\mathtt{Choice\ isEmpty}() : \mathsf{join}(S, \mathtt{int\ pop}() : X)],\\ & \mathtt{this} : \mathtt{StackUser}[\mu X.\mathsf{join}(\mathtt{end}, X)]\end{aligned}$$

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

Rule If types the conditional expression in a similar way as rule 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 `bool`.

Method Call. Rule 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 initT, defined by $T \neq C \implies \mathsf{initT}(T) = T$ and $\mathsf{initT}(C) = C[\mathtt{end}]$.

Rule 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 Call, consider the following code that uses class `StackUser`:

```
s = c.pushN(s)
```

and the input typing context:

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

By applying rule AsgnR on the above assignment with input Δ_0, the output typing context in the premise of the rule is:

$$\begin{aligned}\Delta_1 \ = \ & \mathtt{s} : \mathtt{Stack[end]},\\ & \mathtt{c} : \mathtt{StackUser}[\{\mathtt{Stack\ popAll(Stack)} : \mathtt{end}\}]\end{aligned}$$

At this point we can apply rule Call on `c.pushN(s)` and have the following derivation:

$$\textsc{Call}\ \frac{\begin{array}{lr}\mathtt{Stack\ pushN(Stack\ x)}\ \{\mathtt{x.push(2); x}\} & \\ \qquad\qquad \in \mathsf{methods}(\mathtt{StackUser}) & (1)\\ \Delta_2 \vdash (\mathtt{x.push(2); x})\{\mathtt{c/this}\} : \mathtt{Stack}[S] \dashv \Delta_1 & (2)\\ \Delta \vdash \mathtt{s} : \mathtt{Stack}[\{\mathtt{void\ push(int)} : S\}] \dashv \Delta_3 & (3)\end{array}}{\Delta \vdash \mathtt{c.pushN(s)} : \mathtt{Stack}[S] \dashv \Delta_1}$$

The premise of Call, given in (1), performs a lookup in the methods of the class of the receiver, `ListCons`, to obtain the definition of method `Stack prod(Stack)`. Next, in (2), the premise infers a typestate for the body of the method in which 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:

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

Then, in (3) 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 PathR is used where Δ_3 also updates the type of the receiver:

$$\begin{aligned}\Delta_3 \ = \ & \mathtt{s} : \mathtt{Stack[end]},\\ & \mathtt{c} : \mathtt{StackUser}[\{\mathtt{Stack\ pushN(Stack)} :\\ & \qquad\qquad \{\mathtt{Stack\ popAll(Stack)} : \mathtt{end}\}\}]\\ \Delta \ = \ & \mathtt{Stack}[\{\mathtt{void\ push(int)} : S\}],\\ & \mathtt{c} : \mathtt{StackUser}[\{\mathtt{Stack\ pushN(Stack)} :\\ & \qquad\qquad \{\mathtt{Stack\ popAll(Stack)} : \mathtt{end}\}\}]\end{aligned}$$

Rule 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 $=_{\mathsf{sbt}}$. This is to respect the abstraction principle: the client would know how a method uses its receiver. For example, assume method `Stack pushN(Stack)` is defined as:

```
Stack pushN(Stack x)
          { x.push(2); x = this.popAll(x); x }
```

If we infer a typestate for the body of `Stack pushN(Stack)` with input context Δ_1 we get: an output typing context, Δ', such that:

$$\begin{aligned}\Delta'(\mathtt{c}) \ = \ & \mathtt{StackUser}[\mathtt{Stack\ popAll(Stack)} :\\ & \qquad\qquad \{\{\mathtt{Stack\ popAll(Stack)} : \mathtt{end}\}\}]\end{aligned}$$

$$\textsc{Method-St}\ \dfrac{\begin{array}{c}\Delta' \vdash C[S] \qquad T_1\ m(T_2\ x)\ \{e\} \in \mathsf{methods}(C) \qquad \forall \bar{\ell}, S' \xrightarrow{\bar{\ell}} S \implies S \xrightarrow{\bar{\ell}} S \\ \Delta, \mathtt{this} : C[S'], x : \mathsf{infer}(T_2) \vdash e : \mathsf{infer}(T_1) \dashv \Delta', \mathtt{this} : C[S], x : \mathsf{initT}(T')\end{array}}{\Delta \vdash C[\{T\ m(T') : S\}]}$$

$$\textsc{Set-St}\ \dfrac{\begin{array}{c}\forall H \in \widetilde{H}.\ \Delta_H \vdash C[\{H\}] \\ \Delta = \mathsf{join}(\{\Delta_H\}_{H \in \widetilde{H}})\end{array}}{\Delta \vdash C[\widetilde{H}]}$$

