Typestate Inference for Mungo: Algorithm and Implementation

Iaroslav Golovanov
Department of Computer Science, Aalborg University, Aalborg, Denmark
igolov14@student.aau.dk

Mathias Steen Jakobsen
Department of Computer Science, Aalborg University, Aalborg, Denmark
msja15@student.aau.dk

Mikkel Klinke Kettunen
Department of Computer Science, Aalborg University, Aalborg, Denmark
mkettu16@student.aau.dk

Abstract

We present a usage inference algorithm for the Mungo language. Mungo is an object-oriented programming language with usage annotations describing the permitted sequence of method calls on objects. A typestate in Mungo is a class name and a usage, and describes the type of an object. The type system for Mungo ensures that well typed programs follows the specified usages of all classes, and that null-dereferencing cannot occur at runtime. We show that the inference algorithm correctly infers the principal usage, the largest usage that does not result in runtime errors. Furthermore we show that a class is well-typed with an inferred usage. We present an implementation of both the type system and the inference algorithm, and provide an analysis on time and space complexity of both the algorithm and the inferred usage.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation

Keywords and phrases Behavioural Types, Mungo, Type System, Usages, Typestate

Supplement Material `https://github.com/MungoTypesystem/Mungo-Typechecker`

1 Introduction

Typestates are a programming language concept introduced by Strom & Yemini in [21] that allow us to statically verify that the order of operations on a variable is correct such that we avoid null-dereferencing errors and other semantically undefined expressions. Typestates were originally applied to variables as pre and postconditions. Since then typestates have been applied in a variety of situtations. One such example is a typestate oriented programming paradigm [1, 2] where objects are modelled as states in a labelled transition system, and the transitions are defined by pre and post-conditions on methods.

In this paper, we focus on the object-oriented calculus Mungo presented by Kouzapas et al. in [15] where classes are associated with protocols called *usages*, inspired by the session types presented by Gay et al. in [9]. Usages allow programmers to specify the allowed method call sequences for each class, as opposed to distributing the same logic across each method using pre and postconditions. In usage definitions, we allow three behaviours. The first is branching $\{m_i; w_i\}_{i\in I}^{\vec{E}}$ where m_i describe the available set of methods names along with their remaining protocols w_i. The second is choice $\langle l_i : u_i\rangle_{l_i\in L}^{\vec{E}}$ where l_i are labels and usages u_i are chosen based on the return label of a method. Finally, recursive usages $X^{\vec{E}\uplus\{X=u\}}$ allow a usage variable to be used in its definition and is then unfolded to allow recursive behaviour. The allowed behaviour is dependent on the state of an object. we call this state a typestate and it consists of a class name and a usage.

The calculus was originally presented as part of the paper describing the tools Mungo and

St. Mungo [15]. Their typechecker works by identifying the usage followed in the program and then check that it is a sub-usage of the one declared for the class. Later, a different approach to typechecking was presented by Golovanov et al. in [10] as a continuation of the work by Bravetti et al. in [5]. In their work typechecking is done by following and typechecking usage transitions. This way only the reachable methods are type checked. The research on Mungo is part of the work applying typestates to mainstream programming languages. Similarly, work in applying typestates to C# has been proposed by DeLine & Fähndrich in [8] and to C in [7]. Bierhoff et al. proposes a typestate checker for Java in [4]. A central difference between Mungo and these approaches is that they use pre and postconditions, similar to the approach by Strom & Yemini [21], whereas Mungo uses usages.

In Mungo, programmers manually specify usages for each class; however, in any given OOP program we can have a class where we only care about null-dereferencing errors and not about following a specific protocol. For example, a simple model class that only contains accessor methods would require a usage to include the accessor methods of each field. In Example 1 we illustrate that usages can become large, when no particular order of method calls is required. Usage inference allow programmers to reserve explicit usage specification to classes where they are concerned about protocol errors, thus making typestate definitions less intrusive in a practical setting. For the remaining classes, usage inference can ensure that no errors captured by the type system occur, while still avoiding have to write the large usages.

▶ **Example 1.** Conside the `Product` class below. The class has two fields with setters, and a method for adding the product information to a catalog. We assume that the `Catalog` class contains a method `add` that takes as arguments two initialised objects of classes `String` and `PInfo`.

The information can be added to the product class in any order, but all information should be available when calling `addToCatalog`, and afterwards the protocol is finished.

```
1  class Product {
2      String sku;
3      PInfo info;
4
5      void setSku(String x) { sku = x;  }
6      void setInfo(PInfo x) { info = x; }
7
8      Catalog addToCatalog(Catalog c) {
9          c.add(sku, info);
10         return c;
11     }
12 }
```

The protocol for the `Product` class can be seen in Figure 1, and allows calling `setSku` and `setInfo` in any order. This is still manageable with two fields, but if the class had many fields, the usage would quickly become large. Generally this would happen when a class has many linear fields that can be affected in separate methods. In this particular example, adding one more field would result in 9 distinct states, while adding two more fields would result in 17 distinct states. Generally for k linear fields, we have $O(2^{kl})$ states where l is the number of states in the field usages.

Another problem in the current Mungo toolchain is that we can only check our own modules and assume external modules are safe. With usage inference we can verify the safety of external open source modules and that our use of them is safe with regard to

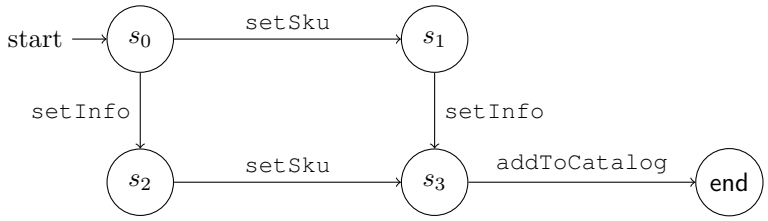


Figure 1 Protocol for the `Product` class

the Mungo typesystem. This could of course also be used more generally to check external modules, and verify the absence of run-time errors captured by the type system. This would give guarantees that a program is safe, without the original author introducing usages to their program.

Most work in the area of typestate and session type inference is based on extracting constraints from the program text and then solving the constraints to create an inferred type [22, 18, 14, 12]. For example, we can infer session types by creating pre and postconditions for each channel and then solve the constraints defined by these conditions. For usages, we could employ a similar approach by creating pre and postconditions for each method of a class, however, we would run into issues concerning protocol completion. If we overwrite a field in a method then we cannot be sure it is a terminated field since we only consider a local aspect of the program. Since we cannot guarantee protocol completion because its a liveness property, we cannot ensure that the inferred usage is well-typed in the type system.

We propose the following approach to usage inference that ensures protocol completion. We start by creating a dependency graph between classes such that a class is dependent on the classes of its fields. We require that the graph is acyclic in which case an ordering of the acyclic graph gives us the order of inference. Once we have this order, we consider possible field environments and how method calls could effect these environments. A field environment contains the current typestate of all fields of the class. Calling a method of the class can change the field environment. If the updates to the field environment, described by the method body, are allowed with regards to field environment, then a transition is added to the updated environment. We check every method and every field state of a class and filter non-terminating states from the labelled transition system before creating the usage. Creating a usage from the labelled transition system is simple since each state can be represented as a recursion variable and transitions are paths in a branch usage for the associated recursion variable.

We claim our contribution is a novel approach to usage inference and we prove that inferring usages with our proposed approach guarantees that usages are principal, meaning they can simulate any usage that can well-type a class. Furthermore, we prove that the inferred usage is itself well-typed hence we avoid null-dereferencing errors and guarantee protocol completion. Finally we present a prototype tool `mungoi` that implements the usage inference as well as the type system for Mungo.

The remainder of the article is structured as follows: Section 2 describes the Mungo calculus that we consider. Section 3 formally defines the inference problem and the properties that must hold for inferred usages. Section 4 describes our approach to usage inference. Section 5 shows an in-depth example of usage inference. In Section 6 we show that the required properties holds for inferred usages. The implementation of the tool `mungoi` is discussed in Section 7. In Section 8 we propose an extension to the type system, to allow inferring method parameter usages. In Section 9 we discuss related work to inference of

behavioural types. Finally, in Section 10 we discuss our findings, and presents ideas for future work.

2 The Language

A Mungo program is a set of declarations D describing a collection a classes and enumerated types. A class C consist of methods $\overrightarrow{M}$ and private fields $\overrightarrow{F}$, and is annotated with a usage $\mathcal{U}$. The syntax with corresponding syntactic categories for the program definitions are shown below. x ranges over method parameter names.

$$
\begin{array}{ll}
\textbf{(Declarations)} & D ::= \mathsf{enum}\ L\ \{\overrightarrow{l}\} \mid \mathsf{class}\ C\ \{\mathcal{U}, \overrightarrow{M}, \overrightarrow{F}\} \\
\textbf{(Fields)} & F ::= z\ f \\
\textbf{(Methods)} & M ::= t\ m(t\ x)\{e\} \\
\textbf{(Values)} & v ::= \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid l \mid \mathsf{null} \\
\textbf{(References)} & r ::= x \mid f \\
\textbf{(Expressions)} & e ::= v \mid r \mid \mathsf{new}\ C \mid f\ =\ e \mid r.m(e) \mid e; e \\
 & \quad \mid \mathsf{if}\ (e)\ \{e\}\ \mathsf{else}\ \{e\} \mid \mathsf{switch}_{r.m}\ (r.m(e))\ \{l_i : e_i\}_{i_i \in L} \\
 & \quad \mid k : e \mid \mathsf{continue}\ k
\end{array}
$$

For a class defined as $\mathsf{class}\ C\ \{\mathcal{U}, \overrightarrow{M}, \overrightarrow{F}\}$ we define the following notation.

$$
C.\mathsf{methods} \stackrel{\text{def}}{=} \overrightarrow{M} \qquad C.\mathsf{fields} \stackrel{\text{def}}{=} \overrightarrow{F} \qquad C.\mathsf{usage} \stackrel{\text{def}}{=} \mathcal{U}
$$

The syntax of types is shown in the following abstract syntax. We distinguish between base types, class types and field types. Base types describe the primitives of our language, and class types are typestates as previously introduced, and is composed of a class name C and a usage $\mathcal{U}$ into a typestate $C[\mathcal{U}]$. The distinction between a field type z and a type t is that for classes we do not consider the usage, hence a field can contain an object of the class regardless of its usage.

$$
\begin{array}{ll}
\textbf{(Base types)} & b ::= \mathsf{void} \mid \mathsf{bool} \mid L \\
\textbf{(Field types)} & z ::= b \mid C \\
\textbf{(Types)} & t ::= b \mid C[\mathcal{U}] \\
\textbf{(Usages)} & \mathcal{U} ::= u^{\overrightarrow{E}} \\
 & u ::= \{m_i; w_i\}_{i \in I} \mid X \\
 & w ::= u \mid \langle l_i : u_i \rangle_{l_i \in L} \\
 & E ::= X = u
\end{array}
$$

A special type $\bot$ is used to represent the type of a null value. Usages, as previously mentioned, describe the protocol of a class. Following a step of the protocol, is represented by a transition on the usage. Table 1 shows the two kinds of transitions a usage can describe. The rule (Branch) allows choosing one of the available methods defined in the branch usage, while (Sel) describes choosing a remaining protocol based on the return value of a method. Finally, (Unfold) allows recursive behaviour by replacing usage variables with the definition of said variable. The usage end represents the finished protocol, where no more transitions are available.

$$\text{(Branch)}\quad \frac{j \in I}{\{m_i : w_i\}_{i\in I}^{\vec{E}} \xrightarrow{m_j} w_j^{\vec{E}}} \qquad \text{(Unfold)}\quad \frac{u^{\vec{E}\cup\{X=u\}} \xrightarrow{m} \mathcal{U}}{X^{\vec{E}\cup\{X=u\}} \xrightarrow{m} \mathcal{U}}$$

$$\text{(Sel)}\quad (\langle l_i : u_i\rangle_{l_i\in L})^{\vec{E}} \xrightarrow{l_i} u_i^{\vec{E}}$$

Table 1 Transitions for usages

In Mungo we employ a linear typing discipline, where there can be only one reference to an object, and where the protocol of an object must be finished, before discarding the object. To that end, we use the concept of linear types from [10].

▶ **Definition 2** (Linear type). *Let t be a type. We say that t is linear, written* $\mathsf{lin}(t)$*, if it a type state with a usage different from* end.

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

We use this concept to ensure protocol completion, as we can ensure that linear objects are not lost when overriding fields, or using objects as parameters to methods. Having only a single reference to an object makes it possible to reason about state changes in the object, as we know that changes can happen only through that reference. We extend this concept to the fields of a class, and say that the class fields are *terminated* if the types of all fields are non-linear.

3 The Usage Inference Problem

In this section, we define the problem of usage inference. The goal of inference is to allow the programmer to specify the usage, only when a specific protocol is required to be followed. If no particular protocol is required to be followed, a usage must be inferred such that no run-time errors caught by the type system can occur. This usage must also be the most permissible usage that will not result in errors, such that valid method call sequences are not disallowed. This section will formalise the inference problem, as well as the properties that inferred usages must adhere to.

