Papaya: Global Typestate Analysis of Aliased Objects

Mathias Jakobsen
University of Glasgow
School of Computing Science
United Kingdom
m.jakobsen.1@research.gla.ac.uk

Alice Ravier
University of Glasgow
School of Computing Science
United Kingdom
2206245r@student.gla.ac.uk

Ornela Dardha
University of Glasgow
School of Computing Science
United Kingdom
ornela.dardha@glasgow.ac.uk

ABSTRACT

Typestates are state machines used in object-oriented programming to specify and verify correct order of method calls on an object. To avoid inconsistent object states, typestates enforce *linear typing*, which eliminates—or at best limits—*aliasing*. However, aliasing is an important feature in programming, and the state-of-the-art on typestates is too restrictive if we want typestates to be adopted in real-world software systems.

In this paper, we present a type system for an object-oriented language with typestate annotations, which allows for *unrestricted* aliasing, and as opposed to previous approaches it does not require linearity constraints. The typestate analysis is *global* and tracks objects throughout the entire program graph, which ensures that well-typed programs *conform* and *complete* the declared protocols. We implement our framework in the Scala programming language and illustrate our approach using a running example that shows the interplay between typestates and aliases.

CCS CONCEPTS

• **Theory of computation** → **Formalisms**; • **Software and its engineering** → *Object oriented languages.*

KEYWORDS

typestates, object-oriented programming

ACM Reference Format:
Mathias Jakobsen, Alice Ravier, and Ornela Dardha. 2021. Papaya: Global Typestate Analysis of Aliased Objects. In *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021), September 6–8, 2021, Tallinn, Estonia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3479394.3479414

1 INTRODUCTION

In class-based object-oriented programming languages, a class defines a number of methods that can be invoked on an object of that class. Often, however, there is an implicit *order* imposed on methods, where some methods should be called before others. For example, a server connection must be opened before sending data, or we might want a clean-up method to be called before freeing resources. These method orderings, or *protocols*, are often defined in varying degrees of formality through documentation or comments, which makes the process difficult and error-prone. Work has been undertaken to include these protocols in the program itself with the introduction of *typestates* for object-oriented languages [1, 3, 8, 21].

Common to many of these approaches is that they rely on a *linear* type system, where only a single reference to an object can exist, thus eliminating–or limiting–aliasing. In Mungo [3, 21] linearity is always enforced, whereas other approaches in languages such as Plaid [2] and Vault [7] allow limited aliasing, while preserving compositionality of the type system such that each class can be type checked in isolation [11, 24]. These approaches often require programmer annotations to deal with aliasing control or they simply eliminate aliasing altogether.

The difficulty with aliasing in the presence of typestates is that if multiple references exist to a single object, then operations on one object reference can affect the type of multiple other references as well. This is further complicated if we allow aliases to be stored in fields on multiple objects. Consequently, operations on objects of one class impact the well-typedness of other classes, potentially leading to inconsistent object states. Looking at the problem from a more 'technical' angle, the difficulty with aliasing in the presence of typestates is due to the discrepancy between the *compositional* nature of typestate-based type systems and the *global* nature of aliasing. To address aliasing one can either (i) allow limited access to an object through aliasing control mechanisms or (ii) if we want unrestricted aliasing then use a form of global analysis. The problem with (i) is that it not trivial to find an alias control mechanism to capture OO programming idioms, and for (ii) while we benefit from the most flexible form of aliasing, the drawback is that we lose compositionality. With the above in mind we pose our research question:

RQ: *Can we define a typestate-based type system for object-oriented languages that guarantees protocol conformance and completion while allowing unrestricted aliasing?*

In this paper, we answer positively our research question and introduce a global approach to type checking object-oriented programs with typestates, which allows *unrestricted aliasing*, meaning that objects can be freely aliased, and stored in fields of other objects. This is more representative of the sort of aliasing that can occur in real-world programs. In this work we treat typestates in a similar fashion to the line of work on Mungo [3, 21] and along the same lines, we introduce Papaya, an implementation of a typestate-based type system for Scala.

Contributions. The contributions of this paper are as follows.

- **Typestates for Aliased Objects**. We formalise an object-oriented language with typestate annotations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPDP 2021, September 6–8, 2021, Tallinn, Estonia
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8689-0/21/09…$15.00
https://doi.org/10.1145/3479394.3479414

  - Section 3 presents the syntax; Section 4 presents the type system that performs global typestate analysis of unrestricted aliased objects and Section 5 presents the operational semantics.
  - Section 6 covers the meta-theory of our formalisation and we show that our type system is safe by proving subject reduction (Theorem 1), progress (Theorem 2), protocol conformance (Corollary 1) and protocol completion (Lemma 2).
- **Papaya Tool**. Section 7 presents the Papaya tool, an implementation of our type system for Scala. Protocols are expressed as Scala objects and are added to Scala classes with the `@Typestate` annotation. Following the formalisation, our implementation allows for unrestricted aliasing, where objects are checked if they conform and complete their declared protocols.
- **The BankAccount Example.** We illustrate our work with a running example (starting in Section 2), which features aliasing. We show how the program is typed in our type system (from Section 4) and we implement it in Scala (in Section 7) where use Papaya to perform typestate checking.

In Section 8 we discuss related work on typestates and aliasing. Finally, in Section 9 we conclude the paper and present ideas for future work.

2 OVERVIEW

We introduce our approach with an example, which is inspired by [20]. The example is shown using the calculus that will be defined in Section 3 with the addition of some base types and operations on those. For completeness, since the calculus requires a formal parameter for all methods, one could pass the unit value as an argument. For readability we omit the argument instead.

Consider the class `BankAccount` shown in Listing 1. It is a simple wrapper class around a field storing an amount of money. Notice that there is an implicit ordering of method calls, which the programmer might assume will be followed when using the class: the amount of money should be set *prior* to using the value of the field, and interest should be applied *after* setting the money; finally, the money variable should only be read *after both* setting the money and applying the interest has occurred, so that an intermediate value is not returned.

```
1  class BankAccount[{setMoney;
2                     {applyInterest;
3                     {getMoney; end}}}] {
4    val amount:float;
5    fun setMoney(d:float):void  {
6      this.amount = d;
7    }
8    fun getMoney():float {
9      this.amount;
10   }
11   fun applyInterest(rate:float):void {
12     this.amount = this.amount * rate;
13   }
14 }
```

Listing 1: Wrapper class around an amount of money

We can express this implicit order of method calls as an explicit *usage*:

$$\{\texttt{setMoney}; \{\texttt{applyInterest}; \{\texttt{getMoney}; \texttt{end}\}\}\}$$

where $\{m_i; w_i\}_{i \in I}$ denotes that a method m_j where $j \in I$ can be called, with the continuation usage w_j. This usage states that the first method called should be `setMoney`, followed by a call to `applyInterest` and finally one to `getMoney`.

We introduce two additional classes as shown in Listings 2 and 3. The `SalaryManager` class adds money to a `BankAccount` and applies a fixed interest rate. The `DataStorage` class fetches the value of a `BankAccount` and stores it in a database.

```
15 class SalaryManager[{setAccount;
16                      {addSalary; end}}] {
17   val account:BankAccount
18   fun setAccount(ms:BankAccount):void {
19     this.account = ms;
20   }
21   fun addSalary(amount:float):void {
22     this.account.setMoney(amount);
23     this.account.applyInterest(1.05);
24   }
25 }
```

Listing 2: Salary manager that adds funds to a `BankAccount` object

```
26 class DataStorage[{setAccount;
27                    {store; end}}] {
28   val account:BankAccount
29   fun setAccount(ms:BankAccount):void {
30     this.account = ms;
31   }
32   fun store():void {
33     this.account.getMoney();
34     // store value in database
35   }
36 }
```

Listing 3: Data storage class that reads the funds of a `BankAccount` object

Note that in the three classes we defined so far, there is no explicit mentioning of possible aliasing. However, as shown in Listing 4, an instance of class `BankAccount` can be aliased and shared between the manager and data store, as long as the joined operations on the instance respect its usage.

```
37  account = new BankAccount;
38  manager = new SalaryManager;
39  db = new DataStorage;
40
41  manager.setAccount(account);
42  db.setAccount(account);
43
44  manager.addSalary(100.0);
45  db.store();
```

Listing 4: Aliasing of a `BankAccount` object

If we were to swap lines 44 and 45, then they would no longer follow the protocol, as the data store would call `getMoney` before `setMoney` and `applyInterest` were called.

3 THE LANGUAGE

We introduce an object-oriented calculus with classes and enumeration types, similar to previous work on Mungo [3, 6, 21, 22, 37].

