ABSTRACT

We report on two tools that extend Java with support for static type-checking 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 type-checked SMTP client (§ 4), illustrates the end-to-end toolchain provided by Scribble, StMungo and Mungo.

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

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

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

```java
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():
      if(head == 0) return Check.EMPTY;
      return Check.NONEMPTY;
    void deallocate(): } }
```

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 call 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:

```java
@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.

```java
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,...,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 un-
bounded number of push() operations, looping back to NonEmpty
each time.

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

\[
\text{label: do switch block while(true)}
\]

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 fol-
low a typestate protocol; aliasing on objects allows for different
method calls on an object that might lead to an inconsistent types-
tate. 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:

\[
s = su.pushH(s,64); s = su.popAll(s);
\]

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

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

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

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 commu-
nication 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 pro-
col 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.

```plaintext
BuyTicket_R(self R, role A, role F){ request(Travel) 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; 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.

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

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

```plaintext
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].

```plaintext
smtp(role R, role C) { // Global interaction between server and client.
_220(String) from S to C; rec X1 { choice at S { _250(String) from S to C; continue X1; } or { _250DASH(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.

```plaintext
local protocol SMTP_C(role S, self C) { _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; ... } } }
```

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.

```plaintext
BuyTicket_R(self R, role A, role F){
request(Travel) from F 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;
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.

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

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

```plaintext
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}
```
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.

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.
customise CMain by adding SSL connection code for authentication with the gmain server. Mungo is able to statically check CMain, or any code that uses a CRole object, to ensure that methods of the protocol are called in a valid sequence and that all possible responses are handled. The programmer is not required to use the skeleton implementation of CMain, or even the CRole API. It is possible to write new code that uses the API, or to use the typestate specification to guide the development of an alternative API, or to refactor the typestate specification itself.

5. A CORE CALCULUS FOR MUNGO

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

Syntax. The syntax of the calculus is given in Fig. 1. We use to denote a possibly empty set of elements that range over the subject meta-variable. A program is a set of type declarations \( \mathcal{D} \), each of which declares either a class or an enumerated type. A class declaration defines a class named \( C \) with typestate specification \( S \), fields \( \bar{F} \) and methods \( M \). An enumeration declaration defines an enumerated type named \( E \) with a non-empty set \( I \) 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 \( \mathcal{D} \), which can be accessed by the following functions: given that a class \( C : S \{ \bar{F}; M \} \in \mathcal{D} \) we define fields \( \text{fields}(C) = \bar{F}, \text{methods}(C) = M, \text{typestate}(C) = S \); and \( \text{enum}(E) = I \) if \( E \{ \bar{I} \} \in \mathcal{D} \). A typestate definition \( S \) specifies a state machine that has as actions the methods of a class. A typestate definition is either an internal choice \( H \) of method signatures, or a recursive typestate \( \mu XS \), 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 \( E \{ \bar{I} \} \{ \bar{S} \} \in \mathcal{D} \) with the method signature defining the transition to one of the possible states \( \{ S \} \in \mathcal{D} \); 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 \( \mu XS \) 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 \( T \{ \bar{M} \} \{ \bar{f} \} \{ \bar{S} \} \in \mathcal{D} \) specifies a return type \( T \), the name \( m \) of the method, the type \( T' \) of the parameter \( x \), and the expression \( e \) that comprises the method body. A path is either the atomic path \( \text{this} \) denoting the current object (receiver), the composite path \( r.f \) denoting the field \( f \) of the object denoted by \( r \), or a parameter \( x \). At runtime paths are resolved to heap locations (see runtime syntax below). A constant is the special value \( \text{null} \), which is assignable to any class type, a boolean or void literal, or an enum value \( l \). A constant or a path is an expression. In the expression forms method call \( r.m(e) \), field assignment \( r.f = e \), and object creation \( \text{new} \ C \), have the target object of the invocation or assignment is restricted to a path \( r \), rather than an arbitrary expression. The other expression forms include sequential composition \( e; e' \), switch expressions, if else expression, labelled expressions \( \lambda : e \), and continue expressions that jump to the enclosing expression labelled by \( \lambda \).

