AALBORG UNIVERSITY
STUDENT REPORT

Behavioural Separation with Parallel Usages for a Core Object-Oriented Language

Aalborg University

Department of Computer Science

Master Thesis

Iaroslav Golovanov Mathias Steen Jakobsen

Mikkel Klinke Kettunen

supervised by
Hans Hüttel

June 2020

AALBORG UNIVERSITY

STUDENT REPORT

Department of Computer Science
Aalborg University
https://www.cs.aau.dk/

Title:
Behavioural Separation with Parallel Usages for a Core Object-Oriented Language

Theme:
Master Thesis

Project Period:
Spring 2020

Group:
PT103F20

Participants:
Iaroslav Golovanov
Mathias Steen Jakobsen
Mikkel Klinke Kettunen

Supervisor:
Hans Hüttel

Pages:
64

Date:
June 2020

Abstract:

This report extends work on the Mungo language with behavioural separation. Mungo is an object-oriented calculus that employs typestates with a behavioural type system to ensure the absence of null-dereferencing. The typestates are expressed with usages, which are protocols that specify the admissible sequences of method calls on objects. Previous type systems for Mungo have all had a linearity constraint on objects. We lessen this constraint by extending the usage specifications with a parallel usage construct, where the parallel constituents describe separate local behaviour. We use the parallel usage to reason about aliasing, as a parallel usage describes a separation of the heap where we can reason about each constituent in isolation. Furthermore, parallel usages allow for arbitrary interleaving of local protocols, solving a problem of exponential growth of the size of usages, for unrelated linear fields in classes. We show that the new type system, with support for behavioural separation, retains the properties of previously presented type systems for Mungo, namely the safety and progress results, as well as protocol fidelity. Finally, we present an implementation of the Mungo language with support for parallel usages, along with a suite of examples.

Preface

Acknowledgements: We would like to thank our supervisor Hans Hüttel for his valuable insight and input, not only for this thesis, but during all of our work within this topic. He has gone beyond our expectations for a supervisor, and far beyond what is required. For that we are grateful.

Iaroslav, Mikkel, and Mathias
Aalborg, June 15th, 2020

Summary

In this thesis, we present work on a behavioural type system for the object-oriented calculus Mungo, which was originally presented by Kouzapas et. al [KDPG16] and has been further explored by Bravetti et. al [BFG+20] and Golovanov et. al [GJK19, GJK20]. Based on the concept of typestates introduced by Strom & Yemini [SY86], object types in Mungo are composed of a class name and a *usage* describing the admissible sequences of method invocations on the object. We analyse different approaches to typestates in object-oriented programming languages and compare the approaches of Mungo, Plaid [BBA09] and Fugue [DF04].

A weakness of current type systems for Mungo, compared to the other approaches, is its inability to concisely express unrelated operations in usages, leading to exponential size protocol specifications. Furthermore, a shared weakness between all of the analysed approaches, and a common weakness amongst many other behavioural type systems is the need for a *linear* typing discipline, where aliasing is disallowed. Reasoning about aliased objects is difficult in a behavioural type system, as types evolve during their lifetime, and changes in one alias can affect the others. Therefore, a common solution is to disallow aliasing completely and instead apply a linear type system, where only a single reference to an object is allowed at a given time. This, however, is in contrast to common practice in object-oriented programming, where aliasing is commonplace. List iteration, linked lists, and shared resources are just some examples of where aliasing occurs in practice.

We analyse multiple approaches for lessening this linearity constraint. One approach is particularly interesting, namely that of behavioural separation as described by Caires & Seco [CS13]. Based on separation logic, behavioural separation seeks to reason about *interference* in a behavioural type system, where aliasing is one type of interference. They do so by allowing the memory to be partitioned according to the types in the program, to reason locally about changes in the memory, rather than requiring a global view of the program. One of the constructs they use is a parallel construct describing how a value can be split into two constituents and used separately.

We extend the usages of Mungo with a parallel construct $(u_1 \mid u_2).u_3$ describing how an object can be split into two aliases, each with a local protocol of u_1 and u_2 respectively, and after completing the local protocols, it can be merged back into a single reference with protocol u_3. We draw upon behavioural separation to reason about multiple references to the same object and allow a limited form of aliasing in method calls, where the same objects can be referenced by multiple parameters. Furthermore, even without aliasing, the introduction of parallel usages solves the earlier mentioned issue of unrelated operations, as the local protocols u_1 and u_2 can be followed in any order, hence a parallel usage concisely describes that u_1 and u_2 are unrelated, without enumerating all interleavings of following the two protocols.

Two run-time semantics were created, a big-step semantics and a small-step semantics, to be able to naturally express and show results for the type system. We show that for terminating programs, the two semantics are equivalent, and as such the properties shown for these programs

can be applied to either of the two semantics. Furthermore, we present a type system and show that properties shown for Mungo in previous work, also hold for this new type system. These properties include protocol fidelity, safety and progress. A consequence of the progress result is that null-dereferencing cannot occur at runtime. To further illustrate the strengths of the type system, an implementation of the Mungo language is presented, and a suite of program examples is available at `https://mungotypesystem.github.io/MungoBehaviouralSeparation`. These examples show how parallel usages can express real-life protocols, and how the type system helps ensure that the protocols are followed.

Finally, we conclude that the extension of the Mungo language solves many of the issues discovered with Mungo, while preserving the guarantees and strengths of the Mungo approach to type-state declarations.

Contents

Chapter 1

Introduction

As software systems become increasingly complex, reasoning about the behaviour of the systems becomes harder. Type systems ensure that operations are only allowed for values of the correct types, and strongly typed languages even provide compile-time guarantees that type-related runtime-errors cannot occur. Common for most of these type systems is that they only capture the static type information, and does not capture the dynamic nature of values in a program. This is particularly noticeable in object-oriented programming languages. Objects can encode complex structures, and will often require that methods are called in a specific order. For example, in objects of a class modelling a database, it is important that the connection is initialised, before attempting to extract data from the database.

Despite the nature of these objects, the type systems for the most widely used object-oriented languages such as Java or C# does not allow the programmers to express the dynamic properties of a class, and programmers must instead resolve to informal specifications, not verified by the type systems.

Type systems for dynamic properties are not a new phenomenon. Typestates were originally a concept developed by Strom and Yemini [SY86], as an extension to the concept of a type. The static type information such as *int*, *string* or *float* is now extended with a state, describing the current *protocol* of a value. Through *typestate tracking* a type system can statically ensure that the dynamic properties expressed in the protocol are followed at run-time.

In this thesis, we work with the object-oriented language Mungo presented by Kouzapas et. al [KDPG16]. In particular, this thesis is a continuation of the line of research started by Bravetti et. al [BFHR19, BFG$^+$20] and continued by Golovanov et al [GJK19, GJK20]. In Mungo, a class is annotated with a *usage* $\mathcal{U}$, describing the permitted method sequences for an object of that class. The type system then ensures the following: (i) the defined usage for a class is followed by all instances, (ii) all protocols will be terminated, and (iii) following the usage will not result in null-dereferencing. This is achieved with typestate tracking, where objects in the language have types $C[\mathcal{U}]$ and where the usage $\mathcal{U}$ is updated throughout the lifetime of an object and specifies the currently available operations.

A common problem with behavioural type systems is the need for *linearity*. As types of objects in Java are immutable, objects can be freely aliased, as operations on one alias do not impact the type of the other aliases. This is not the case when working with behavioural types, as types can now change as operations are performed. So to ensure that the types of all aliases are consistent, a *linear typing discipline* is often employed, where only a single reference to an object is allowed. This has been the approach of previous type systems for Mungo. In the following sections, we explore ways to lessen this constraint.

1.1 Typestates

In 1986, Strom & Yemini introduced the concept of typestates where they use Hoare triplets to define the operations that are available to variables based on its state [SY86]. Each variable is assigned a type which contains a partially ordered set of typestates that define sequences of allowed operations on that type. For example, we can specify that a variable must have a value assigned to it before it can be used in an operation or that a file must be opened before being read etc. Originally, the work of Strom & Yemini was established for a procedural language but since then, several approaches have been proposed for object-oriented languages [BBA09, SNS+11, CV11, KDPG16, BFG+20, DF04]. In this section, we focus on Fugue [DF04], Plaid [SNS+11], and Mungo [KDPG16, BFG+20] since they represent three distinct approaches to typestates, namely, pre and post conditions, splitting objects into hierarchies of states, and globally specifying the allowed sequences of method calls.

The following examples are based on a `HouseController` class that controls a light switch, temperature device and door lock of a house. In the examples, we only include the methods associated with light control, in order to keep the examples brief, unless including some of the other methods is necessary for illustrating a specific strength or weakness. The protocol for the `LightController` part of the `HouseController` class is displayed in Figure 1.1. The protocol is presented as a labelled transition system. The state $\perp$ is the initial state, the transitions originating from states are annotated with an allowed method call, finally, the state end describes the completed protocol.

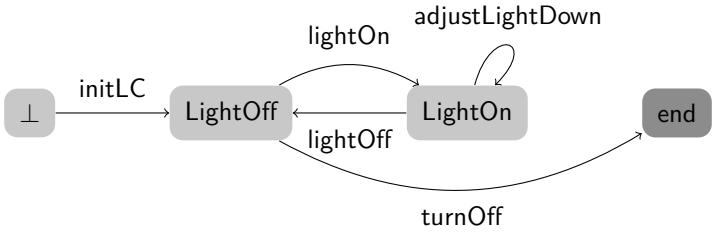

Figure 1.1: Shows `LightController` part of the protocol graph for the `HouseController` class.

Fugue

Fugue is an example of a modular analysis tool developed for object-oriented languages that compile to the Common Language Runtime (CLR) [DF04]. Fugue allows programmers to specify state-machine protocols as pre and post conditions and Fugue can then check that all method invocations are correct in relation to avoiding null-dereferencing and following the specified protocol. In Fugue, pre and post conditions are specified using custom attributes that are associated with each method declaration and the possible states are enumerated and associated with the class declaration. For example, we can specify that a File object can be either open or closed as WithProtocol("Open","Closed"). In order to define the pre and post conditions we use the following attributes: Creates("Open") defines the initial state, ChangeState("Open","Closed") defines a transition from the open to the closed state and InState("Open") specifies that the object must be in the open state.

The previously mentioned attributes work on symbolic state names; however, naming each distinct state can be problematic in large state spaces. Fugue, therefore, allows *Custom States* where fields are types that can be used as attribute parameters in pre and post conditions. The example in Listing 1.1 uses custom states since there is an exponential number of states related

to the various states of fields lc, tc, and dc; and their combinations. When using custom states, the attributes are slightly different from the ones presented but follows the same concept.

```
1  class HouseControllerState : CustomState {
2      bool LightOn;
3      bool TempOn;
4      bool DoorOpen;
5  }
6
7  [WithProtocol(CustomState=typeof(HouseControllerState))]
8  class HouseController {
9
10     LightController lc;
11     TempController tc;
12     DoorController dc;
13
14     [Creates,
15      OutHouseControllerState(
16         LightOn=false,
17         TempOn=false,
18         DoorOpen=false)]
19     public void initLightController(){
20         lc = new LightController();
21     }
22
23     ...
24
25     [InHouseControllerState(LightOn=false),
26      OutHouseControllerState(LightOn=true)]
27     public void lightOn(){lc.on()}
28
29     [InHouseControllerState(LightOn=true)]
30     public void adjustLightDown(){lc.down(1)}
31
32     [InHouseControllerState(LightOn=true),
33      OutHouseControllerState(LightOn=false)]
34     public void lightOff(){lc.off()}
35
36     ...
37 }
```

Listing 1.1: An example of the `HouseController` class in Fugue

The advantage of the typestate approach Fugue uses is that pre and post conditions, in small programs, are straightforward. Furthermore, since Fugue is compiled to CLR and uses class attributes to make assertions, it supports typestates in multiple practical languages. The downsides are mainly related to readability and reasoning about protocols in large programs. For example, classes with a large number of distinct states, that use custom states, can become unreadable if many field states are considered in the attributes of each method. Furthermore, by distributing the protocol information across various methods it can be hard to reason about large and complex protocols. Finally, Fugue does not have a way to specify an end state, hence protocol completion cannot be guaranteed for a terminating program.

Plaid

Plaid is a dynamically typed programming language that allows specification of protocols by dividing a class into a hierarchy of states and associating the allowed methods and variables with each state [SNS+11]. Methods can then update the global state of an object by assigning a new state from the hierarchy. In Plaid, the **case of** keyword specifies inclusion in a state dimensions. For example, `Open` **case of** `File` tell us that the Open state is in the File state dimension. Plaid also allows *and-states* using the **with** keyword; the **with** keyword specifies that a state must include all states part of the *with-expression* and they must be from separate dimensions. For example, **state** `Open` **case of** `File = Status` **with** `ContentType` which tell us that the state `Open` is in the `File` dimension and includes both state `Status` and state `ContentType`. Finally, Plaid allows *or-states* by declaring that two states are part of the same dimension, we can specify that the parent state is one of the two using **case of**. Plaid handles exponential state-spaces in the update rule by only updating the parts that are mutually exclusive with the new state.

```
1  val houseController = new HouseController @
2              LightUninitialsed with TempUninitialsed with DoorUninitialsed;
3
4  state HouseController { }
5
6  state LightUninitialsed case of HouseController {
7      method initLightController(){
8          lc = new LightController();
9          this <- LightOff;
10     }
11 }
12
13 state LightOff case of HouseController {
14     method lightOn(){
15         lc.on();
16         this <- LightOn;
17     }
18 }
19
20 state LightOn case of HouseController {
21     method adjustLightDown(){
22         lc.down(1);
23     }
24     method lightOff(){
25         lc.off();
26         this <- LightOff;
27     }
28 }
29
30 ...
```

Listing 1.2: An example of the HouseController definition in Plaid

Plaid represents a new typestate-oriented programming paradigm as an extension to object-oriented programming where the focus is on object states and their transitions. The Plaid project is fairly well-developed in the sense that it supports both gradual typing and some forms of aliasing; also more involved program examples have been written in Plaid. That said, there

are a number of disadvantages. Plaid does not suffer from the same issues of readability as custom states from Fugue, but state transitions are still distributed across methods and various states, for example in Listing 1.2 at lines 9, 16, and 26 which can result in protocols being hard to reason about. Furthermore, Plaid also does not include a way to define a final state or specify an initial state in the state declarations.

Mungo

Mungo is a tool that was originally developed to allow typestate definitions in Java [KDPG16]. Later work was presented for a core calculus also named Mungo [BFG+20], in which we will present the following example. Typestate definitions in Mungo are specified for each class declaration as sequences of allowed method calls. The typestate definitions in Mungo are called usages and are significantly different from the previous approaches since a usage in Mungo is specified globally for a single class instead of being distributed across each method. A usage is defined using three forms of behaviours: branching of the form $\{m_i; w_i\}_{i \in I}$ where a method out of several can be selected, choices of the form $\langle l : u_l \rangle_{l \in L}$ where a usage is chosen based on the return value l_i of a method, and recursion of the form $u^{\overrightarrow{X = u}}$ where whenever X is encountered in u, it is replaced by its associated usage thereby allowing recursion.

```
1  class HouseController {
2
3      {initLightController; LightOff
4       initTempController; TempOff
5       initDoorController; DoorLocked}
6      [
7        LightOff = {lightOn; LightOn
8                    turnOff; end
9                    initTempController; LightOffTempOff
10                   initDoorController; LightOffDoorLocked},
11       LightOn = {adjustLight; LightOn
12                  lightOff; LightOff},
13       ...
14     ]
15
16     LightController lc;
17     TempController tc;
18     DoorController dc;
19
20     void initLightController(){
21         lc = new LightController()
22     }
23
24     ...
25
26     void lightOn(){lc.on()}
27
28     void adjustLightDown(){lc.down(1)}
29
30     void lightOff(){lc.off()}
31
32     ...
```

```
33 }
```

Listing 1.3: An example of the `HouseController` class in Mungo

Usages are in a sense separated from the method implementation since usage transitions are entirely managed by the type system in contrast to manual transitions and annotations. Hence method implementations are simplified by not including protocol logic like Fugue and Plaid. Additionally, Mungo includes an end state which allows Mungo to ensure protocol completion for terminating programs. A downside of Mungo is that usage specification does not handle exponential state-spaces and in some cases, we are forced to enumerate all possible combinations of states in the usages. For example, in Listing 1.3 initialisation of the fields `lc`, `tc`, and `dc` may occur in any order which results in large usages with many distinct states. This is illustrated on line 9 and 10 where the resulting usage, after calling either `initTempController` or `initDoorController`, should include the behaviour of a newly initialised field and the previously initialised field `lc` which results in two new recursive variables `LightOffTempOff` and `LightOffDoorLocked`.