To define what is meant by *most permissible*, we define an ordering of usages. The intuition of this ordering is that if $\mathcal{U} \sqsubseteq \mathcal{U}'$ then $\mathcal{U}'$ can describe the same behaviour as $\mathcal{U}$ and possibly more. We define this as a simulation ordering [20] where $\mathcal{U}'$ must be able to match any transition sequence from $\mathcal{U}$.

▶ **Definition 3** (Usage subtyping). *Let $R \subseteq$ **Usages** $\times$ **Usages**. R is 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 there exists a usage simulation R such that $(\mathcal{U},\mathcal{U}') \in R$.

With the ordering relation $\sqsubseteq$ defined, the principal usage is the largest usage for a class that will not experience errors caught by the type system.

▶ **Definition 4** (Principal Usage). *A usage $\mathcal{U}$ is a* principal usage *for class C if $\vdash_{\vec{D}}$* class $C\{\mathcal{U}, \vec{F}, \vec{M}\}$ *and for all $\mathcal{U}'$ where $\vdash_{\vec{D}}$* class $C\{\mathcal{U}', \vec{F}, \vec{M}\}$ *we have that $\mathcal{U}' \sqsubseteq \mathcal{U}$.*

The reason that we require the inferred usages to be principal is that otherwise the usage $\mathsf{end}^{\emptyset}$ would be a valid inferred usage for any class. The usage will result in no run-time errors, as no method calls will be allowed, but will of course not allow any well-behaving method call sequence either.

A consequence of inferring principal types is that inferred usages will always be recursive. As we require protocol completion, every object will, at the end of its protocol, have only non-linear fields. This particular state is similar to the initial object state, hence the protocol can be followed once more. So if a class is well-typed with usage $\mathcal{U} = u^{\vec{E}}$, then it will also be well-typed with the usage $u^{\vec{E}}\{\mathsf{end} \ / \ u\}$. The only way to capture this in the inferred usages is to have a recursive structure, where a class can either terminate with usage end or return to the initial usage.

We now formally define the inference problem. We allow classes to not have declared usages, meaning that for a class C, $C.\mathsf{usage}$ can be undefined. The problem of usage inference is then to define a substitution for unspecified usages with a principal usage for the given class.

▶ **Definition 5** (Inference Problem). *Given a program $\vec{D}$, find for each $C \in \vec{D}$ where $C.\mathsf{usage}$ is not defined, a usage $\mathcal{U}$ such that $\mathcal{U}$ is a principal usage for C.*

When inferring usages, we require that the usages of fields are known at inference time. This has the obvious implication that inference for a class with a field of its own type, is not supported by this type of inference. Also, we cannot infer the usages of classes that contains fields of each other. We therefore need a way to represent dependencies between classes, in order to ensure we only attempt to infer usages when we know the usage of each field. This is defined formally as follows.

▶ **Definition 6** (Inference Graph). *An* inference graph $G_{\vec{D}}$ *for a program $\vec{D}$ is a directed graph and is given by $G_{\vec{D}} = (\vec{C}, \{(C_2, C_1) | C_1, C_2 \in \vec{D} \wedge (C_2\ f) \in C_1.\mathsf{fields} \wedge C_2.\mathsf{usage}$ is undefined $\wedge$ $C_1.\mathsf{usage}$ is undefined$\})$. The order of inference is given by a topological ordering of the vertices of G.*

We only consider inference graphs which are acyclic, where there are no cyclic dependencies in field usages, because all fields must be known in order to do inference. Furthermore, edges are only added between classes with undefined usages, meaning that cycles can be avoided by specifying the usage for one of the classes that would otherwise have a cyclic dependency.

4 Usage Inference

To ensure that the inferred usages do not allow for null-dereferencing, we use an approach similar to that of the original type system, to track the usages of fields. Since we do not allow overriding linear values, we cannot use the approach from [16] where pre-conditions are extracted right-to-left of method bodies. Instead, the idea that we employ is to scan the method bodies left-to-right and create a transition system between object states. These object states then act as pre-conditions for calling other methods.

Extracting the Method Availability Transition System

The first step of inferring the usage is to identify what methods are available for a class at what times. For this we use a *rooted labelled transition system* (RLTS). A RLTS is a 4-tuple

$(Q, A, \rightarrow, q_0)$ where Q is the set of states, A are the labels, $\rightarrow \subseteq Q \times A \times Q$ are the transitions and $q_0 \in Q$ is the initial state [11].

In this transition system, we wish to reflect the change on an object caused by a method call. For this, we use the field typing environment as defined in [10]. The effect a method can have is a change to the fields on the object, which will be reflected by a change in type in the field typing environment.

▶ **Definition 7** (Field typing environment). *The field typing environment is a partial function and maps field names to types. We write* $\boldsymbol{EnvT_F}$ *to denote the set of all field typing environments.*

$$envT_F : \boldsymbol{FNames} \rightharpoonup (\boldsymbol{Typestates}_\perp \cup \boldsymbol{BTypes}).$$

For the method availability transition system, the states are the field typing environments $Q = \mathbf{EnvT_F}$ and the labels are the method names $A = \{m \mid t\ m(t'\ x)\{e\} \in C.\mathsf{methods}\}$. The initial state is the initial field typing environment $q_0 = \vec{F}.\mathsf{inittypes}$. We define $\vec{F}.\mathsf{inittypes}$ to be a field typing environment where fields are mapped to their initial type. For fields with class types, the initial type is $\perp$, otherwise the field of base types are simply mapped to their base type. The transition relation $\rightarrow$ is given by the (Class) rule in Table 2. The $\rightarrow$ relation contains a transition between two states $envT_F \xrightarrow{m} envT'_F$ if calling method m in the field typing environment $envT_F$ does not result in errors and the resulting field typing environment is $envT'_F$.

$$\text{(Class)}\quad \frac{\begin{array}{c}\{\mathsf{this} \mapsto envT_F\}; \emptyset \cdot (\mathsf{this}, [x \mapsto t]) \vdash^{\emptyset} e : t' \triangleright \{\mathsf{this} \mapsto envT'_F\}; \emptyset \cdot (\mathsf{this}, [x \mapsto t'']) \\ t'\ m(t\ x)\ \{e\} \in C.\mathsf{methods} \qquad\qquad \neg\mathsf{lin}(t'')\end{array}}{envT_F \xrightarrow{m} envT'_F}$$

Table 2 Method availability transition relation $\rightarrow$

The (Class) rule types an expression in its premise using the type system presented in [10]. Here we present the intuition of the type system and defer a more detailed explanation to Appendix A. The general intuition underlying the type system is that we follow the usage transitions of a class, and check that each transition does not result in a protocol error. The typing rules for classes are shown in Table 3. The judgments are of the form $\Theta; envT_F \vdash_{\vec{D}} C[\mathcal{U}] \triangleright envT'_F$, and can be read as: "By following usage $\mathcal{U}$ starting with the field environment $envT_F$, the resulting field environment is $envT'_F$". The environment Θ is used to handle recursive usage variables in the rules (TCRec) and (TCVar). The central rule (TCBr) checks that method sequences defined by the usage, do not lead to the errors captured by the type system. It does so by type checking the method body of the method specified by the usage, given the current bindings of fields in $envT_F$. The rule (TCCh) is for type checking choice usages and does so by verifying that no matter what branch is chosen, the resulting method sequences will not experience errors.

Type judgements for expressions are of the form $\Lambda; \Delta \vdash^{\Omega} e : t \triangleright \Lambda'; \Delta'$ where Λ is used to store the current field type environment, Δ stores the current object context and parameter bindings, and Ω stores environments used for labelled expressions.

Notice that the premise of (Class) is similar to that of (TCBr), this is the connection between inference and the type system, where a transition is added to $\rightarrow$, if that same method call would be well-typed with (TCBr).

$$\text{(TCBr)}\quad \frac{\begin{array}{c} I \neq \emptyset \qquad\qquad \forall i \in I \;.\; \exists envT_F'' \;. \\ \{\mathsf{this} \mapsto envT_F\}; \emptyset \cdot (\mathsf{this}, [x_i \mapsto t_i']) \vdash_{\vec{D}} e_i : t_i \triangleright \{\mathsf{this} \mapsto envT_F''\}; \emptyset \cdot (\mathsf{this}, [x_i \mapsto t_i'']) \\ \mathsf{terminated}(t_i'') \qquad t_i\ m_i(t_i'\ x_i)\{e_i\} \in C.\mathsf{methods}_{\vec{D}} \qquad \Theta; envT_F'' \vdash_{\vec{D}} C[u_i^{\vec{E}}] \triangleright envT_F' \end{array}}{\Theta; envT_F \vdash_{\vec{D}} C[\{m_i; u_i\}_{i \in I}^{\vec{E}}] \triangleright envT_F'}$$

$$\text{(TCCh)}\quad \frac{\forall l_i \in L \;.\; \Theta; envT_F \vdash_{\vec{D}} C[u_i^{\vec{E}}] \triangleright envT_F'}{\Theta; envT_F \vdash_{\vec{D}} C[\langle l_i : u_i \rangle_{l_i \in L}^{\vec{E}}] \triangleright envT_F'}$$

$$\text{(TCEn)}\quad \Theta; envT_F \vdash_{\vec{D}} C[\mathsf{end}^{\vec{E}}] \triangleright envT_F$$

$$\text{(TCVar)}\quad (\Theta, [X \mapsto envT_F]); envT_F \vdash_{\vec{D}} C[X^{\vec{E}}] \triangleright envT_F'$$

$$\text{(TCRec)}\quad \frac{(\Theta, [X \mapsto envT_F]); envT_F \vdash_{\vec{D}} C[u^{\vec{E}}]_{\vec{D}} \triangleright envT_F'}{\Theta; envT_F \vdash_{\vec{D}} C[X^{\vec{E} \uplus \{X = u\}}] \triangleright envT_F'}$$

Table 3 Typing class usage definitions

Usage Graph

From the method availability transition system, we create another RLTS $(S, A, \Rightarrow, s_0)$, representing the usage of a class. We call this transition system the *usage graph*. The construction of the usage graph is similar to what we presented informally in Example 1.

For simplicity, we say that all terminated field typing environments are equal. Hence if $\mathsf{terminated}(envT_F)$ and $\mathsf{terminated}(envT_F')$ then $envT_F = envT_F'$. This will result in smaller inferred usages, and will not affect correctness, as all terminated field typing environments allow the same method calls. We write $envT_{F_\perp}$ to indicate a terminated field typing environment.

The set of states in the usage graph is a subset of the method availability transitions, with the addition of the state end, indicating that the usage is in the end state. end is not a field typing environment like the remaining states and denotes the terminated protocol, rather than a terminated field typing environment.

$$S = \{envT_F \mid envT_{F_\perp} \rightarrow^* envT_F \wedge envT_F \rightarrow^* envT_{F_\perp}\} \cup \{\mathsf{end}\}$$

We require that only reachable field typing environments are used as states, furthermore we require that any field typing environment can also execute a sequence of method calls to return to a terminated field typing environment $envT_{F_\perp}$. The reason for this is to ensure that all method sequences allowed in the final usage can reach end.

The labels of the transition system are the method names of C, $A = \{m \mid t\ m(t'\ x)\{e\} \in C.\mathsf{methods}\}$ as the previous RLTS. The initial state is again $s_0 = \vec{F}.\mathsf{inittypes}$. The transition relation $\Rightarrow$ is described by the rules in Table 4. The (Trans) rule tells us that a class usage allows a method m from environments $envT_F$ to $envT_F'$ if the same transition can be extracted by rule (Class). Rule (End) handles the case where a field typing environment has a transition to a terminated field typing environment, in which case the usage graph should contain a transition to the end state.

$$(\textsc{Trans}) \quad \frac{envT_F \xrightarrow{m} envT'_F}{envT_F \overset{m}{\Rightarrow} envT'_F} \qquad (\textsc{End}) \quad \frac{envT_F \xrightarrow{m} envT_{F_\perp}}{envT_F \overset{m}{\Rightarrow} \mathsf{end}}$$

Table 4 The usage graph transition relation $\Rightarrow$

Generating a Usage

We now describe how to convert a usage graph to a usage that can be type checked. The idea behind the algorithm is to represent each state of the usage graph as a usage variable. Transitions from a state in the usage graph will be represented as a branch usage $\{m_i; X_i\}_{i \in I}$ where m_i is the method name, and X_i is the usage variable of the next state in the usage graph. We use the notation X_q to describe a usage-variable that is unique for a particular state $q \in S$.

In Table 5 and Table 6 a declarative description of usage generation is shown. The judgements for usage generation are of the form $(S, A, \Rightarrow, s_0) \triangleright \mathcal{U}$ and tell us how to create a usage from the usage graph, by extracting usage fragments and assign them to usage variables. The judgements for usage fragment generation are of the form $(S, A, \Rightarrow, s_0) \vdash s \triangleright w$ and describe how we create individual usage fragments w with respect to a state $s \in S$.

