



Behavioural Types for Memory and Method Safety in a Core Object-Oriented Language

Mario Bravetti¹, Adrian Francalanza², Iaroslav Golovanov³,
Hans Hüttel³, Mathias S. Jakobsen³, Mikkel K. Kettunen³,
and António Ravara⁴

¹ Dipartimento di Informatica, Università di Bologna, Bologna, Italy

² Department of Computer Science, University of Malta, Msida, Malta

³ Institut for Datalogi, Aalborg Universitet, Aalborg, Denmark

⁴ NOVA-LINCS and NOVA School of Science and Technology, Caparica, Portugal
`aravara@fct.unl.pt`

Abstract. We present a type-based analysis ensuring memory safety and object protocol completion in the Java-like language Mungo. Objects are annotated with usages, typestates-like specifications of the admissible sequences of method calls. The analysis entwines usage checking, controlling the order in which methods are called, with a static check determining whether references may contain null values. It prevents null pointer dereferencing in a typestate-aware manner and memory leaks and ensures that the intended usage protocol of every object is respected and completed. The type system admits an algorithm that infers the most general usage with respect to a simulation preorder. The type system is implemented in the form of a type checker and a usage inference tool.

1 Introduction

The notion of reference is central to object-oriented programming, which is thus particularly prone to the problem of *null-dereferencing* [18]: a recent survey [30, Table 1.1] analysing questions posted to StackOverflow referring to `java.lang` exception types notes that, as of 1 November 2013, the most common exception was precisely null-dereferencing. Existing approaches for preventing null-dereferencing require annotations, e.g. in the form of pre-conditions or type qualifiers, together with auxiliary reasoning methods. For instance, Fähndrich and Leino [12] use type qualifiers with data flow analysis to determine if fields are used safely, while Hubert *et al.* rely on a constraint-based flow analysis [20]. Recently, type qualifiers to prevent issues with null pointers were adopted in mainstream languages, like nullable types in C#, Kotlin, and Swift, and option

Work partially supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233 (BehAPI), the UK EPSRC grant EP/K034413/1 (ABCD), and by NOVA LINCS (UIDB/04516/2020) via FCT.

© Springer Nature Switzerland AG 2020

B. C. d. S. Oliveira (Ed.): APLAS 2020, LNCS 12470, pp. 105–124, 2020.

https://doi.org/10.1007/978-3-030-64437-6_6

types in OCaml, Scala, and Java. These approaches rely on programmer intervention, what can be viewed as a limitation, since the absence of null-dereferencing does not come “for free”, just as a consequence of a program being well-typed.

Static analysis tools that are “external” with respect to the type-system, like the Checker framework [10] or the Petri Net based approach in [8], can be used to check the code once it is in a stable state. However, both type qualifiers and “external” static analyses suffer from a common problem: they are restrictive and require additional explicit checks in the code (e.g., if-then-else statements checking for null), resulting in a “defensive programming” style.

On the contrary, by including the analysis *as part of the type system*, one obviates the need for additional annotations and auxiliary reasoning mechanisms. For instance, the Eiffel type system [24] now distinguishes between attached and detachable types: variables of an attached type can never be assigned a void value, which is only allowed for variables of detachable type. However, enriching the type system in this way is not enough, in that *it is the execution of a method body that typically changes the program state*, causing *object fields* to become nullified. The *interplay* between *null-dereferencing* and the *order in which methods of an object are invoked* is therefore important. A recent manifestation of this is the bug found in Jedis [32], a Redis [33] Java client, where a `close` method could be called even after a socket had timed out [32, Issue 1747]. One should therefore see an object as following a *protocol* describing the *admissible sequences of method invocations*. The intended protocol can, thus, be expressed as a *behavioural type* [3, 6, 21]: our idea is to use such types to ensure no null-dereferencing via static type checking.

There are two main approaches to behavioural type systems. The notion of *typestates* originates with Strom and Yemini [29]; the idea is to annotate the type of an object with information pertaining to its current state. Earlier work includes that of Vault [11], Fugue [9] and Plaid [2, 31]. In the latter, an object-oriented language, the programmer declares for each class *typestates* (the significant states of objects) and annotates each method with (statically checked) pre and post-conditions. Pre-conditions declare typestates that enable the method; post-conditions define in which typestate the method execution leaves the object. One has also to declare in these assertions the states of fields and parameters. Garcia *et al.* [14] describe a gradual typestate system following the approach of Plaid and combining access permissions with gradual types [28] to also control aliasing in a robust way. The other approach taken is that of *session types* [19]. This originated in a π -calculus setting where the session type of a channel is a protocol that describes the sequence of communications that the channel must follow. Channels are *linear*: they are used exactly once with the given protocol, and evolve as a result of each communication that it is involved in.

1.1 Our Approach

The approach of Gay *et al.* [15], adopted in this paper, combines the two approaches above to behavioural types: the type of a class C is endowed with a *usage type* denoting a behaviour that any instance of C must follow. Consider the class `File` in Listing 1.1 [1]; the usage defined on lines 4–8 specifies the

admissible sequences of method calls for any object instance of `File`. The usage is a set of defining equations where `Init` is the initial *variable* (denoting a typestate). It tells the object must first be `opened`, going then into typestate `Check` (another usage variable) where only the method `isEOF` can be called. When `isEOF` is called the continuation of the protocol depends on the method result: if it returns `EOF` one can only `close` the file and the protocol is completed (denoted by `end`); if it returns `NOTEOF` one can `read` and check again. This ensures that all methods of a `File` object are called according to the safe order declared.

```

enum FileStatus { EOF , NOTEOF }           1
                                           2
class File {                               3
  {Init = {open; Check}                   4
    Check = {isEOF;                       5
              (EOF: {close; end} , NOTEOF: {read; Check})}
        }                                 6
  }                                       7
                                           8
                                           9
  void open(void x) { ... }              10
  FileStatus isEOF(void x) { ... }      11
  Char read(void x) { ... }             12
  void close(void x) { ... }            13
}                                         14

```

Listing 1.1. An example class describing files

In Listing 1.2 an additional class `FileReader` is introduced. Its usage type, at line 18, requires that the `init()` method is called first, followed by method `readFile()`.

```

class FileReader {                         17
  {Init = {init ; {readFile; end}}}       18
                                           19
  File file;                              20
                                           21
  void init() { file = new File }        22
                                           23
  void readFile() {                       24
    file.open(unit);                      25
    loop: switch(file.isEOF()) {          26
      EOF: file.close()                   27
      NOTEOF: file.read(); continue loop  28
    }
  }                                       29
}                                         30
}                                         31

```

Listing 1.2. An example class intended for reading files

Class `FileReader` uses class `File` for its field `file` declared at line 20: method calls on `file` will have to follow the usage type of `File`. Indeed, since `FileReader`

class usage imposes to call method `init` before method `readFile`, we have that `FileReader` class code correctly deals with objects of class `File`: first method `init` code creates a `File` object inside field `file`, then method `readFile` code follows `File` usage for such an object, by first opening it, entering a loop (lines 26 to 29) to read until its end, and closing it. So, in general, type checking of a class (as `FileReader`) entails checking that: assuming the usage type of the class is followed, all fields (e.g. `file`) are correctly dealt with according to the usage type of their class (e.g. the usage of `File`). Moreover, since methods on field `file` are called by code of method `readFile` only, and since, by considering for `FileReader` class tpestates, we know that such method can only be performed after the `init` method, no null-dereferencing can occur when `FileReader` class code is executed. In spite of this, previous work considering tpestates and no null-dereferencing checking would not allow to type check the `FileReader` class. For example the above mentioned approach of [2,14,31] would require the programmer, besides declaring tpestates imposing `init` to be performed before method `readFile`, to explicitly annotate `readFile` method with a precondition stating that `file` cannot be null. Such annotations are quite demanding to the programmer and sometimes even redundant. When an object is in a given tpestate (e.g. when `readFile` is enabled) the values of its fields are, implicitly, already constrained (`file` cannot be null). Therefore type-checking based on a tpestate-aware analysis of `FileReader` class code (i.e., assuming that the usage type of the class is followed) makes it possible to guarantee no null-dereferencing without any additional annotation. Tpestate-aware analysis of null-dereferencing is one of the novel contributions of this paper.