1.2 Aliasing

One of the central challenges encountered, when developing a modular type system with typestates, is that of handling aliases. Tracking the state of objects requires the type system to reason about object states which is complicated in a modular type system when aliasing is allowed.

Example 1. Consider type checking the code in Listing 1.4. The method `useFiles` takes as arguments two file objects that have not been opened and open them. When type checking the method call on Line 11, the type of `f1` is `File[Closed]`, hence both arguments matches the type of the formal parameters. However, when the method is evaluated, then after opening the file referenced by `x1`, the call to `open` on `x2` is no longer valid, hence the call should not be well-typed. On line 14 we see a call to the same method, with two different objects as arguments. The types are exactly the same, the method will execute correctly, and the method call should be well-typed.

```
1  void useFiles(File[Closed] x1, File[Closed] x2) {
2      x1.open();
3      x2.open();
4  }
5  [...]
6
7  f1 = new File();
8  f2 = new File();
9
10 // Should fail
11 useFiles(f1, f1);
12
13 // Should work
14 useFiles(f1, f2);
```

Listing 1.4: Illustrates the challenges of reasoning about aliases

To circumvent this problem, aliasing is often disallowed by using a linear type system. However, several approaches have been proposed to reduce the restrictions on aliasing that comes with using a linear type system.

Adaption and Focus

Fähndrich & DeLine developed two constructs, for creating aliases of linear objects in the Vault programming language, called Adoption and Focus [FD02]. Vault is a programming language that supports tracking protocols of objects via typestates by using pre and post conditions to define protocols. Aliasing control works by allowing a linear type to be allocated to a nonlinear container by using Adoption and then within a limited scope tracking statements on it with Focus. Focus is inferred around statements that work on linear types and in a Focus scope statements on potential aliases of the linear object, that is currently in focus, are prohibited. Potential aliases are specified by annotating nonlinear container objects with guards.

Capabilities

In part inspired by Adoption and Focus, an approach was presented by Voinea et al. Here a capability-based method is used to handle aliasing in the context of concurrent multiparty session types [VDG19]. A session is split into a nonlinear channel and a linear capability. The session channel can only be used when it has the associated capability, hence reasoning about protocols with concurrency becomes similar to ensuring mutual exclusion with locks. An advantage of this approach compared with Adoption and Focus is that potential aliases do not have to be annotated.

Permission Borrowing

Naden et al. developed a type system capable of handling some forms of aliasing by using permission borrowing [NBAB12]. Permissions tell a pointer variable the operations it is allowed to perform on a linear object. They present three forms of permissions: `unique` representing a variable that is not aliased, `immutable` that allows a variable to be aliased but not mutated, and `shared` that allows both aliasing and mutation. In this work, two forms of permission manipulation are facilitated in order to enable aliasing: splitting and borrowing. Permission borrowing is similar to the capabilities approach where a permission to a field variable can be temporarily borrowed in order to access its members. Permission splitting allows a permission to be converted into a multiple of other permissions; for example, an unique permission can be split into multiple immutable permissions.

1.3 Separation Logic

Separation logic was originally developed as a way to reason about programs that mutate data in the presence of aliasing by extending Hoare Logic [O'H12, Rey00]. The basic idea is similar to pre and post conditions where assertions are inserted at various program points and correctness is proved by ensuring these inserted assertions are correct. In addition to Hoare logic, separation logic includes a separation conjunction which is central to separation logic since it represents a request to partition the memory such that the involved assertions are satisfied. The separation conjunction allows us to localise changes to variables and thereby avoid having to consider all possibly aliased variables to create valid assertions in Hoare logic.

Behavioural Separation

Behavioural separation builds upon separation logic but focuses on separation for behavioural types rather than data held in memory. Caires & Seco presented a type system with behavioural separation that is able to handle various forms of aliasing, manipulation of linked-lists and concurrency [CS13]. The purpose of behavioural separation is managing the interference of different sub-usages from the same object and thereby ensuring safety in the presence of concurrency and aliasing. They extend usage specification with, for example, $(U_1 \mid U_2)$ which describes a parallel usage where both usages can be performed without interfering with the other. $\circ U$ describes an isolated usage that is not dependent on global constraints. Finally, shared usages $!U$ allow unbounded splitting into parallel usages or in other words unbounded aliasing. These usages combined with usages similar to the ones presented for Mungo, allow us to define type assertions that describe behavioural dependencies. The type system then ensures that only safe interference occurs by using principles from separation logic to reason about the effects of expressions while preserving modularity.

Separation with Typestates

Militão et al. presented a typestate oriented language with behavioural separation [MaAC10]. In this language, a class is composed of *views*, *view equations* and methods. Views are collections of fields and define the current state of an object. For example, a class EmptyPair could contain the views Left, Right and Pair indicating states where either some or all data has been initialised. View equations then define how the behavioural separation can be used on a class. If a object of class EmptyPair is viewed as a Pair, then by the view equation $\mathsf{Pair} = \mathsf{Left} * \mathsf{Right}$ it can be decomposed into the two views Left and Right. Where the two new views will only have access to the left or right element of the pair respectively. Militão et al. distinguishes between *linear* views and *unbounded* views. For linear views, the fields mentioned in a decomposition must be disjoint, so that a field cannot be used by two different aliases at the same time. For unbounded views, which can be aliased arbitrarily many times, it is required that the fields are read-only.

The behavioural separation is handled by a subtyping relation on the typing environment, and the splitting operator $*$ is lifted to environments, allowing the environments to be split according to the view equations of the fields. These environments allow multiple bindings of the same variable so that a field f can be bound to both Right and Left at the same time, indicating that the same object (bound to f) has been aliased according to the views equations.

1.4 Problem Statement

In this chapter, we have outlined challenges for object-oriented programming languages. We have seen that the presented approaches, Fugue, Plaid, and Mungo, employ different methods for working with typestates.

We have described multiple approaches to lessening the linearity constraint for behavioural type systems. Both in terms of handling aliasing, or more generally handling interference with separation logic.

We have seen that using custom states in Fugue and and-states in Plaid, it is possible to abstract over exponential-size state spaces when expressing type states. In the current work on Mungo however, no such abstraction is possible. We hypothesise that behavioural separation can be used to both lessen the linearity constraint and abstract over exponential state-spaces and propose the following problem statement for the remainder of the report.

Can behavioural separation be used to reason about aliasing and exponential state spaces in the Mungo language, while preserving modularity in type checking?

The remainder of the report is structured as follows. Chapter 2 presents an overview of our solution to both aliasing and exponential state spaces and presents the formal core calculus for our version of Mungo. In Chapter 3 we present the type system for the calculus and present soundness results. An implementation is presented in Chapter 4 through multiple program examples written in Mungo, highlighting features of the implementation and in turn the type system itself. Here we also present a time complexity analysis of the implementation. Finally in Chapter 5 we discuss our findings and propose further work.

Chapter 2

The Language

In this chapter we provide an overview of our solution to the problem statement, describe the version of the Mungo language we are working with, and present two semantics that are equivalent for terminating programs, modelling the run-time behaviour of the language.

2.1 Introducing Parallel Usages

There are some similarities between the types defined by Caires & Seco and the usages employed in Mungo, so this work expands on this similarity by introducing concepts from behavioural separation to the Mungo language to express new types of behaviour. Inspired by the work by Militão et al. [MaAC10] we wish to allow the separation of objects according to the fields they access. However, instead of doing this with *views*, we wish to enforce this through the usages directly.

The usage constructs used in the previous research on Mungo were branching, choice and recursion. A branching usage $\{m_i; w_i\}_{i \in I}$ describes a protocol where a method m_j $(j \in I)$ can be called, and the object will continue with the remaining usage w_j. A choice usage $\langle l : u_l \rangle_{l \in L}$ describes that based on a return value of a method (a label in the enumeration type L), the corresponding usage u_l is chosen. We introduce the concept of *parallel usages* $(u_1|u_2).u_3$ describing how an object can be decomposed into two objects with the usages u_1 and u_2 respectively, and composed back into a single object with usage u_3 after completion of the two local protocols. This construction encodes both the parallel and sequential types presented by Caires & Seco [CS13]. The parallel usage construct describes the behavioural separation where an object can safely be aliased since the two parallel usages cannot interact with the same linear fields. This restriction is similar to the restriction for view equations previously mentioned. This property will be formally defined as well-formed usages later. So while behavioural separation was initially introduced to manage interference in a concurrent language, we apply them to *parallel protocols* in an object-oriented language.

Parallel protocols offer other benefits than just aliasing. Inspired by parallel processes, we allow parallel usages to advance the two local usages separately, meaning that for a field f with usage $(\{m; \mathsf{end}\} \mid \{n; \mathsf{end}\}).u$, both $f.m()$ and $f.n()$ will be valid method calls leading to usages $(\mathsf{end} \mid \{n; \mathsf{end}\}).u$ or $(\{m; \mathsf{end}\} \mid \mathsf{end}).u$ respectively. This neatly solves the problem of exponential usage sizes in Mungo programs, that was described in Section 1.1. Methods that modify a single linear field can now be placed in parallel with other such methods, and no longer requires an enumeration of all possible sequences of the method calls. In fact, this treatment of

parallel usages is similar to the *with*-states of Plaid, as each parallel constituent can be thought of as a separate dimension of the object, from the other.

Example 2. Consider the task of modelling a home automation system. For each subsystem in the house, a controller class is defined. These controllers are then aggregated into controllers for larger subsystems. The `HouseController` class below aggregates three controllers for smaller subsystems. For simplicity, we ignore the initialisation of the fields in this example. This aggregation results in a class with unrelated linear fields. Locking the doors should not impact what methods are available for controlling the lights. In previous work on Mungo this resulted in large usages for the wrapper class, where all possible method sequences on the fields should be encoded in the usage for the wrapper class, as was illustrated in Section 1.1. Employing parallel usages, however, gives us the ability to express that that the methods are unrelated, and can be called in any order. For an instance `hc` of the class, the method sequences `hc.toggleLight();hc.setTemperature();hc.lockDoors()` and `hc.setTemperature();hc.toggleLight();hc.lockDoors()` will both be allowed (as will any other permutation of the method sequence).

```
1  class HouseController {
2      U=({toggleLight;end}|{setTemperature;end}|{lockDoors;end}).end
3
4      LightController lc
5      TempController tc
6      DoorController dc
7
8
9      void toggleLight(){if (lc.isOn()){lc.off()} else {lc.on()}}
10     void setTemperature(){tc.setTemp(22)}
11     void lockDoors(){dc.lock()}
12 }
```

The behavioural separation is apparent in the modularity it allows on method parameters. For the method below which states that the formal parameter should have usage `{toggleLight ; end}`, an object with usage `U`, defined on line 2, is a valid parameter. The type system allows for the parallel usages to be split into its component, and that each component is treated separately. This means that the actual parameter of type `HouseController[U]` can be split into three objects with types `HouseController[{toggleLight; end}]`, `HouseController[{setTemperature;end}]`, and `HouseController[{lockDoors;end}]` and then the method can be called with the correct type.

```
1  void controlLights(HouseController[{toggleLight; end}] →
2                     HouseController[end] x) {
3      x.toggleLight()
4  }
```

After a method call, the usage of the actual parameter is updated according to the method signature, and the usages can be composed back into a parallel usage, as illustrated below.

```
1  // Usage is ({toggleLight; end} | {setTemperature; end} | {lockDoors; end
       }).end
2  hc = new HouseController;
3  controlLights(hc)
4  //Usage is now (end | {setTemperature; end} | {lockDoors; end}).end
```

2.2 Syntax

In this section we present the syntax for a version of the Mungo calculus. The syntax is given in Figure 2.1 and Figure 2.2 with run-time parts of the syntax highlighted in grey. The syntax is based on the Mungo calculus presented by Bravetti et. al [BFG+20] with minor simplifications and extended with parallel usages.

2.2.1 Classes, Expressions and Values

A program declaration in the Mungo calculus is a set of class declarations $\vec{D}$. A class declaration defines a class C with usage $\mathcal{U}$, a set of field declarations $\vec{F}$, and a set of method declarations $\vec{M}$. Method declarations are of the form $t_5\ m(t_1 \to t_2\ x_1, t_3 \to t_4\ x_2)\ \{e\}$ where t_5 denotes the return type, the parameters of the form $t \to t'\ x$ specify that an argument x has type t before the method body is evaluated and afterwards it has type t'. Finally, the body of method m is an expression e. The declarations for methods have been extended with a second argument, compared with earlier versions of Mungo, to allow aliasing on method arguments. With this notation for method parameters, we now require both pre and post conditions on method parameters. The reason for this requirement is that there is no longer a linearity assumption, hence changes to parameters in a method call can induce changes in a field of the caller. The post condition allows the type system to track these changes and update the field accordingly.

(FNames)	f
(PNames)	x
(MNames)	m
(CNames)	C
(Locations)	o
(Declarations)	$D ::= \mathsf{class}\ C\ \{\mathcal{U}, \vec{F}, \vec{M}\}$
(Fields)	$F ::= z\ f$
(Methods)	$M ::= t\ m(t \to t_\perp\ x, t \to t_\perp\ x)\ \{e\}$
(References)	$r ::= f \mid x$
(Expressions)	$e ::= e; e \mid f = e \mid r.m(v,v) \mid v \mid \mathsf{new}\ C \mid$ $\mathsf{if}_{\boxed{r}}\ (r.m(v,v))\ \{e\}\ \mathsf{else}\ \{e\} \mid k : e \mid \mathsf{continue}\ k$ $\boxed{\mathsf{return}_{r.m(v,v)}\{e\}}$
(Values)	$v ::= b \mid r \mid \boxed{o}$
(BValues)	$b ::= \mathsf{unit} \mid \mathsf{null} \mid \mathsf{true} \mid \mathsf{false}$

Figure 2.1: Syntax of Mungo

Expressions e include typical imperative statements as well as values. A difference between this version of Mungo and the version presented by Bravetti et al. [BFG+20] is that *switch*-expressions along with enumeration types have been removed. The switch expression would, based on an enumeration type, choose the corresponding expression, and follow that branch in a

usage as well. We simplify this by using the *if*-expression to choose a branch based on a returned boolean from a method and update the choice-usage of the callee according to the boolean as well. As we can encode switch-expressions using this if-expression, we do not consider them integral to the language and omit them for simplicity. Iterations are expressed as jumps where a label is given to an expression $k : e$ and by writing $\mathsf{continue}\ k$ we jump back to expression e. The remaining *user (non-runtime) expresssions* are sequential composition $e; e$, field assignments $f = e$, method calls $r.m(v, v)$, values v and object initialisations $\mathsf{new}\ C$. Finally we have the *return*-expression $\mathsf{return}_{r.m(v,v)}\{e\}$, annotated with a method call. This is used in the small-step semantics to wrap the method body during an evaluation, to know when to pop the call-stack, and the method call is used to determine the callee to update the usage accordingly.