The syntax of terms in the calculus is shown in Figure 1. For a sequence $\phi_1, \phi_2, \ldots \phi_n$ we write $\overline{\phi}$ and let $|\overline{\phi}| = n$. A program is a list of class and enum-definitions $\overline{D}$, followed by a class Main which contains the main method. A class definition contains the initial protocol, or usage $\mathcal{U}$, field declarations $\overline{F}$ and method declarations $\overline{M}$. For expressions, the language supports assignment, object initialisation, method calls (on fields, parameters or on the object itself). Note that for simplicity and readability of typing rules later on, method calls and field access use an object-reference o as the target, thus call-chaining and nested field access is not allowed. However, the language can be easily extended to facilitate these features, requiring an extra object look-up in the relevant typing rules. The only object reference that can occur in program text is the this reference. The language also supports control structures (conditionals, loops, and sequential composition) and match expressions (switch on an enumeration type). Loops are formalised with a jump-style loop with labelled expressions and continue statements in line with Mungo work.

$$
\begin{aligned}
D &::= \mathsf{class}\ C\{\mathcal{U}, \overline{F}, \overline{M}\} \mid \mathsf{enum}\ L\{\overline{l}\} \\
F &::= \mathsf{val}\ f : t \\
M &::= \mathsf{fun}\ m(x : t) : t\ \{e\} \\
r &::= o \mid o.f \mid x \\
e &::= o.f = e \mid o.f = \mathsf{new}\ C \mid e; e \mid r.m(e) \mid \mathsf{unit} \mid o.f \mid x \\
&\quad \mid \mathsf{if}\ (e)\ \{e\}\ \mathsf{else}\ \{e\} \mid o.l \mid \mathsf{match}(e)\{\overline{l : e}\} \mid \mathsf{null} \\
&\quad \mid \mathsf{true} \mid \mathsf{false} \mid k : e \mid \mathsf{continue}\ k
\end{aligned}
$$

Figure 1: Syntax of class definitions

The syntax of types is shown in Figure 2 and it contains the object types $o[C, \mathcal{U}]$, base types bool and void, the *null-type* $\bot$, and enumeration types L and L link o. The shaded production rules indicate run-time syntax. An object type $o[C, \mathcal{U}]$ is composed of an object reference o, which is a unique identifier to a single object, a class name C and a current usage $\mathcal{U}$ describing the remaining protocol of the object. The enumeration type L link o introduced in [36] is used to track updates in switch-statements and are not declared in the program text.

$$
\begin{aligned}
t &::= C \mid \mathsf{void} \mid \mathsf{bool} \mid L \\
T &::= o[C, \mathcal{U}] \mid \mathsf{void} \mid \bot \mid \mathsf{bool} \mid L \mid L\ \mathsf{link}\ o \\
\mathcal{U} &::= \mu X.\mathcal{U} \mid X \mid \{\overline{m; w}\} \mid \mathsf{end} \\
w &::= \langle \overline{l : \mathcal{U}} \rangle \mid \mathcal{U}
\end{aligned}
$$

Figure 2: Syntax of types

Definition 1 presents a labelled transition system for usages, annotated with the method call or the enumeration label, depending on the action performed. If an object has type $o[C, \mathcal{U}]$, then the transitions of $\mathcal{U}$ describe the permitted operations on the object referenced by o. As previously described, branch usages $\{m_i; w_i\}_{i \in I}$ describe a set of available methods, each with a continutation usage. Choice usages $\langle l_i : \mathcal{U}_i \rangle_{i \in I}$ describe that based on a enumeration label l_j, the protocol continues with protocol $\mathcal{U}_j$. Recursive behaviour can be specified with recursive usages $\mu X.\mathcal{U}$ and the end usage denotes the terminated protocol which has no transitions.

Definition 1 (LTS on Usages)**.**

$$
\frac{j \in I}{\{m_i; w_i\}_{i \in I} \xrightarrow{m_j} w_j} \qquad \frac{j \in I}{\langle l_i : \mathcal{U}_i \rangle_{i \in I} \xrightarrow{l_j} \mathcal{U}_j}
$$

$$
\frac{\mathcal{U}\{X/\mu X.\mathcal{U}\} \to \mathcal{U}'}{\mu X.\mathcal{U} \xrightarrow{\alpha} \mathcal{U}'}
$$

We define a notion of well-formedness for expressions (Definition 2), which requires that continue expressions do not show up in places where, after loop unfolding, they would be followed by other expressions. Examples of ill-formed expressions include $o.m(\mathsf{continue}\ k)$ and $\mathsf{continue}\ k; o.m(\mathsf{unit})$. Furthermore, well-formedness also requires a labelled expression has a terminating branch so that $k : \mathsf{if}\ (\mathsf{true})\ \{\mathsf{continue}\ k\}\ \mathsf{else}\ \{\mathsf{unit}\}$ is well-formed whereas $k : \mathsf{continue}\ k$ is not.

Definition 2 (Well-formedness)**.** An expression e is well-formed if:

(1) No expression follows a continue expression after unfolding continue expressions in e
(2) No free loop-variables in e
(3) All continue expressions in e are guarded by a branching (if or match) expression
(4) There must be a branch in all labelled expressions in e that does not end with a continue expression

We conclude with the definition of well-formed methods.

Definition 3 (Well-formed methods)**.** A method declaration $\mathsf{fun}\ m(x : t) : t\ \{e\}$ is well formed if e is well formed and recursive calls are guarded by a branching expression.

4 TYPE SYSTEM

As opposed to previous type systems for Mungo [3, 21, 22] the type system presented here performs a *global* analysis of the program, in order to maintain a global view of aliasing while guaranteeing correct objects' states. This means that instead of relying on compositionality during type checking, we must explore the entire program graph. Consequently when a method call is encountered during type checking, the type system must ensure that the body of the method is well typed in the current typing environment.

We define a typing environment Γ using the production rules shown in Figure 3. A typing environment maps object references to an object-type as well as a field typing environment λ that contains the types for all fields in the object. Furthermore we use the notation $\Gamma[o \mapsto (T, \lambda)]$ to indicate an update of an existing binding for object o, and $\Gamma[o.f \mapsto o']$ to update the existing binding of a field of object o. A typing environment can only contain a single binding for each object reference o. Similarly, a field typing environment can only contain a single binding for each field name.

$$
\begin{aligned}
\Gamma &::= \emptyset \mid \Gamma, o \mapsto (T, \lambda) \\
\lambda &::= \emptyset \mid \lambda, f \mapsto z \\
z &::= \mathsf{basetype\ bool} \mid \mathsf{basetype\ void} \\
&\quad \mid \mathsf{basetype}\ \bot \mid\ \mathsf{basetype}\ L \\
&\quad \mid \mathsf{reference}\ o
\end{aligned}
$$

Figure 3: Syntax of typing environments

We define the initial field environment given a set of field declarations $\overline{F}$.inittypes. This is used when initialising new objects. Fields with class types are given the initial type of $\bot$ whereas fields of base types retain that type in the field environment.

$$
\begin{aligned}
(\overline{F}, \mathsf{var}\ f : C).\mathsf{inittypes} &= \overline{F}.\mathsf{inittypes}, f \mapsto \mathsf{basetype}\ \bot \\
(\overline{F}, \mathsf{var}\ f : \mathsf{bool}).\mathsf{inittypes} &= \overline{F}.\mathsf{inittypes}, f \mapsto \mathsf{basetype\ bool} \\
(\overline{F}, \mathsf{var}\ f : \mathsf{void}).\mathsf{inittypes} &= \overline{F}.\mathsf{inittypes}, f \mapsto \mathsf{basetype\ void} \\
(\overline{F}, \mathsf{var}\ f : L).\mathsf{inittypes} &= \overline{F}.\mathsf{inittypes}, f \mapsto \mathsf{basetype}\ L \\
\emptyset.\mathsf{inittypes} &= \emptyset
\end{aligned}
$$

We also define the following shorthand functions for extracting information from the typing environment and class definitions.

$$
\begin{aligned}
(o[C, \mathcal{U}], \lambda).\mathsf{class} &\triangleq C & (o[C, \mathcal{U}], \lambda).f &\triangleq \lambda(f) \\
(o[C, \mathcal{U}], \lambda).\mathsf{usage} &\triangleq \mathcal{U} & (T, \lambda).\mathsf{type} &\triangleq T \\
(o[C, \mathcal{U}], \lambda).\mathsf{reference} &\triangleq o & (T, \lambda).\mathsf{fields} &\triangleq \lambda
\end{aligned}
$$

For a class name C where class $C\{\mathcal{U}, \overline{F}, \overline{M}\} \in \overline{D}$ we let $\overline{D}(C) = \mathsf{class}\ C\{\mathcal{U}, \overline{F}, \overline{M}\}$ and define the following functions.