The rule (ICLASS) tells us that in the context of a RLTS $(S, A, \Rightarrow, s_0)$ we can create a usage $X^{\{X_{s_i} = \mathcal{U}_i\}}_{envT_{F_\perp}}$ if for each state s_i in S we can create the associated usage fragment u_i.

$$(\textsc{IClassEnd}) \quad (S, A, \emptyset, envT_{F_\perp}) \triangleright \mathsf{end}^{\emptyset}$$

$$(\textsc{IClass}) \quad \frac{\Rightarrow \neq \emptyset \qquad \forall s_i \in S \qquad (S, A, \Rightarrow, envT_{F_\perp}) \vdash s_i \triangleright \mathcal{U}_i}{(S, A, \Rightarrow, envT_{F_\perp}) \triangleright X^{\{X_{s_i} = \mathcal{U}_i\}}_{envT_{F_\perp}}}$$

Table 5 Usage generation rules for constructing usage variables and assigning usage fragments to them.

The rule (ICALL) handles the case where we want to build a branch usage fragment. It tells us that given a state $envT_F$ we can create a usage fragment $\{m_i; X_{s_i}\}_{m_i \in A, s_i \in S}$ if a transition exists in the transition system such that starting in $envT_F$ we can reach state s_i using method m_i. In (ICALLE) we handle the case where the method called returns an enum label. It works in a similar way to rule (ICALL) but creates an additional choice step with the labels of enum L. We now have a situation where a usage fragment w can non-deterministically choose between the choice usage with the fragment $\{m_i; \langle l_j : X_{s_i} \rangle_{l_j \in L}\}$, or the usage variable with the fragment $\{m_i; X_{s_i}\}$. There is no difference in the usage X_{s_i} for each choice label, simply because imposing any constraints on one branch would make the usage non-principal.

$$\text{(ICall)} \quad \frac{envT_F \overset{m_i}{\Longrightarrow} s_i \quad t\ m_i(t'\ x)\ \{e\} \in C.\mathsf{methods} \quad t \neq L}{(S, A, \Rightarrow, envT_{F_\perp}) \vdash envT_F \triangleright \{m_i; X_{s_i}\}_{m_i \in A, s_i \in S}}$$

$$\text{(ICalle)} \quad \frac{envT_F \overset{m_i}{\Longrightarrow} s_i \quad L\ m_i(t'\ x)\ \{e\} \in C.\mathsf{methods}}{(S, A, \Rightarrow, envT_{F_\perp}) \vdash envT_F \triangleright \{m_i; \langle l_j : X_{s_i}\rangle_{l_j \in L}\ m_i; X_{s_i}\}_{m_i \in A, s_i \in S}}$$

$$\text{(IEnd)} \quad (S, A, \Rightarrow, envT_{F_\perp}) \vdash \mathsf{end} \triangleright \mathsf{end}$$

Table 6 Usage generation rules for constructing usage fragments.

From the rules it can be seen that the inferred usage of a class will be on the form $X^{\{X_i = \{m_i; X_i'\} \quad X_j = \{m_j; \langle l_k : X_j'\rangle_{l_k \in L}\ m_j; X_j'\}\}_{i \in I, j \in J}}$. In other words, each usage variable definition $X_{envT_F} = \{m; X_{envT_F'}\}$ describes a transition $envT_F \overset{m}{\Rightarrow} envT_F'$ in the usage graph.

The rules of Table 5 and Table 6 declaratively defines how a usage is generated from the usage graph. Based on this, we present an imperative algorithm for generating the usage, as shown in Algorithm 1. We use the following notation to build branch usages:

$$\{m_i : u_i\}_{i \in I} \cup \{m_j : u_j\}_{j \in J} \overset{\text{def}}{=} \{m_i : u_i\ m_j : u_j\}_{i \in I, j \in J}.$$

Algorithm 1 Usage Generation

```
1:  function Infer(S, A, ⇒, envT_F⊥)
2:      function Reach(envT_F)
3:          return {envT_F' | ∃m ∈ A.envT_F ⇒^m envT_F'}
4:      function CreateState(envT_F)
5:          u ← ∅
6:          for all m ∈ A do
7:              for all s ∈ S do
8:                  if envT_F ⇒^m s then
9:                      if s = end then
10:                         u ← u ∪ {m; end}
11:                         if L m(_ x){_} ∈ C.methods then
12:                             u ← u ∪ {m; ⟨l_i : end⟩_{l_i∈L}}
13:                     else
14:                         u ← u ∪ {m; X_s}
15:                         if L m(_ x){_} ∈ C.methods then
16:                             u ← u ∪ {m; ⟨l_i : X_s⟩_{l_i∈L}}
17:         return u
18:     if Reach(envT_F⊥) = ∅ then
19:         return end^∅
20:     E ← ∅
21:     for all envT_F ∈ S \ {end} do
22:         E ← E ∪ {X_envTF = CreateState(envT_F)}
23: return X^E_envTF⊥
```

5 Example

In this section we show an example of how a usage is inferred. We consider a simple bank example with two classes: `Account` and `Customer`, where the usage for the `Account` class is known:

$$\texttt{Account}.\mathsf{Usage} = \{\texttt{init}; X\}^{\vec{E}}$$

where

$$\vec{E} = \{X = \{\texttt{deposit}; X \ \ \texttt{enoughFunds}; \left\langle \begin{array}{l} \mathsf{true} : \{\texttt{withdraw}; X\} \\ \mathsf{false} : X \end{array} \right\rangle \ \ \texttt{close}; \mathsf{end}\}\}$$

It specifies that the `Account` class has five public methods: `init`, `withdraw`, `deposit`, `enoughFunds`, and `close`. The usage defines a protocol where after initialisation the protocol enters a recursive state where money can be added to an account or withdrawn given there are enough funds in the account. At some point, the protocol can be finished by closing the account.

In Listing 1 a `Customer` class is declared with two `Account` fields. Inferring a usage for `Customer` proceeds by following the approach described in Section 4. The algorithm starts by creating a method availability transition system for the class. Then non-terminating transitions are filtered out, resulting in a usage graph. Finally, the usage graph is converted into a usage. In the remaining part we use the abbreviations `sa` $\stackrel{\text{def}}{=}$ `savingsAccount`, `da` $\stackrel{\text{def}}{=}$ `debitAccount`, and `Acc` $\stackrel{\text{def}}{=}$ `Account`.

```
1  class Customer{
2      Account savingsAccount
3      Account debitAccount
4
5      void createDebitAccount(){
6          debitAccount = new Account();
7          debitAccount.init()
8      }
9      void createSavingsAccount(){
10         savingsAccount = new Account();
11         savingsAccount.init();
12         savingsAccount.deposit()
13     }
14     void transferFunds(){
15         switch(savingsAccount.enoughFunds()){
16             true:  savingsAccount.withdraw();
17                    debitAccount.deposit()
18             false: unit
19         }
20     }
21     void closeAccounts(){
22         debitAccount.close();
23         savingsAccount.close()
24     }
25 }
```

Listing 1 `Customer` class definition

The algorithm starts by considering the initial field typing environment $envT_F$ = Acc .fields.inittypes = {sa : ⊥, da : ⊥}. From $envT_F$ it then examines all methods of the class and attempts to derive a transition to some $envT'_F$ using the rule (Class). For example, Table 7 shows the derivation of a transition from $envT_F$ to $envT_F\{\texttt{da} \mapsto \texttt{Account}[X]\}$ using method createDebitAccount where $\Lambda_\perp = \{\mathsf{this} \mapsto envT_F\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Lambda_\perp;\emptyset\cdot(\mathsf{this},[x\mapsto \mathsf{void}])\vdash_{\vec{D}} \mathsf{new}\ \texttt{Acc} : \texttt{Acc}[\texttt{Acc.Usage}] \triangleright \Lambda_\perp;\emptyset\cdot(\mathsf{this},[x\mapsto\mathsf{void}])}\,\text{TNew} \quad \neg\mathsf{lin}(\perp) \quad \texttt{Acc} = \Lambda(\mathsf{this}).\mathsf{class} \quad \mathsf{agree}(\texttt{Acc.fields}(\texttt{da}), \texttt{Acc}[\texttt{Acc.Usage}])}{\Lambda_\perp;\emptyset\cdot(\mathsf{this},[x\mapsto\mathsf{void}])\vdash_{\vec{D}} \texttt{da} = \mathsf{new}\ \texttt{Acc} : \mathsf{void} \triangleright \Lambda_\perp\{\mathsf{this}.\texttt{da}\mapsto\texttt{Acc}[\texttt{Acc.Usage}]\};\emptyset\cdot(\mathsf{this},[x\mapsto\mathsf{void}])}\,\text{TFld} \quad \texttt{Acc.Usage} \xrightarrow{\texttt{init}} X \quad \mathsf{void}\ \texttt{init}(\mathsf{void}\ x)\ \{\_\} \in \texttt{Acc.methods}}{\Lambda_\perp\{\texttt{da}\mapsto\texttt{Acc}[\texttt{Acc.Usage}]\};\emptyset\cdot(\mathsf{this},[x\mapsto\mathsf{void}])\vdash_{\vec{D}} \texttt{da.init}() : \mathsf{void} \triangleright \Lambda_\perp\{\mathsf{this}.\texttt{da}\mapsto\texttt{Acc}[X]\};\emptyset\cdot(\mathsf{this},[x\mapsto\mathsf{void}])}\,\text{TCallF}}{\Lambda_\perp;\emptyset\cdot(\mathsf{this},[x\mapsto\mathsf{void}])\vdash_{\vec{D}} \texttt{da} = \mathsf{new}\ \texttt{Acc};\texttt{da.init}() : \mathsf{void} \triangleright \Lambda_\perp\{\mathsf{this}.\texttt{da}\mapsto\texttt{Acc}[X]\};\emptyset\cdot(\mathsf{this},[x\mapsto\mathsf{void}])}\,\text{TSeq} \quad \mathsf{void}\ \texttt{createDebitAccount}(\mathsf{void}\ x)\{\mathsf{new}\ \texttt{Acc};\texttt{da.init}()\} \in \texttt{Acc.methods}}{envT_F \xrightarrow{\texttt{createDebitAccount}} envT_F\{\texttt{da}\mapsto\texttt{Acc}[X]\}}\,\text{Class}$$

Table 7 Derivation for a transition from initial state using method createDebitAccount

Since a derivation from $envT_F$ to $envT_F\{\texttt{da} \mapsto \texttt{Account}[X]\}$ with createDebitAccount is possible, the transition is included in the method availability transition system. Continuing with the same approach for every method in Customer with respect to $envT_F$ results in the following transitions: Calling createSavingsAccount from $envT_F$ is allowed and results in a transition to $envT_F\{\texttt{sa} \mapsto \texttt{Account}[X]\}$. Finally, calling transferFunds or closeAccounts from $envT_F$ is impossible hence no transitions are added for these methods.

At this point, all methods of class Customer have been considered as transitions from $envT_F$. Now the same procedure is repeated for every other field typing environment. Since we know that in the usage graph, unreachable states from $envT_F$ are filtered out, we only consider the transitions in the reachable subgraph of the method availability transitions. Figure 2 shows the section of the method availability transitions, reachable from $envT_F$ where $envT_F^{(1)} = \{\texttt{da} : \texttt{Account}[X],\ \texttt{sa} : \perp\}$, $envT_F^{(2)} = \{\texttt{da} : \perp,\ \texttt{sa} : \texttt{Account}[X]\}$, and $envT_F^{(3)} = \{\texttt{da} : \texttt{Account}[X],\ \texttt{sa} : \texttt{Account}[X]\}$.

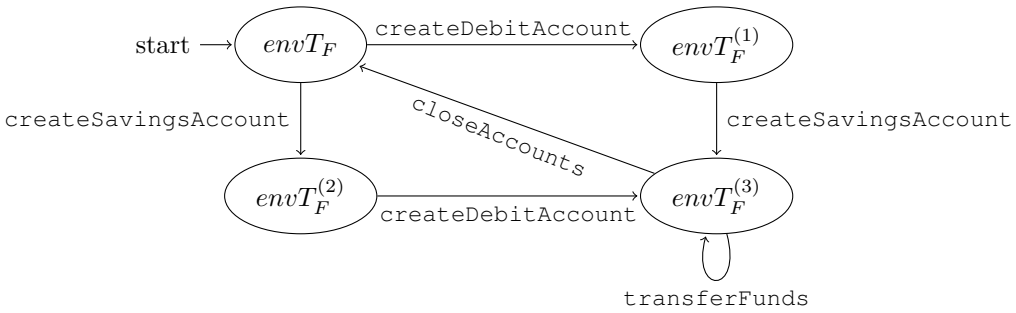


Figure 2 Method availability transition system of class Customer

The method availability transition system in Figure 2 is transformed into a usage graph by filtering out non-terminating transitions and using the rules in Table 4. In the case of the Customer no states from the figure is filtered, but an end state is added with a transition from $envT_F^{(3)}$ using closeAccounts as stated in rule (End).