$$\textsc{Enum-St}\ \dfrac{\begin{array}{c}\forall l \in E.\ \Delta_l \vdash C[S_l] \qquad E\ m(T\ x)\ \{e\} \in \mathsf{methods}(C) \\ \Delta, x : C[S'] \vdash E\ m(T\ x)\ \{e\} : E \dashv \Delta'' \qquad \Delta'' = \mathsf{join}(\{\Delta_l\}_{l \in E})\end{array}}{\Delta \vdash C[E\ m(T) : \langle S_l \rangle_{l \in E}]}$$

$$\textsc{End-St}\ \dfrac{\begin{array}{c}\Delta = \{f : C'[\mathsf{end}] \mid C'\ f \in \mathsf{fields}(C)\} \cup \\ \{f : T \mid T\ f \in \mathsf{fields}(C) \text{ and } T \neq C\}\end{array}}{\Delta \vdash C[\mathsf{end}]}$$

$$\textsc{Rec-St}\ \dfrac{\begin{array}{c}\Delta' \vdash C[S] \qquad \Delta = \{r : C[\mu X.S] \mid r : C[S] \in \Delta'\} \cup \\ \{r : U \mid r : U \in \Delta' \text{ and } U \neq C'[S']\}\end{array}}{\Delta \vdash C[\mu X.S]}$$

$$\textsc{Var-St}\ \dfrac{\begin{array}{c}\Delta = \{f : C'[X] \mid C'\ f \in \mathsf{fields}(C)\} \\ \cup \{f : T \mid T\ f \in \mathsf{fields}(C) \text{ and } T \neq C\}\end{array}}{\Delta \vdash C[X]}$$

$$\textsc{Class}\ \dfrac{\begin{array}{c}\Delta \vdash C[S] \\ \forall f : C'[S] \in \Delta \implies S = \mathsf{end}\end{array}}{\vdash \mathtt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\}}$$

$$\textsc{Program}\ \dfrac{\forall D \in \widetilde{D} \qquad D = \mathtt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \implies \vdash D}{\vdash \widetilde{D}}$$

Figure 7: Typestate inference rules for methods, classes and programs

Given that $\Delta_1(\mathtt{c}) = \mathtt{StackUser}[\{\{\mathtt{Stack\ popAll(Stack)} : \mathsf{end}\}\}]$, it is revealed that the body of `Stack pushN(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 Set-St requires the inference and join of the typestates of all branches in an internal choice. Rule Method-St relies on the $\mathsf{infer}(T)$ definition that maps a type T to the corresponding inferred type U as: $T \neq C \implies \mathsf{infer}(T) = T$ and $\mathsf{infer}(C) = C[S]$, for some S. Rule 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 $\mathsf{infer}(T)$ is used to check that the return and parameter types of the method match the types of the inferred ones. As in Call, a self-call should preserve the typestate of the receiver up to type equivalence. Rule Enum-St is similar to rule Method-St. It requires the inference and join of the typestates of all the external choices and then infers the method prefix. Rule End-St requires all fields of the class to finish in the inactive typestate. Rules Rec-St and Var-St are similar to rules LExpr and Continue, where they bind and use a recursive variable, respectively. Rule 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 Class. A program is well-typed if all of its classes are well-typed, as stated by rule 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 Switch-AtR is similar to Switch, the difference being the condition of the switch, which is evaluated to an active receiver rather than a method call. Rule AtR infers a typestate for $e@r$, by first inferring a typestate for e. 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.

$$\textsc{Switch-AtR}\ \dfrac{\begin{array}{c}\forall l \in E.\ \Delta_l, r : C[S_l] \vdash e_l : U_l \dashv \Delta' \qquad \Delta \vdash e : E \dashv \Delta'', r : C[\langle S_l \rangle_{l \in E}] \\ \Delta'' = \mathsf{join}(\{\Delta_l\}_{l \in E})\end{array}}{\Delta \vdash \mathtt{switch}\ (e@r)\ \{e_l\}_{l \in E} : \mathsf{join}(\{U_l\}_{l \in E}) \dashv \Delta'}$$

$$\textsc{AtR}\ \dfrac{\Delta \vdash e : U \dashv \Delta'}{\Delta \vdash e@r : U \dashv \Delta'} \qquad \textsc{Config}\ \dfrac{\Delta \vdash h \qquad \Delta \vdash e : U \dashv \Delta'}{\Delta \vdash h, e : U \dashv \Delta'}$$

$$\textsc{Heap}\ \dfrac{\begin{array}{c}\forall r : U \in \Delta. \\ h(r) = o \qquad \Delta \vdash o : U \dashv \Delta\end{array}}{\Delta \vdash h} \qquad \textsc{Object}\ \dfrac{\begin{array}{c}\mathsf{typestate}(C) = S \\ S \xrightarrow{\bar{s}} S'\end{array}}{\Delta \vdash C[\widetilde{f : o}] : C[S'] \dashv \Delta}$$