$$
\begin{aligned}
(\mathsf{class}\ C\{\mathcal{U}, \overline{F}, \overline{M}\}).\mathsf{usage} &\triangleq \mathcal{U} \\
(\mathsf{class}\ C\{\mathcal{U}, \overline{F}, \overline{M}\}).\mathsf{fields} &\triangleq \overline{F} \\
(\mathsf{class}\ C\{\mathcal{U}, \overline{F}, \overline{M}\}).\mathsf{methods} &\triangleq \overline{M}
\end{aligned}
$$

The type system is driven by the following (Main) rule, which states that if the main method is well typed, then the entire program is well typed. As previously mentioned, the type system will expand method calls, hence the type system will visit all reachable parts of the program. In the (Main) rule we require term(Γ) meaning that the resulting type environment must be *terminated*, meaning that protocols must be finished for all objects. term is defined as:

$$
\mathsf{term}(\Gamma) \Leftrightarrow \forall o \in \mathsf{dom}(\Gamma).\ \Gamma(o).\mathsf{usage} = \mathsf{end}
$$

(Main)

$$
\frac{\begin{array}{c}\mathsf{Main}\{\mathcal{U}, \overline{F}, \overline{M}\} \in \overline{D} \\ \overline{M} = \{\mathsf{fun\ main(void}\ x)\ \{e\}\} \qquad \mathcal{U} = \{\mathsf{main; end}\} \\ \emptyset; \emptyset; \{o_{\mathsf{main}} \mapsto (\mathsf{Main[end]}, \overline{F}.\mathsf{initvals})\} \vdash e : T \dashv \Gamma \qquad \mathsf{term}(\Gamma)\end{array}}{\vdash \overline{D} : \mathsf{ok}}
$$

Judgments for type checking expressions are of the form $\Theta; \Omega; \Gamma \vdash e : T \dashv \Gamma'$. The environments Ω and Θ are used to track labelled expressions and recursive method calls respectively. The label environment Ω relates loop labels k to typing environments Γ such that when encountering a continue expression we can compare the current typing environment to the initial typing environment when entering the loop. This will be explained in detail later. The recursion environment Θ serves the same purpose but for recursive method calls instead. As method calls are expanded in the type systems, recursive method definitions will lead to infinite type checking if not handled carefully. By keeping track of the currently expanded methods, the type system can terminate type checking after a single expansion of each method.

Returning back to the format of judgments, $\Theta; \Omega; \Gamma \vdash e : T \dashv \Gamma'$, we can now describe the meaning of the judgment. Given initial environments Θ, Ω, Γ, evaluating the expression e will result in a value of type T and a possibly updated typing environment Γ'. We assume that $\overline{D}$ is globally available in the rules, and contains the class definitions of the program.

The first rules found in Figure 4 describe object operations such as reading fields or parameters, assigning fields and object initialisation. They are for the most part standard, although the rules for method calls require some further description.

The rules for direct method calls, which is used for typing method calls on parameters or the this object, are defined in (Call-d) and (Call-d-rec). In (Call-d) the method body is expanded, and the current typing environment from before the unfolding is stored in the recursion environment Θ. Upon reaching a recursive call inside the method body the (Call-d-rec) rule is used to compare the current typing environment to the one stored in Θ. This enforces that upon making a recursive call, the typing environment should be the same as it was for the initial method call. If this is the case, then we can terminate type checking of the call-chain, as we have checked this exact configuration already with the initial call. So the combination of the two rules (Call-d) and (Call-d-rec) gives us a recursive typing behaviour, with the base case defined by (Call-d-rec). This exact behaviour is repeated for indirect calls which are used for fields, as illustrated in the rules (Call-ind) and (Call-ind-rec).

Four auxiliary functions are used in the rules:

agree checks that a value of type T matches the one defined in the program text as t. This allows null to be written to fields with

$\boxed{\Theta; \Omega; \Gamma \vdash e : T \dashv \Gamma'}$

(Assign)
$$\frac{\Theta; \Omega; \Gamma \vdash e : T \dashv \Gamma' \qquad \Gamma'(o).\mathsf{class.fields}(f) = \mathsf{var}\ f : t \qquad \mathsf{agree}(t, T)}{\Theta; \Omega; \Gamma \vdash o.f = e : \mathsf{void} \dashv \Gamma'[o.f \mapsto \mathsf{vtype}(T)]}$$

(Field)
$$\frac{\Gamma(o).\mathsf{fields}(f) = z \qquad T = \mathsf{getType}(\Gamma, z)}{\Theta; \Omega; \Gamma \vdash o.f : T \dashv \Gamma}$$

(New)
$$\frac{o' \text{ fresh} \qquad \overline{D}(C) = \mathsf{class}\ C\{\mathcal{U}, \overline{F}, \overline{M}\} \qquad \mathsf{val}\ f : C \in \overline{D}(\Gamma'(o).\mathsf{class}).\mathsf{fields}}{\Theta; \Omega; \Gamma \vdash o.f = \mathsf{new}\ C : \mathsf{unit} \dashv (\Gamma, o' \mapsto (o'[C, \mathcal{U}], \overline{F}.\mathsf{inittypes}))[o.f \mapsto o']}$$

(Unit)
$$\frac{}{\Theta; \Omega; \Gamma \vdash \mathsf{unit} : \mathsf{void} \dashv \Gamma}$$

(Bool)
$$\frac{v \in \{\mathsf{true}, \mathsf{false}\}}{\Theta; \Omega; \Gamma \vdash v : \mathsf{bool} \dashv \Gamma}$$

(Enum)
$$\frac{l \in L}{\Theta; \Omega; \Gamma \vdash o.l : L\ \mathsf{link}\ o \dashv \Gamma}$$

(Null)
$$\frac{}{\Theta; \Omega; \Gamma \vdash \mathsf{null} : \bot \dashv \Gamma}$$

(Const)
$$\frac{l \in L}{\Theta; \Omega; \Gamma \vdash o.l : L \dashv \Gamma}$$

(Obj)
$$\frac{\Gamma(o) = (o[C, \mathcal{U}], \lambda)}{\Theta; \Omega; \Gamma \vdash o : o[C, \mathcal{U}] \dashv \Gamma}$$

(Call-d)
$$\frac{\begin{array}{c}\Theta; \Omega; \Gamma \vdash e : T \dashv \Gamma'' \qquad \Gamma''(o) = (o[C, \mathcal{U}], \lambda) \qquad \mathcal{U} \xrightarrow{m} \mathcal{U}' \\ \mathsf{fun}\ m(x : t) : t'\{e'\} \in \overline{D}(C).\mathsf{methods} \qquad \mathsf{agree}(t, T) \\ (\Theta, o.m \mapsto \Gamma'''); \Omega; \Gamma''' \vdash e'\{\mathsf{this}/o\}\{x/\mathsf{getValue}(T')\} : T' \dashv \Gamma' \qquad \mathsf{returns}(t', T')\end{array}}{\Theta; \Omega; \Gamma \vdash o.m(e) : T' \dashv \Gamma'}$$

where $\Gamma''' = \Gamma''[o \mapsto (o[C, \mathcal{U}'], \lambda)$

(Call-d-rec)
$$\frac{\begin{array}{c}(\Theta, o.m \mapsto \Gamma''); \Omega; \Gamma \vdash e : T \dashv \Gamma''' \qquad \Gamma'''(o) = (o[C, \mathcal{U}], \lambda) \qquad \mathcal{U} \xrightarrow{m} \mathcal{U}' \\ \mathsf{agree}(t, T) \qquad \Gamma'' = \Gamma'''[o \mapsto (o[C, \mathcal{U}'], \lambda)]\end{array}}{(\Theta, o.m \mapsto \Gamma''); \Omega; \Gamma \vdash o.m(e) : T' \dashv \Gamma'}$$

(Call-ind)
$$\frac{\begin{array}{c}\Theta; \Omega; \Gamma \vdash e : T \dashv \Gamma'' \qquad \Gamma''(o).f = o' \\ \Gamma''(o') = (o'[C, \mathcal{U}], \lambda) \qquad \mathcal{U} \xrightarrow{m} \mathcal{U}' \\ \mathsf{fun}\ m(x : t) : t'\{e'\} \in \overline{D}(C).\mathsf{methods} \qquad \mathsf{agree}(t, T) \\ (\Theta, o'.m \mapsto \Gamma'''); \Omega; \Gamma''' \vdash e'\{\mathsf{this}/o\}\{x/\mathsf{getValue}(T')\} : T' \dashv \Gamma' \qquad \mathsf{returns}(t', T')\end{array}}{\Theta; \Omega; \Gamma \vdash o.f.m(e) : T' \dashv \Gamma'}$$

where $\Gamma''' = \Gamma''[o' \mapsto (o'[C, \mathcal{U}], \lambda)$

(Call-ind-rec)
$$\frac{\begin{array}{c}(\Theta, o'.m \mapsto \Gamma''); \Omega; \Gamma \vdash e : T \dashv \Gamma''' \qquad \Gamma'''(o).f = o' \qquad \Gamma'''(o') = (o'[C, \mathcal{U}], \lambda) \\ \mathcal{U} \xrightarrow{m} \mathcal{U}' \qquad \mathsf{agree}(t, T) \qquad \Gamma'' = \Gamma'''[o' \mapsto (o'[C, \mathcal{U}'], \lambda)]\end{array}}{(\Theta, o'.m \mapsto \Gamma''); \Omega; \Gamma \vdash o.f.m(e) : T' \dashv \Gamma'}$$