The type system for `Mungo` (a Java-like language) [23] depends on *linearity* and *protocol fidelity*. Linearity requires that, once an object reference is written to a variable/field whose type is a class with a usage, it can be read from that variable/field at most once (it can also be passed around or written to other variables/fields instead). This avoids aliasing while permitting compositional reasoning via the type system, making it possible to statically verify that objects follow the intended protocol. Protocol fidelity requires that usage of a class is followed when calling methods on a variable/field whose type is that class. Checking protocol compliance merely by such a simple form of protocol fidelity, however, does not suffice to correctly perform all checks guaranteeing correctness of our example. For instance, protocol fidelity in [23] permits:

- Omitting `file = new File` in the body of method `init` at line 22. However, even if one follows the prescribed protocol of `FileReader` by invoking first `init` and then `readFile`, one gets a null-dereferencing when calling `open` on `file`.
- Adding `file = null` after `file = new File` at line 22. This results not only in getting a null-dereferencing, as above, but also in losing the reference to the created object before its protocol is completed, due to object memory being released only at the end of its protocol.
- Adding `file = new File` at line 27 before calling `close`. This result in the loss of the reference to the previous object stored in `file` that has not yet completed its protocol.

Therefore, the `Mungo` type system in [23] does *not* provide guarantees ruling out null-dereferencing and loss of references. In particular, three unpleasant behaviours are still allowed: (i) null-assignment to a field/parameter containing an object with an incomplete tpestate; (ii) null-dereferencing (even `null.m()` is accepted); (iii) using objects without completing their protocol and without returning them. Moreover the type system in [23] is based on a mixture of type checking and type inference that makes it not fully compositional – to be, type checking a class should not depend on type checking other classes; without it, the complexity of the type analysis would not depend only on the structure of the class being typechecked but also on that of other classes.

1.2 Contributions

In this paper we present the first “pure” behavioural type-checking system for `Mungo` that handles all these important unsolved issues. This constitutes (to our knowledge) the *first compositional behavioural type system* for a realistic object-oriented language that *rules out null-dereferencing and memory leaks* as a by-product of a safety property, *i.e.*, *protocol fidelity*, and of a (weak) liveness property, *i.e.*, object *protocol completion* for terminated programs. In particular it is the first type system that checks null-dereferencing in a *tpestate-aware* manner. Note that, while protocol fidelity is an expected property in behavioural type systems, this does not hold for properties like: *protocol completion* for a mainstream-like language or *memory safety*, *i.e.*, no null-dereferencing/memory leaks. Notably our type system:

- Makes it possible to analyze source code of individual classes *in isolation* (by just relying on usages of other classes which are part of their public information, thus respecting correct encapsulation principles).
- Is based on a more complex notion of protocol fidelity, w.r.t. the type system in [23], that includes a special tpestate for variables/fields representing the null value and encompasses tpestate-aware null-dereferencing checking.
- Is based on requiring protocol completion for terminated programs and references that are lost (e.g. by means of variable/field re-assignment or by not returning them).
- Is compositional and uses type checking only, *i.e.* a term is typable if its immediate constituents are (unlike [23] which uses a mixture of type checking and type inference).
- Admits an algorithm for *principal usage inference*. For any class, the algorithm correctly infers the largest usage that makes the class well-typed.

The typing-checking system and the usage inference system presented herein as (rule based) inductive definitions, were implemented (in Haskell) to allow to test not only the examples presented ahead, but also more elaborate programs – a suit of examples and the code implementing both systems is available at GitHub [1]. A new version of `Mungo` following our approach is also available at <https://github.com/jdmota/java-tpestate-checker>.

Table 1. Syntax of Mungo

$D ::= \text{enum } L \{\vec{l}\} \mid \text{class } C \{\mathcal{U}, \vec{M}, \vec{F}\}$ $F ::= z \ f$ $M ::= t \ m(t \ x)\{e\}$ $v ::= \text{unit} \mid \text{true} \mid \text{false} \mid \mid \mid \text{null}$ $r ::= x \mid f \mid \text{this}$ $e ::= v \mid r \mid \text{new } C \mid f = e \mid r.m(e) \mid e; e$ $\quad \mid k : e \mid \text{continue } k \mid \text{if } (e) \{e\} \text{ else } \{e\}$ $\quad \mid \text{switch } (r.m(e)) \{l_i : e_i\}_{i \in L}$	$b ::= \text{void} \mid \text{bool} \mid L$ $z ::= b \mid C$ $t ::= b \mid C[\mathcal{U}]$ $u ::= \{m_i; w_i\}_{i \in I} \mid X$ $w ::= u \mid \langle l_i : u_i \rangle_{l_i \in L}$ $E ::= X = u$ $\mathcal{U} ::= u^{\vec{E}}$
--	---

Due to space restrictions, we omit in this paper some rules and results. The complete formal systems are presented and described in detail in <http://people.cs.aau.dk/~hans/APLAS20/typechecking.pdf> and <http://people.cs.aau.dk/~hans/APLAS20/inference.pdf>.

2 The Mungo Language

Mungo is a typed Java-like language in the style of Featherweight Java [22] that contains usual object-oriented and imperative constructs; the name also refers to its associated programming tool developed at Glasgow University [23,34]. The Mungo language is a subset of Java that extends every Java class with a typestate specification. The syntax of Mungo is given in Table 1. A *program* \vec{D} is a sequence of *enumeration declarations*, introducing a set of n labels $\{l_1, \dots, l_n\}$, for some natural $n > 0$, identified by a name $L \in \mathbf{ENames}$, followed by a sequence of *class declarations* where $C \in \mathbf{CNames}$ is a class name, \vec{M} a set of methods, \vec{F} a set of fields, and \mathcal{U} a usage. A program \vec{D} is assumed to include a main class, called **Main**, with a single method called **main** having void parameter/return type and usage type $\mathcal{U} = \{\text{main}; \text{end}\}^\varepsilon$. In the examples we used a set of defining equations to specify usages, indicating which variable is the initial one. In the formal syntax we omit the initial variable—a usage is just an expression u with a set of defining equations \vec{E} as superscript ($u^{\vec{E}}$). The usage in line 18 of Listing 1.2 is thus written as $\{\text{init}; RF\}^E$, with $E = \{RF = \{\text{readFile}; \text{end}\}\}$.