2.2.2 Well-Formed Continue Expressions

As we treat control structures as expressions, we introduce well-formedness for expression, where we require a correct placement of $\mathsf{continue}$ expressions. The rules in Table 2.1 ensures the following:

- Expressions $\mathsf{continue}\ x; e$ are disallowed
- Free occurrences of labels are disallowed
- There is always a terminating branch in loops

As we handle loops with unfolding, $\mathsf{continue}\ x; e$ would result in nonsensical behaviour where e would be evaluated for each loop iteration, in spite of $\mathsf{continue}\ x$ clearly indicating a jump back to the labelled expression. The last requirement is mostly technical, as it allows us to always be able to reason about a loop. It is not required that the terminating branch is ever chosen. In programs, we require that for all methods $t\ m(t \to t_\bot\ x, t \to t_\bot\ x)\ \{e\}$ we have $\emptyset \vdash e \dashv \theta$ where $\top \in \theta$.

$$\text{(Seq)}\quad \frac{\theta \vdash e \dashv \theta' \cup \{\top\} \qquad \theta \vdash e' \dashv \theta''}{\theta \vdash e; e' \dashv \theta''} \qquad \text{(Call)}\quad \frac{}{\theta \vdash r.m(v_1, v_2) \dashv \{\top\}}$$

$$\text{(If)}\quad \frac{\theta \vdash e_1 \vdash \theta' \qquad \theta \vdash e_2 \vdash \theta'' \qquad \top \in \theta' \cup \theta''}{\theta \vdash \mathsf{if}_r\ (r.m(v, v))\ \{e\}\ \mathsf{else}\ \{e'\} \dashv \{\top\}} \qquad \text{(Assign)}\quad \frac{\emptyset \vdash e \dashv \theta \cup \{\top\}}{\theta \vdash f = e \dashv \{\top\}}$$

$$\text{(Val)}\quad \frac{}{\theta \vdash v \dashv \{\top\}} \qquad \text{(New)}\quad \frac{}{\theta \vdash \mathsf{new}\ C \dashv \{\top\}}$$

$$\text{(Lab)}\quad \frac{\theta \cup \{k\} \vdash e \dashv \theta' \cup \{\top\}}{\theta \vdash k : e \dashv \{\top\}} \qquad \text{(Con)}\quad \frac{k \in \theta}{\theta \vdash \mathsf{continue}\ k \dashv \{k\}}$$

Table 2.1: Well-formed continue expression judgments

2.2.3 Types

Types t in the language can be either base types $\mathsf{bool}, \mathsf{void}$ or a typestate $C[\mathcal{U}]$ composed of a class name C and a usage $\mathcal{U}$. Furthermore, we introduce the special type $\bot$ to describe the type of null. We write $t_\bot$ in the syntax where it is allowed to use $\bot$ in the user syntax. This is only allowed in the post conditions for method parameters, to indicate that a parameter was

consumed (linearly read) in the method. Field types can be base types or a class name, meaning that a field can hold an object of class C no matter the current usage and it can also contain the null-value.

$$
\begin{aligned}
&\textbf{(FTypes)} && z ::= \mathsf{bool} \mid \mathsf{void} \mid C \\
&\textbf{(Types)} && t ::= C[\mathcal{U}] \mid \mathsf{bool} \mid \bot \mid \mathsf{void} \\
&\textbf{(Usages)} && \mathcal{U} ::= u^s \mid u^? \\
& && u ::= \mu X.u \mid \{m_i; w_i\}_{i \in I} \mid (u_1 \mid u_2).u_3 \mid \mathsf{end} \mid \circledcirc \\
& && w ::= \langle u_1, u_2 \rangle \mid u \\
& && s ::= s \cdot \mathtt{l} \mid s \cdot \mathtt{r} \mid \epsilon
\end{aligned}
$$

Figure 2.2: Syntax of types in Mungo

The syntax for recursive usages $\mu X.u$, matches closely to the syntax of labelled expressions. Choice usages $\langle u_1, u_2 \rangle$ correspond to the two branches of an if-expression, u_1 describing the true-branch and u_2 describing the false-branch. Branch usages $\{m_i; w_i\}_{i \in I}$ remain unchanged from the earlier work on Mungo. We then introduce the parallel usages $(u_1 \mid u_2).u_3$ to support behavioural separation. The goal is to use each side of the usage disjointly, so we introduce a runtime usage $\circledcirc$ to indicate a *hole* in parallel usages. For example, $(\circledcirc \mid u_2).u_3$ indicates that a usage is missing from the parallel usage. This placeholder is used to retain the parallel structure of usages, even when splitting an object reference. As $\circledcirc$ does not describe any behaviour, an object with usage $(\circledcirc \mid \circledcirc).u_3$ cannot perform the behaviour described by u_3 before the holes in the usage have been filled. Finally, the end usage describes a terminated protocol, where no operations are permitted. Even though neither $\circledcirc$ nor end allow any operations they are not equivalent. end describes a terminated protocol, whereas $\circledcirc$ describes a (potentially unfinished) missing protocol.

The syntax for usages $\mathcal{U}$ are on the form u^s, where u is the protocol and s is used to keep track of usage splits. A usage that has not been split is tagged with ϵ. During a split of the usage $(u_1 \mid u_2).u_3^s$, the constituents are tagged such that their initial configuration can be reconstructed, hence a split would result in the following usages $u_1^{s \cdot \mathtt{l}}$, $u_2^{s \cdot \mathtt{r}}$ and $(\circledcirc \mid \circledcirc).u_3^s$. Usages in class definitions are always implicitly on the form u^ϵ, as instantiating an object is always the full object. Usage for method parameters are always annotated as $u^?$. As methods are invariant to the particular split of an object, the wildcard ? is only instantiated with a split s during the type-checking of the call.

Terminated Types

In previous versions of Mungo [BFG+20, KDPG16] linear type systems were used to track changes in object states. In the linear type systems the notion of linearity defined that objects with a usages different from end are linear and linear values cannot be overwritten. This notion of linearity, among other things, is important to protocol completion since we can use it to check whether an object is linear and ensure we do not lose references to objects with incomplete protocols. Similarly, in our present work we want to ensure protocol completion; however, we are not employing a strictly linear type system and therefore define a notion called *terminated* instead. A type is considered to be terminated if it is a base type or if the type is a typestate where the usage is end. The predicate term is used to test whether an object is terminated

to handle destructive reads in the semantics and in the type system ensure references to non-terminated objects are not lost during sequential expressions and assignments. We retain the notion that a type is linear, if it is not terminated.

Definition 1 (Terminated type)**.** For a type t we say that t is terminated written $\mathsf{term}(t)$ with the following predicate:

$$\mathsf{term}(\mathsf{bool}) \quad \mathsf{term}(\mathsf{void}) \quad \mathsf{term}(\bot) \quad \mathsf{term}(C[\mathsf{end}^s])$$

2.2.4 Usage Transitions

Usages describe a labelled transition system. The transition rules are shown in Table 2.2. Rule (Branch) describes that usages of the form $\{m_i; w_i\}_{i \in I}$ allow any method m_j, where $j \in I$, to be called and the usage continues as the associated usage w_j. Rule (Rec) describes transitions for recursive usages of the form $\mu X.u^s \xrightarrow{m} u'^s$ where X is a recursion variable mapped to usage u. The rule tells us that occurrences of variable X in usage u are unfolded when performing a transition. Rules (SelTrue) and (SelFalse) specify that choice usages of the form $\langle u_1, u_2 \rangle^s$ either continue as u_1 or u_2 depending on a boolean value. Notice that parallel usages have no transition rules. We argue that parallel usages do not describe behaviour directly, but only describe how an object is perceived. The behaviour of a parallel usage is only implicitly described by each branch of the parallel usage. So instead of treating an object as having a parallel usage, we should treat an object as having either one of the two sides of the parallel usage instead. A further description of this will follow in the next section. Finally, we define that two end usages in a parallel usage are congruent with the succeeding usage, as illustrated below. This allows us to continue from a parallel usage into the succeeding usage, precisely when both local protocols have finished.

$$(\mathsf{end} \mid \mathsf{end}).u^s \equiv u^s$$

$$\text{(Branch)} \quad \frac{j \in I}{\{m_i; w_i\}^s_{i \in I} \xrightarrow{m_j} w^s_j} \qquad \text{(SelTrue)} \quad \langle u_1, u_2 \rangle^s \xrightarrow{\mathsf{true}} u^s_1$$

$$\text{(SelFalse)} \quad \langle u_1, u_2 \rangle^s \xrightarrow{\mathsf{false}} u^s_2 \qquad \text{(Rec)} \quad \frac{(u\{\mu X.u/X\})^s \xrightarrow{m} u'^s}{\mu X.u^s \xrightarrow{m} u'^s}$$

Table 2.2: Labelled transitions for usages

2.2.5 Usage Well-Formedness

As parallel usages represent a behavioural separation, we must ensure that that each parallel constituent can be treated separately. The first step of ensuring this, is on a syntactical level, with well-formed usages. Figure 2.3 illustrates the desired property. An object with a parallel usage can be split into disjoint parts of the whole. To achieve this separation, we must ensure that operations on one part, do not impact any other.

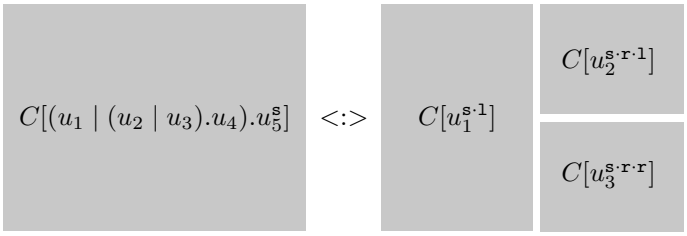

Figure 2.3: Illustration of parallel usages

$$(\textsc{Par})\ \frac{\begin{array}{c}\mathsf{fields}(C,u_1)\cap\mathsf{fields}(C,u_2)=\emptyset\\ \mathsf{methods}(u_1)\cap\mathsf{methods}(u_2)=\emptyset\\ \emptyset,C\vdash u_1\ \mathtt{ok}\quad \emptyset,C\vdash u_2\ \mathtt{ok}\quad \theta,C\vdash u_3\ \mathtt{ok}\end{array}}{\theta,C\vdash (u_1\mid u_2).u_3\ \mathtt{ok}} \qquad (\textsc{Br})\ \frac{\forall i\in I.\theta,C\vdash w_i\ \mathtt{ok}}{\theta,C\vdash \{m_i;w_i\}_{i\in I}\ \mathtt{ok}}$$

$$(\textsc{Sel})\ \frac{\theta,C\vdash u_1\ \mathtt{ok}\quad \theta,C\vdash u_2\ \mathtt{ok}}{\theta,C\vdash \langle u_1,u_2\rangle\ \mathtt{ok}} \qquad (\textsc{Rec})\ \frac{\theta\cup\{X\},C\vdash u\ \mathtt{ok}}{\theta,C\vdash \mu X.u\ \mathtt{ok}}$$

$$(\textsc{Var})\ \frac{}{\theta\cup\{X\},C\vdash X\ \mathtt{ok}} \qquad (\textsc{En})\ \frac{}{\theta,C\vdash \mathsf{end}\ \mathtt{ok}}$$

Table 2.3: Well-formed usage judgments for parallel usages

Table 2.3 shows the conditions for a usage being well-formed. The most interesting rule is (Par), which states that the methods and fields mentioned in the parallel constituents are disjoint. The methods and fields functions are defined to extract the methods a usage mention and the fields accessed in those methods, respectively. The cases omitted from the presentation are the obvious definitions of recursively calling the function on subusages or subexpressions. The mbody function extracts the method body of a method m from a class C.

$$\mathsf{methods}(\{m_i;w_i\}_{i\in I})=\bigcup_{i\in I}\{m_i\}\cup\mathsf{methods}(w_i)$$

$$\mathsf{fields}(C,u)=\bigcup_{m\in\mathsf{methods}(u)}\mathsf{fields}(\mathsf{mbody}(C,m))$$
$$\mathsf{fields}(f=e)=\mathsf{fields}(e)\cup\{f\}$$
$$\mathsf{fields}(f)=\{f\}$$

The next set of judgments for usage well-formedness requires protocols to always be able to terminate. That is, we disallow infinite behaviour such as $\mu X.\{m;X\}$, but do allow the behaviour $\mu X.\{m;X\ n;\mathsf{end}\}$. The reason for this requirement is mostly technical. When following an infinite usage, we cannot reason about the resulting typing environment as it can never be reached. By requiring the usage to be able to terminate, we can reason about the resulting environment, in the case where the loop is terminated. It is important to note that we do not disallow infinite programs, it is still possible to create an infinite loop, we only require protocols *to be able* to terminate at some point in the future. The judgments for well-formedness are shown

in Table 2.4 and works by collecting the possible end-states, which are either recursion variables or the end usage. We require for a well-formed usage u where $\emptyset \vdash u \dashv \theta$, that $\mathsf{end} \in \theta$.

$$(\textsc{Par})\quad \frac{\theta \vdash u_3 \dashv \theta' \quad \emptyset \vdash u_1 \dashv \theta_1 \cup \{\mathsf{end}\} \quad \emptyset \vdash u_2 \dashv \theta_2 \cup \{\mathsf{end}\}}{\theta \vdash (u_1 | u_2).u_3 \dashv \theta'} \qquad (\textsc{Bra})\quad \frac{\forall i \in I. \theta \vdash w_i \dashv \theta_i}{\theta \vdash \{m_i; w_i\}_{i \in I} \dashv \bigcup_{i \in I} \theta_i}$$

$$(\textsc{Sel})\quad \frac{\theta \vdash u_1 \dashv \theta_1 \quad \theta \vdash u_2 \dashv \theta_2}{\theta \vdash \langle u_1, u_2 \rangle \dashv \theta_1 \cup \theta_2} \qquad (\textsc{Rec})\quad \frac{\theta \cup \{X\} \vdash u \dashv \theta' \cup \{\mathsf{end}\}}{\theta \vdash \mu X.u \dashv \{\mathsf{end}\}}$$

$$(\textsc{Var})\quad \frac{X \in \theta}{\theta \vdash X \dashv \{X\}} \qquad (\textsc{End})\quad \frac{}{\theta \vdash \mathsf{end} \dashv \{\mathsf{end}\}}$$

Table 2.4: Well-formed usage judgments for termination

We require that usages defined in the program text are well-formed, and as well-formedness is preserved by the usage transitions presented in Table 2.2, all usages encountered during execution will be well-formed.

2.2.6 Expressive Power of Usages

We now explore the expressive power of usages. To do so, we analyse the language class of the *traces* of usages, that is, the operation sequences they can describe. We write $\mathcal{L}(\mathcal{U})$ to describe the language of method traces of $\mathcal{U}$. This treatment is simplified by assuming the existence of the ordinary transition rules for parallel constructs, where each side can advance individually. While not present in the transition rules described in Table 2.2, the behaviour is simulated in the type systems, as will be illustrated later.

To illustrate the language class of usages traces, consider the following usage:

$$\mathcal{U} = \mu X.\{m; X \quad n; \mathsf{end}\}$$

We see that the usage allows any number of method calls to m, but terminates after a call to n, hence the strings generated by the regular expression m^*n are in the language $\mathcal{L}(\mathcal{U})$, but as no termination is required the infinite sequence of calling m is also allowed, hence $m^\omega \in \mathcal{L}(\mathcal{U})$. As usages generate both finite and infinite strings, we see directly that the languages are neither regular nor ω-regular. But if we construct an automaton, that for finite strings accepts as a NFA, and for infinite strings accepts as a Büchi automaton, we see that in fact the language of a usage with transition labels Σ is $\mathcal{L}(\mathcal{U}) \subseteq \Sigma^\infty$ where, following the definition by Thomas [Tho91], $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$.

For a usage $\mathcal{U}$, we define the following two automatas as a 5-tuples: $(Q, \Sigma, \delta, q_0, F_{\text{NFA}})$, and $(Q, \Sigma, \delta, q_0, F_{\text{Büchi}})$ for recognising the traces of $\mathcal{U}$. The automatas recognise the finite and infinite

traces of $\mathcal{U}$ respectively.

$$
\begin{aligned}
Q &= \{U' \mid \mathcal{U} \to^* \mathcal{U}'\} \\
\Sigma &= \mathsf{methods}(\mathcal{U}) \cup \{\mathsf{true}, \mathsf{false}\} \\
\delta(\mathcal{U}', a) &= \{\mathcal{U}'' \mid \mathcal{U}' \xrightarrow{a} \mathcal{U}''\} \\
q_0 &= \mathcal{U} \\
F_{\text{NFA}} &= \{\mathsf{end}\} \\
F_{\text{Büchi}} &= Q
\end{aligned}
$$

We argue that for a finite usage $\mathcal{U}$, both Q and Σ are finite. Σ is finite since we can only mention finitely many method names in a finite usage, and by inspection of the transition rules for usages, we see that only a single unfolding of recursive usage definitions are enough to reach all usage states by the transition relation. We see that the definitions follow that of an NFA or a Büchi automaton, hence based on the acceptance criteria we will accept regular or ω-regular languages. The set of acceptance states are different depending on the input is a finite or infinite traces. For finite traces, the usage must terminate in end. For infinite traces, we allow all infinite behaviour specified by the usage, hence $F_{\text{Büchi}} = Q$.

2.3 Semantics

In this section, we present both a big-step and a small-step operational semantics for the presented version of Mungo. The purpose of presenting both semantics is to simplify proving and stating properties about the type system. The small-step semantics is used to prove temporal properties such as protocol fidelity and the big-step semantics simplifies proving properties about type safety.

Previous work on Mungo only employed a small-step semantics. Proving soundness for the type system then entailed showing that the type system is an approximation of the small-step semantics. This made for some unwieldy proofs, where the properties could not be naturally expressed. We seek to remedy this by introducing big-step semantics for the language and show that a big-step transition can be matched by a terminating sequence of small-step semantics and vice versa. This equivalence result lets us express the desired properties in a natural way, and apply the result to both semantics.

2.3.1 Configurations

The big-step and small-step semantics both use the *environment-store* binding model. Values v, which are stored in the environments can be base values b or locations o.

$$\mathbf{Values} = \mathbf{Locations} \cup \mathbf{BValues}$$

Locations, are only references to objects and do not store any information about the object itself. The information about the fields and the type information is stored in a heap h, which maps object references to a typestate for the object, as well as the field environment env_f which stores the values of the fields.

$$\mathbf{Env_F} = \mathbf{FNames} \rightharpoonup \mathbf{Values}$$

$$\mathbf{Heap} = \mathcal{P}(\mathbf{Locations} \rightharpoonup \mathbf{Typestates} \times \mathbf{Env_F})$$

As can be seen in the definition of **Heap**, a heap can contain multiple bindings of an object o, as heaps are sets of partial functions. We use this to handle parallel usages, where each constituent of the parallel usage is stored as a binding of o. Syntactically, we describe heaps with multiple bindings using *heap separation* with the operator $*$. A heap h can be separated, using heap subtyping, into $h' * h''$, where bindings of an object o can appear in both h' and h''. When extracting a value from a separated heap $(h' * h'')(o)$ we allow the value to be extracted from either h' or h''. Due to the well-formedness condition on usages, there is only one binding of o that makes sense in a given context. For example, if a method call of method m occurs, and the binding of o should be updated to reflect the usage, then only one binding of o will have a usage that allows a m-transition.