(If)
$$\frac{\Theta; \Omega; \Gamma \vdash e_1 : \mathsf{bool} \dashv \Gamma'' \qquad \Theta; \Omega; \Gamma'' \vdash e_2 : T \dashv \Gamma' \qquad \Theta; \Omega; \Gamma'' \vdash e_3 : T \dashv \Gamma'}{\Theta; \Omega; \Gamma \vdash \mathsf{if}\ (e_1)\ \{e_2\}\ \mathsf{else}\ \{e_3\} : T \dashv \Gamma'}$$

(Comp)
$$\frac{\Theta; \Omega; \Gamma \vdash e : T \dashv \Gamma'' \qquad \Theta; \Omega; \Gamma'' \vdash e' : T' \dashv \Gamma'}{\Theta; \Omega; \Gamma \vdash e; e' : T' \dashv \Gamma'}$$

(Label)
$$\frac{\Theta; \Omega, k \mapsto \Gamma; \Gamma \vdash e : T \dashv \Gamma'}{\Theta; \Omega; \Gamma \vdash k : e : T \dashv \Gamma'}$$

(Continue)
$$\frac{\Omega(k) = \Gamma}{\Theta; \Omega; \Gamma \vdash \mathsf{continue}\ k : T \dashv \Gamma'}$$

(Case)
$$\frac{\Theta; \Omega; \Gamma \vdash e : L\ \mathsf{link}\ o \dashv \Gamma'' \qquad \forall l_i \in L. \begin{cases} \Gamma''(o).\mathsf{usage} \xrightarrow{l_i} \mathcal{U}_i \\ \Theta; \Omega; \Gamma''[o.\mathsf{usage} \mapsto \mathcal{U}_i] \vdash e_i : T \dashv \Gamma' \end{cases}}{\Theta; \Omega; \Gamma \vdash \mathsf{match}(e)\{\overline{l : e}\} : T \dashv \Gamma'}$$

Figure 4: Typing rules for expressions

class types, and allows objects to be stored in fields with matching classes, no matter the particular protocol of the object.

$$\text{agree}(C, \bot) \quad \text{agree}(C, o[C, \mathcal{U}]) \quad \text{agree}(\text{bool}, \text{bool})$$
$$\text{agree}(\text{void}, \text{void}) \quad \text{agree}(L, L)$$

The returns predicate extends the agree predicate with an option to return a link type from a method, to support switching on choice usages by linking the enumeration type to an object.

$$\text{returns}(t, T) \Leftrightarrow \text{agree}(t, T) \vee (t = L \wedge T = L \text{ link } o)$$

getType and vtype are used for tagging and unpacking values for storing them in the typing environment. The reason we need this is to handle the indirection of an object reference o, so that we can look up the type of a field, with an extra access to the typing environment.

$$\begin{aligned}
\text{getType}(\text{reference } o, \Gamma) &= \Gamma(o).\text{type} \\
\text{getType}(\text{basetype bool}, \Gamma) &= \text{bool} \\
\text{getType}(\text{basetype void}, \Gamma) &= \text{void} \\
\text{getType}(\text{basetype } \bot, \Gamma) &= \bot \\
\text{getType}(\text{basetype } L, \Gamma) &= L
\end{aligned}$$

$$\begin{aligned}
\text{vtype}(o[C, \mathcal{U}]) &= \text{reference } o \\
\text{vtype}(\text{bool}) &= \text{basetype bool} \\
\text{vtype}(\text{void}) &= \text{basetype void} \\
\text{vtype}(\bot) &= \text{basetype } \bot \\
\text{vtype}(L) &= \text{basetype } L
\end{aligned}$$

Next in Figure 4 follows the rules for control structures. The rule for sequential composition is straightforward, and uses the resulting environment from e to type check e'. The rule (If) requires the same resulting environment for both branches. The (Case) rule checks that all branches of the choice-usage results in the same final environment, similar to (If) and updates the usage accordingly. Finally (Label) and (Continue) type checks loops. (Label) simply adds the current typing environment to Ω, while (Continue) can result in an arbitrary type and typing environment. This behaviour is safe, since we know that the expression is well formed, meaning that any continue statement is guarded by an if case or match statement, in which case only the choice of an environment that matches the other branch, can be chosen if the expression must be well typed.

Notice that in the (Case) rule we make use of the enumeration type L link o. As such types do not *agree* with any other types, they cannot be stored in fields or used as method arguments. So the only way for these types to be show up in a well-typed program, is as the return value of a method, used for matching in a case statement.

Example 1. Consider again the bank account example presented in Section 2. The typing environments is an approximation of the heap, and after typing lines 41-46, the environment is:

$$\begin{aligned}
o_{\text{main}} &\mapsto (\text{Main}[\text{end}], \{\texttt{account} \mapsto o_{\text{acc}}, \texttt{manager} \mapsto o_{\text{man}}, \texttt{db} \mapsto o_{\text{d}}, \}) \\
o_{\text{acc}} &\mapsto (\text{BankAccount}[\ \{\text{setMoney}; \{\text{applyInterest}; \text{getMoney}; \text{end}\}\}\}], \{\texttt{amount} \mapsto \texttt{float}\}) \\
o_{\text{man}} &\mapsto (\text{SalaryManager}[\{\text{addSalary}; \text{end}\}], \{\texttt{account} \mapsto o_{\text{a}}\}) \\
o_{\text{db}} &\mapsto (\text{DataStorage}[\{\text{store}; \text{end}\}], \{\texttt{account} \mapsto o_{\text{a}}\})
\end{aligned}$$

After type checking line 48, where the salary manager adds funds to the account, the following bindings are updated in the typing environment, while the remaining bindings are unchanged.

$$\begin{aligned}
o_{\text{acc}} &\mapsto (\text{BankAccount}[\{\text{getMoney}; \text{end}\}], \{\texttt{amount} \mapsto \texttt{float}\}) \\
o_{\text{man}} &\mapsto (\text{SalaryManager}[\{\text{addSalary}; \text{end}\}], \{\texttt{account} \mapsto o_{\text{a}}\})
\end{aligned}$$

This allows line 49 to be typechecked, since the o_{acc} has been updated to allow a call to the method `getMoney`, this is an example of how the global type checking approach allow us to track changes to aliased fields, even if they happen through seemingly unrelated objects.

5 SEMANTICS

In this section we define the run-time semantics of the language. It follows the standard model where object references are used to look up values in the heap. The heap itself is similar in some respects to the typing environment we have previously discussed. The heap maps object references o to their class and a field environment. In the semantics, we do not consider typestates, hence instead of mapping the object reference to a full type, we only map it to its class in order to look up method definitions and field declarations. The field bindings in the heap is a mapping from field names to values, which themselves can be object references or base values such as true, null, or unit. The initial field environment is defined similarly to $\overline{F}$.inittypes, but instead it maps the fields to values instead, in $\overline{F}$.initvals.

$$\begin{aligned}
(\overline{F}, \text{var } f : C).\text{initvals} &= \overline{F}.\text{initvals}, f \mapsto \text{null} \\
(\overline{F}, \text{var } f : \text{bool}).\text{initvals} &= \overline{F}.\text{initvals}, f \mapsto \text{false} \\
(\overline{F}, \text{var } f : \text{void}).\text{initvals} &= \overline{F}.\text{initvals}, f \mapsto \text{unit} \\
(\overline{F}, \text{var } f : L).\text{initvals} &= \overline{F}.\text{initvals}, f \mapsto l \\
&\qquad \text{where enum } L\{l, \overline{l}\} \in \overline{D} \\
\emptyset.\text{initvals} &= \emptyset
\end{aligned}$$

We now define configurations, which are of the form $\langle h, e \rangle$. When evaluating the expression e, both the expression and the heap can change. To model this, we let a computation step be of the form $\langle h, e \rangle \Rightarrow \langle h', e' \rangle$. The reduction rules are shown in Figure 5.

To simplify the reduction rules, we make use of an evaluation context to guide the evaluation of composite expressions.

$$\mathcal{E} ::= [\_] \mid o.f = \mathcal{E} \mid \mathcal{E}; e \mid o.m(\mathcal{E}) \mid o.f.m(\mathcal{E})$$
$$\mid \text{if } (\mathcal{E}) \{e\} \text{ else } \{e\} \mid \text{match}(\mathcal{E})\{\overline{l : e}\}$$

The (CTX) rules ensures that inner expressions are evaluated first (e.g. left-hand side of a sequential expression are evaluated before right-hand side). The remaining rules handle the interesting base cases of the semantics.