Fields, classes and methods are annotated with *types*, ranged over by t . The set of base types \mathbf{BTypes} contains the void type (that of value **unit**), the type **bool** of values, and enumerations types, ranged over by L , for sets of labels. Typestates $C[\mathcal{U}] \in \mathbf{Typestates}$ are a central component of the behavioural type system, where C is a class name and \mathcal{U} is a *usage*, specifying the admissible sequences of method calls allowed for an object. In our setting typestates are used to type *non-uniform objects* [27], instead of the usual class type (i.e., we write $C[\mathcal{U}]$ instead of just C). A *branch* usage $\{m_i; w_i\}_{i \in I}$ describes that any one of the methods m_i can be called, following which the usage is then w_i . We

let `end` denote the *terminated*, empty usage – a branch with $I = \emptyset$. The usage $\{\text{readFile}; \text{end}\}$ is a simple example of a terminating branch usage. A *choice* usage $\langle l_i : u_i \rangle$ specifies that the usage continues as u_i depending on a label l_i . This is useful when the flow of execution depends on the return value of a method. The usage in line 6 of Listing 1.1 is an example of a choice usage. Recursive usages allow for iterative behaviour. A defining equation $X = u$ specifies the behaviour of the usage variable X , which can occur in u or in any of the other equations.

The set **Values** is ranged over by v and contains boolean values, unit, labels, and null. References ranged over by r describe how objects are accessed – as method parameters ranged over by $x \in \mathbf{PNames}$, as field names ranged over by $f \in \mathbf{FNames}$ or as the enclosing object, `this`. Values and references are expressions. To follow a linear discipline, reading either a field or a parameter containing an object nullifies the reference (as the object is passed somewhere else). Moreover, assigning objects to references is only possible if they contain null or if their protocol is terminated. Expressions also include standard method calls $r.m(e)$. Methods have exactly one argument; the extension to an arbitrary number of arguments is straightforward. Sequential compositions is denoted by $e; e'$, conditionals are `if (e) {e1} else {e2}`, and there is a restricted form of selection statements, `switch (r.m(e)) {li : ei}i ∈ L`, that branches on the result of a method call that is supposed to return a label (an example of this construct is in lines 26–29 of Listing 1.2).

Iteration in Mungo is possible by means of jumps. This lets us give more expressive behavioural types as mutual recursions in nested loops [23]. Expressions can be labelled as $k : e$, where k must be a unique label; the execution can then jump to the unique expression labelled k by evaluating the expression `continue k` (an example of this construct is in lines 26 and 28 of Listing 1.2). We require that labelled expressions are well-formed: in a labelled expression $k : e$, the label k is bound within e , and all occurrences of `continue k` must be found in this scope; moreover, in e , `continue` cannot be part of the argument of a method call; the expression on the right side of an assignment or on the left of a sequential composition must not be a `continue` expression; finally, within $k : e$ there must be at least a branch, considering all `if` and `switch` expressions possibly included in the code e , that does not end up in `continue k` (if there is no `if` and `switch`, e must not end up in `continue k`). This last condition rules out pathological infinite behaviours such as $k : \text{if } (\text{true}) \{ \text{continue } k \} \{ \text{continue } k \}$ and $k : \text{continue } k$.

A method named m is declared as $t_2 m(t_1 x)\{e\}$, where t_2 denotes the return type of m , while the argument type is t_1 . The body of m is an expression e . Classes in Mungo are instantiated with `new C`. When an object is created all its fields are initialised to null. Assignment is always made to a field inside the object on which a method is evaluated (fields are considered private). For an object to modify a field in another object, it must use a method call.

We introduce a dot-notation that refers directly to components of a class definition; we let $C.\text{methods}_{\vec{B}} \stackrel{\text{def}}{=} \vec{M}$, $C.\text{fields}_{\vec{B}} \stackrel{\text{def}}{=} \vec{F}$ and $C.\text{usage}_{\vec{B}} \stackrel{\text{def}}{=} U$.

To describe (partially) evaluated expressions, we extend the syntax to include *run-time expressions* and values:

$$\begin{aligned} v &:: = \dots \mid o \\ e &:: = \dots \mid \text{return}\{e\} \mid \text{switch}_{r.m} (e) \{l_i : e_i\}_{l_i \in L} \end{aligned}$$

The $\text{return}\{e\}$ construct encapsulates the ongoing evaluation of a method body e . We also introduce a general switch construct that allows arbitrary expressions and thus the partial evaluation of this construct.

3 The Type System

Our type system is a sound approximation of the reduction relation (*cf.*, Sect. 4) and rejects programs that “may go wrong” [26]. We return to this type safety result as Theorem 2 in Sect. 5. The main intentions behind our type system are to ensure that every object will follow its specified protocol, that no null pointer exceptions are raised, and no object reference is lost before its protocol is completed. The system lets us type classes separately and independently of a main method or of client code. Following Gay et al. [15], when we type a class, we type its methods according to the order in which they appear in the class usage. This approach to type checking is crucial. For suppose we call a method m of an object o that also has a field containing another object o' . This call will not only change the typestate of o (admitting the method was called at a moment allowed by the usage). The call can also change the state of o' , since the code of method m may contain calls to methods found in o' . With the type-checking system we present herein, we take an important step further: by giving a special type to null and make a careful control of it in the typing rules, we manage to prevent memory leaks and detect previously allowed null-pointer exceptions.

Example 1. Recall the `FileReader` class in Listing 1.2. Its usage requires calling first method `init` to guarantee that field `file` gets a new `File` object. Then the usage of `FileReader` requires calling the method `readFile` which then call methods on `file` according to its usage (*cf.* Listing 1.1): first `open` then iterate testing for end-of-file (using the method `isEOF`), reading while this is not the case. Failure to follow the usage of `FileReader` and, for instance, calling `readFile` without having called `init` first causes null-dereferencing. The behavioural type system of Mungo presented herein prevents this, alleviating the programmer from having to consider all possible negative situations that could lead to errors for each method. Usages are also simpler than the (sometimes redundant) assertions required by `Plaid` [2, 14, 31] and the defensive programming style required by tools such as `Checker` [10] is not needed. ■

Typing program definitions \vec{D} requires judgements like $\vdash \vec{D}$. Rule `TPROG` (below) says a program is well-typed if each of its enumeration and class declarations are well-typed; in turn declarations require judgements $\vdash_{\vec{D}} D$, saying an enumeration is well-typed if all labels in the set do not occur in any other

enumeration declaration (labels are uniquely associated with a type L – rule TENUM omitted) and a class is well-typed if its usage is well-typed.

Judgements for typing class usages are of the form

$$\Theta; \Phi \vdash_{\vec{D}} C[\mathcal{U}] \triangleright \Phi'$$

where Φ is a *field typing environment* assigning types to fields of the class C (Φ' is the corresponding result of the typing derivation, reflecting the changes in the tpestates of objects stored in the fields) and Θ is an environment assigning field type environments to usage equation variables, to deal with recursive behaviour, as explained below. The judgement also considers the program definition \vec{D} in which the class occurs, but since it never changes in a type derivation, we will not refer it in the rules henceforth presented (a subset of all rules; the omitted ones are in the technical reports referred at the end of Sect. 1). The judgement takes Θ , Φ , \vec{D} , and $C[\mathcal{U}]$ as input and if there is a derivation, it outputs Φ' .