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

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

The operational semantics is annotated with labels \( \ell \) that denote the creation of a new object \( \text{new} \ C \), an enum value choice \( c(\ell) \), method call \( r.(T \{ \bar{M} \}) \), assigning a field \( r.f = v \), the conditional label \( (\ell, f) \), and the silent label \( (\ell) \). The definition of states is extended to the set of enum values \( \{ \ell : S \} \in \mathcal{D} \) and we define action labels \( s \) for labels: internal choice \( T \{ \bar{M} \} \{ \bar{S} \} \), external choice \( E \{ \bar{M} \} \{ \bar{S} \} \). The access of a field \( h.f \) is inductively defined on the access of \( h \).

Labelled reduction semantics. We define heap access and update functions that are used by the reduction relation in Fig. 3: 

- \( h.(\text{root}) = h \)
- \( h.(r.f = o) = h[r.f = o] \)
- \( h.(r.f \rightarrow o') = h[r.f = C[f \mapsto o']] \)

The root object is accessed via \( h.(\text{root}) \). The access of a field \( h.f \) is inductively defined on the access of \( h \). Similarly, we use the heap access function to update object fields as in \( h[r.f \rightarrow o] \).
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[f = inith(T)] whose fields are all initialised to the value inith(T), where T is the type of the field, defined as: inith(C) = null; inith(E) = Enull; inith(bool) = ff; and inith(void) = *, where for every enumerated type E we require there to be a distinguished element Enull ∈ 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-AssignC 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-AssignR rule is to relocate the object from r′ to h.r, leaving null at its old location. The annotation label for R-AssignR is r.f = v. The R-Call rule is labelled with r.T m T′ and resolves the method m by first looking up the receiver r in the heap, which must be an object C[f = inith(T)], and then selecting the method m from the definition of C. Prior to executing the selected method, we convert its body e 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.(l), of the switch expression to the branch indicated by l. The r is used in the reduction label to indicate which object made the choice. Rule R-Label is labelled with τ, and says that a labelled expression l : e discards the l and substitutes a copy of the labelled expression for every occurrence of continue l 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 fields(C) = T f, each field of C is initialised with the corresponding value inith(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:

\[ U ::= C[S] \mid E \mid \text{bool} \mid \text{void} \mid \text{bot} \]
\[ \Delta ::= \emptyset \mid \Delta, r : U \mid \Delta, \lambda : X \]

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 a constant type.

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

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

- The subtyping relation \( \prec_{\text{st}} \) is defined by the rules in Fig. 4.
- The equivalence relation is defined as \( =_{\text{st}} = \prec_{\text{st}} \cap \prec_{\text{st}}^{-1} \).
- The join operator \( \text{join}(\cdot, \cdot) \) 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.

\[ R_{-}\text{SEQ} \quad h,(v,e) \xrightarrow{\tau} h,e \]
\[ R_{-}\text{TRUE} \quad h,\text{if}(tt)\ e_{1}\ \text{else}\ e_{2} \xrightarrow{\text{if}} h,e_{1} \]
\[ R_{-}\text{FALSE} \quad h,\text{if}(ff)\ e_{1}\ \text{else}\ e_{2} \xrightarrow{\text{if}} h,e_{2} \]
\[ R_{-}\text{NEW} \quad h,r.f = \text{new}\ C \xrightarrow{\text{new}} h[r.f \mapsto C[f : \text{inith}(T)]], \star \]
\[ R_{-}\text{VALUE} \quad (v \neq \ell) \quad h,v@r \xrightarrow{\ell} h,v \]
\[ R_{-}\text{ASGNR} \quad (h' = h[r' \mapsto \text{null}]) \quad h,r.f = r' \xrightarrow{r'\cdot f} h'[r.f \mapsto h(r')], \star \]
\[ R_{-}\text{SWITCH} \quad (l' \in E) \quad h,\text{switch}(l'@r)\ \{e\}_{e\in E} \xrightarrow{(l')} h,e \]
\[ R_{-}\text{LABEL} \quad h,\ell : e \xrightarrow{\ell} h,e[\ell/\ell] \]
\[ R_{-}\text{CTX} \quad h,e \xrightarrow{\ell} h',e' \]

**Figure 3: Operational semantics**
A disjoint union of the rest of the methods is performed. Join on relation on typestates as follows.

For the methods in common, the continuation typestates are joined. The most interesting case of join on typestates is the join of method signatures. Recursive typestates is done up to unfolding. The relation generalises subtyping with the empty set being treated as a special case. Rule Definition 2.