Example 2. Consider again the bank account example. When reaching line 48, the heap contains the following bindings:

$$o_{\text{main}} \mapsto (\text{Main}, \{\texttt{account} \mapsto o_{\text{acc}}, \texttt{manager} \mapsto o_{\text{man}}, \texttt{db} \mapsto o_{\text{d}}, \})$$
$$o_{\text{acc}} \mapsto (\text{BankAccount}, \{\texttt{amount} \mapsto 0\})$$
$$o_{\text{man}} \mapsto (\text{SalaryManager}, \{\texttt{account} \mapsto o_{\text{a}}\})$$
$$o_{\text{db}} \mapsto (\text{DataStorage}, \{\texttt{account} \mapsto o_{\text{a}}\})$$

After evaluating the expression on line 48, where the salary manager adds funds to the account, the following bindings are updated in the heap, while the remaining bindings are unchanged.

$$o_{\text{acc}} \mapsto (\text{BankAccount}, \{\texttt{amount} \mapsto 100\})$$

We see that compared to the type system, fewer bindings were updated, due to the typestates not being tracked in the semantics. However, the resulting environments from the type system and the semantics remains consistent, meaning that the types mentioned in the type system are consistent with the values in the heap. This property and more will be shown in the following section.

6 PROPERTIES

In this section we show important properties that hold for the defined language. The proofs can be found in a technical report [19]. The first result we show is the fact that we can remove bindings from Θ while the expression remains well-typed. The intuition of this is that Θ serves to denote the base case of checking recursive calls. So when we remove a binding from the environment, we simply have to expand the method body once more, leading to the entry being added again in Θ.

Lemma 1. If in a typing derivation starting from an empty recursion environment we have $(\Theta, o.m \mapsto \Gamma'');\emptyset;\Gamma \vdash e : T \dashv \Gamma'$ then we also have $\Theta;\emptyset;\Gamma \vdash e : T \dashv \Gamma'$.

Proof. Details in [19] Appendix B. □

Along with a similar proof for labelled expressions, where bindings can be removed from Ω, this shows that we often consider situations where Θ and Ω are empty. So for readability of the upcoming properties, we omit writing the environments when they are empty, so $\Gamma \vdash e : T \dashv \Gamma'$ is equivalent to $\emptyset;\emptyset;\Gamma \vdash e : T \dashv \Gamma'$.

We must establish a soundness result, and show that the usages defined for classes are respected at run-time, and no protocol deviation occurs. To establish such a relationship between the type system and the semantics, we first define *consistency* between a heap and a typing environment, which describes that the typing environment and heap agree on the classes of all objects, and agree on the field bindings of all objects.

Definition 4 (Heap consistency)**.** We say that a heap is consistent with a typing environment, written $\Gamma \vdash h$, if Γ and h contains the same objects and the field bindings of each object are also consistent.

$$\frac{\text{dom}(h) = \text{dom}(\Gamma) \quad \forall o \in \text{dom}(\Gamma).h(o).\text{fields} = \Gamma(o).\text{fields} \wedge h(o).\text{class} = \Gamma(o).\text{class}}{\Gamma \vdash h}$$

Furthermore, we lift the transition system for usages to typing environments, with the rules shown in Figure 6. Notice how the transitions match the updates to a typing environment performed by the typing rules shown in Figure 4. This allows us to establish that only a single update is performed to a typing environment when evaluating one step in the reduction semantics.

To complete subject reduction (Theorem 1), we consider a semantics where the transitions in (call-d) and (call-ind) are annotated with $o.m$ and $o'.m$ respectively, (match) is annotated with $o.l'$ and all other transitions are annotated with the empty string ε. We can use these labels to show a correspondence between the transitions on typing environments, and the transitions between run-time configurations.

The subject reduction theorem states that a single reduction preserves well-typedness with a single update to the typing environment.

Theorem 1 (Subject Reduction)**.** If $\Gamma \vdash h$, $\Gamma \vdash e : T \dashv \Gamma'$ and $\langle h, e\rangle \overset{\alpha}{\Rightarrow} \langle h', e'\rangle$ then $\exists \Gamma''.\Gamma'' \vdash e' : T \dashv \Gamma'$ such that $\Gamma \xrightarrow{\alpha} \Gamma''$ and $\Gamma'' \vdash h'$

Proof. Details in [19] Appendix C. □

Example 3. We return to the configuration just before executing line 44 in Listing 4. In Examples 1 and 2 we have stated what the heap and typing environment contain when reaching this statement. In this example we show the consistency between the heap and typing environment for the expression after a single transition.

The remaining expression of the program at this point is:

$$e = o_{\text{main}}.\texttt{manager}.\texttt{addSalary}(100.0); o_{\text{main}}.\texttt{db}.\texttt{store}(\texttt{unit})$$

In Example 1 we identified the typing environment as:

$$\Gamma = \left\{ \begin{array}{ll} o_{\text{main}} \mapsto & (\text{Main}[\text{end}], \\ & \{\texttt{account} \mapsto o_{\text{acc}}, \texttt{manager} \mapsto o_{\text{man}}, \\ & \texttt{db} \mapsto o_{\text{d}}, \}) \\ o_{\text{acc}} \mapsto & (\text{BankAccount}[\{\texttt{setMoney}; \\ & \{\texttt{applyInterest}; \\ & \{\texttt{getMoney}; \text{end}\}\}\}], \\ & \{\texttt{amount} \mapsto \texttt{float}\}) \\ o_{\text{man}} \mapsto & (\text{SalaryManager}[\{\texttt{addSalary}; \text{end}\}], \\ & \{\texttt{account} \mapsto o_{\text{a}}\}) \\ o_{\text{db}} \mapsto & (\text{DataStorage}[\{\texttt{store}; \text{end}\}], \\ & \{\texttt{account} \mapsto o_{\text{a}}\}) \end{array} \right\}$$

In Example 2 we identified the heap as:

$$\boxed{\langle h, e\rangle \Rrightarrow \langle h', e'\rangle}$$

$$\text{(ctx)}\ \frac{\langle h, e\rangle \Rrightarrow \langle h', e'\rangle}{\langle h, \mathcal{E}[e]\rangle \Rrightarrow \langle h', \mathcal{E}[e']\rangle} \qquad \text{(assign)}\ \frac{}{\langle h, o.f = v\rangle \Rrightarrow \langle h[o.f \mapsto v], \mathsf{unit}\rangle} \qquad \text{(seq)}\ \frac{}{\langle h, v; e\rangle \Rrightarrow \langle h, e\rangle}$$

$$\text{(if-true)}\ \frac{}{\langle h, \mathsf{if}\ (\mathsf{true})\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \Rrightarrow \langle h, e_1\rangle} \qquad \text{(if-false)}\ \frac{}{\langle h, \mathsf{if}\ (\mathsf{false})\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \Rrightarrow \langle h, e_2\rangle}$$

$$\text{(lab)}\ \frac{}{\langle h, k : e\rangle \Rrightarrow \langle h, e\{\mathsf{continue}\ k/k : e\}\rangle} \qquad \text{(match)}\ \frac{l_j : e_j \in \overline{l : e}}{\langle h, \mathsf{match}(o.l_j)\{\overline{l : e}\}\rangle \rightarrow \langle h, e_j\rangle}$$

$$\text{(call-d)}\ \frac{h(o).\mathsf{class.methods} \ni \mathsf{fun}\ m(x : t) : t'\{e\}}{\langle h, o.m(v)\rangle \Rrightarrow \langle h, e\{\mathsf{this}/o\}\{x/v\}\rangle} \qquad \text{(call-ind)}\ \frac{h(o).f = o' \qquad h(o').\mathsf{class.methods} \ni \mathsf{fun}\ m(x : t) : t'\{e\}}{\langle h, o.f.m(v)\rangle \Rrightarrow \langle h, e\{\mathsf{this}/o'\}\{x/v\}\rangle}$$

$$\text{(new)}\ \frac{o' \text{ fresh} \qquad h' = (h, o' \mapsto (C, C.\mathsf{fields.initvals}))[o.f \mapsto o']}{\langle h, o.f = \mathsf{new}\ C\rangle \Rrightarrow \langle h', \mathsf{unit}\rangle} \qquad \text{(fld)}\ \frac{h(o).\mathsf{fields}(f) = v}{\langle h, o.f\rangle \Rrightarrow \langle h, v\rangle}$$