Rule TCLASS uses an empty usage variable type environment and a field typing environment assigning *initial* types to the fields of the class: since when a new object is created, all its fields of a class type are initialised with `null`, their respective type in the initial field typing environment is \perp (to control dereferencing), a new type not available to the programmer – we extend the syntax of types with it (see below). So,

$$\vec{F}. \text{inittypes} \triangleq \{f \mapsto \text{inittype}(z) \mid z \in \vec{F}\}$$

where $\text{inittype}(b) \triangleq b$ and $\text{inittype}(C) \triangleq \perp$. Moreover, following the class usage \mathcal{U} must result in a *terminated* field typing environment Φ , *i.e.*, fields with class types must have either usage `end` or type \perp . The judgement $\Theta; \Phi \vdash C[\mathcal{U}] \triangleright \Phi$, discussed below, makes use of the method set \vec{M} via $C.\text{methods}$.

$$\text{(TPROG)} \frac{\forall D \in \vec{D} . \vdash_{\vec{D}} D}{\vdash_{\vec{D}}} \quad \text{(TCLASS)} \frac{\emptyset; \vec{F}. \text{inittypes} \vdash C[\mathcal{U}] \triangleright \Phi \quad \text{terminated}(\Phi)}{\vdash_{\vec{D}} \text{class } C\{\mathcal{U}, \vec{F}, \vec{M}\}}$$

Linear and Terminated Types. Values have a type that can be either a base type b , \perp , the type of `null`, or “general” tpestates $C[\mathcal{U}]$, with \mathcal{U} now being either branch or choice usages.

$$W ::= = w^{\vec{E}} \quad U ::= = \mathcal{U}|W \quad t ::= = b|C[\mathcal{U}]|\perp$$

An important distinction in our type system is the one between linear and non-linear types, as it is this distinction that makes the type system able to detect potential null dereferencings and memory leaks.

A type t is *linear*, written $\text{lin}(t)$, if it is a class type $C[\mathcal{U}]$ with a usage \mathcal{U} that is different from `end`. This use of linearity forces objects to be used only by a single “client”, thus preventing interference in the execution of its protocol.

$$\text{lin}(t) \triangleq \exists C, \mathcal{U} . t = C[\mathcal{U}] \wedge \mathcal{U} \neq \text{end}$$

All other types are non-linear, and we call such types *terminated*.

Table 2. Typing class usage definitions
$$\begin{array}{c}
I \neq \emptyset \quad \forall i \in I. \exists \Phi'' \\
\{\text{this} \mapsto \Phi\}; (\emptyset \cdot (\text{this}, [x_i \mapsto t'_i])) \vdash e_i : t_i \triangleright \{\text{this} \mapsto \Phi''\}; (\emptyset \cdot (\text{this}, [x_i \mapsto t''_i])) \\
\text{TCBR} \frac{\wedge \text{terminated}(t''_i) \wedge t_i m_i(t'_i x_i)\{e_i\} \in C.\text{methods} \wedge \Theta; \Phi'' \vdash C[w_i^{\vec{E}}] \triangleright \Phi'}{\Theta; \Phi \vdash C[\{m_i; w_i\}_{i \in I}^{\vec{E}}] \triangleright \Phi'} \\
\text{TCCCH} \frac{\forall l_i \in L. \Theta; \Phi \vdash C[u_i^{\vec{E}}] \triangleright \Phi'}{\Theta; \Phi \vdash C[(l_i : u_i)_{i \in L}^{\vec{E}}] \triangleright \Phi'} \quad \text{TCEND} \frac{}{\Theta; \Phi \vdash C[\text{end}^{\vec{E}}] \triangleright \Phi} \\
\text{TCVAR} \frac{}{(\Theta, [X \mapsto \Phi]); \Phi \vdash C[X^{\vec{E}}] \triangleright \Phi'} \quad \text{TCREC} \frac{(\Theta, [X \mapsto \Phi]); \Phi \vdash C[u^{\vec{E}}] \triangleright \Phi'}{\Theta; \Phi \vdash C[X^{\vec{E}\omega\{X=u\}}] \triangleright \Phi'}
\end{array}$$

Typing class usages requires the rules in Table 2. Each one applies to a different constructor of usages: rule TCEND is a base case: `end` does not change the field environment. Rules TCREC and TCVAR are used to type a class with respect to a recursive usage X , which is well-typed if the class C can be well-typed under the body u of X defined in \vec{E} (note that in the premise of TCREC, the defining equation for X does not occur); rule TCVAR handles recursion variables appearing in usages, associating a type with the variable in the environment. Rule TCCCH deals with choice usages, requiring all possible evolutions to lead to the same usage to guarantee determinism. Finally, the important rule TCBR deals with (non-terminated) branch usages: for each method m_i mentioned, its body e_i is checked with its return type t_i ; and the initial type t'_i of the parameter x , declared in the method signature; following the effect of executing the method body, yields the resulting type t''_i of x and the resulting type of `this`.

Example 2. Recall the class `File` from Listing 1.1. To type it, we inspect its usage. As it starts as a branch, we apply TCBR and check that the body of method `open` is well-typed and move on to check usage `Check`. As it is a recursion variable, we use TCREC; we now have a branch usage where we check that the body of method `isEOF` is well-typed, using rule TCCCH. If the method returns `EOF`, we then check method `close` and terminate with rule TCEND; if it returns `NOTEOF`, we check method `read` and finish the type-checking process (the derivation ends) since we find again the recursion variable `Check`. Both cases result in identical field typing environments. ■

Typing expressions requires rules for values, with atomic and composite constructors. Our semantics is stack-based and introduces run-time extensions to the language. In the type system, the counterpart of the stack is an environment that plays a limited role in typing the language we show herein but is essential for typing code resulting from computational steps (details in the technical report).

Let the *object field environment* A record the type information for fields in objects and $S = [x \mapsto t]$ be a parameter type environment.

The key element when typing the programmer's code is the pair (o, S) , where o is the main object. Type judgements are of the form

$$A; (o, S) \vdash_{\bar{\Omega}}^{\Omega} e : t \triangleright A'; (o, S')$$

Their intended meaning is as follows: evaluating e in the initial environments A and S will produce final typing environments A' and S' , results yield only if the derivation succeeds. Here $\bar{\Omega}$ is a label environment that is used to map a label k to a pair of environments (A, S) , $\bar{\Omega}$ is only used in continue expressions and is therefore omitted in most rules (as well as \bar{D}). Since typing environments are functions, $A\{o \mapsto t\}$, for instance, denotes the result of a substitution, *i.e.*, the environment equal to A everywhere but in the image of o , that is now t .