Figure 8: Typestate inference rules for runtime syntax

Rule Object checks that the typestate of the objects in the context match the declared typestate of their class. Finally, rule Config 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

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 Ty-Id states that Δ remains unchanged under a τ-reduction. Rule Ty-New states that a path in Δ mapped to an inactive typestate reduces under $r.f.\mathtt{new}\ C$ and its typestate is updated accordingly. Rules Ty-AsgnR and Ty-AsgnC 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 Ty-Call performs a reduction of a method-prefixed typestate with the method prefix itself being the label. Similarly, rule Ty-Label reduces with an enumerated value for paths that have a runtime switch typestate. The behaviour of the `if` label is captured by rule Ty-If. In both the

$$\textsc{Ty-Id}\ \frac{}{\Delta \xrightarrow{\tau} \Delta} \qquad \textsc{Ty-If}\ \frac{\Delta' \preccurlyeq_{\mathsf{sbt}} \Delta}{\Delta \xrightarrow{\mathtt{if}} \Delta'} \qquad \textsc{Ty-AsgnC}\ \frac{}{\Delta \xrightarrow{r.f=c} \Delta}$$

$$\textsc{Ty-Call}\ \frac{}{\Delta, r : C[\{T\ m(T') : S\}] \xrightarrow{r.T\ m\ T'} \Delta, r : C[S]}$$

$$\textsc{Ty-AsgnR}\ \frac{}{\Delta, r.f : C[\mathsf{end}], r' : C[S] \xrightarrow{r.f=r'} \Delta, r.f : C[S], r' : C[\mathsf{end}]}$$

$$\textsc{Ty-New}\ \frac{(S \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C) \wedge \forall r.f.f' : C'[S'] \in \Delta.\ S' = \mathsf{end})}{\Delta, r.f : C[\mathsf{end}] \xrightarrow{r.f.\mathtt{new}\ C} \Delta, r.f : C[S]}$$

$$\textsc{Ty-Label}\ \frac{\Delta' \preccurlyeq_{\mathsf{sbt}} \Delta}{\Delta, r : C[\langle S_l \rangle_{l \in E}] \xrightarrow{r.\langle l' \rangle} \Delta', r : C[S_{l'}]}$$

Figure 9: Reduction relation on typing contexts

last two rules the result of the reduction is a subtype of the starting typing context.

We state the progress and subject reduction theorem in the following. The proof is given in [30].

Theorem 1. (*Progress and Subject Reduction*) Let a set of declarations $\widetilde{D}$ with $\vdash \widetilde{D}$. Assuming $\widetilde{D}$ is the program context, let e be a run time expression and suppose $\Delta \vdash h, e : U \dashv \Delta''$. Then, either e is a value, or there exist ℓ, h' and e' such that $h, e \xrightarrow{\ell} h', e'$, and there exist Δ' and U' such that $\Delta \xrightarrow{\ell} \Delta'$ and $\Delta' \vdash h', e' : U' \dashv \Delta''$ and $U' \preccurlyeq_{\mathsf{sbt}} U$.

Subject reduction requires that the trace of the execution of a program is included in the trace of the inferred typestates of the program. Furthermore, we require a progress property on expressions: an expression that is not a value can always reduce. As a corollary of Theorem 1, we further observe that the trace of the inferred context of a program is included in the declared typestate of the program. This is stated by the following.

Corollary 1. (*Coherence of Typestate Inference*) Let $\widetilde{D}$ be a set of declarations such that $\vdash \widetilde{D}$. Assuming $\widetilde{D}$ to be the ambient program context, let e be a run time expression and suppose $\Delta \vdash h, e : U \dashv \Delta'$. If $h, e \xrightarrow{\widetilde{\ell}\, r.f.\mathtt{new}\ C} h', e'$, for some $\widetilde{\ell}$, then $\Delta = \Delta'', r : C[\mathsf{end}]$ with $\Delta' = \Delta'', r : C[S]$ and $S \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C)$.

7. RELATED WORK

Session types and programming languages. The Session Java (SJ) language [28] builds on earlier work [15, 14, 17] to add binary session type channels to Java. SJ has been applied to a range of situations including scientific computation [38] 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 usecase 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.

Capecchi 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 [37] typechecks the operations of a library that implements multiparty session types using a restricted set of MPI [31] primitives. In contrast, our framework typechecks Java statements and expressions, instead of higher-level operations. The work in [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. [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, [34] 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 [40].