The usage graph is then converted into an usage by using the rules in Table 5 and Table 6. Starting from the root state of the usage graph $envT_F$, rule (ICLass) specifies that the usage is of the form $X^{\vec{E}'}_{envT_F}$ where $\vec{E}'$ contains a number of recursive variables

assigned to usage fragments. Since there exists a transition from $envT_F$ to $envT_F^{(1)}$ with label `createDebitAccount` and from $envT_F$ to $envT_F^{(2)}$ with label `createSavingsAccount`, then using rule (ICALL) a usage fragment $\left\{ \begin{array}{l} \texttt{createDebitAccount}; X_{envT_F^{(1)}} \\ \texttt{createSavingsAccount}; X_{envT_F^{(2)}} \end{array} \right\}$ is created and assigned in $\overrightarrow{E'}$ to variable X_{envT_F} by rule (ICLASS). The same procedure is applied for each state in the usage graph. Finally, the inferred usage is

$$\texttt{Customer}.\mathsf{Usage} = X_{envT_F}^{\overrightarrow{E'}}$$

where

$$\overrightarrow{E'} = \left\{ \begin{array}{l} X_{envT_F} = \left\{ \begin{array}{l} \texttt{createDebitAccount}; X_{envT_F^{(1)}} \\ \texttt{createSavingsAccount}; X_{envT_F^{(2)}} \end{array} \right\} \\ X_{envT_F^{(1)}} = \{\texttt{createSavingsAccount}; X_{envT_F^{(3)}}\} \\ X_{envT_F^{(2)}} = \{\texttt{createDebitAccount}; X_{envT_F^{(3)}}\} \\ X_{envT_F^{(3)}} = \left\{ \begin{array}{l} \texttt{transferFunds}; X_{envT_F^{(3)}} \\ \texttt{closeAccounts}; X_{envT_F} \\ \texttt{closeAccounts}; \mathsf{end} \end{array} \right\} \end{array} \right\}.$$

6 Inference Algorithm Properties

In this section, we describe the properties and prove the correctness of the inference algorithm. The most interesting result is that the inferred usage is a principal usage, meaning that it is the largest usage that makes a class well-typed.

The first property that we show is, that the inference algorithm terminates and that a usage is returned.

▶ **Lemma 8** (Inference Algorithm Termination). *Given a class C, the inference algorithm returns a usage.*

Proof. By inspection of Algorithm 1 we see that all loops are bounded, hence the algorithm will terminate and return a usage, possibly the trivial usage $\mathsf{end}^{\emptyset}$. At the end of this section, we provide an analysis of the runtime complexity of the algorithm.

◀

The next lemma shows the relation between the method availability transitions and the usage graph. If a usage makes a class well-typed, then all field typing environments reachable with that usage is present in both the method availability transition system and the usage graph. This lemma is used in the following proofs because it shows that the interesting field typing environments are not filtered out as part of the construction of the usage graph.

▶ **Lemma 9** (RLTS Filtering). *Let $\vdash_{\overrightarrow{D}} \mathsf{class}\ C\ \{\mathcal{U}, \overrightarrow{M}, \overrightarrow{F}\}$ and let $m_1, m_2, \ldots, m_k$ be a method sequence. If $\mathcal{U} \xrightarrow{m_1} \cdots \xrightarrow{m_k} \mathcal{U}'$ ignoring label-transitions, and $envT_F''$ is the resulting field typing environment starting from $envT_{F_\perp}$, then $envT_{F_\perp} \xrightarrow{m_1} \cdots \xrightarrow{m_k} envT_F''$ and $envT_{F_\perp} \overset{m_1}{\Longrightarrow} \cdots \overset{m_k}{\Longrightarrow} envT_F''$.*

Proof. Let be C be a well typed class $\vdash_{\overrightarrow{D}} \mathsf{class}\ C\{\mathcal{U}, \overrightarrow{F}, \overrightarrow{M}\}$. Since we have that $\emptyset; \overrightarrow{F}.\mathsf{inittypes} \vdash_{\overrightarrow{D}} C[\mathcal{U}] \triangleright envT_F'$ and $\mathsf{terminated}(envT_F')$, we can create a labelled transition system for the class, as illustrated in Figure 3. The transitions represent following a transition of the usage, and each node is a usage along with the corresponding field typing environment.

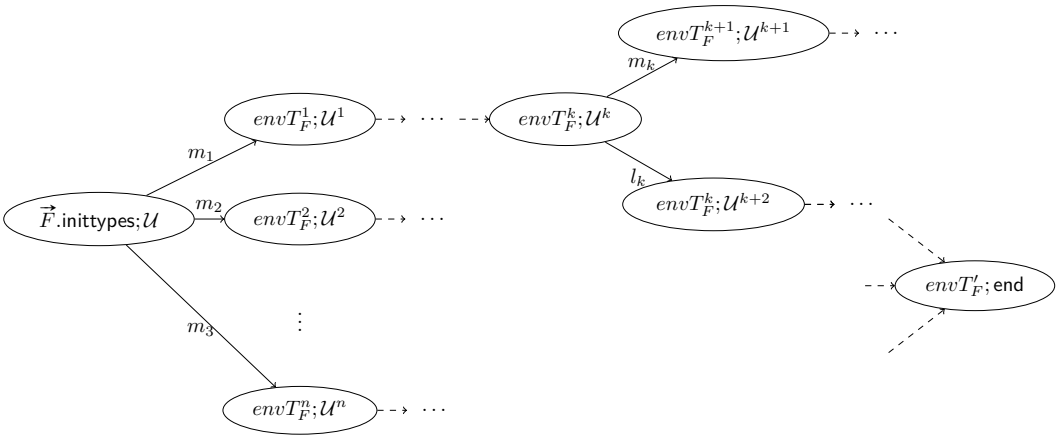


Figure 3 LTS for $\mathcal{U}$

Consider the rules (TCBR) and (TCCH) in Table 8 (repeated from Table 3). Starting from the pair $(\vec{F}.\mathsf{inittypes};\mathcal{U})$ where we know that $\emptyset;\vec{F}.\mathsf{inittypes} \vdash_{\vec{D}} C[\mathcal{U}] \triangleright envT_F'$, the two rules ensures that for each following state $(envT_F;\mathcal{U}')$ in the LTS we have that $\emptyset;envT_F \vdash_{\vec{D}} C[\mathcal{U}'] \triangleright envT_F'$. This gives us that if $(envT_F;\mathcal{U}') \xrightarrow{m} (envT_F'';\mathcal{U}'')$ then by (TCBR) the method body of m is well typed, and the parameter is terminated, hence we have $envT_F \xrightarrow{m} envT_F''$. Since this is the case for any pair in the graph, we have that all nodes can reach a terminated environment $envT_F'$, hence no reachable states are filtered out in the usage LTS and we have $envT_F \xRightarrow{m} envT_F''$.

$$\text{(TCBR)}\quad \frac{\begin{array}{c} I \neq \emptyset \qquad \forall i \in I \;.\; \exists envT_F'' \;. \\ \{\mathsf{this} \mapsto envT_F\};\emptyset \cdot (\mathsf{this},[x_i \mapsto t_i']) \vdash_{\vec{D}} e_i : t_i \triangleright \{\mathsf{this} \mapsto envT_F''\};\emptyset \cdot (\mathsf{this},[x_i \mapsto t_i'']) \\ \mathsf{terminated}(t_i'') \qquad t_i\ m_i(t_i'\ x_i)\{e_i\} \in C.\mathsf{methods}_{\vec{D}} \qquad \Theta;envT_F'' \vdash_{\vec{D}} C[u_i^{\vec{E}}] \triangleright envT_F' \end{array}}{\Theta;envT_F \vdash_{\vec{D}} C[\{m_i;u_i\}_{i\in I}^{\vec{E}}] \triangleright envT_F'}$$

$$\text{(TCCH)}\quad \frac{\forall l_i \in L \;.\; \Theta;envT_F \vdash_{\vec{D}} C[u_i^{\vec{E}}] \triangleright envT_F'}{\Theta;envT_F \vdash_{\vec{D}} C[\langle l_i : u_i\rangle_{l_i\in L}^{\vec{E}}] \triangleright envT_F'}$$

Table 8 Typing class usage definitions

◀

The following theorem proves that the usage inferred by Algorithm 1 and the rules in Section 4, is the largest usage. This is one of the two properties that must be true, for the inferred usage to be considered principal.

▶ **Theorem 10** (Largest Usage Inferred). *Let C be a class and $\mathcal{U}_I$ be an inferred usage, then $\mathcal{U}_I$ must be the largest usage. In other words, we have that $\forall \mathcal{U}. \vdash_{\vec{D}} \mathsf{class}\ C\ \{\mathcal{U},\vec{F},\vec{M}\} \implies \mathcal{U} \sqsubseteq \mathcal{U}_I$.*

Proof sketch. We assume that C is well-typed with some usage $\mathcal{U}$, and show that this usage can be simulated by the inferred usage $\mathcal{U}_I$. To show that this is the case, we build a relation R and show that this is a simulation, showing that $\mathcal{U} \sqsubseteq \mathcal{U}_I$.

The relation R is built iteratively, starting from the pair $(\mathcal{U},\mathcal{U}_I)$. Then each iteration adds three types of pairs to the relation. Figure 4 illustrates the different pairs. The first

type of pair, connected by dots, captures the transition caused by a method call, where the return value is not used in a switch (i.e. the following usage is not a choice usage). The second type of pairs, indicated by dashes, adds pairs where a method call is followed by a choice. The final type of pairs adds pairs where a label is chosen in a choice-usage.

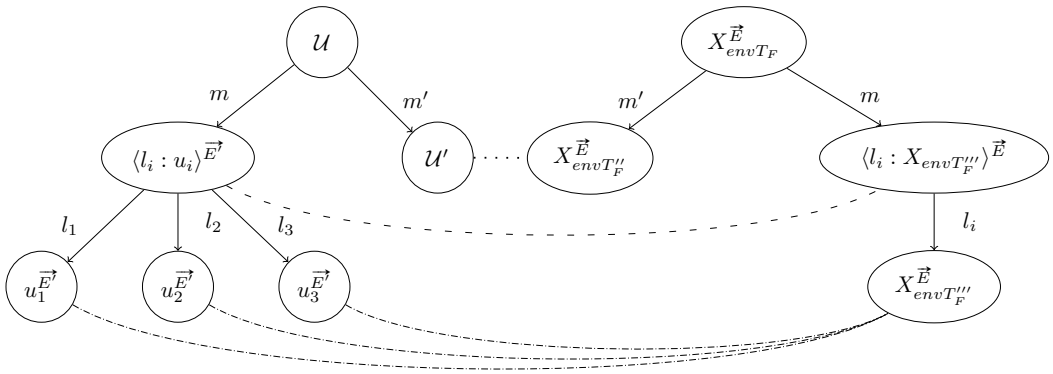


Figure 4 Snapshot of the iterative building of simulation relation

As long as new pairs are added, the iterations continue, and when we reach a fixed-point, the relation will be a simulation. We will always reach such a fixed-point since the usage is finite in size, hence there will be finitely many usages it can transition to. We show that R is a simulation by choosing an element from R and showing that transitions $\mathcal{U} \xrightarrow{m} \mathcal{U}$ is covered by the first type of pairs, $\mathcal{U} \xrightarrow{m} \langle l_i : u_i \rangle^{\vec{E}}$ is covered by the second type and $\langle l_i : u_i \rangle^{\vec{E}} \xrightarrow{l} u_i^{\vec{E}}$ is covered by the third type, in such a way that the inferred usage has matching transitions. ◀

Proof. We prove this with co-induction. Assume $\vdash_{\vec{D}} \mathsf{class}\{\mathcal{U}_0, \vec{F}, \vec{M}\}$, then we know that $\emptyset; \vec{F}.\mathsf{inittypes} \vdash_{\vec{D}} C[\mathcal{U}_0] \triangleright envT'_F$ and $\mathsf{terminated}(envT'_F)$.

From (ICLASS) we know that U_I is generated s.t.

$$U_I = X^{\vec{E}}_{envT_{F_\perp}}$$

We now define a relation R to contain all reachable usages from $\mathcal{U}_0$ and the corresponding reachable usage from $\mathcal{U}_I$. In accordance to the proof sketch, we add three types of pairs, as well as the initial pair $(\mathcal{U}_0, \mathcal{U}_I)$.

$$\begin{aligned}
R = &\{(\mathcal{U}_0, \mathcal{U}_I) \quad \text{(1a)}\} \\
&\cup \left\{ (\mathcal{U}', X^{\vec{E}}_{envT''_F}) \;\middle|\; \begin{array}{ll} (\mathcal{U}, X^{\vec{E}}_{envT_F}) \in R & \text{(1b)} \\ \mathcal{U} \xrightarrow{m} \mathcal{U}' & \text{(1c)} \\ \mathcal{U}' \neq \langle l_i : u_i \rangle^{\vec{E'}}_{l_i \in L} & \text{(1d)} \\ \emptyset; envT''_F \vdash_{\vec{D}} C[\mathcal{U}'] \triangleright envT'_F & \text{(1e)} \end{array} \right\} \\
&\cup \left\{ (\langle l_i : u_i \rangle^{\vec{E'}}_{l_i \in L}, \langle l_i : X^{\vec{E}}_{envT''_F} \rangle_{l_i \in L}) \;\middle|\; \begin{array}{ll} (\mathcal{U}, X^{\vec{E}}_{envT_F}) \in R & \text{(1f)} \\ \mathcal{U} \xrightarrow{m} \langle l_i : u_i \rangle^{\vec{E'}}_{l_i \in L} & \text{(1g)} \\ \emptyset; envT''_F \vdash_{\vec{D}} C[\langle l_i : u_i \rangle^{\vec{E'}}_{l_i \in L}] \triangleright envT'_F & \text{(1h)} \end{array} \right\} \\
&\cup \{(u_i^{\vec{E'}}, X^{\vec{E}}) \mid (\langle l_i : u_i \rangle^{\vec{E'}}_{l_i \in L}, \langle l_i : X \rangle^{\vec{E}}_{l_i \in L}) \in R \quad \text{(1i)}\} \qquad (1)
\end{aligned}$$

In (1a) we see that R is non-empty and contains at least the pair $(\mathcal{U}_0, \mathcal{U}_I)$. In (1b), (1f) and (1i) we see that R is defined recursively, hence a solution to R is the least fixed point of (1). We only consider the least fixed point of (1), as this already captures all reachable usages from $\mathcal{U}$. We now show that R is a usage simulation s.t. $\mathcal{U}_0 \sqsubseteq \mathcal{U}_I$.