When typing values, unit has type `void`, null has type \perp , and the boolean values have type `Bool` where the initial and final environments in the judgements do not change. Typing a parameter is similar to typing (reading) a field, so we only describe the latter here. The rules are presented below. The rule `TOBJ` handles object typing and it says that once we type an object, corresponding to reading it, we remove it from the object type environment.

$$(\text{TOBJ}) \frac{}{A\{o \mapsto t\}; (o', S) \vdash o : t \triangleright A; (o', S)}$$

`TNOFLD` describes how to type non-linear parameters and fields: no updates happen to the environments. The rule `TLINFLD` deals with linear parameters and fields: after typing the value, the linear parameter or field is updated to the type \perp (to prevent aliasing) in either the parameter stack environment or field type environment (only the rules for fields are presented, as those for parameters are similar).

$$(\text{TLINFLD}) \frac{t = A(o).f \quad \text{lin}(t)}{A; (o, S) \vdash f : t \triangleright A\{o.f \mapsto \perp\}; (o, S)}$$

$$(\text{TNOFLD}) \frac{\neg \text{lin}(t)}{A\{o.f \mapsto t\}; (o, S) \vdash f : t \triangleright A\{o.f \mapsto t\}; (o, S)}$$

The key atomic constructors are the creation of objects and assignment to fields and parameters. The typing rules are given below; we omit the rule typing assignment to parameters, as it is similar to that of fields.

$$(\text{TNEW}) A; (o, S) \vdash \text{new } C : C[C.\text{usage}] \triangleright A; (o, S)$$

$$(\text{TFLD}) \frac{C = A(o).\text{class} \quad \text{agree}(C.\text{fields}(f), t') \quad A; (o, S) \vdash e : t' \triangleright A', o.f \mapsto t; (o, S') \quad \neg \text{lin}(t)}{A; (o, S) \vdash f = e : \text{void} \triangleright A'\{o.f \mapsto t'\}; (o, S')}$$

The assignment rules are designed to avoid overwriting fields containing objects with incomplete protocols. To that purpose we use a binary predicate `agree` that only holds if both arguments are the same base type or class type.

$$\text{agree}(z, t) \stackrel{\text{def}}{=} z = t \vee \exists C. z = C \wedge (t = \perp \vee \exists U. t = C[U])$$

Notice that in rule (TFLD) `null` agrees with any class type declared in the program only if the field does not contain an object with a linear type.

Example 3. Consider a field f declared in the program with some class type C . Rule (TFLD) only lets us assign to f if its type is not linear, *i.e.*, it must be either \perp or $C[\text{end}]$. So, f either contains `null` or an object with a terminated protocol. The `agree` predicate lets us assign to f either `null` or, more significantly, any object with a usage (that in the subsequent code will be followed, as the rules we present below show). In particular, one can assign `new C` to f . ■

To type a method call on a field using the rule (TCALLF) (the rule for typing method calls on parameters is similar) first the type environments must be updated by typing the argument and then the usage of the callee object must offer the method. Usages have a labelled transition semantics. Transitions are of the form $u \xrightarrow{m} u'$ and $u \xrightarrow{l} u'$ and defined by the rules below.

$$\begin{array}{c}
 \text{(BRANCH)} \quad \frac{j \in I}{\{m_i : w_i\}_{i \in I} \xrightarrow{\vec{E}} \xrightarrow{m_j} w_j \vec{E}} \quad \text{(UNFOLD)} \quad \frac{u \xrightarrow{\vec{E} \cup \{X=u\}} \xrightarrow{m} \mathcal{W}}{X \xrightarrow{\vec{E} \cup \{X=u\}} \xrightarrow{m} \mathcal{W}} \\
 \text{(SEL)} \quad (\langle l_i : u_i \rangle_{l_i \in L}) \xrightarrow{\vec{E}} \xrightarrow{l_i} u_i \vec{E}
 \end{array}$$

The rule (TRET) for typing `return` is not surprising: once the environments are updated by typing the expression e , we remove from the object environment the last entry, with the identity of the caller and the type of the value in the parameter identifier, given that this type is terminated (to prevent memory leaks and dangling objects with incomplete protocols).

Note that the body of the method is not typed in this rule, since it was handled at the class declaration.

$$\begin{array}{c}
 \text{(TCALLF)} \quad \frac{\Delta; (o, S) \vdash e : t \triangleright \Delta' \{o.f \mapsto C[\mathcal{U}]\}; (o, S') \quad t' \ m(t \ x)\{e'\} \in C.\text{methods}_{\vec{B}} \quad \mathcal{U} \xrightarrow{m} \mathcal{W}}{\Delta; (o, S) \vdash f.m(e) : t' \triangleright \Delta' \{o.f \mapsto C[\mathcal{W}]\}; (o, S')} \\
 \text{(TRET)} \quad \frac{\Delta; \Delta \vdash_{\vec{B}} e : t \triangleright \Delta'; \Delta' \quad \Delta' = \Delta'' \cdot (o', [x \mapsto t']) \quad \text{terminated}(t')}{\Delta; \Delta \cdot (o, S) \vdash_{\vec{B}} \text{return}\{e\} : t \triangleright \Delta'; \Delta'' \cdot (o, S)}
 \end{array}$$

Example 4. Recall class `File` from Listing 1.1. Rule (TCALLF) states that in order for a call of the `close` method to be well-typed, there must be a transition labelled with `close`. This method call thus fails to typecheck under usage `Init`. ■

We conclude this section presenting the typing rules for control structures. The rule TSEQ for sequential composition requires the left expression not to produce a linear value (that would be left dangling): $e; e'$ is well-typed only if the type of e is not linear. Moreover, e' is typed with the environments resulting from typing e (*i.e.*, we take into account the evolution of the protocols of objects in e).

The rules TLAB and TCON for labelled expressions allows environments to change during the evaluation of continue-style loops. However, if a `continue` expression is encountered, the environments must match the original environments, in order to allow an arbitrary number of iterations. Since (o, S) is not relevant, we refer it as Δ .

$$\begin{array}{c}
\text{(TSEQ)} \quad \frac{\Lambda; (o, S) \vdash e : t \triangleright \Lambda''; (o, S'') \quad \text{-lin}(t) \quad \Lambda''; (o, S'') \vdash e' : t' \triangleright \Lambda'; (o, S')}{\Lambda; (o, S) \vdash e; e' : t' \triangleright \Lambda'; (o, S')} \\
\text{(TLAB)} \quad \frac{\Omega' = \Omega, k : (\Lambda, \Delta) \quad \Lambda; \Delta \vdash^{\Omega'} e : \text{void} \triangleright \Lambda'; \Delta'}{\Lambda; \Delta \vdash^{\Omega} k : e : \text{void} \triangleright \Lambda'; \Delta'} \quad \text{(TCON)} \quad \frac{\Omega' = \Omega, k : (\Lambda, \Delta)}{\Lambda; \Delta \vdash^{\Omega'} \text{continue } k : \text{void} \triangleright \Lambda'; \Delta'}
\end{array}$$

In Subsect. 1.1 a class `FileReader` was introduced with a loop repeated in Listing 1.3. Even though calling the `close` method leaves the field in another state than calling `read`, the code is well typed. The reason is that after calling `read`, the field is left in the initial state when entering the loop, and another iteration occurs. When calling `close` the loop is ended. Hence the only resulting state for the field after the loop, is `File[end]`.

```

[... ]                                     26
file.open(unit)                             27
loop: switch( file.isEOF() ) {               28
    EOF: file.close() (*@\label{1st:code:2}@\*) 29
    NOTEOF: file.read();                     30
        continue loop;                       31
}                                             32
[... ]                                     33

```

Listing 1.3. Loop from class `FileReader`

4 The Dynamic Semantics of Mungo

The operational semantics of Mungo is a reduction relation and uses a stack-based binding model, the semantic counterpart of that of the type system.