Typestate. There have been many efforts to add typestate to practical languages, since their introduction in [43]. 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, 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 classes, states are organised into an inheritance hierarchy. The most recent work [20, 45] 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. [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 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. 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 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. [32]. 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 StMungo 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 Voinea for her contribution to Mungo and StMungo. We thank Garrett Morris, Raymond Hu and Nobuko Yoshida for useful comments and discussion.

9. REFERENCES

[1] Mungo Webpage. http://www.dcs.gla.ac.uk/research/mungo/.

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *OOPSLA '09*, pages 1015–1022. ACM Press, 2009.

[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. In *OOPSLA '05*, pages 345–364. ACM Press, 2005.

[4] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA '08*, pages 227–244. ACM Press, 2008.

[5] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07*, pages 301–320. ACM Press, 2007.

[6] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *ECOOP '09*, volume 5653 of *Springer LNCS*, pages 195–219, 2009.

[7] Eric Bodden and Laurie J. Hendren. The Clara framework for hybrid typestate analysis. *Software Tools for Technology Transfer*, 14(3):307–326, 2012.

[8] Viviana Bono, Chiara Messa, and Luca Padovani. Typing 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.*, 410:142–167, 2009.

[10] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR'08*, volume 5201 of *Springer LNCS*, pages 402–417, 2008.

[11] Silvia Crafa and Luca Padovani. The chemical approach to typestate-oriented programming. In *OOPSLA '15*, pages 917–934. ACM Press, 2015.

[12] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01*, pages 59–69. ACM Press, 2001.

[13] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP '04*, volume 3086 of *Springer LNCS*, pages 465–490, 2004.

[14] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and session types. *Information and Computation*, 207(5):595–641, 2009.

[15] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded session types

for object oriented languages. In *FMCO '06*, volume 4709 of *Springer LNCS*, pages 207–245, 2006.

[16] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopolou. Session types for object-oriented languages. In *ECOOP '06*, volume 4067 of *Springer LNCS*, pages 328–352, 2006.

[17] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopolou. A distributed object-oriented language with session types. In *TGC '05*, volume 3705 of *Springer LNCS*, pages 299–318, 2005.

[18] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys '06*, pages 177–190. ACM Press, 2006.

[19] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI '02*, pages 13–24. ACM Press, 2002.

[20] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, 2014.

[21] Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[22] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

[23] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL '10*, pages 299–312. ACM Press, 2010.

[24] Görel Hedin. An introductory tutorial on JastAdd attribute grammars. In *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Springer LNCS*, pages 166–200, 2011.

[25] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08*, pages 273–284. ACM Press, 2008.

[26] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *ECOOP '10*, volume 6183 of *Springer LNCS*, pages 329–353, 2010.

[27] Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE '16*, volume 9633 of *Springer LNCS*, pages 401–418, 2016.

[28] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *ECOOP '08*, volume 5142 of *Springer LNCS*, pages 516–541, 2008.

[29] Idris language homepage. www.idris-lang.org.

[30] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo (long version). http://www.dcs.gla.ac.uk/research/mungo/papers/mungo.pdf.

[31] Message Passing Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, 1994.

[32] Filipe Militão, Jonathan Aldrich, and Luís Caires. Aliasing control with view-based typestate. In *FTfJP '10*. ACM Press, 2010.

[33] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *PADL '04*, volume 3057 of *Springer LNCS*, pages 56–70, 2004.

[34] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *COORDINATION '14*, volume 8459 of *Springer LNCS*, pages 131–146, 2014.

[35] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: local verification of global protocols. In *RV '13*, volume 8174 of *Springer LNCS*, pages 358–363, 2013.

[36] Nicholas Ng, José Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. Protocols by default - safe MPI code generation based on session types. In *CC '15*, volume 9031 of *Springer LNCS*, pages 212–232, 2015.

[37] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: safe parallel programming with message optimisation. In *TOOLS '12*, volume 7304 of *Springer LNCS*, pages 202–218, 2012.

[38] Nicholas Ng, Nobuko Yoshida, Olivier Pernet, Raymond Hu, and Yiannos Kryftis. Safe parallel programming with Session Java. In *COORDINATION '11*, volume 6721 of *Springer LNCS*, pages 110–126, 2011.

[39] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, pages 25–36. ACM Press, 2008.

[40] Rust language homepage. www.rust-lang.org.

[41] Scribble project homepage. www.scribble.org.

[42] Extended simple mail transfer protocol, RFC 5321. https://tools.ietf.org/html/rfc5321.

[43] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[44] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in Plaid. In *OOPSLA '11*, pages 713–732. ACM Press, 2011.

[45] Roger Wolff, Ronald Garcia, Eric Tanter, and Jonathan Aldrich. Gradual typestate. In *ECOOP '11*, volume 6813 of *Springer LNCS*, pages 459–483, 2011.

[46] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *TGC '13*, volume 8358 of *Springer LNCS*, pages 22–41, 2013.