We illustrate the most important rules using examples. The full implementation of class enums is read from right to left. It takes as input the typing context and of the expression on inactive typestate assumptions. The second judgement infers the typestates of the fields of a class when the class is used according to its declared right in a type system fashion, where the expression “consumes” Δ in order to produce ∆. 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 effects 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, AsyncR, AsyncC, and New that require treatment of linearity. Consider the following code that uses the implementation of class Stack in Section §1:

\[ s = \text{new} \text{Stack}; \, k = s \quad (1) \]

The code expression matches rule Seq. We assume

\[ \Delta_0 = s : \text{Stack[end]}, \, k : \text{Stack[s]} \]
as an input typing context and $S \triangleq_{\text{StackProtocol}} \text{Stack}$. 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:

$\Delta \vdash e_1 : U_1 + \Delta'$  
$\Delta \vdash e_2 : U_2 + \Delta'$  
$\Delta = \{ r : C[S] \mid r : C[S] \in \Delta' \}\cup \{ r : U' \mid r : U' \in \Delta' \text{ and } U' \neq C[S'] \}$  
$\Delta \vdash \text{if}(e) \ e_1 \text{ else } e_2 : \text{join}(U_1, U_2) + \Delta'$

meaning that $k$ has an inactive typestate before assignment. Rule ParnR "guesses" a type for a path expression. However, the combination of ParnR and AsgnR is the key to this inference since it enforces a match on the type of $s$ in the output typing context $\Delta_2$ and the type of $k$ in the input typing context $\Delta_0$. For the first expression in (1) we use rule New. By assumption we satisfy its premise: we have $S \triangleq_{\text{New}} \text{Stack}$. For 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 void to forbid dead code after a continue $\lambda$ 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:

$$\Delta_3 \vdash s : \text{new Stack}; \ k : \text{Stack}\{\text{end}\} \quad \text{AsgnR} \quad \ldots$$

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

$$s = \text{new Stack}; \ k = s : \text{void} + \Delta_0 \quad (3)$$

To conclude, in rules AsgnR and New the path this is not assignable. In rule ParnR the path this is not inferable.

The other rules for paths and assignments are as follows. Rule ParnR infers a constant type $U$ for a path $r$ and has no effect in the input typing context, if $r$ is mapped to $U$ in the input typing context. Rule AsgnR 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 LeExp, Continue, Switch, and If. Consider the following class StackUser that defines methods that use a Stack object:

```java
class StackUser{
    1. {{Stack pushN( Stack):}
    2.  {Stack popAll( Stack):} end}}
    3. Stack popN( Stack x) { x.push(2); x }
    4. Stack popAll( Stack x)
    5. {loop: switch( x.isEmpty())
```

Figure 6: Typestate inference rules for expressions
{case EMPTY: x,
  case NOTEMPTY: x.pop();
  continue loop}}

and the input typing context:
\[ \Delta_0 = x : \text{Stack[end]}, \text{this} : \text{StackUser[end]} \]

The body of method popAll in line 4 is a labelled expression, and so the rule LEExp applies. The premise requires an inference for the switch expression by using in input \( \Delta_0 \) augmented with the assumption \( \text{isEmpty} : X \), where \( X \) is fresh. Let \( \Delta_1 = \Delta_0 \cup \text{loop} : X \). LEExp closes all 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 \( \Delta_1 \). The inferred output contexts of the branches are then joined and used in input to infer a typestate for the method call expression in the condition of the 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 PmR:
\[
\Delta_2 = \begin{cases} x : \text{Stack}[S], \text{this} : \text{StackUser[end]}, \text{loop} : X & \Delta_2 + x : \text{Stack}[S] + \Delta_1 \\
\end{cases}
\]

For the FALSE branch we first use rule Seq0 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:
\[
\text{CONTINUE} \quad \Delta_3 = \begin{cases} x : \text{Stack}[X], \text{this} : \text{StackUser}[X] & \Delta_3 + \text{continue} : \text{loop} + \text{bot} + \Delta_1 \\
\end{cases}
\]

The type of the continue expression is bot, since we want \( \text{join}(...) \) to be defined (cf. Fig. 5). To complete the typing of the FALSE branch, we apply rule Call for \( x.\text{pop()} \) and conclude with rule Seq0. The output typing context is:
\[
\Delta_4 = x : \text{Stack}[(\text{int} \text{ pop}) : X], \text{this} : \text{StackUser}[X]
\]