Let $(\mathcal{U}, X^{\vec{E}}_{envT_F}) \in R$ be a pair from R and $\mathcal{U} \xrightarrow{m} \mathcal{U}'$. First we consider the case where $\mathcal{U}' \neq \langle l_i : u_i \rangle^{\vec{E}}_{l_i \in L}$. We must then show that $X^{\vec{E}}_{envT_F} \xrightarrow{m} X^{\vec{E}}_{envT''_F}$ and $(\mathcal{U}', X^{\vec{E}}_{envT''_F}) \in R$.

We know that $\emptyset; envT_F \vdash_{\vec{D}} C[\mathcal{U}] \triangleright envT'_F$. Since we have that $\mathcal{U} \xrightarrow{m} \mathcal{U}'$ this must have been concluded using (TCBr), hence we have that

$$\{\mathsf{this} \mapsto envT_F\}; \emptyset \cdot (\mathsf{this}, [x \mapsto t']) \vdash_{\vec{D}} e : t \triangleright \{\mathsf{this} \mapsto envT''_F\}; \emptyset \cdot (\mathsf{this}, [x \mapsto t''])$$

We also know that $\mathsf{terminated}(t'')$ and, hence we can conclude with (Class) that $envT_F \xrightarrow{m} envT''_F$. From Lemma 9 we know that $envT_F \overset{m}{\Rightarrow} envT''_F$. From (IClass) and (ICall) we can conclude that $X^{\vec{E}}_{envT_F} \xrightarrow{m} X^{\vec{E}}_{envT''_F}$.

Since we know from (TCBr) that $\emptyset; envT''_F \vdash_{\vec{D}} C[\mathcal{U}'] \triangleright envT'_F$, we have from (1) that $(\mathcal{U}', X^{\vec{E}}_{envT''_F}) \in R$.

Now we consider the case where $\mathcal{U}' = \langle l_i : u_i \rangle^{\vec{E'}}_{l_i \in L}$. We must show that $X^{\vec{E}}_{envT_F} \xrightarrow{m} \langle l_i : X_{envT''_F} \rangle^{\vec{E}}_{l_i \in L}$ and that $(\langle l_i : u_i \rangle^{\vec{E'}}_{l_i \in L}, \langle l_i : X_{envT''_F} \rangle^{\vec{E}}_{l_i \in L}) \in R$.

Since $(\mathcal{U}, X^{\vec{E}}_{envT_F}) \in R$ we know that $\emptyset; envT_F \vdash_{\vec{D}} C[\mathcal{U}] \triangleright envT'_F$. Since $\mathcal{U} \xrightarrow{m} \mathcal{U}'$ we must have from (TCBr) that

$$\{\mathsf{this} \mapsto envT_F\}; \emptyset \cdot (\mathsf{this}, [x \mapsto t']) \vdash_{\vec{D}} e : t \triangleright \{\mathsf{this} \mapsto envT''_F\}; \emptyset \cdot (\mathsf{this}, [x \mapsto t'']).$$

From (Class) we know that $envT_F \xrightarrow{m} envT''_F$ and from Lemma 9 we know that $envT_F \overset{m}{\Rightarrow} envT''_F$. From (IClass) and (IClassE) we know that $X^{\vec{E}}_{envT_F} = \{m; X_{envT''_F}\ m; \langle l_i : X_{envT''_F} \rangle_{l_i \in L}\}^{\vec{E}}$, hence $X^{\vec{E}}_{envT_F} \xrightarrow{m} \langle l_i : X_{envT''_F} \rangle^{\vec{E}}$.

We also know that $\emptyset; envT''_F \vdash_{\vec{D}} C[\langle l_i : u_i \rangle^{\vec{E}}_{l_i \in L}] \triangleright envT'_F$. But then from (1) we must have that $(\langle l_i : u_i \rangle^{\vec{E'}}_{l_i \in L}, \langle l_i : X_{envT''_F} \rangle^{\vec{E}}_{l_i \in L}) \in R$.

Now let $(\mathcal{U}, \mathcal{U}_I) \in R$ be a pair from R and $\mathcal{U} \xrightarrow{l} \mathcal{U}'$, hence $\mathcal{U} = \langle l_i : u_i \rangle^{\vec{E'}}_{l_i \in L}$. We know from (1) that $\mathcal{U}_I$ must be on the form $\langle l_i : X_{envT''_F} \rangle^{\vec{E}}_{l_i \in L}$. Hence we know that all transitions in $\mathcal{U}$ can be matched by $\mathcal{U}_I$.

From (1i) we know directly that $(u_i^{\vec{E'}}, X^{\vec{E}}_{envT''_F}) \in R$.

We have now shown that R is a usage simulation where $(\mathcal{U}_0, \mathcal{U}_I)$, hence we know that $\mathcal{U}_0 \sqsubseteq \mathcal{U}_I$. ◀

The remaining property that must hold for the inferred usage to be principal, is that the usage makes the class well-typed.

▶ **Lemma 11** (Usage Inference Well-typedness). *Let C be a class and $\mathcal{U}_I$ be an inferred usage, then $\vdash_{\vec{D}} \mathsf{class}\ C\ \{\mathcal{U}_I, \vec{F}, \vec{M}\}$*

Proof. This follows trivially from the construction of the inference. A transition $X^{\vec{E}}_{envT_F} \xrightarrow{m} X^{\vec{E}}_{envT'_F}$ requires the transition $envT_F \overset{m}{\Rightarrow} envT'_F$ which in turns require $envT_F \xrightarrow{m} envT'_F$. From (Call) we know that this means calling m will not result in errors as the method body is well typed, given starting environment $envT_F$. Since we have that a terminated

environment must be reachable from any state in our usage RLTS, we know that the usage can reach end, meaning that the typing rules are terminated with (TCEn).

For a l-transition, $\mathcal{U}_I \to^* \mathcal{U}_I'' \xrightarrow{m} \mathcal{U}_I' \xrightarrow{l} X_{q_i}^{\vec{E}}$ this transition must have been introduced by the (ICalle) rule, and $\mathcal{U}_I''$ must have been on the form $\mathcal{U}_I'' = \{m_i; \langle l_j : X_{q_i} \rangle_{l_j \in L}\ m_i; X_{q_i}\}_{i \in I}^{\vec{E}}$ (where $\mathcal{U}_I' = \langle l_j : X_{q_k} \rangle_{l_j \in L}^{\vec{E}}$) for some $k \in I$. From the previous case we have that the call of m is well-typed, and that following method calls are also well-typed. But then it will also be the case for the l-transition, since the field typing environment remains the same when selecting in a choice-usage, hence by using (TCCh) once, we return to the previous case of typing method bodies with (TCBr), which are known to be well-typed. ◀

It now only remains to conclude that the inferred usage is, in fact, a principal usage for a class.

▶ **Theorem 12** (Principal Usage Inference). *Let C be a class and $\mathcal{U}_I$ be an inferred usage, then $\mathcal{U}_I$ is a principal usage for C.*

Proof. Follows directly from Theorem 10 and Lemma 11. ◀

6.1 Complexity Analysis of Algorithm

We end this section with a worst-case complexity analysis of the inference algorithm, in terms of space and time.

In the following sections, we use the notation $|\mathcal{U}|$ to mean the maximum size (the maximum number of distinct states) of a usage in our program, $|f|$ to mean the maximum number of fields of any class in the program similarly, $|m|$ to mean the maximum number of methods of any class, $|e|$ to mean the maximum size of a method body in our program and finally, $|L|$ to mean the maximum number of labels of an enum in our program.

Space Complexity

We analyse the space complexity of the algorithm, by analysing the size of the labelled transition systems that are constructed during the inference algorithm. This will give an upper bound for the space required to run the algorithm.

Method Availability Transitions

As evident by the typing rules in Table 12, Table 13 and Table 14, the size of the derivation tree is linear in the size of the expression and the size of the field typing environment which is the number of fields multiplied by the size of the usages. We get that the space requirement of the type system is $O(|e| + |f| \cdot |\mathcal{U}|)$ by a nondeterministic algorithm. Using Savitch's Theorem we have that the space requirements for a deterministic algorithm are $O((|e| + |f| \cdot |\mathcal{U}|)^2)$.

The number of states in the labelled transition system is bounded by the number of field typing environments. Since a field typing environment maps fields to typestates[1], we have that the size of $|Q| = O(2^{|f| \cdot |\mathcal{U}|})$ where Q is the set of states in the RLTS. The number of edges $| \to |$ is bounded by $O(|Q|^2 \cdot |m|) = O(|m| \cdot 2^{2 \cdot |f| \cdot |\mathcal{U}|})$ since the (Call) rule runs the

[1] A field can also be an enum type or a boolean, we disregard those for simplicity, since the number of typestates will be much larger than the number of enums

type system on each method body in the class and each state can have a non-deterministic transition to the initial state.

The total space complexity for the RLTS extraction is

$$\begin{aligned} & O((|e| + |f| \cdot |\mathcal{U}|)^2 + 2^{|f| \cdot |\mathcal{U}|} + |m| \cdot 2^{2 \cdot |f| \cdot |\mathcal{U}|}) \\ = \; & O((|e| + |f| \cdot |\mathcal{U}|)^2 + 2^{2 \cdot |f| \cdot |\mathcal{U}|} \cdot |m|) \end{aligned}$$

Usage Graph Construction

From the definition of the usage graph $(S, A, \Rightarrow, envT_{F_\perp})$ we see that the size is bounded by the size of the method availability transition system. Similarly, the size of the resulting usage generated by Algorithm 1 is also bounded by the size of method availability transition system. Hence the total amount of space required is $O((|e| + |f| \cdot |\mathcal{U}|)^2 + 2^{2 \cdot |f| \cdot |\mathcal{U}|} \cdot |m|)$.

This also means that the size of the inferred usages is exponential in the number of fields and field usages.

Runtime Complexity

First, we analyse the runtime complexity of creating the method availability transitions using Algorithm 2 in Appendix B. The algorithm is a naive implementation. We only consider the statements which are not in constant time. On line 3 there is an implicit loop over methods bounded by $|m|$. On line 4 and 5 we have two loops both bounded by $|S|$. The innermost loop on line 6, bounded by $|m|$. The if-statement on line 7 runs in $O(|e|)$ time when using non-determinism, converting it to deterministic time it is bounded by $2^{|e|}$. Combining all bounds, we end up with $O(|s|^2 \cdot |m| \cdot 2^{|e|})$. Creating the set of states S can be completed in $O(2^{|f| \cdot |u|})$ by enumerating all possible states. The runtime can be simplified to $O(2^{2 \cdot |f| \cdot |u|} \cdot |m| \cdot 2^{|e|})$.

We now conduct a complexity analysis for the runtime of Algorithm 1. First, we analyse the function Reach on lines 2-3. The runtime is $O(|\Rightarrow|)$ as it considers all transitions in $\Rightarrow$.

Now we analyse the function CreateState on lines 4-17, we assume that the updates to u and the comparisons in the if-statements happen in constant time, hence we only look at the interesting part of the algorithm. On line 6, we have a loop bounded by $|A|$ and on line 7 we have a loop bounded by $|S|$ iterations. Both branches of the if statement on line 9, are equivalent in the number of steps. On line 12 and 16, we have an implicit loop bounded by $|L|$. This gives the function a runtime of $O(|S| \cdot |A| \cdot |L|)$.

Lastly, we analyse the entire outer function Infer. On line 18 the comparison happens in constant time and the function call is bounded by $O(|\Rightarrow|)$, as previously shown. On line 21 we have a loop bounded by $|S|$ steps. On line 22 we have the call to CreateState, which was analysed to be $O(|S| \cdot |A| \cdot |L|)$. This gives us the combined runtime Infer to be $O(|\Rightarrow| + |S| \cdot (|S| \cdot |A| \cdot |L|))$, which can be simplified to $O(|\Rightarrow| \cdot |L|)$. By substituting $|\Rightarrow|$ for the size earlier identified, we see that the runtime complexity can be represented as $O(((|e| + |f| \cdot |\mathcal{U}|)^2 + 2^{2 \cdot |f| \cdot |\mathcal{U}|} \cdot |m|) \cdot |L|)$.