We use the following notation when updating bindings in the heap: We write $h[o.\mathsf{usage} \mapsto \mathcal{U}']$ which specifies that we choose an o in h and the usage of o is updated to $\mathcal{U}'$. Additionally, we write $h[o.f \mapsto v]$ to update a field in the object referenced by o. Given that multiple occurrences of o is allowed in h a choice need to be made concerning which o to update, this is straightforward due to usage well-formedness since there is only one o that contains f. We write h, h' to denote the heap containing the bindings of both h and h'. If there are overlapping bindings of fields in h and h', then the separation should be such that every separated heap contains at most one binding of a particular object, and all bindings from h and h' are contained in h, h'. For example, consider the heap $h = \{o_1 \mapsto (C_1[\mathcal{U}_1], env_{f_1})\} * \{o_2 \mapsto (C_2[\mathcal{U}_2], env_{f_2})\}$, then $h, o_1 \mapsto (C_1[\mathcal{U}_3], env_{f_3}) = \{o_1 \mapsto (C_1[\mathcal{U}_1], env_{f_1})\} * \{o_2 \mapsto (C_2[\mathcal{U}_2], env_{f_2}), o_1 \mapsto (C_1[\mathcal{U}_3], env_{f_3})\}$. We use this notation to extract and insert values in the heap.

We assume that the static information about classes such as methods can be accessed directly on the class, such that for a class definition $\mathsf{class}\ C\{\mathcal{U}, \overrightarrow{F}, \overrightarrow{M}\}$ we have that $C.\mathsf{methods} \triangleq \overrightarrow{M}$, $C.\mathsf{fields} \triangleq \overrightarrow{F}$ and $C.\mathsf{usage} \triangleq \mathcal{U}$. For extracting information from the heap, we define the following:

$$(C[\mathcal{U}], env_f).\mathsf{type} \triangleq C[\mathcal{U}] \qquad (C[\mathcal{U}], env_f).\mathsf{usage} \triangleq \mathcal{U}$$
$$(C[\mathcal{U}], env_f).\mathsf{fields} \triangleq env_f \quad (C[\mathcal{U}], env_f).f \triangleq env_f(f)$$
$$(C[\mathcal{U}], env_f).\mathsf{class} \triangleq C$$

The configuration in both semantics keep track of the *active object*, which is the topmost object of the call-stack. Furthermore, they both use a parameter environment env_P to track the parameter bindings for the method that is being evaluated.

$$\mathbf{Env_P} = \mathbf{PNames} \rightharpoonup \mathbf{Values}$$

While it is enough to keep track of the current active object and the current parameter bindings in the big-step semantics, it is necessary to model the entire call-stack in the small-step semantics. So for configurations in the small-step semantics, we introduce a stack environment env_S. The stack environment is a sequence of elements (o, env_P) where o is the active object and env_P is the related parameter environment. Each element in the stack corresponds to a nested method call of the current expression, and as such corresponds to the number of nested returns in the expression.

$$\mathbf{Env_S} = \overrightarrow{\mathbf{Locations} \times \mathbf{Env_P}}$$

Heap subtyping

We introduce behavioural separation to the heap with a subtyping relation. The goal of the subtyping relation is to split the objects in the heap according to their usage and allow multiple bindings of the same object names. Table 2.5 shows the subtyping relation $<:$. In the (ParL)

and (ParR) rules we create additional bindings of the object o, according to the parallel usage. Notice that we add the placeholder usage $\circledcirc$ to the place that was previously occupied by the extracted usage. This ensures that the only way the continuation usage u_3^s can be used, is by merging the bindings of o back into the parallel usage, to reach the usage $(\mathsf{end} \mid \mathsf{end}).u_3^s$ which by definition is the same as u_3^s, and o now has a type of $C[u_3^s]$. This ensures that evaluation of the operations in u_3^s can only start when all aliases introduced by the parallel usage have been reobtained. The transitivity property (by rule (Trans)) allows the subtyping relation to extract and merge arbitrarily deeply nested parallel components.

$$\text{(ParL)}\ \frac{env_f = env'_f \cdot env''_f}{h, o \mapsto \langle C[(u_1 \mid u_2).u_3], env_f\rangle <:> \begin{array}{l} h, o \mapsto \langle C[(\circledcirc \mid u_2).u_3^{\mathtt{s}}], env'_f\rangle * \\ \{o \mapsto \langle C[u_1^{\mathtt{s}\cdot\mathtt{l}}], env''_f\rangle\}\end{array}}$$

$$\text{(ParR)}\ \frac{env_f = env'_f \cdot env''_f}{h, o \mapsto \langle C[(u_1 \mid u_2).u_3], env_f\rangle <:> \begin{array}{l} h, o \mapsto \langle C[(u_1 \mid \circledcirc).u_3^{\mathtt{s}}], env'_f\rangle * \\ \{o \mapsto \langle C[u_2^{\mathtt{s}\cdot\mathtt{r}}], env''_f\rangle\}\end{array}}$$

$$\text{(Concat)}\ \frac{\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset}{h_1, h_2 <:> h_1 * h_2} \qquad \text{(Id)}\ \frac{}{h * \emptyset <:> h <:> \emptyset * h}$$

$$\text{(Trans)}\ \frac{h <: h' \qquad h' <: h''}{h <: h''}$$

Table 2.5: Heap subtyping

In the rules (ParL) and (ParR) we make use of a context-split operation, inspired by the work of Thiemann & Vasconcelos [TV16], on the field environment env_f. The operations are defined in Table 2.6 and distributes the field environment between the two objects. Since the usage is well-formed, we know that the two aliases will access a disjoint set of fields, so we know that a split exists, such that all accessed fields will be available to both bindings of o in the heap.

$$\text{(SplitR)}\ \frac{env_f = env_{f_1} \cdot env_{f_2}}{env_f, f \mapsto v = env_{f_1}, f \mapsto v \cdot env_{f_2}}$$

$$\text{(SplitL)}\ \frac{env_f = env_{f_1} \cdot env_{f_2}}{env_f, f \mapsto v = f \mapsto v \cdot env_{f_1}, env_{f_2}}$$

$$\text{(Base)}\ \frac{}{env_f = env_f \cdot \emptyset = \emptyset \cdot env_f}$$

Table 2.6: Field environment split

Method Traces

Method traces are sequences of method calls along with the invoking object. We let A be the set of method invocations $o.m$ and branch selections $o.b$ where b is true or false. Then a method trace α is an element of the free monoid A^*.

$$A = \{o.m \mid o \in \textbf{Locations}, m \in \textbf{MNames}\} \cup \{o.b \mid o \in \textbf{Locations}, b \in \{\mathsf{true}, \mathsf{false}\}\}$$

An example of a method trace could be $\langle o.\texttt{init} \cdot o.\texttt{isValid} \cdot o.\mathsf{true} \cdot o.\texttt{save}\rangle$, indicating the method sequence for the object o as well as the branches selected in if-cases. There is a strong similarity between method traces and usage transitions as defined in Section 2.2. In fact, method traces form the basis for establishing protocol fidelity for the big-step semantics, as they allow us to inspect the ordering of method calls, even if they all happen in a single transition.

2.3.2 Big-Step Semantics

The transitions in the semantics are of the form $env_P, h \vdash \langle o, e\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$ where h is a heap, env_P is a parameter environment, expression e is the expression being evaluated, v is the resulting value, and α is the method trace of the evaluation. For readability of the rules we omit writing α when $\alpha = \varepsilon$.

We define a function $\mathsf{extract}$ to extract information from the heap and parameter environment. The $\mathsf{extract}$ function is used in rules where method calls occur to extract information about passed parameters and the callee of the method. Base values are simply returned, but for fields and parameters, the object name of the corresponding environment is returned.

$$\mathsf{extract}(v, h, env_P, o) = \begin{cases} env_P(x) & \text{if } v = x \\ h(o).f & \text{if } v = f \\ v & \text{otherwise} \end{cases}$$

Control Structures

Table 2.7 shows the big-step transitions for control structures. They are for the most part unsurprising, except for the rules for if-constructs which ensures that the usages are followed and updates the method sequence to reflect the transition of either true or false. The if-rules are further complicated by the fact that r can have a parallel usage, which is why we allow subtyping on the heap to extract the correct binding of r to allow the transition.

$$(\textsc{Seq}_\text{B}) \quad \frac{env_P, h \vdash \langle o, e\rangle \xrightarrow{\alpha} v'' \dashv env''_P, h'' \quad env''_P, h'' \vdash \langle o, e'\rangle \xrightarrow{\alpha'} v' \dashv env'_P, h'}{env_P, h \vdash \langle o, e; e'\rangle \xrightarrow{\alpha\cdot\alpha'} v' \dashv env'_P, h'}$$

$$(\textsc{IfTrue}_\text{B}) \quad \frac{\begin{array}{c} env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \xrightarrow{\alpha} \mathsf{true} \dashv env''_P, h'' \\ h''' <: h'' \quad h'''(o').\mathsf{usage} \xrightarrow{\mathsf{true}} \mathcal{U} \quad h'''' <: h'''[o'.\mathsf{usage} \mapsto \mathcal{U}] \\ env''_P, h'''' \vdash \langle o, e_1\rangle \xrightarrow{\alpha'} v \dashv env'_P, h' \end{array}}{env_P, h \vdash \langle o, \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \xrightarrow{\alpha\cdot o'.\mathsf{true}\cdot\alpha'} v \dashv env'_P, h'}$$

where $o' = \mathsf{extract}(r, h, env_P, o)$

$$(\textsc{IfFalse}_\text{B}) \quad \frac{\begin{array}{c} env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \xrightarrow{\alpha} \mathsf{false} \dashv env''_P, h'' \\ h''' <: h'' \quad h''(o').\mathsf{usage} \xrightarrow{\mathsf{false}} \mathcal{U} \quad h'''' <: h'''[o'.\mathsf{usage} \mapsto \mathcal{U}] \\ env''_P, h'''' \vdash \langle o, e_1\rangle \xrightarrow{\alpha'} v \dashv env'_P, h' \end{array}}{env_P, h \vdash \langle o, \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \xrightarrow{\alpha\cdot o'.\mathsf{false}\cdot\alpha'} v \dashv env'_P, h'}$$

where $o' = \mathsf{extract}(r, h, env_P, o)$

$$(\textsc{Lab}_\text{B}) \quad \frac{env_P, h \vdash \langle o, e\{\mathsf{continue}\ k/k : e\}\rangle \xrightarrow{\alpha} v \dashv env'_P, h'}{env_P, h \vdash \langle o, k : e\rangle \xrightarrow{\alpha} v \dashv env'_P, h'}$$

Table 2.7: Big-step semantics for control structures

Method Calls

We define the function update below to update the heap and parameter environment with the updated bindings after a method call. The function is used in the ($\textsc{Call}_\text{B}$) rule in Table 2.8 to improve readability.

$$\mathsf{update}(v, v_{arg}, h, env_P, o) = \begin{cases} h, env_P[x \mapsto v] & \text{if } v_{arg} = x \\ h[o.f \mapsto v], env_P & \text{if } v_{arg} = f \\ h, env_P & \text{otherwise} \end{cases}$$

To handle destructive reads of linear values as method parameters, methods are call-by-reference. The ($\textsc{Call}_\text{B}$) rule constructs a parameter environment using the method parameters, and execute the method body with the active object set to the callee reference. Furthermore, the subtyping relation is used on the heap when extracting parallel usages for use in the parameters or as the callee. For example, the binding $f \mapsto \langle C[(\{m; \mathsf{end}\}|\{n; \mathsf{end}\}).u^s_3], env_f\rangle$ in the heap would allow a method call m on f.

$$(\textsc{Call}_{\mathrm{B}})\ \frac{\begin{array}{c} h'' <: h \qquad\qquad h''(o_r).\mathsf{usage} \xrightarrow{m} \mathcal{U} \\ t\ m(t_1 \to t_1'\ x_1, t_2 \to t_2'\ x_2)\ \{e\} \in h''(o_r).\mathsf{class.methods} \qquad h' <: h'''' \\ \{x_1 \mapsto v', x_2 \mapsto v''\}, h''[o_r.\mathsf{usage} \mapsto \mathcal{U}] \vdash \langle o_r, e\rangle \xrightarrow{\alpha} v \dashv \{x_1 \mapsto v^{(3)}, x_2 \mapsto v^{(4)}\}, h'''' \end{array}}{env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \xrightarrow{o_r.m\cdot\alpha} v \dashv env_P', h'}$$

where $v' = \mathsf{extract}(v_1, h'', env_P, o)$,
$v'' = \mathsf{extract}(v_2, h'', env_P, o)$,
$o_r = \mathsf{extract}(r, h'', env_P, o)$,
$h''', env_P'' = \mathsf{update}(v^{(3)}, v1, h'''', env_P, o)$, and
$h''''', env_P' = \mathsf{update}(v^{(4)}, v2, h''', env_P'', o)$.

Table 2.8: Big-step semantics for method calls

Fields, Parameters, and Objects

Finally, Table 2.9 shows the big-step semantics for operations on fields, parameters, and objects. Notice that in ($\textsc{LinRefP}_{\mathrm{B}}$) and ($\textsc{LinRefF}_{\mathrm{B}}$) the resulting environments have lost the binding of the reference. This illustrates how references are treated as linear, and can only be aliased through method calls, and not assignments. For terminated types, such as base values, the rules ($\textsc{UnRefP}_{\mathrm{B}}$) and ($\textsc{UnRefF}_{\mathrm{B}}$) the binding is not removed upon reading, as the type cannot evolve due to aliasing.

To access the type of a value in a heap, we use the $\mathsf{getType}$ function defined below:

$$\mathsf{getType}(v, h) = \begin{cases} \mathsf{void} & \text{if } v = \mathsf{unit} \\ \mathsf{bool} & \text{if } v \in \{\mathsf{true}, \mathsf{false}\} \\ \mathsf{null} & \text{if } v = \mathsf{null} \\ h(o).\mathsf{type} & \text{if } v = o \end{cases}$$

$$(\textsc{Asgn}_\text{B}) \quad \frac{env_P, h \vdash \langle o, e\rangle \xrightarrow{\alpha} v \dashv env'_P, h'}{env_P, h \vdash \langle o, f = e\rangle \xrightarrow{\alpha} \mathsf{unit} \dashv env'_P, h'[o.f \mapsto v]}$$

$$(\textsc{BVal}_\text{B}) \quad \frac{}{env_P, h \vdash \langle o, v\rangle \to v \dashv env_P, h} \; v \in \{\mathsf{null}, \mathsf{true}, \mathsf{false}, \mathsf{unit}\}$$

$$(\textsc{LinRefP}_\text{B}) \quad \frac{\neg\mathsf{term}(\mathsf{getType}(v, h)) \qquad env_P(x) = v}{env_P, h \vdash \langle o, x\rangle \to v \dashv env_P[x \mapsto \mathsf{null}], h}$$

$$(\textsc{LinRefF}_\text{B}) \quad \frac{\neg\mathsf{term}(\mathsf{getType}(v, h)) \qquad h(o).f = v}{env_P, h \vdash \langle o, f\rangle \to v \dashv env_P, h[o.f \mapsto \mathsf{null}]}$$

$$(\textsc{UnRefP}_\text{B}) \quad \frac{\mathsf{term}(\mathsf{getType}(v, h)) \qquad env_P(x) = v}{env_P, h \vdash \langle o, x\rangle \to v \dashv env_P, h}$$

$$(\textsc{UnRefF}_\text{B}) \quad \frac{\mathsf{term}(\mathsf{getType}(v, h)) \qquad h(o).f = v}{env_P, h \vdash \langle o, f\rangle \to v \dashv env_P, h}$$

$$(\textsc{New}_\text{B}) \quad \frac{o' \text{ fresh}}{env_P, h \vdash \langle o, \mathsf{new}\ C\rangle \to o' \dashv env_P, h[o' \mapsto (C[C.\mathsf{usage}], C.\mathsf{initvals})]}$$

Table 2.9: Big-step semantics for method calls

2.3.3 Small-Step Semantics

The transition rules in the small-step semantics are of the form $\langle env_S, h, e\rangle \overset{\alpha}{\Rightarrow} \langle env'_S, h', e'\rangle$ where env_S is a parameter stack, h is a heap, e is the expression being evaluated, and α is the method trace of the reduction. Furthermore, we define the small-step rules in terms of evaluation contexts, defined by the following syntax.