We join the output typing contexts \( \Delta_2 \) and \( \Delta_4 \) of the TRUE and FALSE branch, respectively and use the result as an input typing context for the method call \( x.\text{isEmpty}() \), as stated by the premises of Switch. The output typing context of Switch is:
\[
\Delta_5 = x : \text{Stack}[(\text{Choice isEmpty}) : \text{join}(S, \text{int} \text{ pop} : X)], \text{this} : \text{StackUser}[\text{join}(\text{end}, X)]
\]

To complete the inference of LEExp we close the recursive variable \( X \) in \( \Delta_5 \) and obtain the output typing context for the labelled expression in lines 4-5, which is:
\[
\Delta_6 = x : \text{Stack}[(\mu X. \text{Choice isEmpty}) : \text{join}(S, \text{int} \text{ pop} : X)], \text{this} : \text{StackUser}[\mu X. \text{join}(\text{end}, X)]
\]

Notice the equivalence of the type \( \mu X. \text{join}(\text{end}, X) \), that appears in the mapping of path this, and the type end, meaning that rule Eqv 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 \( T \neq C \Rightarrow \text{initT}(T) = T \) and \( \text{initT}(C) = C[\text{end}] \).

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

and the input typing context:
\[
\Delta_0 = s : \text{Stack}[S], c : \text{StackUser}[\text{Stack popAll}(\text{Stack}) : \text{end}]
\]

By applying rule AseqR on the above assignment with input \( \Delta_0 \), the output typing context in the premise of the rule is:
\[
\Delta_1 = s : \text{Stack[end]},
\quad c : \text{StackUser}[\text{Stack popAll}(\text{Stack}) : \text{end}]
\]

At this point we can apply rule Call on \( c.\text{pushN}(s) \) and have the following derivation:
\[
\begin{align*}
\text{Stack pushN(} \text{Stack } x \{ x.\text{push}(2); x \} & \in \text{methods}(\text{StackUser}) \quad (1) \\
\Delta_2 + x.\text{push}(2); x[\text{this}] & : \text{Stack}[S] + \Delta_1 \\
\Delta + s : \text{Stack}[\text{void push(int)} : S] & + \Delta_2 \quad (3)
\end{align*}
\]

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

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 PmR is used where \( \Delta_3 \) also updates the type of the receiver:
\[
\Delta_3 = s : \text{Stack[end]},
\quad c : \text{StackUser}[\text{Stack pushN}(\text{Stack}) :
\quad \text{Stack popAll}(\text{Stack}) : \text{end}]
\]

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 \( \equiv_{\text{str}} \). This is to respect the abstraction principle: the client would know how a method uses its receiver. For example, assume method Stack pushN(\text{Stack}) is defined as:
\[
\text{Stack pushN(} \text{Stack } x \{ x.\text{push}(2); x = \text{this}.\text{popAll}(x); x \} 
\]

If we infer a typestate for the body of Stack pushN(\text{Stack}) with input context \( \Delta_1 \) we get: an output typing context, \( \Delta' \), such that:
\[
\Delta'(c) = \text{StackUser}[\text{Stack popAll}(\text{Stack}) После] : [\text{Stack popAll}(\text{Stack}) : \text{end}]
\]
Given that $\Delta_1(c) = \text{StackUser}([\text{Stack popAll(Stack) : end}])$, 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 infer($T$) definition that maps a type $T$ to the corresponding inferred type $U$ as: $T \neq C \implies \text{infer}(T) = T$ and $\text{infer}(C) = C[S]$, 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 infer($T$) is used to check that the return and parameter types of the method match the types of the inferred ones. As in Cal, 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 LEq 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-ArR 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 ArR 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.

**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-If states that $\Delta$ remains unchanged under a $r$-reduction. Rule Ty-New states that a path in $\Delta$ mapped to an inactive typestate reduces under $r.f.new C$ and its typestate is updated accordingly. Rules Ty-AsmR and Ty-AsmC label the reduction with an assignment of a path and a constant, respectively. The former reduction ensures linearity conditions when an assignment takes place. The latter leaves the typing context unchanged. Rule 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
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 case study, 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.

Capcechi 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].

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 case study, 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.

Capcechi 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 g-mail 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 Voina for her contribution to Mungo and StMungo. We thank Garrett Morris, Raymond Hu and Nobuko Yoshida for useful comments and discussion.

9. REFERENCES