Combining the two parts of the algorithm, the complexity becomes

$$O(((|e| + |f| \cdot |\mathcal{U}|)^2 + 2^{2 \cdot |f| \cdot |\mathcal{U}|} \cdot |m|) \cdot |L| + 2^{2 \cdot |f| \cdot |u|} \cdot |m| \cdot 2^{|e|})$$

7 Implementation

We present the tool `mungoi`. `mungoi` is a prototype implementation of the work presented in this article. It is an implementation of the type system for Mungo, as well as an inference module. The syntax of the language is a slightly modified version of the calculus presented in this article, as opposed to the original Mungo tool that works on a subset of Java. In this section, we describe the tool and the most interesting technical details and discuss the implications of the implementation. The implementation can be found in a GitHub repository on `https://github.com/MungoTypesystem/Mungo-Typechecker`.

`mungoi` implements the static phases of a compiler for the language and does not contain any run-time system. The goal of the implementation is to serve as a proof-of-concept that this type system is feasible in practice, and not to provide a compiler for an actual programming language. As shown in [10] the type system is enough to provide guarantees about the run-time execution of the programs, therefore we do not consider run-time semantics in our implementation.

The type system in [10] support generic classes, while the language and type system presented in this article does not. The addition of generics does not make the inference problem more interesting, so for readability of the typing rules, we omit generic classes. The implementation of the type system does, however, support generics. This support of generic classes in the type system, due to the tight coupling of the type system and the inference module, carries over to the inference module, so we can infer usages for generic classes as well. Furthermore the language also supports an integer base type, as well as boolean expressions. Again the typing rules for these extensions does not make the type system more interesting, and has been omitted from our presentation of the type system.

The goal of `mungoi` is to verify statically that classes written in the Mungo language do not experience null-dereferences and protocol errors. Furthermore, the tool should automatically infer missing usages, wherever possible. The sequence of operations of `mungoi` is to parse the Mungo program, then infer the missing usages before type checking all classes. This is reflected in the three modules of `mungoi`, the parsing module, the inference module and the type checking module. We now describe each module in turn.

Parsing

The parser is implemented using parsec, a monadic parser combinator library [17]. Parsec provides many higher order functions to drive the parsing, so that is it only necessary to write code for parsing the smallest constituents of a program.

A Mungo program is a file containing a collection of enum declarations, followed by a number of class definitions. The structure of a class definition is illustrated below.

```
1  class C {
2      // Usage
3      {m; <l1 : end l2 : {n; X}>}[X = {o; end}]
4
5      // Fields
6      bool f1
7      OtherClass f2
8
9      // Methods
10     State m(void x) { ...; l1 }
11     void n(void x) { ... }
12     void o(void x) { ... }
```

```
13 }
```

The class contains the usage on the form $u[X_i = u_i]_{i \in I}$. The syntax is only slightly adapted from the normal $u^{\vec{E}}$ for writability.

Usage Inference

The usage inference module first identifies the dependencies between classes, as described in Definition 6. The program graph in Listing 2 is not acyclic, as illustrated in Figure 5, hence `mungoi` cannot infer the usages. It can become acyclic by specifying the usage for class C or B. If class C is specified manually, the inference order will be B before inferring A.

```
1 class A {infer[] B f1 }
2 class B {infer[] C f1 }
3 class C {infer[] B f2 }
```

Listing 2 Invalid program for inference

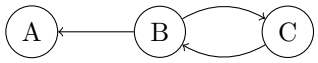


Figure 5 Cycle in inference graph for classes A, B and C

The inference algorithm follows the process defined in Section 4. We define the method availability transitions, the labelled transition systems between field typing environments. As earlier described, we collapse all terminated field typing environments under a single state $envT_{F_\perp}$ to reduce the size of the resulting usage. After this LTS has been created, the filtering step is performed by graph operations, where reachability is checked between states, and the states that are reachable from the initial field typing environment and can reach a terminated field typing environments are preserved in the filtered LTS.

The inferred usage is displayed to the user and used directly in the remaining inference and type checking.

Type Checking

Type checking is an essential part of `mungoi` in terms of ensuring correctness of the programs written in Mungo. The type system is split into two components, as indicated by the typing rules. One component type checks the individual expressions, while the other component implements the typestate checking with the rules presented earlier in Section 4. The expression checking component is used by the typestate checking component, but also in the inference module to extract the method availability transitions by use of the (Class) rule.

The typestate checking module takes a class C $\{\mathcal{U}, \vec{F}, \vec{M}\}$ and follows the transitions of usage $\mathcal{U}$ from the initial field typing environment. It checks that we can reach a terminated field typing environment and that for all transitions of usage $\mathcal{U}$, the class remains well-typed.

We describe the implementation of type checking with an example. Consider field `f` in Figure 6 with class `C2` and usage $X^{\{X=\{\texttt{m2;end}\ \texttt{m2};X\}\}}$. Let the class of method `m1` have the usage {`m1;end`}, the method body of `m1` assigns a new object to field `f` and calls method `m2` on `f`. As previously discussed, the type system is nondeterministic, and in this case calling `m2` can result in `f` being in two different states, illustrated in Figure 6b. The implementation represents non-determinism using a set of states. It is implemented using

the monad presented in Listing 3. The monad works by simulating procedural code, by having a state explicitly passed from one function to the next. States which experience errors are filtered by the Either monad, it represents errors as `Left` values and results as `Right` values.

```
1 data MState s a = MState { runState :: s -> Either String (a, s)}
2                 | MStateError String
```

Listing 3 Specialized state monad for handling non-determinism

Two instances of the monad are created, one to handle non-determinism and one for determinism, as shown in Listing 4. Non-determinism is handled by `NDTypeSystem` through having a list of states and return types from the latest method invocation. `DTypeSystem` is the deterministic counterpart, used for type-checking an expression with a state, which is $\mathsf{Configuration} \times \textbf{Expression} \to \mathcal{P}(\mathsf{Configuration})$ where the result is the set of states which can be reached. To combine the non-deterministic and deterministic monad, we created the function:

```
1 forAll :: DTypeSystem [(MyState, Type)] -> NDTypeSystem ()
```

The function, when given a deterministic type system that returns a list of states and types, gives us a non-deterministic type system, it works by running the deterministic type system on all current states then filters the states that experience errors.

```
1 type NDTypeSystem a = MState [(MyState, Type)] a
2 type DTypeSystem  a = MState (MyState, Type) a
```

Listing 4 The two uses of the `MState` data type

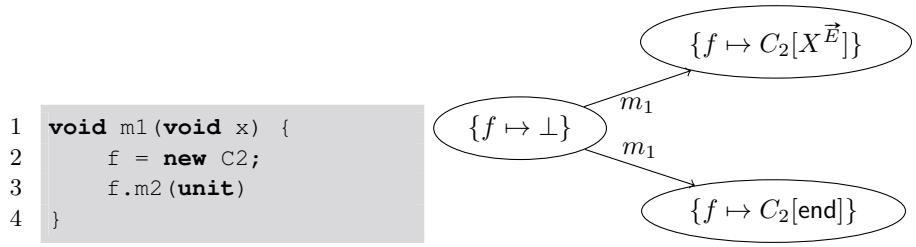
Finally, type-checking is done by starting at the terminated environment and checking that there exists a reachable state, which is terminated if none is found the program is rejected.

The type system has been used on several examples. In Section 5, we present a class which models a Customer, this example has been modified such that it switches on a enum instead of a boolean as that is a requirement for the type system and with the change, the example is accepted.

In the worst case, the type system is going to have an exponential runtime. However, in preliminary testing, the exponential runtime is hardly seen. The only way we have found to slow down the type-checker is by writing non-deterministic usages in which a branch is rejected late and thus the type-checker needs to check all possible states, hence it is exponential. This has only been observed in usages written such that they exploit this weakness of the implementation. The reason for this is that lazy evaluation in Haskell typically does not evaluate all possible states as it only needs one accepting path through the type system, which often happens to be one of the first paths it checks. We expect non-determinism in usages to be used sparsely, hence the amount of branches to check does not become too big to type check.

8 Method Signature Inference

A current limitation is related to usages of parameters, which we cannot infer at this stage. This makes it difficult to use inferred classes as parameters to methods since the usage cannot be specified. We propose an extension to Mungo to overcome this limitation. The idea is to extend the language with ad hoc polymorphism using method overloading, where multiple method signatures can share the same method name and implementation. Then



(a) Calling a method on a class **(b)** Possible states reachable by the call

Figure 6 Type checking a method call

the goal is to infer the possible method signatures so that it is not necessary to specify the usage for a class when used as a method parameter. This has not been integrated into the main body of this article, as it introduces changes to the type system, hence it is not directly applicable to previous results shown about the type system.

First, we update the syntax of method declarations, to allow multiple method signatures for the same body.

$$M ::= \overrightarrow{t \to t}\ m(x)\ \{e\}$$

We also extend the branch usage to be annotated with the method signature. The reason for this is because the parameter could be written to a field, hence based on the usage of the parameters different methods become available. Only with the annotation of a method signature, can we tell the initial type of the parameter, and track further updates to the type.

$$u ::= \{(m_{t \to t})_i; w_i\}_{i \in I} \mid X$$

The change to the type system is minor, as the usage of an object will specify the exact method signature to be used. The rule (TCBr) is updated to type check given the method signature by the usage. The rules (TCallF) and (TCallP) are updated to use the method signature according to the type of the actual parameter. The updated rules are shown in Table 9.

$$\text{(TCBr)}\quad \frac{\begin{array}{c} I \neq \emptyset \qquad \forall i \in I\ .\ \exists envT''_F\ . \qquad \overrightarrow{t'_p \to t'_r}\ m_i(x_i)\{e_i\} \in C.\mathsf{methods}_{\vec{D}} \\ \{\mathsf{this} \mapsto envT_F\}; \emptyset \cdot (\mathsf{this}, [x_i \mapsto (t_p)_i]) \vdash_{\vec{D}} e_i : (t_r)_i \triangleright \{\mathsf{this} \mapsto envT''_F\}; \emptyset \cdot (\mathsf{this}, [x_i \mapsto t'_i]) \\ \mathsf{terminated}(t'_i) \qquad ((t_p)_i \to (t_r)_i) \in \overrightarrow{t'_p \to t_r} \qquad \Theta; envT''_F \vdash_{\vec{D}} C[u_i^{\vec{E}}] \triangleright envT'_F \end{array}}{\Theta; envT_F \vdash_{\vec{D}} C[\{(m_{t_p \to t_r})_i; u_i\}_{i \in I}^{\vec{E}}] \triangleright envT'_F}$$

$$\text{(TCallF)}\quad \frac{\begin{array}{c} \Lambda; \Delta \cdot (o, S) \vdash e : t \triangleright \Lambda'\{o.f \mapsto C[\mathcal{U}]\}; \Delta' \cdot (o, S') \\ \overrightarrow{t_p \to t_r}\ m(x)\{e'\} \in C.\mathsf{methods}_{\vec{D}} \qquad (t \to t') \in \overrightarrow{t_p \to t_r} \qquad \mathcal{U} \xrightarrow{m_{t \to t'}} \mathcal{W} \end{array}}{\Lambda; \Delta \cdot (o, S) \vdash f.m(e) : t' \triangleright \Lambda'\{o.f \mapsto C[\mathcal{W}]\}; \Delta' \cdot (o, S')}$$

$$\text{(TCallP)}\quad \frac{\begin{array}{c} \Lambda; \Delta \cdot (o, S) \vdash_{\vec{D}} e : t \triangleright \Lambda'; \Delta' \cdot (o, [x \mapsto C[\mathcal{U}]]) \\ \overrightarrow{t_p \to t_r}\ m(x)\{e'\} \in C.\mathsf{methods}_{\vec{D}} \qquad (t \to t') \in \overrightarrow{t_p \to t_r} \qquad \mathcal{U} \xrightarrow{m_{t \to t'}} \mathcal{W} \end{array}}{\Lambda; \Delta \cdot (o, S) \vdash_{\vec{D}} x.m(e) : t' \triangleright \Lambda'; \Delta' \cdot (o, [x \mapsto C[\mathcal{W}]])}$$

Table 9 Updated rules for method overloading

Consider a method m with method body e and a parameter of class C. The available method signatures are the ones where the parameter has type $C[\mathcal{U}]$ where $\mathcal{U}$ is reachable

from the initial state, and the method body does not experience errors. We can formally define this, and state that the available method signatures $\overrightarrow{t_p \to t_r}$ are defined as:

$$\left\{ (C[\mathcal{U}] \to t_r) \;\middle|\; \begin{array}{l} \exists \Lambda.\Lambda; \emptyset \cdot (\mathsf{this}, [x \mapsto C[\mathcal{U}]]) \vdash_{\overrightarrow{D}} e : t_r \triangleright \Lambda'; \emptyset \cdot (\mathsf{this}, [x \mapsto t]) \\ C.\mathsf{usage} \to^* \mathcal{U} \\ \mathsf{terminated}(t) \end{array} \right\}$$

With this extension, we can allow the programmer to omit writing the specific usage for method parameters of class types, hence it is possible to use classes with inferred usages as method parameters, even if the inferred usage is unknown to the programmer when writing the program.