$$\mathcal{E} ::= [\_] \mid \mathcal{E}; e \mid f = \mathcal{E} \mid \mathsf{return}_{r.m(v_1,v_2)}\{\mathcal{E}\} \mid \mathsf{if}\ (\mathcal{E})\ \{e_1\}\ \mathsf{else}\ \{e_2\}$$

The evaluation context defines the order of operations, and the ($\textsc{Ctx}_\text{S}$) rule drives the evaluation according to the evaluation context. The remaining small-step rules handle the base-cases where the updates to both the environments and the expression actually occurs.

$$(\textsc{Ctx}_\text{S}) \quad \frac{\langle env_S, h, e\rangle \overset{\alpha}{\Rightarrow} \langle env'_S, h', e'\rangle}{\langle env_S, h, \mathcal{E}[e]\rangle \overset{\alpha}{\Rightarrow} \langle env'_S, h', \mathcal{E}[e']\rangle}$$

Control Structures

Table 2.10 shows the small-step rules for control structures. They are for the most part again unsurprising. However for the if-expressions, we see that the expressions are tagged with a reference r, which is the reference that was originally used for the method call. This is similar to how the switch-construct was handled in previous work on Mungo [BFG+20]. The reference tag is necessary since we need to know which reference should have its usage updated after the branch is chosen in the if-case. Furthermore, similar to the big-step semantics, we allow subtyping on the heap, to the correct binding of r.

$$\text{(SEQ-B}_\text{S}\text{)} \quad \frac{}{\langle env_S, h, v; e' \rangle \Rightarrow \langle env_S, h, e' \rangle}$$

$$\text{(IF-TRUE}_\text{S}\text{)} \quad \frac{env_S = env'_S \cdot (o, env_P) \quad h'' <: h \quad h''(o').\mathsf{usage} \xrightarrow{\mathsf{true}} \mathcal{U} \quad h' <: h''[o'.\mathsf{usage} \mapsto \mathcal{U}]}{\langle env_S, h, \mathsf{if}_r\ (\mathsf{true})\ \{e_1\}\ \mathsf{else}\ \{e_2\} \rangle \overset{o'.\mathsf{true}}{\Longrightarrow} \langle env_S, h', e_1 \rangle}$$

where $o' = \mathsf{extract}(r, h, env_P, o)$

$$\text{(IF-FALSE}_\text{S}\text{)} \quad \frac{env_S = env'_S \cdot (o, env_P) \quad h'' <: h \quad h''(o').\mathsf{usage} \xrightarrow{\mathsf{false}} \mathcal{U} \quad h' <: h''[o'.\mathsf{usage} \mapsto \mathcal{U}]}{\langle env_S, h, \mathsf{if}_r\ (\mathsf{false})\ \{e_1\}\ \mathsf{else}\ \{e_2\} \rangle \overset{o'.\mathsf{false}}{\Longrightarrow} \langle env_S, h', e_2 \rangle}$$

where $o' = \mathsf{extract}(r, h, env_P, o)$

$$\text{(LBL}_\text{S}\text{)} \quad \langle env_S, h, k : e \rangle \Rightarrow \langle env_S, h, e\{\mathsf{continue}\ k/k : e\} \rangle$$

Table 2.10: Small-step semantics for control structures

Method Calls

Table 2.11 shows the small-step rules for method calls. Here we see a major difference between the two semantics. In the small-step semantics, we introduce a new piece of syntax, the return-expression. The expression encapsulates the method body of the called method, and it is tagged with the original method call so that usages can be updated after the method call. Finally, we append the parameter bindings and current object context to the parameter stack environment and wrap the method body in a return evaluation context. In (RET-BS$_\text{S}$) we pop the stack as the method body has been fully evaluated, and evaluation returns to the caller. The subtyping on the heap, which was used in the call-rule for the big-step semantics is also allowed in the small-step semantics, but split over the two rules (CALL$_\text{S}$) and (RET-B$_\text{S}$).

$$(\textsc{Call}_S)\quad \frac{\begin{array}{c} env_S = env'_S \cdot (o, env_P) \quad h' <: h \quad h'(o_r).\mathsf{usage} \xrightarrow{m} \mathcal{U} \\ t\ m(t_1 \to t'_1\ x_1, t_2 \to t'_2\ x_2)\ \{e\} \in h'(o_r).\mathsf{class.methods} \end{array}}{\begin{array}{l} \langle env_S, h, r.m(v_1, v_2)\rangle \xRightarrow{o_r.m} \langle env_S \cdot (o_r, \{x_1 \mapsto v', x_2 \mapsto v''\}), \\ \qquad\qquad\qquad\quad h'[o_r.\mathsf{usage} \mapsto \mathcal{U}], \\ \qquad\qquad\qquad\quad \mathsf{return}_{r.m(v1,v2)}\{e\}\rangle \end{array}}$$

where $v' = \mathsf{extract}(v_1, h', env_P, o)$,
$v'' = \mathsf{extract}(v_2, h', env_P, o)$, and
$o_r = \mathsf{extract}(r, h', env_P, o)$

$$(\textsc{Ret-B}_S)\quad \frac{env_S = env'_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v', x_2 \mapsto v''\}) \quad h' <: h'''}{\langle env_S, h, \mathsf{return}_{r.m(v1,v2)}\{v\}\rangle \Rightarrow \langle env'_S \cdot (o, env'_P), h', v\rangle}$$

where $h'', env''_P = \mathsf{update}(v', v1, h, env_P, o)$ and
$h''', env'_P = \mathsf{update}(v'', v2, h'', env''_P, o)$

Table 2.11: Small-step semantics for method calls

Fields, Parameters, and Objects

Finally, we have the small-step rules for object operations. There are no surprises, and they are all similar to the previously defined big-step rules.

$$(\textsc{Asgn-B}_S)\quad \frac{env_S = env_S \cdot (o, env_P)}{\langle env_S, h, f = v\rangle \Rightarrow \langle env_S, h[o.f \mapsto v], \mathsf{unit}\rangle}$$

$$(\textsc{LinFld}_S)\quad \frac{env_S = env'_S \cdot (o, env_P) \quad h(o).f = v \quad \neg\mathsf{term}(\mathsf{getType}(v, h))}{\langle env_S, h, f\rangle \Rightarrow \langle env_S, h[o.f \mapsto \mathsf{null}], v\rangle}$$

$$(\textsc{LinPar}_S)\quad \frac{env_S = env'_S \cdot (o, env_P) \quad env_P(x) = v \quad \neg\mathsf{term}(\mathsf{getType}(v, h))}{\langle env_S, h, x\rangle \Rightarrow \langle env'_S \cdot (o, env_P[x \mapsto \mathsf{null}]), h, v\rangle}$$

$$(\textsc{UnFld}_S)\quad \frac{env_S = env'_S \cdot (o, env_P) \quad h(o).f = v \quad \mathsf{term}(\mathsf{getType}(v, h))}{\langle env_S, h, f\rangle \Rightarrow \langle env_S, h, v\rangle}$$

$$(\textsc{UnPar}_S)\quad \frac{env_S = env'_S \cdot (o, env_P) \quad env_P(x) = v \quad \mathsf{term}(\mathsf{getType}(v, h))}{\langle env_S, h, x\rangle \Rightarrow \langle env'_S \cdot (o, env_P), h, v\rangle}$$

$$(\textsc{New}_S)\quad \frac{o \text{ fresh}}{\langle env_S, h, \mathsf{new}\ C\rangle \Rightarrow \langle env_S, h[o \mapsto (C[C.\mathsf{usage}], C.\mathsf{initvals})], o\rangle}$$

Table 2.12: Small-step semantics for fields and objects

2.3.4 Equivalence of the Two Semantics

In this section, we present an equivalence result showing that our two semantics are equivalent for terminating programs, and will result in identical values and environments. This allows us to apply results shown for one semantics, to the other and express the desired properties for our language as naturally as possible, as we can choose the semantics that most naturally models the property.

We first show that we can extend expressions in the small-step semantics, and still evaluate the original expression. This is useful in the rule for showing equivalence between the two semantics, in the case for sequential expressions.

Lemma 2. If

$$\langle env_S, h, e\rangle \overset{\alpha}{\Rightarrow}^* \langle env'_S, h', v\rangle$$

then

$$\langle env_S, h, e; e'\rangle \overset{\alpha}{\Rightarrow}^* \langle env'_S, h', e'\rangle$$

Proof. By induction on the length of transition sequence k. The case for $k = 0$ is trivial since no transitions of length $k = 0$ exist. We assume it holds for a k and consider a transition sequence $\langle env_S, h, e\rangle \overset{\alpha}{\Rightarrow}^{k+1} \langle env_S, h'\rangle$. We now have to show $\langle env_S, h, e; e'\rangle \overset{\alpha}{\Rightarrow}^{k+1} \langle env_S, h', e'\rangle$. By applying the rule ($\textsc{Ctx}_\textsc{S}$) k times we see that we have $\langle env_S, h, e; e'\rangle \overset{\alpha}{\Rightarrow}^{k} \langle env_S, h', v; e'\rangle$. One further application of ($\textsc{Seq-B}_\textsc{S}$) shows that we have $\langle env_S, h, e; e'\rangle \overset{\alpha}{\Rightarrow}^{k+1} \langle env_S, h', e'\rangle$. □

We can now prove the lemma that a big-step transition can be matched by a small-step transition sequence.

Lemma 3 (Big-step equivalence)**.** Let e be a user-expression. If

$$env_P, h \vdash \langle o, e\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$$

then

$$\langle env_S \cdot (o, env_P), h, e\rangle \overset{\alpha}{\Rightarrow}^* \langle env_S \cdot (o, env'_P), h', v\rangle$$

Proof. By induction in the height of the derivation-tree. We show the case for method calls, as the big-step semantics and the small-step differs the most here. The remaining cases can be found in Appendix A.

Case ($\textsc{Call}_\textsc{B}$)**:** Assume $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \xrightarrow{o_r.m\cdot\alpha} v \dashv env'_P, h'$. By ($\textsc{Call}_\textsc{B}$) we have $h'' <: h$. Now let $o_r = \mathsf{extract}(r, h'', env_P, o)$, $v' = \mathsf{extract}(v_1, h'', env_P, o)$, and $v'' = \mathsf{extract}(v_2, h'', env_P, o)$. By ($\textsc{Call}_\textsc{B}$) we also have $\{x_1 \mapsto v', x_2 \mapsto v''\}, h''[o_r.\mathsf{usage} \mapsto \mathcal{U}] \vdash \langle o_r, e\rangle \xrightarrow{\alpha} v \dashv \{x_1 \mapsto v^{(3)}, x_2 \mapsto v^{(4)}\}, h'''$, where $h''', env''_P = \mathsf{update}(v^{(3)}, h''', env_P, o)$, and $h'''', env'_P = \mathsf{update}(v^{(4)}, h''', env''_P, o)$. Finally from ($\textsc{Call}_\textsc{B}$) we also know $h' <: h''''$. By IH we have $\langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v', x_2 \mapsto v''\}), h''[o_r.\mathsf{usage} \mapsto \mathcal{U}], e\rangle \overset{\alpha}{\Rightarrow}^* \langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v^{(3)}, x_2 \mapsto v^{(4)}\}), h'''', v\rangle$. By Lemma 4 we have $\langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v', x_2 \mapsto v''\}), h''[o_r.\mathsf{usage} \mapsto \mathcal{U}], \mathsf{return}_{r.m(v_1,v_2)}\{e\}\rangle \overset{\alpha}{\Rightarrow}^* \langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v^{(3)}, x_2 \mapsto v^{(4)}\}), h'''', \mathsf{return}_{r.m(v_1,v_2)}\{v\}\rangle$. With ($\textsc{Call}_\textsc{S}$) we can conclude $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \overset{o'.m}{\Longrightarrow} \langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v', x_2 \mapsto v''\}), h''[o_r.\mathsf{usage} \mapsto \mathcal{U}], \mathsf{return}_{r.m(v_1,v_2)}\{e\}\rangle$. Finally, using ($\textsc{Ret-B}_\textsc{S}$), we can conclude that $\langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v^{(3)}, x_2 \mapsto v^{(4)}\}), h''', \mathsf{return}_{r.m(v_1,v_2)}\{v\}\rangle \Rightarrow \langle env_S \cdot (o, env'_P), h', v\rangle$, hence in total we can conclude $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \overset{\alpha}{\Rightarrow}^* \langle env_S \cdot (o, env'_P), h', v\rangle$. □

We must now prove the opposite direction, that a terminating transition sequence in the small-step semantics can be matched by a big-step transition. First, we present a lemma stating that expressions are evaluated according to an evaluation context. This lemma lets us establish a correspondence between the evaluation of sub-expressions in the two sets of semantics.

Lemma 4.

$$\langle env_S, h, \mathcal{E}[e]\rangle \overset{\alpha}{\Rightarrow}^k \langle env'_S, h', \mathcal{E}[v]\rangle$$

iff

$$\langle env_S, h, e\rangle \overset{\alpha}{\Rightarrow}^k \langle env'_S, h', v\rangle.$$

Proof. By induction in the length k of the transition sequence. We show one direction of the proof, the other is similar.
Case k = 1**:** Assume $\langle env_S, h, e\rangle \overset{\alpha}{\Rightarrow} \langle env'_S, h', e'\rangle$. Then by (CTXS) we have $\langle env_S, h, \mathcal{E}[e]\rangle \overset{\alpha}{\Rightarrow} \langle env'_S, h', \mathcal{E}[e']\rangle$.

Case Inductive case**:** Assume lemma holds for $1 \leq i \leq k$ and show for $k+1$. By definition of $\Rightarrow^{k+1}$ we have that $\langle env_S, h, e\rangle \overset{\alpha'}{\Rightarrow} \langle env''_S, h'', e''\rangle \overset{\alpha''}{\Rightarrow}^k \langle env'_S, h', e'\rangle$ where $\alpha = \alpha' \cdot \alpha''$. As in the base case, we use (CTXS) to conclude $\langle env_S, h, \mathcal{E}[e]\rangle \overset{\alpha}{\Rightarrow} \langle env''_S, h'', \mathcal{E}[e'']\rangle$. By IH we have $\langle env''_S, h'', \mathcal{E}[e'']\rangle \overset{\alpha''}{\Rightarrow}^k \langle env'_S, h', \mathcal{E}[e']\rangle$ hence combined we have $\langle env_S, h, \mathcal{E}[e]\rangle \overset{\alpha}{\Rightarrow}^{k+1} \langle env'_S, h', \mathcal{E}[e']\rangle$ □

Next, we present a lemma stating that a reduction of a user-expression only changes the active object of the stack.

Lemma 5. Let e be a user-expression. If

$$\langle env_S \cdot (o, env_P), h, e\rangle \overset{\alpha}{\Rightarrow}^* \langle env'_S \cdot (o', env'_P), h', v\rangle$$

then $env_S = env'_S$ and $o = o'$.

Proof. By induction in the length of the transition sequence. No transitions exist for case $k = 0$. Assume true for k and show for $k+1$. The expression can be written as

$$\langle env_S \cdot (o, env_P), h, e\rangle \overset{\alpha}{\Rightarrow} \langle env''_S \cdot (o'', env''_P), h'', e'\rangle \overset{\alpha}{\Rightarrow}^k \langle env'_S \cdot (o', env'_P), h', v\rangle$$

By considering the rules that could have been used for transition $\langle env_S \cdot (o, env_P), h, e\rangle \overset{\alpha}{\Rightarrow} \langle env''_S \cdot (o'', env''_P), h'', e'\rangle$ we see that in the small-step semantics, except the rules for method calls, we only change the parameter environment in the top element of the stack. In the rules for method calls, an element is appended to the stack, but the expression is wrapped in a return statement, hence the top element will be removed again, when evaluating the return value. □

We can now prove the opposite direction of our equivalence result, namely that a terminating transition sequence in the small-step semantics can be matched by a single transition in the big-step semantics.