Figure 5: Run-time semantics

$$\text{(empty)}\ \frac{}{\Gamma \xrightarrow{\varepsilon} \Gamma} \qquad \text{(trans)}\ \frac{\Gamma(o).\mathsf{usage} \xrightarrow{\alpha} \mathcal{U}}{\Gamma \xrightarrow{o.\alpha} \Gamma[o.\mathsf{usage} \mapsto \mathcal{U}]} \qquad \text{(update)}\ \frac{\Gamma(o).f = z}{\Gamma \xrightarrow{\varepsilon} \Gamma[o.f \mapsto z']}$$

$$\text{(new)}\ \frac{o \in \mathrm{dom}(\Gamma) \qquad o' \text{ fresh} \qquad \mathsf{class}\ C\{\mathcal{U}, \overline{F}, \overline{M}\} \in \overline{D}}{\Gamma \xrightarrow{\varepsilon} (\Gamma, o' \mapsto (o'[C, \mathcal{U}], \overline{F}.\mathsf{inittypes}))[o.f \mapsto o']}$$

Figure 6: Transition system for typing environments

$$h = \left\{ \begin{array}{l} o_{\mathsf{main}} \mapsto (\mathsf{Main}, \{\mathsf{account} \mapsto o_{\mathsf{acc}}, \\ \qquad\qquad \mathsf{manager} \mapsto o_{\mathsf{man}}, \mathsf{db} \mapsto o_{\mathsf{d}}, \}) \\ o_{\mathsf{acc}} \mapsto (\mathsf{BankAccount}, \{\mathsf{amount} \mapsto 0\}) \\ o_{\mathsf{man}} \mapsto (\mathsf{SalaryManager}, \{\mathsf{account} \mapsto o_{\mathsf{a}}\}) \\ o_{\mathsf{db}} \mapsto (\mathsf{DataStorage}, \{\mathsf{account} \mapsto o_{\mathsf{a}}\}) \end{array} \right\}$$

We have $\Gamma \vdash e : \mathsf{void} \dashv \Gamma''$ where Γ'' is the terminated environment containing the objects of Γ. Using the (ctx) and (call-ind) rule we can conclude the following transition (we let e' denote the updated expression):

$$\begin{array}{l} \langle h, o_{\mathsf{main}}.\mathsf{manager.addSalary}(100.0); o_{\mathsf{main}}.\mathsf{db.store}(\mathsf{unit})\rangle \\ \xrightarrow{o_{\mathsf{man}}.\mathsf{addSalary}} \langle h, (o_{\mathsf{man}}.\mathsf{account.setMoney}(100.0); \\ \qquad\qquad o_{\mathsf{man}}.\mathsf{account.applyInterest}(1.05)); \\ \qquad\qquad o_{\mathsf{main}}.\mathsf{db.store}(\mathsf{unit})\rangle \end{array}$$

Now let Γ' be the updated environment where a single transition has been performed on the salary manager object:

$$\Gamma' = \left\{ \begin{array}{l} o_{\mathsf{main}} \mapsto (\mathsf{Main}[\mathsf{end}], \\ \qquad\qquad \{\mathsf{account} \mapsto o_{\mathsf{acc}}, \mathsf{manager} \mapsto o_{\mathsf{man}}, \\ \qquad\qquad \mathsf{db} \mapsto o_{\mathsf{d}}, \}) \\ o_{\mathsf{acc}} \mapsto (\mathsf{BankAccount}[\ \{\mathsf{setMoney}; \\ \qquad\qquad \{\mathsf{applyInterest}; \\ \qquad\qquad \{\mathsf{getMoney}; \mathsf{end}\}\}\}], \\ \qquad\qquad \{\mathsf{amount} \mapsto \mathtt{float}\}) \\ o_{\mathsf{man}} \mapsto (\mathsf{SalaryManager}[\mathsf{end}], \\ \qquad\qquad \{\mathsf{account} \mapsto o_{\mathsf{a}}\}) \\ o_{\mathsf{db}} \mapsto (\mathsf{DataStorage}[\{\mathsf{store}; \mathsf{end}\}], \\ \qquad\qquad \{\mathsf{account} \mapsto o_{\mathsf{a}}\}) \end{array} \right\}$$

We can conclude $\Gamma \xrightarrow{o_{\mathsf{man}}.\mathsf{addSalary}} \Gamma'$ with the (trans) rule. It is clear that we have $\Gamma' \vdash h$ since we have only updated a usage which is not considered in the consistency relation. Finally we can also conclude $\Gamma' \vdash e' : \mathsf{void} \dashv \Gamma''$ since the remaining usages in Γ' corresponds to the remaining method calls in e' (and also directly from the typing rule of $\Gamma \vdash e : \mathsf{void} \dashv \Gamma''$).

As previously mentioned, we use the labels of the run-time semantics to establish a correspondence between updates to the typing environment and the run-time configurations. In Corollary 1, which follows from Theorem 1, we make this correspondence explicit by showing that when a method call or label selection occurs at run-time, this always follows the protocol of the object.

Corollary 1 (Protocol conformance)**.** If $\Gamma \vdash e : T \dashv \Gamma''$, $\Gamma \vdash h$, $\langle h, e\rangle \overset{o.\alpha}{\Longrightarrow} \langle h', e'\rangle$ then $\exists \Gamma'.\ \Gamma' \vdash e' : T \dashv \Gamma''$ and $\Gamma(o).\mathsf{usage} \xrightarrow{\alpha} \Gamma'(o).\mathsf{usage}$

Lemma 2 (Protocol completion)**.** Let $\overline{D}$ be a well-typed program and let c be the initial configuration of $\overline{D}$. If $c \Rightarrow^* \langle h, v\rangle$ then all objects in h has finished their protocol.

Proof. Since $\vdash \overline{D}$ ok we know from (Main) that $\mathsf{Main}\{\mathcal{U}, \overline{F}, \overline{M}\} \in \overline{D}, \overline{M} = \{\mathsf{fun\ main()}\ \{e\}\}$, and $\{o_{\mathsf{main}} \mapsto (\mathsf{Main[end]}, \overline{F}.\mathsf{inittypes})\} \vdash e : T \dashv \Gamma'$ where term(Γ'). Since term(Γ') we know that all objects have terminated protocols, and from Corollary 1 we know that all objects has followed their protocols. □

We can now conclude with progress property, which states that well-typed programs do not get stuck.

Theorem 2 (Progress)**.** If $\Theta; \Omega; \Gamma \vdash e : T \dashv \Gamma', \Gamma \vdash h$, then either e is a value or $\exists h', e'.\ \langle h, e\rangle \Rightarrow \langle h', e'\rangle$

Proof. Details in [19] Appendix D. □

7 IMPLEMENTATION

In this section we present Papaya, an implementation of our framework and type system for a subset of the Scala programming language [28]. The source code can be found on our Github repository [30].

The Papaya tool is implemented as a plugin for the Scala compiler, meaning that programs with protocol violations will produce compilation errors and the program will not be compiled.

Scala is an object-oriented language which compiles to JVM bytecode and consequently is compatible with Java code and applications, while introducing new language features such as lazy evaluation, immutability, type inference and pattern matching. These features make Scala an expressive and powerful high-level language that supports object-oriented programming, functional programming, and a mix of both.

The main features supported by Papaya are:

- **Control flow structures** Papaya supports users using loops, if-else statements, match statements, and functions.
- **Recursion** Papaya handles recursive function calls as described for the formalisation.
- **Fields** Objects with typestates can be stored in class fields and dealt with appropriately.
- **Unrestricted aliasing** Papaya offers the user unrestricted aliasing of variables.

To compare with the earlier implementations of Mungo for Java, we can see that Papaya introduces new features:

- **Uncertain states** For increased flexibility, Papaya allows multiple branches to result in different typestates, as long as the type state of all objects eventually become consistent. This is a deviation between the formalism of this paper and the implementation.
- **Unrestricted aliasing** Mungo enforces linearity in its program, disallowing the user to alias objects with a protocol. With unrestricted aliasing in Papaya, the user is free to alias as much as they want to.

The implemented algorithm follows the structure defined for the formalism, by analysing the program graph, starting from entry-point into the program and expanding the program graph upon reaching method calls. At each method call encountered during verification, the transition system of the protocol of the callee is consulted to ensure that the method call is currently allowed for the object.

This is different from the previous implementations of the Mungo tool. In the original implementation [6, 21, 22, 37] the tool infers the typestate of objects in the program and then checks that this respects the typestate defined for the classes. Since that version of the Mungo tool requires a linear treatment of references, the tool works similar to a classic type system for object-oriented language where each class is checked in isolation. In a recent implementation of Mungo [25], the Java Checker Framework is used to analyse the control flow graph of a Java program, and perform typestate analysis. The tool allows for a modular approach to type checking, however the use of aliases in the new implementation is still restricted to fractional permissions where write-access is only given to linear references.

Example 4. We return to our bank account example. This time we write a working implementation in Scala and use Papaya to verify the correctness of the implementation.