Expressions transitions are relative to a program definition \bar{D} with the form

$$\vdash_{\bar{D}} \langle h, env_S, e \rangle \rightarrow \langle h', env'_S, e' \rangle$$

A *heap* h records the bindings of object references. In the heap, every object reference o is bound to some pair $(C[\mathcal{W}], env_F)$ where $C[\mathcal{W}]$ is a typestate and $env_F \in \mathbf{Env}_F$ is a field environment. A *field environment* is a partial function $env_F : \mathbf{FNames} \rightarrow \mathbf{Values}$ that maps field names to the values stored in the fields. Given a heap $h = h' \uplus \{o \mapsto (C[\mathcal{W}], env_F)\}$, we write $h\{\mathcal{W}'/h(o).\text{usage}\}$ to stand for the heap $h' \uplus \{o \mapsto (C[\mathcal{W}'], env_F)\}$ ¹; $h\{v/o.f\}$ for $h' \uplus \{o \mapsto (C[\mathcal{W}], env_F\{f \mapsto v\})\}$; and we use the notation $h\{o.f \mapsto C[\mathcal{W}]\}$

¹ We use \uplus to denote disjoint union.

to denote the heap $h' \uplus \{o \mapsto \langle C[\mathcal{W}'], env'_F \rangle\}$ where $env'_F = env_F\{f \mapsto C[\mathcal{W}]\}$. Moreover, if $h = h' \uplus \{o \mapsto _ \}$ then $h \setminus \{o\} = h'$ and when $o \notin \text{dom}(h)$, $h \setminus \{o\} = h$. Finally, we say a heap is *terminated* if for all objects in its domain, their usages are terminated.

The *parameter stack* env_S records to the bindings of formal parameters. It is a sequence of bindings where each element (o, s) contains an object o and a parameter instantiation $s = [x \mapsto v]$. In a parameter stack $env_S \cdot (o, s)$ we call the bottom element o the *active object*. Often, we think of the parameter stack as defining a function. The domain $\text{dom}(env_S)$ of the parameter stack env_S is the multiset of all object names on the stack. The range of the parameter stack $\text{ran}(env_S)$ is the multiset of all parameter instantiations on the stack. We refer to the attributes of an object o bound in heap h , where $h(o) = \langle C[\mathcal{W}], env_F \rangle$, as:

$$\begin{aligned} h(o).\text{class} &\stackrel{\text{def}}{=} C & h(o).\text{env}_F &\stackrel{\text{def}}{=} env_F \\ h(o).\text{usage} &\stackrel{\text{def}}{=} \mathcal{W} & h(o).f &\stackrel{\text{def}}{=} env_F(f) & h(o).\text{fields} &\stackrel{\text{def}}{=} \text{dom}(env_F) \end{aligned}$$

The Transition Rules. Linearity also appears in the semantics, and the linearity requirement is similar to that of the type system. Here, a value v is said to be linear w.r.t. a heap h written $\text{lin}(h, v)$ iff v has type $C[\mathcal{U}]$ and $\mathcal{U} \neq \text{end}$. If the field denotes a terminated object or a ground value, field access is *unrestricted*.

Below we show the most important transition rules. The rules for reading linear fields illustrate how linearity works in the semantics. In LDREF we update a linear field or parameter to null after we have read it, while the rule UDREF tells us that the value contained in an unrestricted fields remains available.

$$\begin{aligned} (\text{UDREF}) & \frac{h(o).f = v \quad \neg \text{lin}(v, h)}{\vdash_{\vec{D}} \langle h, (o, s) \cdot env_S, f \rangle \rightarrow \langle h, (o, s) \cdot env_S, v \rangle} \\ (\text{LDREF}) & \frac{h(o).f = v \quad \text{lin}(v, h)}{\vdash_{\vec{D}} \langle h, (o, s) \cdot env_S, f \rangle \rightarrow \langle h\{\text{null}/o.f\}, (o, s) \cdot env_S, v \rangle} \\ (\text{UPD}) & \frac{h(o).f = v' \quad \neg \text{lin}(v', h)}{\vdash_{\vec{D}} \langle h, (o, s) \cdot env_S, f = v \rangle \rightarrow \langle h\{v/o.f\}, (o, s) \cdot env_S, \text{unit} \rangle} \\ (\text{LBL}) & \vdash_{\vec{D}} \langle h, env_S, k : e \rangle \rightarrow \langle h, env_S, e\{k : e/\text{continue } k\} \rangle \\ (\text{CALLF}) & \frac{env_S = (o, s) \cdot env'_S \quad o' = h(o).f \quad _ m(_)\{e\} \in h(o').\text{class.methods}_{\vec{D}} \quad h(o').\text{usage} \xrightarrow{m} \mathcal{W}}{\vdash_{\vec{D}} \langle h, env_S, f.m(v) \rangle \rightarrow \langle h\{\mathcal{W}/h(o').\text{usage}\}, (o', [x \mapsto v]) \cdot env_S, \text{return}\{e\} \rangle} \\ (\text{RET}) & \frac{v \neq v' \Rightarrow \neg \text{lin}(v', h)}{\vdash_{\vec{D}} \langle h, (o, [x \mapsto v']) \cdot env_S, \text{return}\{v\} \rangle \rightarrow \langle h, env_S, v \rangle} \end{aligned}$$

The rule LBL shows how a loop iteration is performed by substituting instances of `continue k` with the expression defined for the associated label. In CALLF the premise describes how an object must follow the usage described by its current typestate. A method m can only be called if the usage of the object allows an m transition and the result of this evaluation is that the usage of the object is updated and the next evaluation step is set to the method body

e by wrapping the expression in a special $\text{return}\{e\}$ statement; this is a run-time extension of the syntax that lets us record method calls waiting to be completed. The **RET** rule describes how a value is returned from a method call, by unpacking the **return** statement into the value, while popping the call stack. For details on the run-time syntax, see the technical report.

5 Results About the Type System

The first important result is the soundness of the type system, as usual shown in two steps: subject-reduction and type-safety. So, firstly, well-typed programs remain well-typed during execution. In our setting this means that when a well-typed configuration reduces, it leads to a configuration that is also well-typed. A configuration is well-typed if its bindings match the type information given: The heap matches the field typing environment Λ , the stack Δ in the type system matches the stack from the semantics, the objects mentioned in the type system match those of e , the expression e itself is well typed and the field type environment Λ is compatible with the program \vec{D} . If this is the case, we write $\Lambda, \Delta \vdash_{\vec{D}} \langle h, env_S, e \rangle : t \triangleright \Lambda'$. Let Δ and Δ' be sequences of object type and parameter type environments, defined as $\Delta = envT_O \cdot envT_S$. An object type environment $envT_O$ maps object names to typestates.

$$envT_O : \mathbf{ONames} \rightarrow \mathbf{Typestates}$$

A parameter type environment $envT_S$ is a sequence of pairs $(o, [x \mapsto t])$ mapping an object o to a parameter binding $[x \mapsto t]$.

To prove Theorem 1, we need typing rules for the dynamic syntax. Rule (**TRET**) is one such rule; it also becomes central when proving Theorem 3.

$$(\mathbf{TRET}) \frac{\Lambda; \Delta \vdash_{\vec{D}} e : t \triangleright \Lambda'; \Delta' \quad \Delta' = \Delta'' \cdot (o', [x \mapsto t']) \quad \text{terminated}(t')}{\Lambda; \Delta \cdot (o, S) \vdash_{\vec{D}} \text{return}\{e\} : t \triangleright \Lambda'; \Delta'' \cdot (o, S)}$$

The rule tell us that a $\text{return}\{e\}$ expression is well-typed if the expression body e is well-typed and the method parameter used in the expression body is terminated after the execution. The final type environment is one, in which the parameter stack environment does not mention the called object and its associated parameter. The intention is that this mirrors the modification of the stack environment in the semantics as expressed in the reduction rule (**RET**).