Lemma 6 (Small-step equivalence)**.** Let e be a user-expression. If

$$\langle env_S \cdot (o, env_P), h, e\rangle \overset{\alpha}{\Rightarrow}^* \langle env_S \cdot (o, env'_P), h', v\rangle$$

then

$$env_P, h \vdash \langle o, e\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$$

Proof. By induction in the length of the transition sequence. We show the case for if-expressions. The remaining cases are shown in Appendix A.
Case (CTXS) - If**:** Assume

$$\langle env_S \cdot (o, env_P), h, \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \overset{\alpha}{\Rightarrow}^k \langle env_S \cdot (o, env'_P), h', v\rangle$$

We can rewrite this as follows

$$\langle env_S \cdot (o, env_P), h, \mathsf{if}_r\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle$$
$$\stackrel{\alpha'}{\Longrightarrow}{}^{k_1} \langle env_S \cdot (o, env''_P), h'', \mathsf{if}_r\ (v')\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle$$
$$\stackrel{\alpha''}{\Longrightarrow}{}^{k_2} \langle env_S \cdot (o, env'_P), h', v\rangle$$

Lemma 4 tell us that $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \stackrel{\alpha'}{\Rightarrow}{}^{k_1} \langle env_S \cdot (o, env''_P), h'', v'\rangle$ where $v' \in \{\mathsf{true}, \mathsf{false}\}$. From the IH we have $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \to v' \dashv env''_P, h''$. We now consider the case where $v' = \mathsf{true}$. The case for false is similar hence it will not be presented. When $v' = \mathsf{true}$ we know the following:

$$\langle env_S \cdot (o, env''_P), h'', \mathsf{if}\ (\mathsf{true})\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \stackrel{o'.\mathsf{true}}{\Longrightarrow} \langle env_S \cdot (o, env''_P), h''', e_1\rangle$$
$$\stackrel{\alpha'''}{\Longrightarrow}{}^{k_2-1} \langle env_S \cdot (o, env'_P), h', v\rangle$$

where $h'''' <: h''$, $h''''(o'.\mathsf{usage}) \xrightarrow{\mathsf{true}} \mathcal{U}$, and $h''' <: h'''[o'.\mathsf{usage} \mapsto \mathcal{U}]$. From the IH we have that $env''_P, h''' \vdash \langle o, e_1\rangle \xrightarrow{\alpha'''} v \dashv env'_P, h'$ and with rule (IfTrueB) we can now conclude $env_P, h \vdash \langle o, \mathsf{if}\ (e)\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \xrightarrow{\alpha' \cdot o'.\mathsf{true} \cdot \alpha'''} v \dashv env'_P, h'$. □

We now have both directions of equivalence, hence we can state our final equivalence theorem, stating that for terminating programs the two semantics are equivalent.

Theorem 1 (Equivalence)**.** Let e be a user-expression.

$$\langle env_S \cdot (o, env_P), h, e\rangle \stackrel{\alpha}{\Rightarrow}{}^{*} \langle env_S \cdot (o, env'_P), h', v\rangle$$

iff

$$env_P, h \vdash \langle o, e\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$$

Proof. Direct consequence of Lemma 3 and Lemma 6. □

Example 3. To illustrate the equivalence result, consider the expression $f = \mathsf{true}; x.m(\mathsf{true}, \mathsf{unit})$ where m is declared as $\mathsf{void}\ m(\mathsf{bool} \to \mathsf{bool}\ x_1, \mathsf{void} \to \mathsf{void}\ x_2)\{f' = x_1\}$. We show that evaluating the expression in the small-step and big-step semantics using the same initial environments, will result in the same terminal environments. The heap contains bindings of two objects, the active object o, and the object bound to the parameter x:

$$h = \{o \mapsto \langle C[\mathcal{U}], \{f \mapsto \mathsf{false}\}\rangle, o_x \mapsto \langle C'[\{m; w\}], \{f' \mapsto \mathsf{false}\}\rangle\}$$

The current parameter bindings contain the object o_x and a base value unit.

$$env_P = \{x \mapsto o_x, x' \mapsto \mathsf{unit}\}$$

Evaluating the expression in the small-step semantics results in the following transition sequence:

$$\begin{aligned}
&\langle h, (o, env_P), f = \mathsf{true}; x.m(\mathsf{true}, \mathsf{unit})\rangle \\
&\quad \Rightarrow \langle h[o.f \mapsto \mathsf{true}], (o, env_P), x.m(\mathsf{true}, \mathsf{unit})\rangle \\
&\quad \stackrel{o_x.m}{\Longrightarrow} \langle h[o.f \mapsto \mathsf{true}], (o, env_P) \cdot (o_x, \{x_1 \mapsto \mathsf{true}, \mathsf{unit}\}), \mathsf{return}_{x.m(\mathsf{true},\mathsf{unit})}\{f' = x_1\}\}\rangle \\
&\quad \Rightarrow \langle h[o.f \mapsto \mathsf{true}, o_x.f' \mapsto \mathsf{true}], (o, env_P) \cdot (o_x, \{x_1 \mapsto \mathsf{true}, \mathsf{unit}\}), \mathsf{return}_{x.m(\mathsf{true},\mathsf{unit})}\{\mathsf{unit}\}\rangle \\
&\quad \Rightarrow \langle h[o.f \mapsto \mathsf{true}, o_x.f' \mapsto \mathsf{true}, o_x.\mathsf{usage} \mapsto w], (o, env_P), \mathsf{unit}\rangle
\end{aligned}$$

The big-step derivation is shown in Figure 2.4, with some conditions omitted for simplicity. We see that the resulting environments for both semantics are:

$$h' = h[o.f \mapsto \mathsf{true}, o_x.f' \mapsto \mathsf{true}, o_x.\mathsf{usage} \mapsto w]$$

and

$$env'_P = env_P$$

2.3.5 Initial Configurations

So far we have discussed the execution of a program with both the big-step and small-step semantics. However, we have only discussed the execution of an already initialised configuration. In this section, we describe how the program is started from the definition $\vec{D}$.

Inspired by the `main` method of Java, and similar concepts in other object-oriented languages, where the `main` method defines the entrypoint to the program, Mungo assumes the existence of a class $\mathsf{class\ Main}\{\mathcal{U}, \vec{F}, \vec{M}\} \in \vec{D}$ where:

1. $\mathcal{U} = \{\mathsf{main; end}\}^\epsilon$
2. $\vec{M} = \{\mathsf{void\ main(void} \to \mathsf{void}\ x_1, \mathsf{void} \to \mathsf{void}\ x_2)\{e\}\}$

We can see from the usage that after evaluating the main method, the usage $\mathcal{U}$ is terminated, and the program is finished. We can now define the initial configurations for both the big-step and small-step semantics.

The heap should only contain information about the object of the class Main, hence we can immediately define $h = \{o_{\mathsf{main}} \mapsto \langle \mathsf{Main}[\mathcal{U}], \vec{F}.\mathsf{initvals}\rangle\}$. Similarly, as the parameters to the method are both of type void, we define $env_P = \{x_1 \mapsto \mathsf{unit}, x_2 \mapsto \mathsf{unit}\}$. Finally, the active object is o_{main}. With all initial environments defined, we can conclude that the initial configurations for the big-step semantics is $env_P, h \vdash \langle o_{\mathsf{main}}, e\rangle$, and the small-step configuration is $\langle (o_{\mathsf{main}}, env_P), h, e\rangle$.

$$\frac{\dfrac{h, env_P \vdash \langle o, f = \mathsf{true} \rangle \to \mathsf{unit} \dashv h[o.f \mapsto \mathsf{true}], env_P}{h, env_P \vdash \langle o, f = \mathsf{true}; x.m(\mathsf{true}, \mathsf{unit}) \rangle \xrightarrow{o_x.m} \mathsf{unit} \dashv h[o.f \mapsto \mathsf{true}, o_x.f' \mapsto \mathsf{true}, o_x.\mathsf{usage} \mapsto w], env_P} \quad \dfrac{\dfrac{h[o.f \mapsto \mathsf{true}], env_P \vdash \langle o, x.m(\mathsf{true}, \mathsf{unit}) \rangle}{\xrightarrow{o_x.m} \mathsf{unit} \dashv h[o.f \mapsto \mathsf{true}, o_x.f' \mapsto \mathsf{true}], env_P} \quad \dfrac{h[o.f \mapsto \mathsf{true}], \{x_1 \mapsto \mathsf{true}, x_2 \mapsto \mathsf{unit}\} \vdash \langle o_x, f' = x_1 \rangle}{\to \mathsf{unit} \dashv h[o.f \mapsto \mathsf{true}, o_x.f' \mapsto \mathsf{true}], \{x_1 \mathsf{true}, x_2 \mathsf{unit}\}}}$$

Figure 2.4: Big-step derivation of expression in Example 3

Chapter 3

The Type System and its Properties

In this chapter, the type system is introduced along with its properties. The type system is a central contribution of this project. It is capable of reasoning about certain forms of aliasing through the parallel usage construct that in addition, allows usages to be declared more efficiently. Two important results are shown for the type system and the semantics: protocol fidelity which tells us that usage declarations are followed in the semantics, and soundness. The soundness result is composed of a safety result, which specifies that the type system is an over-approximation of the semantics, and progress which tells us well-typed programs do not get stuck and by extension that null-dereferencing does not occur.

3.1 Type System

The type system is inspired by the Mungo type system presented by Bravetti et. al [BFG+20]. It uses a modular approach, where each class can be checked independently. It deviates from the Mungo type system by dropping the linearity requirement.

We start this section with an overview of the type system, before presenting the type judgments.

3.1.1 Type Checking with Usages

A goal with the type system is to remain modular so that classes can be type checked in isolation. This modularity is relatively simple to achieve when using a linear type system, as changes in the fields of one object cannot affect fields in another. In this type system, we wish to allow aliasing, while still preserving modularity. To accomplish this, we allow aliasing to be controlled with parallel usages, where we know that the two constituents of the usage do not interfere. This means that we introduce a limited form of aliasing, where aliasing can only be achieved through method calls, and we still perform destructive reads on linear values.

In a type system without behavioural types, it is often sufficient with a single pass over each method in a class to check if the static type information is consistent. In our setting, however, the types of the fields of a class are ever evolving, so the first time a method is called it may be consistent with the current field types, but not the next time. What the type system must ensure is, that when following the usage of a class, then every method invocation allowed by the usage

is consistent with the field types at that particular point in time. We do this by introducing typing judgments for typestates, and by following the usage, check the method invocations each time they appear in the usage. Only when a method name is mentioned in a branch usage, is the method body type checked. To ensure protocol completion, the type system requires that when reaching the end usage, the fields of the class are all terminated, so that no object is left with an unfinished protocol. Protocol completion along with destructive reads ensures that aliasing does not create problems. If one of the aliases is destructively read and stored in a field, then the original parallel usages cannot continue, and cannot reach the end usage, hence the caller of the method that performs the destructive read cannot be well-typed.

3.1.2 Environments

Type information for both fields and variables are collected in a typing environment Γ. Similar to the heap, we allow multiple bindings of the same values in the typing environment. Again, due to well-formedness of usage, only a single binding in Γ makes sense to extract in a given context, as each duplicate binding in Γ corresponds to a split of a parallel usage.

$$\Gamma : \mathcal{P}(\mathbf{Values} \rightharpoonup \mathbf{Types})$$

We write Γ, Γ' to denote the typing environment that contains the bindings of both Γ and Γ'. If Γ and Γ' both contains bindings of the same values, the bindings of Γ' must be placed in separated parts of Γ, such that no bindings are lost. For example in the typing environment $\Gamma = \{f \mapsto t_1\} * \{x \mapsto t_2\}$ we have $\Gamma, f \mapsto t = \{f \mapsto t_1\} * \{x \mapsto t_2, f \mapsto t\}$.

We retain the idea of having a field typing environment Φ from earlier work on Mungo [BFG+20], by doing so we ensure that Φ does not contain the parameters that are appended in rule (TCBr); however, from the definition below it is easy to see that for a field typing environment Φ, $\{\Phi\}$ is a Γ instance.

$$\Phi : \mathbf{FNames} \rightharpoonup \mathbf{Types}$$

To allow for behavioural separation we introduce subtyping on typing environments, as shown in Table 3.1. Rule (Concat) allows us to extract values from the typing environments. This is used to extract types that are not parallel or even types that are parallel but should not be further divided. For example if a method parameter requires type $C[(u_1 \mid u_2).u_3^s]$, then we can use (Concat) to extract this type from the field environment. Conversely, the rule can be used to combine separated environments back into a single typing environment, as long as there are no overlapping bindings in the two separated values. The rules (ParL) and (ParR) allow us to alias a value into two names, where each name only describes part of the object as described earlier in the context of behavioural separation. This is done by separating the typing environment into two, where the parallel constituent is placed in the newly created separated typing environment, and the remaining usage is kept in the original typing environment. To extract both sides of a parallel usage, one would apply each of the rules (ParL) and (ParR). The rules can also be used to merge separated environments back into a single environment. For example, this is used to remove aliases so that when only a single reference exists, the object can continue with usage u_3^s.

$$(\textsc{ParL})\ \frac{}{\Gamma, v \mapsto C[(u_1|u_2).u_3^s] <:> \ (\Gamma, v \mapsto C[(\circledcirc|u_2).u_3^s])* \{v \mapsto C[u_1^{s\cdot \mathtt{l}}]\}}$$

$$(\textsc{ParR})\ \frac{}{\Gamma, v \mapsto C[(u_1|u_2).u_3^s] <:> \ (\Gamma, v \mapsto C[(u_1|\circledcirc).u_3^s])* \{v \mapsto C[u_2^{s\cdot \mathtt{r}}]\}}$$

$$(\textsc{Concat})\ \frac{\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{(\Gamma_1, \Gamma_2) <:> \Gamma_1 * \Gamma_2} \qquad (\textsc{Id})\ \frac{}{\Gamma * \emptyset <:> \Gamma <:> \emptyset * \Gamma}$$

$$(\textsc{Trans})\ \frac{\Gamma <: \Gamma' \qquad \Gamma' <: \Gamma''}{\Gamma <: \Gamma''}$$

Table 3.1: Subtyping on Environments

To type check a parallel usage, we must ensure that the two branches do not interact with the same fields. This is achieved by introducing a split on field typing environments, similar to the field environment split defined in Section 2.3.1. The field typing environment split is defined in Table 3.2 and ensures that fields of objects with parallel usages are handled in a consistent way between the type system and the semantics.

$$\frac{\Phi = \Phi_1 \circ \Phi_2}{\Phi, f : t = \Phi_1, f : t \circ \Phi_2}$$

$$\frac{\Phi = \Phi_1 \circ \Phi_2}{\Phi, f : t = \Phi_1 \circ \Phi_2, f : t}$$

$$\frac{}{\Phi = \Phi \circ \emptyset = \emptyset \circ \Phi}$$

Table 3.2: Field typing environment split

3.1.3 Typing Rules

In this section we define the typing rules for the language defined in Chapter 2.

Program Declarations

In Table 3.3 the typing rules for program and class declarations are defined. Rule (TProg) tell us a program declaration is well-typed if all its class declarations are well-typed. A class declaration is well-typed if its usage well-types the class and results in a field typing environment that is terminated (TClass). The terminated predicate is the term predicate, introduced in Section 2.2.3, lifted to field typing environments such that Φ is terminated if $\forall f \in \mathrm{dom}(\Phi).\mathsf{term}(\Phi(f))$ By requiring that the final field typing environment Φ is terminated we ensure protocol completion where all fields of a terminated class are also terminated.

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

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

Table 3.3: Program well-typedness

Branch and Choice Usages

In Table 3.4 we define the typing rules for branch and choice usages. The rule (TCBr) is central to the type system, in the sense that this is the rule where updates to Φ happen. When type checking a branch usage, we must ensure that all allowed method invocations are consistent with the current field typing environment Φ, and that the resulting field typing environment Φ'' after type checking the method body is consistent with the remaining usage. As this rule uses the typing rules for expressions in the premise, we tie the type checking of classes to the type checking of the individual methods in the class. (TCCh) requires that both branches of a choice usage result in the same field typing environment Φ', which means that no matter what branch is chosen, finishing the protocol leaves the object in a consistent state. Finally, (TCEn) requires no further type derivation, and the final field typing environment has been reached.

$$\text{(TCBr)}\quad \frac{\forall i \in I.\exists \Phi''.\begin{cases} t_i\ m_i(t_i' \to t_i''\ x_i, t_i''' \to t_i''''\ x_i')\ \{e_i\} \in C.\mathsf{methods} \\ \{\Phi, x_i \mapsto t_i', x_i' \mapsto t_i'''\} \vdash e_i : t_i \triangleright \{\Phi'', x_i \mapsto t_i'', x_i' \mapsto t_i''''\} \\ \Theta; \Phi'' \vdash C[u_i^s] \triangleright \Phi' \end{cases}}{\Theta; \Phi \vdash C[\{m_i; u_i\}_{i \in I}^s] \triangleright \Phi'}$$

$$\text{(TCCh)}\quad \frac{\Theta; \Phi \vdash C[u_1^s] \triangleright \Phi' \qquad \Theta; \Phi \vdash C[u_2^s] \triangleright \Phi'}{\Theta; \Phi \vdash C[\langle u_1, u_2 \rangle^s] \triangleright \Phi'}$$

$$\text{(TCEn)}\quad \frac{}{\Theta; \Phi \vdash C[\mathsf{end}^s] \triangleright \Phi}$$

Table 3.4: Typing rules for branch and choice usages

Recursive Usages

Table 3.5 contains the typing rules for recursive usages with the rules (TCRec) and (TCVar). The idea for handling recursive usages is to keep track of the bound recursion variables along with field typing environment Φ at the time of binding. When a recursion variable X is bound with the usage $\mu X.u^s$ the field typing environment is saved in Θ. Any occurrences of the recursion variable in u are well-typed if the field typing environment is the same as the one originally saved in Θ. Recursive usages are used to describe iterative behaviour, so we must ensure that each iteration has the same initial environment, to ensure that the resulting environment is also the same. The guarantee that the resulting environment is the same, is what allows the type system to only check the recursive behaviour once, instead of arbitrarily many times. In (TCVar) the resulting field typing environment can be any field environment Φ'. The reason is, that when

we meet a recursion variable, the resulting environment must match the environment reached by *not* iterating. We illustrate this with an example.