9 Related Work

In this section, we describe relevant work for both behavioural types and usage inference. Behavioural types have over the years been studied in different contexts and using various approaches [13]. We focus on two forms of behaviour types; namely, typestates and session types since these two approaches most closely resemble the work of this article.

9.1 Behavioural Types

In [9] Gay et al. extend session types to object oriented programming by combining session types and typestates. Here they specify a single typestate abstracting over a session type, for each classes, instead of annotating each method with pre and postconditions.

In [15] Kouzapas et al. use a similar approach but for typestate definitions to avoid null-dereferencing errors. Furthermore, they improve the implementation by inferring the sequence of method calls to remove the need for type declarations on parameters and return types. Typestate declarations on classes are still necessary since they check that the inferred sequence of method calls are a subtype of the declared typestate. Finally in [10], Golovanov et al. presents a type system for Mungo where the type checking of typestates is performed entirely by the type system without inference. This is an extension to the work by Bravetti et al. in [5]. The work in this article improve on this approach by allowing optional typestate declarations and typestates are inferred for classes without one. Our approach to inference is different from the one by Kouzapas et al. since we infer the principal usage and do not require a class explicitly to have a declared a usage.

9.2 Usage Inference

There is currently not a lot of work available for usages, however, the problem of inferring usages is not entirely new either. A similar problem appears in the context of session types. The main difference between inference in the context of session types and usages is that for session types we infer the order of channel messages to check that the order is correct between two or more endpoints. In the context of usage types, we infer the order of operations to ensure that we avoid null-dereferencing similar to pre and postconditions from typestates [21].

In [6] Collingbourne & Kelly present an algorithm to infer session types from program control flow. In this work, the authors start by simplifying the program text such as loops and conditional statements. Then they build a graph from the communication statements, where equal nodes are unified. Afterwards the graph is converted into a DFA. Our inference

algorithm has a resemblance to this work since their communication statements are operations on session types and in our case method calls are operations on usages, in both cases a graph is built from the state transitions. The inferred type in their case is checked against a specified or likewise inferred dual session type to ensure the correctness of an implementation with respect to a communication protocol. A limitation of their approach is that they have to infer the type of each session instance. This would be a problem in our setting since we can have several instances of classes that are used distinctly and an inferred usage should cover all cases. This is why we infer a principal usage instead of inferring a usage for each instantiation of a class.

The most recent work is [19] where Padovani presents an inference algorithm for context-free session types is presented. They come across unique challenges in terms of sequential composition for session types and polymorphic recursion which they overcome by introducing a higher-order combinator. Their type system allows context-free session types to be checked and inferred using the type checker and parametric polymorphism implementation of the host programming language. Since we use Haskell we could also have exploited its parametric polymorphism to infer usages. The main disadvantage of using the type system of Haskell as part of the algorithm is that it becomes less general, in the sense that the implementation is tied to languages whose type system support parametric polymorphism. In contrast, our inference algorithm can be applied in other implementations of Mungo; in particular, the Mungo toolchain [15].

10 Discussion and Future Work

The main contribution of this paper is a usage inference algorithm for Mungo that extends the applicability of usages in practical settings. Usage inference ensures that programs do not produce null-dereferencing errors and allows programmers to express tighter protocol control over class methods when needed, hence we decrease the overhead and increase the maintainability of working with usages by not requiring each usage to be written by hand.

We have shown that any inferred usage by our algorithm is the most general usage that avoids null-dereferencing and protocol errors by proving that the inferred usage can simulate every usage that can well-type a class. Furthermore, we have shown that the inferred usage is itself guaranteed to be well-typed in the Mungo type system. This would not have been the case if we employed the approaches of previous work in this area since they could not guarantee protocol completion. We achieve this by considering the state of fields in a class as a state in a method availability transition system where a class method is a transition if given a field state the method is well-typed. From this transition system, we remove all non-terminating paths and include an end state resulting in a usage graph. Finally, an actual usage is constructed from this usage graph. We have demonstrated our approach with an example and described our Haskell implementation. By implementing a prototype version of the inference algorithm, we have illustrated that an implementation is possible, and that that it could be added to the Mungo and St. Mungo toolchain, making the toolchain more practical to use.

The usage inference algorithm has three current limitations. One is complexity since we both have non-determinism and usages can themselves become exponential in size, but we cannot necessarily improve on this limitation since the nature of principal usages results in exponential worst-case size and non-determinism is needed to handle protocol completion. The second limitation is that it is difficult to use classes with inferred usages as method parameters, as the usage cannot be easily specified. We have proposed a solution for this,

as an extension to the language in Section 8. The final limitation is the linear typing discipline since it restricts the use of Mungo in general-purpose programming. It also hinders the applications of usage inference as a safety checker of open source modules which is a significant motivating factor for implementing usage inference. As future work, we intend to implement the parameter usage inference extension and prove the necessary properties. Another line of future work should be on loosening the restrictions of linear typing either by adopting access-permissions similar to [3] or by finding a novel approach.

Finally, given that our inference approach uses type rules for expressions to extract method availability transitions, we can exchange the type rules with a different set and still infer usages. This allows our usage inference to be applied in contexts outside our `mungoi` tool. For example, we could use the type rules for expressions from the original Mungo paper [15] and our parameter extension to, in theory, enable usage inference in their Mungo tool.

11 Acknowledgements

The authors wish to thank Simon Gay and Ornela Dardha for the hospitality extended during a visit to Glasgow as part of this project. The authors are also thankful to the insights and discussions about the Mungo project, and proof techniques that Simon Gay and Ornela Dardha provided.

References

1 Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1639950.1640073`, `doi:10.1145/1639950.1640073`.

2 Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pages 1015–1022, 10 2009. `doi:10.1145/1639950.1640073`.

3 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. *SIGPLAN Not.*, 42(10):301320, October 2007. URL: `https://doi.org/10.1145/1297105.1297050`, `doi:10.1145/1297105.1297050`.

4 Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical api protocol checking with access permissions. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 195–219, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

5 Mario Bravetti, Adrian Francalanza, Hans Hüttel, and António Ravara. A type system for mungo, 2019. Unpublished worksheet.

6 Peter Collingbourne and Paul Kelly. Inference of session types from control flow. *Electr. Notes Theor. Comput. Sci.*, 238:15–40, 06 2010. `doi:10.1016/j.entcs.2010.06.003`.

7 Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. *SIGPLAN Not.*, 36(5):59–69, May 2001. URL: `http://doi.acm.org/10.1145/381694.378811`, `doi:10.1145/381694.378811`.

8 Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, pages 465–490, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

9 Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. *SIGPLAN Not.*, 45(1):299–312, January 2010. URL: `http://doi.acm.org/10.1145/1707801.1706335`, `doi:10.1145/1707801.1706335`.

10 Iaroslav Golovanov, Mathias Steen Jakobsen, and Mikkel Klinke Kettunen. Soundness of the Mungo Type System with Generics, 2019. URL: https://projekter.aau.dk/projekter/da/studentthesis/soundness-of-the-mungo-type-system-with-generics(5516badb-7034-439b-8359-9abca0910ac6).html.

11 Roberto Gorrieri. *Labeled Transition Systems*, pages 15–34. Springer International Publishing, Cham, 2017. URL: https://doi.org/10.1007/978-3-319-55559-1_2, doi:10.1007/978-3-319-55559-1_2.

12 Eva Fajstrup Graversen, Jacob Buchreitz Harbo, Hans Hüttel, Mathias Ormstrup Bjerregaard, Niels Sonnich Poulsen, and Sebastian Wahl. Type inference for session types in the pi-calculus. In *Web Services, Formal Methods, and Behavioral Types*, pages 103–121, Cham, 2016. Springer International Publishing.

13 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, April 2016. URL: http://doi.acm.org/10.1145/2873052, doi:10.1145/2873052.

14 Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session type inference in haskell. In *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010.*, pages 74–91, 2010. URL: https://doi.org/10.4204/EPTCS.69.6, doi:10.4204/EPTCS.69.6.

15 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, pages 146–159, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2967973.2968595, doi:10.1145/2967973.2968595.

16 Dimitris Kouzapas, Simon Gay, and António Ravara. Behavioural type-inference for object-oriented programs, 2019. Unpublished worksheet.

17 Daan Leijen, Paolo Martini, and Antoine Latter. parsec: Monadic parser combinators, 2019. URL: https://hackage.haskell.org/package/parsec.

18 Leonardo Gaetano Mezzina. How to infer finite session types in a calculus of services and sessions. In Doug Lea and Gianluigi Zavattaro, editors, *Coordination Models and Languages*, pages 216–231, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

19 Luca Padovani. Context-free session type inference. *ACM Trans. Program. Lang. Syst.*, 41(2):9:1–9:37, March 2019. URL: http://doi.acm.org/10.1145/3229062, doi:10.1145/3229062.

20 David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, pages 167–183, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

21 Robert Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12:157–171, 01 1986. doi:10.1109/TSE.1986.6312929.

22 Álvaro Tasistro, Ernesto Copello, and Nora Szasz. Principal type scheme for session types. *International Journal of Logic and Computation*, 3:34–43, 12 2012.

A Type System

In the following section, we include the typing environments and other definitions from [10].

To ensure protocol completion, we introduce the concept of *linear types*. We say that a type is linear if it is a class type, with a usage different from end. In other words, a type is linear if its protocol is not finished.

▶ **Definition 13** (Linear type).

$$\mathsf{lin}(t) \stackrel{\text{def}}{=} \exists C, \mathcal{W} \;.\; t = C[\mathcal{W}] \wedge \mathcal{W} \neq \mathsf{end}.$$

We also introduce *field typing environments* to map the fields of a class to the current type state. We then extend the concept of linear types to field typing environments, and say that a field typing environment is *terminated* if all fields in the environment are non-linear.

▶ **Definition 14** (Field type environment). *The field type environment is used to store the type information for fields, and is a partial function*

$$envT_F : \boldsymbol{FNames} \rightharpoonup (\boldsymbol{Typestates}_\perp \cup \boldsymbol{BTypes}).$$

Table 10 describes the rules for typing programs. The rule (TProg) say that a program is well-typed if all its declarations are well-typed. (TClass) type checks a class declaration and describes that a class is well-typed if its usage is well-typed and the associated field typing environment is terminated. Requiring that the final environment is terminated, ensures protocol completion for all fields, since all fields of class-types, must be non-linear.

$$(\text{TProg}) \quad \frac{\forall D \in \vec{D} \,.\, \vdash_{\vec{D}} D}{\vdash D}$$

$$(\text{TClass}) \quad \frac{\emptyset; \vec{F}.\mathsf{inittypes}_{\vec{D}} \vdash_{\vec{D}} C[\mathcal{U}] \triangleright envT_F \quad \mathsf{terminated}(envT_F)}{\vdash_{\vec{D}} \mathsf{class}\ C\{\mathcal{U}, \vec{F}, \vec{M}\}}$$

Table 10 Typing program and class definitions

In order to type check a class we follow its usage transitions and the type rules describing how to type these transitions are defined in Table 11. The central rule is (TCBr) it tells us that a branch usage is well-typed if the initial method of each branch is declared in the class and its method body is well-typed. Furthermore, the remaining usage after the method call is also well-typed in the updated field typing environment. The (TCCh) rule says that a choice usage is well-typed if the usage associated with each label is well-typed and the resulting environments are the same.

To type check recursive usages, we use an environment to keep track of field typing environments, as defined below.

▶ **Definition 15** (Usage variable environment). *The environment Θ is an environment used to keep track of recursive usage variables, and is a partial function*

$$\Theta : \boldsymbol{UVars} \rightharpoonup \boldsymbol{EnvT_F}$$

The rules (TCVar) and (TCRec) handle recursive usages. (TCRec) handles the case where a recursive variable is encountered for the first time, it maps the variable to a field typing environment and substitutes the variable for the associated usage and it is well-typed if the class with the substituted usage is well-typed. Finally, (TCVar) handles the case where a recursive variable, that we have already handled with (TCRec), appears again. In that case (TCVar) tells us such a usage is well-typed with an arbitrary resulting field typing environment, since the resulting environment depends on rule (TCRec).

The (TCBr) rule checks the method bodies, in accordance with the current field typing environment. We define typing judgments for expressions of the form $\Lambda; \Delta \vdash^{\Omega} e : t \triangleright \Lambda'; \Delta'$.