In Listing 5 we show the implementation of the bank account introduced earlier. The typestate is specified using the `@Typestate` annotation where the argument refers a name of a singleton object defining the behaviour of a class.

```
1  @Typestate("BankAccountProtocol")
2  class BankAccount() {
3      var balance:Float = 0
4      def fill(amount:Float):Unit =
5          { balance = amount }
6      def get():Float = balance
7      def applyInterest(ir:Float):Unit = {
8          balance = balance * ir
9      }
10 }
```

Listing 5: Implementation of the BankAccount with attached protocol

The protocol is written in a Scala-like domain specific language. The protocol of the bank account is shown in Listing 6. Notice that the implementation uses state equations (i.e. init = setMoney(Float) → intermediate) instead of the recursive definitions used in the formalism to describe state changes. This change is introduced to allow programmers to specify their protocols more easily.

```
1  object BankAccountProtocol extends
       ProtocolLang with App {
2    in("init")
3      when("setMoney(Float)")
4          goto "intermediate"
5    in("intermediate")
6      when("applyInterest(Float)")
7          goto "filled"
8    in("filled")
9      when("getMoney()")
```

```
10          goto "end"
11    in("end")
12      end()
13  }
```

Listing 6: Protocol for the BankAccount class

The protocol specifies that the BankAccount starts in the `"init"` state and can perform one transition with a call to `setMoney(Float)` to go to the `"intermediate"` state. We can see that it then has one possible transition to the `"filled"` state, whence it has one last possible transition to the `"end"` state. Comparing this to the previously defined usage {setMoney; {applyInterest; {getMoney; end}}} we see that the two descriptions are equivalent.

```
11  class DataStorage() {
12      var money:BankAccount = null;
13      def setMoney(m:BankAccount):Unit =
14          { money = m}
15      def store():Unit = {
16          var amount = money.get()
17          println(amount)
18          // write to the database
19      }
20  }
21  class SalaryManager() {
22      var money:BankAccount = null;
23      def setMoney(m:BankAccount):Unit =
24          { money = m}
25      def addSalary(amount:Float):Unit = {
26          money.fill(amount)
27          money.applyInterest(1.02f)
28      }
29  }
```

Listing 7: Implementations of two classes that will use a shared bank account

In Listing 7 we show the implementation of the two remaining classes previously introduced, and in Listing 8 we show how aliasing is achieved by providing the `account` reference to both the salary manager and the data store.

```
1  object Demonstration extends App {
2      val account = new BankAccount
3      val manager = new SalaryManager
4      val storage = new DataStorage
5      manager.setMoney(account)
6      storage.setMoney(account)
7      manager.addSalary(5000)
8      storage.store()
9  }
```

Listing 8: Program segment that uses aliasing

In the implementation we handle the layer of indirection between references (with potential aliasing) and objects similarly to the treatment in the type system in Section 4 but with more information tracked in order to aid debugging and error handling. This means that the three references introduced in Listing 8 are tracked independently but all point to the same underlying instance, as shown in Figure 7.

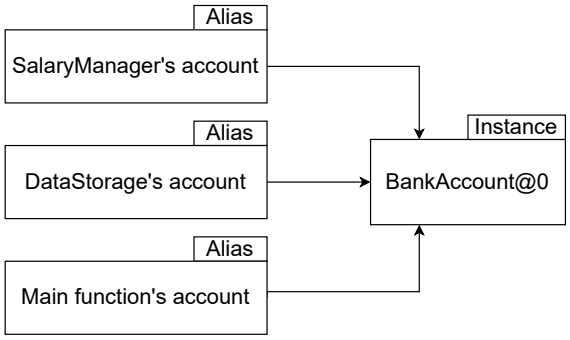

Figure 7: Example of the structure of Instances and Aliases in the BankAccount example. Here we have three Aliases pointing to a single BankAccount Instance. The Instance has an "@0" ID to differentiate it from other potential BankAccount Instances. Each Alias is identified by its name and scope.

8 RELATED WORK

Session types. Session types [16–18] were introduced to ensure type-safe structured communication between multiple parties. The process sending data and the process receiving data must agree on the type of data being transmitted. The concept of session types has been also explored for object oriented languages [10, 35]. A particular application of session types for an object oriented language is that in Bica [14] where session types are used to type communication on channels, but also to perform type-safe interaction with objects themselves. In terms of message passing in object-oriented languages, we can view a session type as a description of the messages we can send to a particular object, corresponding to an ordering of method calls. In the simplest setting we can imagine object initialisation as instantiating a communication channel between the new object and the caller, and subsequent method calls as sending messages on this channel. Scalas et al. [31, 32] integrate binary and multiparty session types in Scala and implement it as a library.

Typestates. While originally introduced to track value initialisation [33], typestates have been explored extensively for object-oriented programming. The approach described for Bica is one approach for typestates in object-oriented languages that inspired the line of research on Mungo [3, 13, 22, 25–27]. In this line of work,

typestates are based on session types and describe the permitted sequence of method calls, in a syntax similar to session types.

Plaid [1, 34] introduces the concept of *Typestate-oriented programming* wherein typestates form the basis for objects, rather than class descriptions. As operations are performed on object, they transition between states, and the set of available operations evolve, ensuring that methods can only be called on objects that are in a state that implements the method.

The Fugue protocol checker [8, 9] extends class definitions for the Common Language Runtime (CLR) [23] with state machines. They use pre and postconditions to describe the transitions between states and preconditions are used as guards, to ensure that methods are only called when the object is in a state that allows the method calls.

Lastly, the work on typestates for concurrent object-oriented languages [5, 29] uses typestates to reason about protocol conformance, but also properties such as deadlock freedom.

Aliasing and Typestates. We have seen multiple approaches to combining typestates with object-oriented programming, but each approach handles the presence of aliasing differently.

Vault [7] introduces *tracked types* where a unique key is created for each object, and operations can only be performed on the object by the current holder of the key.

In an extension to the Vault language [11] the concepts of *adoption* and *focus* are used for introducing aliases. The adoption construct allows a linear value (the adoptee) to be converted into a nonlinear reference for the duration of the adopters lifetime. As linear resources of the adoptee cannot be accessed through the nonlinear type, they introduce the focus operation to temporarily convert the nonlinear type into a linear type, by ensuring that in the linear scope, no other aliases can witness the operations, and that the object is left in a consistent state after the operation, so that the operations remains invisible to other aliases.

Later, in the work on Fugue [8] they allow objects to be marked *NotAliased* and *MayBeAliased*. In the case of an object being marked *NotAliased* the object is treated linearly, whereas objects marked *NotAliased* are tracked to see if they can *escape* from their context (by method calls or assignment, etc.) and emits a warning in case of unsafe aliasing.

Multiple approaches to aliasing have been introduced for the Plaid language. Bierhoff and Aldrich [2] present a fine-grain approach to aliasing. The authors note that an approach such as the one used in Fugue must be able to reason about all aliases to allow state change to an object, hence limiting nonlinear objects to simple operations. Instead they propose a collection of five permissions such as unique (single reference with read/write permissions), share (one reference has read/write permissions, other references has read permissions) or the inverse pure (read access while other reference has read/write permissions). For the different permissions, they introduce the concept of *permission splitting* and *permission joining*, where one alias with a permission can be split into two aliases that are equally or more restricted than the original. Similarly, for joining, two permissions can be merged back into a potentially less restrictive permission. To handle an arbitrary number of aliases, and ensure that all aliases can be collected to regain write access, they introduce fractions denoting how many times a permission has been split, and conversely when all fractions has been recovered.

A typesystem for a language inspired by Plaid [24] uses concepts from behavioural separation [4] to reason about type-states. In this language, classes are composed of *views*, and each view contains a subset of the fields of the class. Through *view equations*, views can be composed or decomposed into a number of other views, similar to permission splitting and joining as previously described. Through view decomposition, each alias is associated with a single view, and hence also follows the view equations. Similar to the previous work on aliasing in Plaid they use fractions to keep track of splits when allowing an unbounded number of aliases, so they can ensure that all aliases are recovered before any updates to the full object.

Mungo generally treats objects as linear values, where only a single reference to an object can exist. While enforcing linearity allows for a common treatment of all object references, it is a deviation from real-world programs where aliasing is used in programming patterns for sharing data etc. Accordingly, work has been undergoing to lessen this constraint. A recent implementation of the Mungo tool [26] supports access permissions similar to those described for Plaid.

In another treatment of aliasing for Mungo [15], the language of usages is extended with a parallel construct $(\mathcal{U}_1 \mid \mathcal{U}_2).\mathcal{U}_3$ where an object can be aliased into two references, with usages $\mathcal{U}_1$ and $\mathcal{U}_2$ respectively. After completion of the local protocol, only a single reference (with usage $\mathcal{U}_3$) exists. This approach is analogous to the view-equations used in [24].