Example 4. If we have a field typing environment Φ where calling the method m leads to field typing environment Φ', and calling n remains in Φ. Then following the usage $\mu X.\{m;\mathsf{end} \quad n; X\}$ should result in the field typing environment Φ'. With the usage $\{m;\mathsf{end}\}$, (TCBr) tells us that the resulting environment is Φ' but for $\{n; X\}$ we do not know what the resulting environment is. We solve this by allowing (TCVar) to choose any resulting environment, and since all branches in a usage must result in the same environment, then only the choice of Φ' would leave the class well-typed.

$$(\text{TCRec}) \quad \frac{\Theta, (X : \Phi); \Phi \vdash C[u^s] \triangleright \Phi'}{\Theta; \Phi \vdash C[\mu X.u^s] \triangleright \Phi'}$$

$$(\text{TCVar}) \quad \frac{}{\Theta, (X : \Phi); \Phi \vdash C[X^s] \triangleright \Phi'}$$

Table 3.5: Typing rules for recursive usages

Parallel Usages

The typing rules for parallel usages can be found in Table 3.6. Rule (TCPar) tell us that a parallel usage $(u_1 \mid u_2).u_3^s$ is well-typed if we can split the context $\Phi = \Phi_1 \circ \Phi_2$ such that u_1 is typable in Φ_1 and u_2 is typable Φ_2. In other words, we split the fields into two field environments that do not overlap and ensure that each local protocol is well-typed in the corresponding field typing environment. The rule (TCPlace) specifies that a placeholder $\circledcirc$ is well-typed. As placeholders do not occur in user-specified usages, this rule will not be encountered while type checking a program, but is useful when proving properties about the heap later on.

$$(\text{TCPar}) \quad \frac{\Theta; \Phi_1 \vdash C[u_1^{s \cdot 1}] \triangleright \Phi_1' \qquad \Theta; \Phi_2 \vdash C[u_2^{s \cdot \mathtt{r}}] \triangleright \Phi_2' \qquad \Theta; \Phi_1' \circ \Phi_2' \vdash C[u_3^s] \triangleright \Phi'}{\Theta; \Phi_1 \circ \Phi_2 \vdash C[(u_1 \mid u_2).u_3^s] \triangleright \Phi'}$$

$$(\text{TCPlace}) \quad \frac{}{\Theta; \Phi \vdash C[\circledcirc^s] \triangleright \Phi'}$$

Table 3.6: Typing rules for parallel usages

Typing judgments for expressions are of the form $\Gamma \vdash^{\Omega} e : t \triangleright \Gamma'$, where e is the expression currently being typed in environment Γ resulting in the type t and the environment Γ'. The recursion environment Ω plays the same role for expressions as Θ played for typestates. It is used for mapping labels to the initial typing environment. It is used in labelled expressions and is omitted from most rules, as it is left unchanged.

Method Calls

In Table 3.7 we define the typing rule for call expressions. Rule (Call) handles method calls on a reference r. Type checking a method call involves checking that parameters match the type given

in the method declaration, as well as ensuring that the usage of r allows a method transition. The rule is complicated by the fact that both r and the parameters v_1 and v_2 can have parallel usages, where constituents should be extracted first. We require that we can extract the types from Γ with the use of the subtyping relation, such that the usage of r has a m-transition and the types of the actual parameters matches those of the formal parameters. As methods do not distinguish types based on the current split of a usage, we use the inst predicate to instantiate the split-placeholder in usages $u^?$, to be the split of the actual parameters. Since we retain the split values with the instantiation, we can use the subtyping relation to merge the parallel constituents back into the original parallel structures in Γ'.

$$\mathsf{inst}(C[u^?], C[u^s], C[u'^?], C[u'^s]) \quad \mathsf{inst}(C[u^?], C[u^s], \bot, \bot)$$
$$\mathsf{inst}(b, b, b, b)$$

We allow some abuse of notation in the case that v_1 or v_2 are base values. In this case, we assume that $\Gamma(b)$ is the type of the base value and that we can separate them from the typing environment.

$$\text{(Call)} \quad \frac{\begin{array}{c}\Gamma'' * \{r : C[\mathcal{U}]\} * \{v_1 : t_{p_1}\} * \{v_2 : t_{p_2}\} <: \Gamma \\ \Gamma' <: \Gamma'' * \{r : C[\mathcal{U}']\} * \{v_1 : t'_{p_1}\} * \{v_2 : t'_{p_2}\} \\ t\ m(t_1 \to t'_1\ x_1, t_2 \to t'_2\ x_2)\ \{e\} \in C.\mathsf{methods} \quad \mathcal{U} \xrightarrow{m} \mathcal{U}' \\ \mathsf{inst}(t_1, t_{p_1}, t'_1, t'_{p_1}) \qquad \mathsf{inst}(t_2, t_{p_2}, t'_2, t'_{p_2})\end{array}}{\Gamma \vdash r.m(v_1, v_2) : t \triangleright \Gamma'}$$

Table 3.7: Typing rules for method call expressions

Control Structures

In Table 3.8 we define the typing rules for control structures. The rule (Seq) is standard in the sense that a sequential expression is well-typed if expression e_1 is well-typed and its resulting environment can be used to type expression e_2. However, we also require that the type of e_1 is terminated, such that objects with non-terminated usages are not lost. if-expressions are used to handle choice usages, hence the conditional expression is a method call on r, returning a bool value and where the usage of r is a choice usage. As with the (Call) rule, the usage of r can be a parallel usage, so we must allow subtyping on Γ'' to extract the choice usage for r. And again use subtyping to restore the original parallel structure before type checking the two branches of the if-expression. Similar to the condition in (TCCh), we require that the two branches have the same resulting typing environment. The typing rules (Lab) and (Con) checks continue-style loops by using a similar approach as for recursive usages. The initial environment is saved in Ω and then later when the label is encountered again in a $\mathsf{continue}$-expression, we check that the new environment matches the initial environment saved in Ω. Again we see that the typing rule for $\mathsf{continue}$-expression allows for an arbitrary resulting environment. This follows the same reasoning that was presented for recursion variables in (TCVar).

$$\text{(Seq)}\quad \frac{\Gamma \vdash e : t \triangleright \Gamma'' \qquad \Gamma'' \vdash e' : t' \triangleright \Gamma' \qquad \mathsf{term}(t)}{\Gamma \vdash e; e' : t' \triangleright \Gamma'}$$

$$\text{(If)}\quad \frac{\begin{array}{cc} \Gamma \vdash r.m(v_1, v_2) : \mathsf{bool} \triangleright \Gamma'' & \Gamma''' * \{r : C[\langle u_1, u_2 \rangle^s]\} <: \Gamma'' \\ \Gamma'''' <: \Gamma''' * \{r : C[u_1^s]\} & \Gamma''''' <: \Gamma''' * \{r : C[u_2^s]\} \\ \Gamma'''' \vdash e' : t \triangleright \Gamma' & \Gamma''''' \vdash e'' : t \triangleright \Gamma' \end{array}}{\Gamma \vdash \mathsf{if}\ (r.m(v_1, v_2))\ \{e'\}\ \mathsf{else}\ \{e''\} : t \triangleright \Gamma'}$$

$$\text{(Lab)}\quad \frac{\Omega' = \Omega, (k : \Gamma) \qquad \Gamma \vdash^{\Omega'} e : \mathsf{void} \triangleright \Gamma'}{\Gamma \vdash^{\Omega} k : e : \mathsf{void} \triangleright \Gamma'}$$

$$\text{(Con)}\quad \frac{\Omega = \Omega', (k : \Gamma)}{\Gamma \vdash^{\Omega} \mathsf{continue}\ k : \mathsf{void} \triangleright \Gamma'}$$

Table 3.8: Typing rules for control structure expressions

Fields, Parameters, and Objects

In Table 3.9 we define the typing rules for field, parameters, and object expressions. Rule (Fld) tells us that a field assignment is well-typed if the current type of the field is terminated, to not lose a non-terminated object reference. Furthermore, we require that the field type of f *agrees* with the type of the expression. The agree predicate is defined below.

$$\mathsf{agree}(C, C[\mathcal{U}]) \quad \mathsf{agree}(C, \bot) \quad \mathsf{agree}(b, b)$$

Base values are typed using rule (Val) that specifies the type of a base value is found by calling function btype defined below, which simply maps base values to their corresponding base type.

$$\mathsf{btype}(\mathsf{true}) = \mathsf{bool} \qquad \mathsf{btype}(\mathsf{false}) = \mathsf{bool}$$
$$\mathsf{btype}(\mathsf{unit}) = \mathsf{void} \qquad \mathsf{btype}(\mathsf{null}) = \bot$$

Rule (New) states that creating an instance of class C results in an object with type $C[C.\mathsf{usage}]$, which is the usage that is defined for the class in the program text. Rule (LinRef) is similar to destructive reads in a linear type system since we overwrite the reference with $\bot$ after reading it in order to avoid aliasing. This is required, since we treat each alias of an object as a linear value, hence reading the alias should be destructive, such that we only introduce new aliases through method calls. Rule (UnRef) handles reads of terminated fields in which case the reference is not overwritten in Γ. If f is mapped to a terminated object, then aliasing cannot cause interference, as no operations that alter the type can be performed on the value.

$$\text{(Fld)}\quad \frac{\Gamma \vdash e : t \triangleright \Gamma', f \mapsto t' \qquad \mathsf{agree}(C.\mathsf{fields}(f), t) \qquad \mathsf{term}(t')}{\Gamma \vdash f = e : \mathsf{void} \triangleright \Gamma', f \mapsto t}$$

where C is the class we are checking

$$\text{(Val)}\quad \frac{\mathsf{btype}(b) = t}{\Gamma \vdash b : t \triangleright \Gamma}$$

$$\text{(LinRef)}\quad \frac{\neg\mathsf{term}(t)}{\Gamma, r \mapsto t \vdash r : t \triangleright \Gamma, r \mapsto \bot}$$

$$\text{(UnRef)}\quad \frac{\mathsf{term}(t)}{\Gamma, r \mapsto t \vdash r : t \triangleright \Gamma, r \mapsto t}$$

$$\text{(New)}\quad \frac{}{\Gamma \vdash \mathsf{new}\ C : C[C.\mathsf{usage}] \triangleright \Gamma}$$

Table 3.9: Typing rules for field and object expressions

Example 5. We present two type derivations of the expressions $f = \mathsf{true}$ and $x.m(\mathsf{true}, \mathsf{unit})$ in Figure 3.1. What we see is that the subtyping on the typing environment rule allows the type system to use behavioural separation in method calls. The (Call) rule is a good example of how the type system is invariant to the amount of nested parallel usages, as the subtyping relation allows the type system to extract the specific component of the usage, no matter how deeply nested in a parallel usage it is. So in a sense, the type system requires certain behaviour from the usage, rather than a specific structure.

3.1.4 Slack in the Type System

As previously discussed, a class is well-typed if following the usage does not lead to problems. This introduces some slack into the type system, as usages are overapproximations of the method traces of the program. Usages describe *all* allowed method sequences of an object, while only requiring the program text to follow one specific method sequence. On one hand, this introduces a situation where the type system disallows programs that will never go wrong, as we will illustrate shortly with an example. On the other hand, this is what allows the type system to remain modular with respect to classes, as we check all method sequences when type checking the classes, which means that *any* method sequences encountered in other classes are guaranteed to be well-typed if they conform to the usage. We illustrate the slack with an example.

Example 6. Consider the class `Err` in Listing 3.1. We see that calling the method `good` on a newly created object would not cause any problems while calling `bad` would result in null-dereferencing on the field `f`. As the usage specifies that calling `bad` is allowed the class is not well-typed. In spite of the class not being well-typed, the `main` method in the `Main` class would not encounter problems at runtime, as the non-erroneous branch is chosen. Removing `bad; end` from the usage on line 2 would make the class well-typed, and the run-time behaviour will be exactly the same.

$$\{m; w\} \xrightarrow{m} w$$

$$\dfrac{\dfrac{}{\{x : C[(\circledcirc \mid \{n; w'\}).u^{\epsilon}]\} * \{x : C[\{m; w\}^{1}]\} <: \Gamma}\,(\textsc{Parl}) \qquad \dfrac{}{\Gamma' <: \{x : C[(\circledcirc \mid \{n; w'\}).u^{\epsilon}]\} * \{x : C[w^{1}]\}}\,(\textsc{Parl})}{\underbrace{\{x : C[(\{m; w\} \mid \{n; w'\}).u^{\epsilon}]\}}_{\Gamma} \vdash x.m(\mathsf{true}, \mathsf{unit}) : \mathsf{void} \triangleright \underbrace{\{x : C[(w \mid \{n; w'\}).u^{\epsilon}]\}}_{\Gamma'}}\,(\textsc{Call})$$

$$\dfrac{\dfrac{\dfrac{}{\{f : \mathsf{bool}\} \vdash \mathsf{true} : \mathsf{bool} \triangleright \emptyset, f : \mathsf{bool}}\,(\textsc{Val}) \quad \mathsf{btype}(\mathsf{true}) = \mathsf{bool} \quad \mathsf{agree}(C.\mathsf{fields}(f), \mathsf{bool}) \quad \mathsf{term}(\mathsf{bool})}{\{f : \mathsf{bool}\} \vdash f = \mathsf{true} \triangleright \{f : \mathsf{bool}\}}\,(\textsc{Fld})$$

Figure 3.1: Type derivations for Example 5

```
1  class Err {
2      U = {good; end bad; end}
3
4      LinC f
5
6      void good(void → void x1, void → void x2) { unit }
7
8      void bad(void → void x1, void → void x2) {
9          f.m(unit, unit)
10     }
11 }
12 class Main {
13     U = {main; end}
14
15     Err e
16
17     void main(void → void x1, void → void x2) {
18         e = new Err;
19         e.good(unit, unit)
20     }
21 }
```

Listing 3.1: A program that is not well-typed, but does not go wrong

In the following sections, we explore properties of both the semantics and the type system, and the correspondence between the two.

3.2 Protocol Fidelity

As usages form the basis for our type system, it is imperative to guarantee that usages are respected in the semantics. That is, method calls in the semantics should follow the defined usages of classes. In this section we present results for protocol fidelity, showing that operations performed on objects follow the usage defined by their class.

Usages describe a temporal property of what operations are available at a given time. As such, properties regarding usages are most naturally expressed using the small-step semantics. First, we define two properties, based on the method traces defined in Section 2.3, stating that method calls and branch selections follow the usages of a given object.

Lemma 7. If

$$\langle env_S, h, e\rangle \xRightarrow{o.m} \langle env'_S, h', e'\rangle$$

then there exists a $h'' <: h$ and a usage $\mathcal{U}$ such that

$$h''(o).\mathsf{usage} \xrightarrow{m} \mathcal{U}$$

Proof. Case analysis on the small-step semantics rules. We see that $\xRightarrow{o.m}$ only happens in the ($\textsc{Call}_\textsc{s}$) rule where $h'' <: h$ and $h''(o).\mathsf{usage} \xrightarrow{m} \mathcal{U}$ is given in the premise. □

Lemma 8. If

$$\langle env_S, h, e\rangle \xRightarrow{o.v} \langle env'_S, h', e'\rangle$$

then there exists a $h'' <: h$ and a usage $\mathcal{U}$ such that

$$h''(o).\mathsf{usage} \xrightarrow{v} \mathcal{U}$$

where $v \in \{\mathsf{true}, \mathsf{false}\}$

Proof. Case analysis on the small-step semantics rules. We see that $\stackrel{o.v}{\Longrightarrow}$ only happens in the rules (IF-TRUE$_S$) and (IF-FALSE$_S$) rule where $h'' <: h$ and $h''(o).\mathsf{usage} \xrightarrow{v} \mathcal{U}$ is given in the premise. □

We can now use these two results to state our protocol fidelity theorem.