The first environment Λ denotes a *object field environment* as defined below.

$$\text{(TCBr)}\quad \frac{\begin{array}{c} I \neq \emptyset \qquad \forall i \in I \;.\; \exists envT_F'' \;. \\ \{\mathsf{this} \mapsto envT_F\}; \emptyset \cdot (\mathsf{this}, [x_i \mapsto t_i']) \vdash_{\vec{D}} e_i : t_i \triangleright \{\mathsf{this} \mapsto envT_F''\}; \emptyset \cdot (\mathsf{this}, [x_i \mapsto t_i'']) \\ \mathsf{terminated}(t_i'') \qquad t_i\ m_i(t_i'\ x_i)\{e_i\} \in C.\mathsf{methods}_{\vec{D}} \qquad \Theta; envT_F'' \vdash_{\vec{D}} C[u_i^{\vec{E}}] \triangleright envT_F' \end{array}}{\Theta; envT_F \vdash_{\vec{D}} C[\{m_i; u_i\}_{i \in I}^{\vec{E}}] \triangleright envT_F'}$$

$$\text{(TCCh)}\quad \frac{\forall l_i \in L \;.\; \Theta; envT_F \vdash_{\vec{D}} C[u_i^{\vec{E}}] \triangleright envT_F'}{\Theta; envT_F \vdash_{\vec{D}} C[\langle l_i : u_i \rangle_{l_i \in L}^{\vec{E}}] \triangleright envT_F'}$$

$$\text{(TCEn)}\quad \Theta; envT_F \vdash_{\vec{D}} C[\mathsf{end}^{\vec{E}}] \triangleright envT_F$$

$$\text{(TCVar)}\quad (\Theta, [X \mapsto envT_F]); envT_F \vdash_{\vec{D}} C[X^{\vec{E}}] \triangleright envT_F'$$

$$\text{(TCRec)}\quad \frac{(\Theta, [X \mapsto envT_F]); envT_F \vdash_{\vec{D}} C[u^{\vec{E}}]_{\vec{D}} \triangleright envT_F'}{\Theta; envT_F \vdash_{\vec{D}} C[X^{\vec{E} \uplus \{X = u\}}] \triangleright envT_F'}$$

Table 11 Typing class usage definitions

▶ **Definition 16** (Object field environment). Λ *is an object field typing environment used to store pairs of class names and field type environments, and is a partial function*

$$\Lambda : \boldsymbol{ONames} \rightharpoonup (\boldsymbol{CNames} \times \boldsymbol{Types}) \times \boldsymbol{EnvT_F}$$

It is an approximation of the run-time heap, and maps objects to field typing environment. When type checking classes, only a single object, this, is present in the environment.

The second environment, $\Delta = envT_O \cdot envT_S$, encapsulates an *object type environment* used as environment to temporary store object types, and an *parameter stack type environment* which is an approximation of the run-time stack.

▶ **Definition 17** (Object type environment). $envT_O$ *is an object type environment used to store the type of objects, and is a partial function*

$$envT_O : \boldsymbol{ONames} \rightharpoonup \boldsymbol{Typestates}$$

▶ **Definition 18** (Parameter stack type environment). $envT_S$ *is a parameter type environment used to store a sequence of pairs* (o, S) *where* o *is an objects and* S *is a parameter binding* $[x \mapsto t]$

$$envT_S : \overrightarrow{\boldsymbol{ONames} \times (\boldsymbol{PNames} \times (\boldsymbol{Typestates}_\perp \cup \boldsymbol{BTypes}))}$$

The judgement is read as evaluating an expression e in a heap approximated by Λ and a stack approximated by Δ results in updated environments Λ' and Δ'. The environment Ω is used for typing labelled expressions, and will be omitted in all rules except (Lab) and (Con).

▶ **Definition 19** (Label environment). Ω *is a label environment used to map a label to a pair of environments* (Λ, Δ)

$$\Omega : \boldsymbol{Labels} \rightharpoonup \Lambda \times \Delta$$

Table 12 shows the typing rules for values, which are all simple as they do not change any fields. The rule (TObj) tells us that typing an object removes it from the environment which is done to enforce linearity.

$$\text{(TLit)}\quad \frac{l \in L}{\Lambda;\Delta \vdash_{\vec{D}} l : L \triangleright \Lambda;\Delta} \qquad \text{(TVoid)}\quad \Lambda;\Delta \vdash_{\vec{D}} \mathsf{unit} : \mathsf{void} \triangleright \Lambda;\Delta$$

$$\text{(TNew)}\quad \Lambda;\Delta \vdash_{\vec{D}} \mathsf{new}\ C : C[C.\mathsf{usage}] \triangleright \Lambda;\Delta \qquad \text{(TBot)}\quad \Lambda;\Delta \vdash_{\vec{D}} \mathsf{null} : \bot \triangleright \Lambda;\Delta$$

$$\text{(TObj)}\quad \frac{envT_O = envT'_O, o \mapsto t}{\Lambda; envT_O \cdot envT_S \vdash_{\vec{D}} o : t \triangleright \Lambda; envT'_O \cdot envT_S} \qquad \text{(TBool)}\quad \frac{v \in \{\mathsf{true}, \mathsf{false}\}}{\Lambda;\Delta \vdash_{\vec{D}} v : \mathsf{Bool} \triangleright \Lambda;\Delta}$$

Table 12 Typing rules for values

Table 13 shows the rules that interact with the field typing environment. The rules (TLinPar) and (TLinFld) handle dereferencing linear fields, in both rules the bindings are updated to $\bot$ after reading, so that we ensure that only one reference to the object is present. Rule (TFld) tell us that assignment to a field requires that the field is unrestricted. Here we also use the helper function agree to ensure that the type of the value is compatible with the type of the field.

▶ **Definition 20** (Agree predicate).

$$\mathsf{agree}(b, b) \qquad \mathsf{agree}(C, C[\mathcal{W}])$$

Finally, rule (TCallF) and (TCallP) say that method calls on fields or object references are well-typed if the associated usage allows the method call and the usage in the environment is updated with the usage following the transition.

Lastly Table 14 shows the typing rules for the composite expressions, which are all similar to those found in [10]. As in the previous work, we require that resulting environments of if-cases and switch-expressions all match.

B Extracting Method Availability Transitions

Algorithm 2 shows a naive implementation of method availability transition system. While the worst-case complexity would remain the same, a better implementation would be to only consider the reachable states from $envT_{F_\bot}$, since all other states are filtered out in the construction of the usage graph.

Algorithm 2 Extracting the method availability transitions

```
1: function MthdAvailTrans(C)
2:     → ← ∅
3:     A ← {m | (t m(t' x){e}) ∈ C.methods}
4:     for all envT_F ∈ EnvT_F do
5:         for all envT'_F ∈ EnvT_F do
6:             for all (t m(t' x){e}) ∈ C.methods do
7:                 if {this ↦ envT_F}; ∅·(this, [x ↦ t']) ⊢_D e : t▷{this ↦ envT'_F}; ∅·(this, [x ↦ t'']) ∧ ¬lin(t'') then
8:                     → ← → ∪{(envT_F, envT'_F, m)}
       return (EnvT_F, A, →, envT_{F_⊥})
```

$$(\textsc{TLinPar})\quad \frac{\mathsf{lin}(t)}{\Lambda;\Delta\cdot(o,[x\mapsto t])\vdash_{\vec{D}} x : t \triangleright \Lambda;\Delta\cdot(o,[x\mapsto \bot])}$$

$$(\textsc{TNoLPar})\quad \frac{\neg\mathsf{lin}(t)}{\Lambda;\Delta\cdot(o,[x\mapsto t])\vdash_{\vec{D}} x : t \triangleright \Lambda;\Delta\cdot(o,[x\mapsto t])}$$

$$(\textsc{TLinFld})\quad \frac{t=\Lambda(o).f \qquad \mathsf{lin}(t)}{\Lambda;\Delta\cdot(o,S)\vdash_{\vec{D}} f : t \triangleright \Lambda\{o.f\mapsto \bot\};\Delta\cdot(o,S)}$$

$$(\textsc{TNoLFld})\quad \frac{\neg\mathsf{lin}(t)}{\Lambda\{o.f\mapsto t\};\Delta\cdot(o,S)\vdash_{\vec{D}} f : t \triangleright \Lambda\{o.f\mapsto t\};\Delta\cdot(o,S)}$$

$$(\textsc{TFld})\quad \frac{\begin{array}{c} C=\Lambda(o).\mathsf{class} \qquad \mathsf{agree}(C.\mathsf{fields}_{\vec{D}}(f),t') \\ \Lambda;\Delta\cdot(o,S)\vdash e : t' \triangleright \Lambda', o.f\mapsto t;\Delta'\cdot(o,S') \qquad \neg\mathsf{lin}(t)\end{array}}{\Lambda;\Delta\cdot(o,S)\vdash_{\vec{D}} f = e : \mathsf{void} \triangleright \Lambda'\{o.f\mapsto t'\};\Delta'\cdot(o,S')}$$

$$(\textsc{TCallF})\quad \frac{\begin{array}{c}\Lambda;\Delta\cdot(o,S)\vdash e : t \triangleright \Lambda'\{o.f\mapsto C[\mathcal{U}]\};\Delta'\cdot(o,S') \\ t'\ m(t\ x)\{e'\}\in C.\mathsf{methods}_{\vec{D}} \qquad \mathcal{U}\xrightarrow{m}\mathcal{W}\end{array}}{\Lambda;\Delta\cdot(o,S)\vdash f.m(e) : t' \triangleright \Lambda'\{o.f\mapsto C[\mathcal{W}]\};\Delta'\cdot(o,S')}$$

$$(\textsc{TCallP})\quad \frac{\begin{array}{c}\Lambda;\Delta\cdot(o,S)\vdash_{\vec{D}} e : t \triangleright \Lambda';\Delta'\cdot(o,[x\mapsto C[\mathcal{U}]]) \\ t'\ m(t\ x)\{e'\}\in C.\mathsf{methods}_{\vec{D}} \qquad \mathcal{U}\xrightarrow{m}\mathcal{W}\end{array}}{\Lambda;\Delta\cdot(o,S)\vdash_{\vec{D}} x.m(e) : t' \triangleright \Lambda';\Delta'\cdot(o,[x\mapsto C[\mathcal{W}]])}$$

Table 13 Typing expressions

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

$$(\text{TSeq})\quad \frac{\Lambda;\Delta \vdash_{\vec{D}} e : t \triangleright \Lambda'';\Delta'' \qquad \neg\mathsf{lin}(t) \qquad \Lambda'';\Delta'' \vdash_{\vec{D}} e' : t' \triangleright \Lambda';\Delta'}{\Lambda;\Delta \vdash_{\vec{D}} e; e' : t' \triangleright \Lambda';\Delta'}$$

$$(\text{TIf})\quad \frac{\begin{array}{c}\Lambda;\Delta \vdash_{\vec{D}} e : \mathsf{Bool} \triangleright \Lambda'';\Delta'' \\ \Lambda'';\Delta'' \vdash_{\vec{D}} e' : t \triangleright \Lambda';\Delta' \qquad \Lambda'';\Delta'' \vdash_{\vec{D}} e'' : t \triangleright \Lambda';\Delta'\end{array}}{\Lambda;\Delta \vdash_{\vec{D}} \mathsf{if}\ (e)\ \{e'\}\ \mathsf{else}\ \{e''\} : t \triangleright \Lambda';\Delta'}$$

$$(\text{TSwP})\quad \frac{\begin{array}{c}\Lambda;\Delta \cdot (o,S) \vdash_{\vec{D}} e : L \triangleright \Lambda'';\Delta'' \cdot (o, [x \mapsto C[(\langle l_i : u_i\rangle_{l_i \in L})^{\vec{E}}]]) \\ \forall l_i \in L.\ \Lambda'';\Delta'' \cdot (o, [x \mapsto C[u_i^{\vec{E}}]]) \vdash_{\vec{D}} u_i : t \triangleright \Lambda';\Delta' \cdot (o,S')\end{array}}{\Lambda\Delta \cdot (o,S) \vdash_{\vec{D}} \mathsf{switch}_{x.m}\ (e)\{l_i : u_i\}_{l_i \in L} : t \triangleright \Lambda';\Delta' \cdot (o,S')}$$

$$(\text{TSwF})\quad \frac{\begin{array}{c}\Lambda;\Delta \cdot (o,S) \vdash_{\vec{D}} e : L \triangleright \Lambda'', o.f \mapsto C[(\langle l_i : u_i\rangle_{l_i \in L})^{\vec{E}}]];\Delta'' \\ \forall l_i \in L.\ \Lambda'', o.f \mapsto C[u_i^{\vec{E}}];\Delta'' \vdash_{\vec{D}} e_i : t \triangleright \Lambda';\Delta' \cdot (o,S')\end{array}}{\Lambda;\Delta \cdot (o,S) \vdash_{\vec{D}} \mathsf{switch}_{f.m}\ (e)\{l_i : e_i\}_{l_i \in L} : t \triangleright \Lambda';\Delta' \cdot (o,S')}$$

$$(\text{TLab})\quad \frac{\Omega' = \Omega, k : (\Lambda,\Delta) \qquad \Lambda;\Delta \vdash^{\Omega'}_{\vec{D}} e : \mathsf{void} \triangleright \Lambda;\Delta}{\Lambda;\Delta \vdash^{\Omega}_{\vec{D}} k : e : \mathsf{void} \triangleright \Lambda;\Delta}$$

$$(\text{TCon})\quad \frac{\Omega' = \Omega, k : (\Lambda,\Delta)}{\Lambda;\Delta \vdash^{\Omega'}_{\vec{D}} \mathsf{continue}\ k : \mathsf{void} \triangleright \Lambda;\Delta}\text{-}$$

Table 14 Typing rules for composite expressions