Common between the approaches to aliasing described in this section is that they adopt a local treatment of aliasing, allowing them to preserve compositionality of the type system, whereas the treatment in this paper is global. The local treatment allows for greater flexibility in a larger system, where components can be replaced without having to re-verify the entire system, whereas the global approach allows for the maximum flexibility for the programmer's work with aliasing.

A typestate verification framework for Java with support for aliasing has been presented in [12]. The tool makes sound approximations to scale to larger programs, at the cost of precision (increased false positives).

9 CONCLUSION AND FUTURE WORK

In this paper we have explored a global approach to reasoning about unrestricted aliasing in the presence of typestates. We have shown the standard soundness properties about the type system, namely subject reduction and progress. Furthermore, we have shown the protocol conformance property–which ensures that protocols defined for classes are respected by instantiated objects, and that no protocol deviation occurs–and the protocol completion property–which ensures that protocols are completed for all objects, meaning that after termination of a program all objects have successfully completed their protocol.

The language presented in this paper is a small object-oriented language that does not correspond directly to any real-life programming language. However it does have similarities to the low level

JVM bytecode language. As future work, we plan to explore this similarity in an attempt at integrating typestates in JVM bytecode.

As we use a global approach of type checking the entire program graph, as opposed to checking each class in isolation, the run-time may suffer for larger programs. To combat this, it would be interesting to split classes into a linear section, and an unrestricted section. Then values that are treated by the class as linear objects (where only a single reference exists at all times) can be checked in isolation, before the global analysis checks the unrestricted sections of all classes. We leave it to future work to check whether such a split of a class can be determined without programmer annotations, and to explore how to integrate the previous approaches to type checking linear objects can be integrated as a step before the global analysis.

ACKNOWLEDGMENTS

Research supported by the EPSRC programme grant “From Data Types to Session Types: A Basis for Concurrency and Distribution" EP/K034413/1 (ABCD), and EU HORIZON 2020 MSCA RISE project 778233 “Behavioural Application Program Interfaces” (BehAPI). We thank Simon Fowler for his valuable comments on the paper, Alceste Scalas for his helpful tips on Scala and Elena Giachino for her (implicit) suggestion on the name Papaya.

REFERENCES

[1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. (2009), 1015–1022. https://doi.org/10.1145/1639950.1640073

[2] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. *ACM SIGPLAN Notices* 42, 10 (2007), 301–319. https://doi.org/10.1145/1297105.1297050

[3] Mario Bravetti, Adrian Francalanza, Iaroslav Golovanov, Hans Hüttel, Mathias S Jakobsen, Mikkel K Kettunen, and António Ravara. 2020. Behavioural Types for Memory and Method Safety in a Core Object-Oriented Language. In *Programming Languages and Systems*, Bruno C d. S Oliveira (Ed.). Springer International Publishing, Cham, 105–124.

[4] Luís Caires and João C. Seco. 2013. The type discipline of behavioral separation. *ACM SIGPLAN Notices* 48, 1 (2013), 275–286. https://doi.org/10.1145/2480359.2429103

[5] Silvia Crafa and Luca Padovani. 2017. The chemical approach to typestate-oriented programming. *ACM Transactions on Programming Languages and Systems* 39, 3 (2017), 917–934. https://doi.org/10.1145/3064849

[6] Ornela Dardha, Simon J. Gay, Dimitrios Kouzapas, Roly Perera, A. Laura Voinea, and Florian Weber. 2017. Mungo and StMungo: tools for typechecking protocols in Java. In *Behavioural Types: from Theory to Tools*, Simon Gay and Antonio Ravara (Eds.). River Publishers, 309–328. http://eprints.gla.ac.uk/146891/

[7] Robert DeLine and Manuel Fahndrich. 2001. Enforcing high-level protocols in low-level software. *In Proc. of PLDI* pages (2001), 59–69.

[8] Robert DeLine and Manuel Fähndrich. 2004. *The Fugue protocol checker: Is your software Baroque?* Technical Report January. Microsoft Research. http://research.microsoft.com/apps/pubs/default.aspx?id=67458%5Cnhttp://research.microsoft.com/en-us/projects/fugue/

[9] Robert Deline and Manuel Fähndrich. 2004. Typestates for Objects. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3086 (2004), 465–490. https://doi.org/10.1007/978-3-540-24851-4_21

[10] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session types for object-oriented languages. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4067 LNCS (2006), 328–352. https://doi.org/10.1007/11785477_20

[11] Manuel Fähndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 13–24. https://doi.org/10.1145/512529.512532

[12] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2 (2008), 9:1–9:34. https://doi.org/10.1145/1348250.1348255

[13] Simon J. Gay, Nils Gesbert, António Ravara, and Vasco T. Vasconcelos. 2015. Modular session types for objects. *Logical Methods in Computer Science* 11, 4 (2015), 1–76. https://doi.org/10.2168/LMCS-11(4:12)2015

[14] Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. 2010. Modular session types for distributed object-oriented programming. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 299–312. https://doi.org/10.1145/1706299.1706335

[15] Iaroslav Golovanov, Hans Hüttel, Mathias Steen Jakobsen, and Mikkel Klinke Kettunen. 2021. Behavioural Separation with Parallel Usages. In *Proceedings of the 23rd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs* (Virtual, Denmark) *(FTfJP 2021)*. Association for Computing Machinery, New York, NY, USA.

[16] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 715)*, Eike Best (Ed.). Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[17] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1381 (1998), 122–138. https://doi.org/10.1007/bfb0053567

[18] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 273–284. https://doi.org/10.1145/1328438.1328472

[19] Mathias Jakobsen, Alice Ravier, and Ornela Dardha. 2021. Papaya: Global Typestate Analysis of Aliased Objects Extended Version. *CoRR* abs/2107.13101 (2021). arXiv:2107.13101 https://arxiv.org/abs/2107.13101

[20] Mathias Steen Jakobsen, Mikkel Klinke Kettunen, and Iaroslav Golovanov. 2020. Behavioural Separation with Parallel Usages for a Core Object-Oriented Language.

[21] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2016. Typechecking protocols with Mungo and StMungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, James Cheney and Germán Vidal (Eds.). ACM, 146–159. https://doi.org/10.1145/2967973.2968595

[22] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2018. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Science of Computer Programming* 155 (2018), 52–75. https://doi.org/10.1016/j.scico.2017.10.006

[23] Microsoft. 2020. Common Language Runtime (CLR) overview - .NET. https://docs.microsoft.com/en-us/dotnet/standard/clr

[24] Filipe Militão, Jonathan Aldrich, and Luís Caires. 2010. Aliasing control with view-based typestate. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs, FTFJP 2010, Maribor, Slovenia, June 22, 2010*. ACM, 7:1–7:7. https://doi.org/10.1145/1924520.1924527

[25] João Mota. 2021. Coping with the reality: adding crucial features to a typestate-oriented language.

[26] João Mota, Marco Giunti, and António Ravara. 2021. Java Typestate Checker. In *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12717)*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer, 121–133. https://doi.org/10.1007/978-3-030-78142-2_8

[27] Mungo project. 2021. Mungo. http://www.dcs.gla.ac.uk/research/mungo/

[28] Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. 2021. Scala Language Specification. https://scala-lang.org/files/archive/spec/2.13/

[29] Luca Padovani. 2018. Deadlock-Free Typestate-Oriented Programming. *Art Sci. Eng. Program.* 2, 3 (2018), 15. https://doi.org/10.22152/programming-journal.org/2018/2/15

[30] Alice Ravier. 2021. Scala-Mungo. https://github.com/Aliceravier/Scala-Mungo

[31] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A linear decomposition of multiparty sessions for safe distributed programming. *Leibniz International Proceedings in Informatics, LIPIcs* 74, March (2017), 241–2431. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

[32] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *Proceedings of the 30th European Conference on Object-Oriented Programming, ECOOP (LIPIcs, Vol. 56)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:28. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

[33] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. , 157–171 pages. https://doi.org/10.1109/TSE.1986.6312929

[34] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Eric Tanter. 2011. First-class state change in plaid. *ACM SIGPLAN Notices* 46, 10 (2011), 713–732. https://doi.org/10.1145/2076021.2048122

[35] Vasco T. Vasconcelos. 2011. Sessions, from Types to Programming Languages. *Bull. EATCS* 103 (2011), 53–73. http://eatcs.org/beatcs/index.php/beatcs/article/view/136

[36] Vasco T. Vasconcelos, Simon J. Gay, António Ravara, Nils Gesbert, and Alexandre Z. Caldiera. 2009. Dynamic interfaces. In *2009 International Workshop on Foundations of Object-Oriented Languages (FOOL'09)*.

[37] A. Laura Voinea, Ornela Dardha, and Simon J. Gay. 2020. Typechecking Java Protocols with [St]Mungo. In *Proceedings of the International Conference on Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 , FORTE (Lecture Notes in Computer Science, Vol. 12136)*. Springer, 208–224. https://doi.org/10.1007/978-3-030-50086-3_12