Theorem 1 (Subject reduction). *Let \vec{D} be such that $\vdash \vec{D}$ and let $\langle h, env_S, e \rangle$ be a configuration. If $\vdash_{\vec{D}} \langle h, env_S, e \rangle \rightarrow \langle h', env'_S, e' \rangle$ then:*

$$\begin{aligned} & \exists \Lambda, \Delta . \Lambda, \Delta \vdash_{\vec{D}} \langle h, env_S, e \rangle : t \triangleright \Lambda'; \Delta' \text{ implies} \\ & \exists \Lambda^N, \Delta^N . \Lambda^N, \Delta^N \vdash_{\vec{D}} \langle h', env'_S, e' \rangle : t \triangleright \Lambda''; \Delta', \text{ where } \Lambda'(o) = \Lambda''(o) \text{ and } o \\ & \text{is the active object in the resulting configuration.} \end{aligned}$$

Secondly, a well-typed program will never go wrong. In our case, this means that a well-typed program does not attempt *null-dereferencing*, and all method calls follow the specified usages. We formalize the notion of run-time error via a predicate and write $\langle h, env_S, e \rangle \longrightarrow_{\text{err}}$ whenever $\langle h, env_S, e \rangle$ has an error. Rules (NC-1) and (NC-2) describe two cases of null-dereferencing that occur when the object invoked by method m has been nullified.

$$\text{(NC-1)} \quad \frac{h(o).f = \text{null}}{\langle h, (o, s) \cdot env_S, f.m(v) \rangle \longrightarrow_{\text{err}}}$$

$$\text{(NC-2)} \quad \langle h, (o, [x \mapsto \text{null}]) \cdot env_S, x.m(v) \rangle \longrightarrow_{\text{err}}$$

Theorem 2 (Type safety). *If $\Lambda; \Delta \vdash_{\vec{D}} \langle h, env_S, e \rangle : t \triangleright \Lambda'; \Delta'$ and $\langle h, env_S, e \rangle \rightarrow^* \langle h', env'_S, e' \rangle$ then $\langle h', env'_S, e' \rangle \not\rightarrow_{\text{err}}$*

The second important result concerning type-checking is that well-typed terminated programs do not leave object protocols incomplete. It is a direct consequence of requiring in rule **TClass** the field typing environment to be terminated and of the fact that the only active object is the main one, o_{main} , which have the end usage).

Theorem 3 (Protocol Completion). *Let $\vdash \vec{D}$. For all reachable configurations $\langle h, env_S, e \rangle_{\vec{D}}$ such that $\langle h, env_S, e \rangle \not\rightarrow$, we have $env_S = (o_{\text{main}}, s_{\text{main}})$, $e = v$, and moreover, $\text{terminated}(h)$.*

6 Usage Inference

In this section, we outline a usage inference algorithm and present results for the inferred usages. The algorithm infers usages from class declarations in order to improve usability: it allows programmers to abstain from specifying usages for all class declarations and facilitates type checking with less usages annotations in the source program. The algorithm works on an acyclic graph of dependencies between class declarations. This dependency graph defines the order in which inference takes place, where the usage for a class C can only be inferred if the usages of its fields have been explicitly declared or previously inferred.

The inference algorithm returns a usage for any given class declaration. It considers how method call sequences of the class affect its field typing environments by using the type rules defined in Table 2. The process can be described by the following three steps: **Step 1.** Extract allowed method sequences. **Step 2.** Filter non-terminating sequences. **Step 3.** Convert sequences into usages.

Step 1 is the most interesting since it establishes the possible sequences of method calls in relation to the type system by using a transition relation \rightarrow given by rule (CLASS) shown below. This rule states that a method call m initiated from field typing environment Φ and results in Φ' is allowed if the method body is well-typed and its parameter is terminated at the end. The (CLASS) rule, when

applied to a class declaration, defines a transition system where field typing environments are states and transitions between them are method calls.

$$(\text{CLASS}) \frac{\frac{\{\text{this} \mapsto \Phi\}; \emptyset \cdot (\text{this}, [x \mapsto t]) \vdash^\emptyset e : t' \triangleright \{\text{this} \mapsto \Phi'\}; \emptyset \cdot (\text{this}, [x \mapsto t''])}{t' m(t x) \{e\} \in C.\text{methods}} \quad -\text{lin}(t'')}{\Phi \xrightarrow{m} \Phi'}}$$

Note that the premise of the (CLASS) rule is similar to the premise of (TCBR) and that a method transition is only available if the method body is well typed according to (TCBR). From the transition system defined by \rightarrow , we filter out all transitions that cannot reach a terminated environment, since these environments will result in protocols that cannot terminate.

Example 5. Recall the `FileReader` example from Listing 1.2. We wish to infer the usage for the `FileReader` class. Since the class has a field of type `File`, the usage of the `File` class must be known. The inference algorithm starts from the initial field typing environment $\{\text{file} : \perp\}$. Using the (CLASS) rule we see that only the method body of `init` is well typed in the initial field typing environment, thus updating the field typing environment $\{\text{file} : \text{File}[\mathcal{U}]\}$. The procedure is then repeated for the updated environment where a call to `readFile` is now possible. The field is now terminated and the algorithm proceeds to convert the transition system into a usage. This conversion is done by representing each state of the transition system as a usage variable and each transition as a branch usage. The resulting usage for class `FileReader` is: $X^{X=\{\text{init}; X'\} \quad X'=\{\text{readFile}; \text{end readFile}; X\}}$ ■

A correctly inferred usage, in our case, is the *principal usage* for a class declaration. A principal usage is the most permissible usage hence it includes the behaviours of all usages that well-type a class. In other words, the most permissible usage allows all sequences of method calls, for a class declaration, that do not lead to null-dereferencing errors and can terminate. We define principality in terms of a simulation ordering \sqsubseteq where the principal usage can simulate any usage that would make the class well typed. Let R be a binary relation on usages. We call R a usage simulation iff for all $(\mathcal{U}_1, \mathcal{U}_2) \in R$ we have that:

1. If $\mathcal{U}_1 \xrightarrow{m} \mathcal{U}'_1$ then $\mathcal{U}_2 \xrightarrow{m} \mathcal{U}'_2$ such that $(\mathcal{U}'_1, \mathcal{U}'_2) \in R$
2. If $\mathcal{U}_1 \xrightarrow{l} \mathcal{U}'_1$ then $\mathcal{U}_2 \xrightarrow{l} \mathcal{U}'_2$ such that $(\mathcal{U}'_1, \mathcal{U}'_2) \in R$

We say that \mathcal{U} is a subusage of \mathcal{U}' , written $\mathcal{U} \sqsubseteq \mathcal{U}'$, if for some usage simulation R we have $(\mathcal{U}, \mathcal{U}') \in R$. Principal usages are always recursive since there is no difference between a field typing environment at the initial state and at the end as all fields are non-linear in accordance with protocol completion. A method sequence that results in a terminated environment can be repeated any number of times and remain well typed. To allow usages to express termination or repetition, we allow a limited form of non-determinism to choose between `end` and repeating the protocol, as in the definition of X' above. If \mathcal{U}_I is the inferred usage for a class C , then C is well-typed with this usage. Moreover, the inferred usage is principal wrt. usage simulation.

Theorem 4 (Principality). *If \mathcal{U}_I is the inferred usage for class C , then \mathcal{U}_I is the largest usage for C that makes C well typed. That is, $\vdash_{\vec{D}} \text{class } C \{ \mathcal{U}_I, \vec{F}, \vec{M} \}$ and $\forall \mathcal{U}. \vdash_{\vec{D}} \text{class } C \{ \mathcal{U}, \vec{F}, \vec{M} \} \implies \mathcal{U} \sqsubseteq \mathcal{U}_I$.*

7 Conclusions and Future Work

In this paper we present a behavioural type system for the **Mungo** language, also implemented in the form of a type-checker. Every class is annotated with usages that describe the protocol to be followed by method calls to object instances of the class. Moreover, object references can only be used in a linear fashion. The type system provides a formal guarantee that a well-typed program will satisfy three important properties: memory-safety, and object protocol fidelity and object protocol completion. Behavioural types are essential, as they allow variables of class type to have a type that evolves to \perp , which is the only type inhabited by the null value. This is in contrast to most type systems for Java like languages that do not let types evolve during a computation and overload null to have any type.

Furthermore, a contribution that is novel and in contrast to other typestate-based approaches is that it is possible to infer usages from a class description. We have implemented a tool that can infer the most general usage wrt. a simulation preorder that makes the class well-typed.

In our type system, variables obey a very simple protocol of linearity: They must be written to once, then read once (otherwise they return a null value). The current version of the **Mungo** tool [23] allows for a representation of fields that can be used k times for $k > 1$ by having k field variables that are each used once. A natural extension of the system is therefore to allow for a richer language of safe variable protocols. An approach currently under investigation is to use ideas from the work on behavioural separation of Caires and Seco [7], by Franco et al. on SHAPES [13] and that of Militão et al. [25].

To be able to type a larger subset of Java than what **Mungo** currently allows, further work also includes adding inheritance to the language in a type-safe manner. Inheritance is common in object oriented programming, and would allow **Mungo** to be used for a larger set of programs. This is particularly important, since classes in languages like Java always implicitly inherit from the class `Object`. However, Grigore has shown that type checking for Java in the presence of full subtyping is undecidable [17]. Therefore, in further work, we need to be extremely careful when introducing subtyping into our system. Moreover, notice that this would require defining subtyping for class usages, a form of behavioural/session subtyping [4, 5, 16].

References

1. <https://github.com/MungoTypesystem/Mungo-Typechecker/ExamplePrograms/>
2. Aldrich, J.: The Plaid programming language (2010)

3. Ancona, D., et al.: Behavioral types in programming languages. *Found. Trends Program. Lang.* **3**(2–3), 95–230 (2016)
4. Bravetti, M., Carbone, M., Zavattaro, G.: Undecidability of asynchronous session subtyping. *Inf. Comput.* **256**, 300–320 (2017)
5. Bravetti, M., Carbone, M., Zavattaro, G.: On the boundary between decidability and undecidability of asynchronous session subtyping. *Theoret. Comput. Sci.* **722**, 19–51 (2018)
6. Bravetti, M., Zavattaro, G.: Process calculi as a tool for studying coordination, contracts and session types. *J. Logical Algebraic Methods Program.* **112**, 100527 (2020)
7. Caires, L., Seco, J.C.: The type discipline of behavioral separation. In: The 40th Symposium on Principles of Programming Languages, POPL 2013, pp. 275–286. ACM (2013)
8. de Boer, F.S., Bravetti, M., Lee, M.D., Zavattaro, G.: A petri net based modeling of active objects and futures. *Fundamenta Informaticae* **159**(3), 197–256 (2018)
9. DeLine, R., Fähndrich, M.: Typestates for objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24851-4_21
10. Dietl, W., Dietzel, S., Ernst, M.D., Muslu, K., Schiller, T.W.: Building and using pluggable type-checkers. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 681–690. ACM (2011)
11. Fähndrich, M., DeLine, R.: Adoption and focus: practical linear types for imperative programming. In: Proceedings of PLDI 2002, pp. 13–24. ACM (2002)
12. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: Proceedings of OOPSLA 2003, pp. 302–312. ACM (2003)
13. Franco, J., Tasos, A., Drossopoulou, S., Wrigstad, T., Eisenbach, S.: Safely abstracting memory layouts. *CoRR*, [abs/1901.08006](https://arxiv.org/abs/1901.08006) (2019)
14. Garcia, R., Tanter, É., Wolff, R., Aldrich, J.: Foundations of typestate-oriented programming. *Trans. Program. Lang. Syst.* **36**(4), 1–44 (2014)
15. Gay, S.J., Gesbert, N., Ravara, A., Vasconcelos, V.T.: Modular session types for objects. *Logical Methods Comput. Sci.* **11**(4), 1–76 (2015)
16. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. *Acta Informatica* **42**(2–3), 191–225 (2005)
17. Grigore, R.: Java generics are turing complete. In: Proceedings of POPL 2017, pp. 73–85. ACM (2017)
18. Hoare, T.: Null references: the billion dollar mistake (2009)
19. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_35
20. Hubert, L., Jensen, T., Pichardie, D.: Semantic foundations and inference of non-null annotations. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 132–149. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68863-1_9
21. Hüttel, H., et al.: Foundations of session types and behavioural contracts. *ACM Comput. Surv.* **49**(1), 1–36 (2016)
22. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for Java and GJ. *Trans. Program. Lang. Syst.* **23**(3), 396–450 (2001)
23. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo: a session type toolchain for Java. *Sci. Comput. Program.* **155**, 52–75 (2018)

24. Meyer, B.: Ending null pointer crashes. *Commun. ACM* **60**(5), 8–9 (2017)
25. Militão, F., Aldrich, J., Caires, L.: Aliasing control with view-based typestate. In: *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs, FTFJP 2010*. ACM (2010)
26. Milner, R.: A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* **17**, 348–375 (1978)
27. Nierstrasz, O.: Regular types for active objects. In: *Proceedings of the 8th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1993)*, pp. 1–15. ACM (1993)
28. Siek, J., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) *ECOOP 2007*. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73589-2_2
29. Strom, R.E., Yemini, S.: Typestate: a programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.* **12**(1), 157–171 (1986)
30. Sunshine, J.: Protocol programmability. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2013). AAI3578659
31. Sunshine, J., Stork, S., Naden, K., Aldrich, J.: Changing state in the plaid language. In: *Companion to OOPSLA 2011*, pp. 37–38. ACM (2011)
32. The Jedis Project: Jedis (2011–2019). <https://github.com/xetorthio/jedis/>
33. The Redis Project: Redis (2011–2019). <https://redis.io/>
34. Voinea, A.L., Dardha, O., Gay, S.J.: Typechecking Java protocols with [St]Mungo. In: Gotsman, A., Sokolova, A. (eds.) *FORTE 2020*. LNCS, vol. 12136, pp. 208–224. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50086-3_12