Theorem 2 (Protocol Fidelity)**.** If

$$\langle env_{S1}, h_1, e_1\rangle \stackrel{\alpha_1}{\Longrightarrow} \langle env_{S2}, h_2, e_2\rangle \stackrel{\alpha_2}{\Longrightarrow} \ldots \langle env_{Sk}, h_k, e_k\rangle$$

then $\forall 1 \leq i < k$ if $\alpha_i = o.m$ then there exists a heap $h' <: h_i$ and a usage $\mathcal{U}$ such that $h'(o).\mathsf{usage} \xrightarrow{m} \mathcal{U}$ otherwise if $\alpha_i = o.v$ then there exists a heap $h' <: h_i$ and a usage $\mathcal{U}$ such that $h'(o).\mathsf{usage} \xrightarrow{v} \mathcal{U}$ where $v \in \{\mathsf{true}, \mathsf{false}\}$.

Proof. Induction in the length of the transition sequence.
Case Base case $k = 2$**:** Here we have $\langle env_{S1}, h_1, e_1\rangle \stackrel{\alpha_1}{\Longrightarrow} \langle env_{S2}, h_2, e_2\rangle$. Assume $\alpha_1 = o.m$, then by Lemma 7 we have that $\exists h', \mathcal{U}$ s.t. $h'(o).\mathsf{usage} \xrightarrow{m} \mathcal{U}$. Now assume $\alpha_1 = o.v$ where $v \in \{\mathsf{true}, \mathsf{false}\}$, then by Lemma 8 we have $\exists h', \mathcal{U}$ s.t. $h'(o).\mathsf{usage} \xrightarrow{v} \mathcal{U}$.

Case Inductive step**:** Assume the lemma holds for $k \leq i$ and show for $i + 1$.

$$\langle env_{S1}, h_1, e_1\rangle \stackrel{\alpha_1}{\Longrightarrow} \ldots \langle env_{Si}, h_i, e_i\rangle \stackrel{\alpha_{i+1}}{\Longrightarrow} \langle env_{Si+1}, h_{i+1}, e_{i+1}\rangle$$

By IH we know that the lemma holds for $\langle env_{S1}, h_1, e_1\rangle \stackrel{\alpha_1}{\Longrightarrow} \ldots \langle env_{Si}, h_i, e_i\rangle$, so it only remains to show that it holds for $\langle env_{Si}, h_i, e_i\rangle \stackrel{\alpha_{i+1}}{\Longrightarrow} \langle env_{Si+1}, h_{i+1}, e_{i+1}\rangle$. This, however, follows the same structure as the base case. If $\alpha_{i+1} = o.m$ then it follows from Lemma 7 that $\exists h', \mathcal{U}$ s.t. $h'(o).\mathsf{usage} \xrightarrow{m} \mathcal{U}$ and if $\alpha_{i+1} = o.v$ where $v \in \{\mathsf{true}, \mathsf{true}\}$ it follows from Lemma 8 that $\exists h', \mathcal{U}$ s.t. $h'(o).\mathsf{usage} \xrightarrow{v} \mathcal{U}$. □

Protocol fidelity has only been shown for the small-step semantics. But from Theorem 1 we know that method traces in the two semantics are equivalent for terminating programs. Since big-step transitions can be matched by a small-step transition sequence, for which the properties hold, we must have that the order of operations for a big-step transition also follows the usages for the classes. With protocol fidelity now shown to hold, we omit writing the method traces on transitions in the remainder of the report.

3.3 Soundness

The next set of properties show that the type system is sound. The main results are one of *safety* stating that the type system is an over-approximation of the runtime semantics, and *progress* stating the well-typed programs will terminate without errors or loop forever.

In the proofs for soundness, we will need to show a correspondence between the type system and semantics. To that end, we will also have to reason in the few areas where the type system and semantics differs. The cases are for labelled expressions and method calls. For labelled expressions, the semantics unfolds the loop and executes the body multiple times whereas the

type system only type checks the body once. To establish the correspondence between the type system and semantics, we introduce a lemma stating the expression remains well-typed after unfolding.

Lemma 9 (Substitution)**.** If

$$\Gamma \vdash k : e : \mathsf{void} \triangleright \Gamma'$$

then

$$\Gamma \vdash e\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$$

Proof. Induction in the height of the derivation of $\Gamma \vdash k : e : \mathsf{void} \triangleright \Gamma'$. We present the case for continue expressions. The full proof is in Appendix A.
Case (Con)**:** We consider the case where $e = \mathsf{continue}\ k'$. There are two sub-cases. First we treat the case where $k' = k$. Assume $\Gamma \vdash^{\Omega} k : e : \mathsf{void} \triangleright \Gamma'$. Here we have that $e\{\mathsf{continue}\ k/k : e\} = k : e$, which is is well-typed due to our assumption. For the case where $k' \neq k$ it follows the same structure as the other base cases. From (Lab) we know $\Gamma \vdash^{\Omega'} \mathsf{continue}\ k' : \mathsf{void} \triangleright \Gamma'$. From (Con) we know that $\Omega'(k') = \Gamma$, but since $k' \neq k$ we must have that $\Omega(k') = \Gamma$. As no substitution happens, we can conclude $\Gamma \vdash^{\Omega} e\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$.

□

The next difference between the semantics and the type system arises in the handling of method calls. The type system remains modular, and trusts the method signature, as that method will already have been checked, or will be checked later with the rule (TCBr). We can do this, in the type system, since the field environment Γ only considers the fields of the active object, hence changes introduced to other objects during a method call does not affect Γ. In the semantics, however, the method body will be evaluated, potentially leading to even more method calls. The issue is that we must establish a correspondence between the type system and the active object, but also between the object that a method was called on, and a type environment that we only know exists (due to the class being well-typed). Two things must be guaranteed for method calls: (i) method calls only affect the fields of the active objects that are mentioned in the parameters or as the callee, and (ii) the objects in the heap can be well-typed with their current usage.

3.3.1 Reachable Objects

To ensure (i) we introduce a function $\mathsf{reach}(env_P, h, o)$, that given a parameter environment, a heap, and an object, returns all possible objects in the heap that can be accessed by o or through methods called on o. The objects are the fields of o, and recursively the objects that the fields can access themselves as well as the objects that can be accessed through the parameters.

$$\mathsf{reach}(env_P, h, o) = \mathrm{range}(env_P) \cup \mathsf{reach}(h, o) \cup \bigcup_{o' \in \mathrm{range}(env_P)} \mathsf{reach}(h, o')$$

$$\mathsf{reach}(h, o) = \mathrm{range}(h(o).\mathsf{fields}) \cup \bigcup_{o' \in \mathrm{range}(h(o).\mathsf{fields})} \mathsf{reach}(h, o')$$

Lemma 10. If $env_P, h \vdash \langle o, e\rangle \to v \dashv env'_P, h'$, then $\forall o' \in \mathrm{dom}(h).o' \notin \mathsf{reach}(env_P, h, o) \implies h(o') = h'(o')$

Proof. Induction in the height of the derivation tree of $env_P, h \vdash \langle o, e\rangle \to v \dashv env'_P, h'$. The full proof can be found in Appendix A. □

To ensure (ii) we introduce a judgment $\vdash h, o$ describing that a heap h is well-typed w.r.t an object o. The judgment requires that all reachable objects from o are well-typed with the current bindings of fields and the usage of the object.

$$\text{(WTH)} \quad \frac{\forall o' \in \mathsf{reach}(h,o).\exists \Phi, \Phi'.\begin{cases}\Phi \vdash^h h(o').\mathsf{fields} \\ \Phi \vdash h(o').\mathsf{class}[h(o').\mathsf{usage}] \triangleright \Phi'\end{cases}}{\vdash h, o}$$

3.3.2 Well-Typed Configurations

We now formally define what we have previously described as a correspondence between the semantics and the type system. The rules in Table 3.10 describe a well-typed configuration. In such a configuration we ensure that the types of the fields in the field typing environment are the same as the type in the field environment in the heap. Similarly for the parameters, their types must match in the heap and Γ. We also require that the typing environment domains matches the domain of the objects' fields and the parameter bindings. Finally, we use the previously defined rule (WTH) to ensure that the heap itself is well-typed.

$$\text{(WTC-S)} \quad \frac{\Gamma_1 \vdash env_P, h_1, o \qquad \Gamma_2 \vdash env_P, h_2, o}{\Gamma_1 * \Gamma_2 \vdash env_P, h_1 * h_2, o}$$

$$\text{(WTC-B)} \quad \frac{\begin{array}{c} h \neq h' * h'' \qquad \Gamma \neq \Gamma' * \Gamma'' \\ \Gamma \vdash^h env_P \qquad \Gamma \vdash^h h(o).\mathsf{fields} \qquad \vdash h, o \\ \mathrm{dom}(\Gamma) \setminus \{x_1, x_2\} = \mathrm{dom}(h(o).\mathsf{fields}) \end{array}}{\Gamma \vdash env_P, h, o}$$

$$\text{(WTP)} \quad \frac{env_P(x_1) \in \mathrm{dom}(h) \implies \Gamma(x_i) = \mathsf{getType}(env_P(x_i), h)}{\Gamma \vdash^h env_P}$$

$$\text{(WTF)} \quad \frac{\forall f \in env_f \qquad \mathsf{getType}(env_f(f), h) = \Gamma(f)}{\Gamma \vdash^h env_f}$$

Table 3.10: Well-typed configurations

Lemma 11 (Well-typed initial configuration)**.** Let $\vec{D}$ be a program with an initial configuration $env_P, h \vdash \langle o, e\rangle$. If $\vdash \vec{D}$ then $\exists \Gamma, \Gamma'.\Gamma \vdash e : t \triangleright \Gamma'$ and $\Gamma \vdash env_P, h, o$

Proof. Per definition of the initial configurations in Section 2.3.5 we know that $h = \{o \mapsto \langle \mathsf{Main}[\{\mathsf{main}; \mathsf{end}\}], \vec{F}.\mathsf{initvals}\rangle\}$ and $env_P = \{x_1 \mapsto \mathsf{unit}, x_2 \mapsto \mathsf{unit}\}$.

By (TCLASS) we know that $\emptyset; \vec{F}.\mathsf{inittypes} \vdash \mathsf{Main}[\{\mathsf{main}; \mathsf{end}\}] \triangleright \Phi$, and by (TCBR) we have $\vec{F}.\mathsf{inittypes}, x_1 \mapsto \mathsf{void}, x_2 \mapsto \mathsf{void} \vdash e : \mathsf{void} \triangleright \Phi'', x_1 \mapsto \mathsf{void}, x_2 \mapsto \mathsf{void}$. So let $\Gamma = \vec{F}.\mathsf{inittypes}, x_1 \mapsto \mathsf{void}, x_2 \mapsto \mathsf{void}$ and $\Gamma' = \Phi'', x_1 \mapsto \mathsf{void}, x_2 \mapsto \mathsf{void}$.

It remains to be shown that $\Gamma \vdash env_P, h, o$. We have that $\mathrm{dom}(\Gamma) = \mathrm{dom}(h(o).\mathsf{fields}) \cup \mathrm{dom}(env_P)$ since $\mathrm{dom}(\Gamma) = \mathrm{dom}(\vec{F}) \cup \{x_1, x_2\}$, $\mathrm{dom}(h) = \mathrm{dom}(\vec{F})$, $\mathrm{dom}(env_P) = \{x_1, x_2\}$. To satisfy (WTP) we notice that $\Gamma(x_1) = \mathsf{void} = \mathsf{getType}(\mathsf{unit}, h)$ as well as $\Gamma(x_2) = \mathsf{void} = \mathsf{getType}(\mathsf{unit}, h)$. The condition of $\forall f \in h(o).\mathsf{fields}\ .\ \mathsf{getType}(h(o).f, h) = \Gamma(f)$ follows from the definitions of $\vec{F}.\mathsf{initvals}$ and $\vec{F}.\mathsf{inittypes}$. Finally $\vdash h, o$ is trivially satisfied as $\mathsf{reach}(h, o) = \emptyset$. □

3.3.3 Type System Correspondence

The final thing to show before establishing the correspondence between the semantics and the type system is to show that behavioural separation in the semantics can be matched in the type system. We show this, by introducing the following lemma for the subtyping relations.

Lemma 12 (Subtyping correspondence)**.** If $\Gamma \vdash env_P, h, o$ then $\exists h'$ s.t. $h' <: h$ iff $\exists \Gamma'$ s.t. $\Gamma' <: \Gamma$ and $\Gamma' \vdash env_P, h', o$.

Proof. By induction in the height of the subtyping judgments $h' <: h$ and $\Gamma' <: \Gamma$. As most subtyping rules are bidirectional we prove both directions. We also prove both directions of the lemma itself. Here we only present the two directions of the (ParL) rule for subtyping in the heap. The remaining cases can be found in Appendix A.

Case (ParL)**:** We prove both directions of the rule. Assume $\Gamma \vdash env_P, h, o$ and $h' <: h$ was concluded with (ParL), hence $h' = h'', o' \mapsto (C[(\circledcirc \mid u_2).u_3^s], env'_f) * \{o' \mapsto (C[u_1^{s\cdot 1}], env''_f)\}$, $h = h'', o' \mapsto (C[(u_1 \mid u_2).u_3^s], env_f)$, and $env_f = env'_f \cdot env''_f$. If no f exists s.t. $h(o).f = o'$ then $\Gamma * \emptyset \vdash env_P, h', o$, and with (Id) we conclude $\Gamma * \emptyset <: \Gamma$. If f does exist, then we know that $\Gamma(f) = \mathsf{getType}(o', h) = C[(u_1 \mid u_2).u_3^s]$, hence $\Gamma = \Gamma'', f \mapsto C[(u_1 \mid u_2).u_3^s]$. By (ParL) we conclude $\Gamma' < \Gamma$ where $\Gamma' = \Gamma'', f \mapsto C[(\circledcirc \mid u_2).u_3^s] * \{f \mapsto C[u_1^{s\cdot 1}]\}$. Using (WTC-S) we conclude $\Gamma' \vdash env_P, h', o$. We now prove the opposite direction. Assume $\Gamma \vdash env_P, h, o$ and $h' <: h$ was concluded with (ParL), hence $h' = h'', o' \mapsto (C[(u_1 \mid u_2).u_3^s], env_f)$, $h = h'', o' \mapsto (C[(\circledcirc \mid u_2).u_3^s], env'_f) * \{o' \mapsto (C[u_1^{s\cdot 1}], env''_f)\}$, and $env_f = env'_f \cdot env''_f$. If $\Gamma \vdash env_P, h, o$ then by (WTC-S) we have $\Gamma = \Gamma_1 * \Gamma_2$ where $\Gamma_1 \vdash env_P, h'', o' \mapsto (C[(\circledcirc \mid u_2).u_3^s], env'_f), o$ and $\Gamma_2 \vdash env_P, \{o' \mapsto (C[u_1^{s\cdot 1}], env''_f)\}, o$. From (WTC-B) we have $\mathrm{dom}(\Gamma_2) \setminus \{x_1, x_2\} = \mathrm{dom}(h(o).\mathsf{fields})$, hence if no f exists s.t. $h(o).f = o'$ then $\mathrm{dom}(\Gamma_2) = \emptyset$, hence trivially we have $\Gamma_1 \vdash env_P, h', o$. Otherwise, if f exists, then we know $\mathsf{getType}(o', \{o' \mapsto C[(u_1^{s\cdot 1})], env''_f\}) = C[u_1^{s\cdot 1}] = \Gamma_2(f)$ and $\mathsf{getType}(o', h'', o' \mapsto C[(\circledcirc \mid u_2).u_3^s], env'_f) = C[(\circledcirc \mid u_2).u_3^s] = \Gamma_1(f)$, hence $\Gamma_2 = \{f \mapsto C[u_1^{s\cdot 1}]\}$ and $\Gamma_1 = \Gamma'', f \mapsto C[(\circledcirc \mid u_2).u_3^s]$. With (ParL) we can conclude $\Gamma' = \Gamma'', f \mapsto C[(u_1 \mid u_2).u_3^s]$ and finally $\Gamma' \vdash env_P, h', o$.

□

We now present our first soundness result, namely that of heap-safety, which ensures that the resulting environments after a big-step transitions remain consistent with the type system.

Theorem 3 (Heap safety)**.** If $env_P, h \vdash \langle o, e\rangle \to v \dashv env'_P, h'$, $\Gamma \vdash e : t \triangleright \Gamma'$, and $\Gamma \vdash env_P, h, o$ then $\Gamma' \vdash env'_P, h', o$ and $\mathsf{getType}(v, h') = t$

Proof. Induction in the height of the derivation tree of $env_P, h \vdash \langle o, e\rangle \to v \dashv env'_P, h'$. The proof can be found in Appendix A. □

The next result is that of progress, stating that a well-typed program does not get stuck. In a big-step semantics, this means showing that a well-typed program can perform a transition and thereby finish evaluating. However, as our big-step semantics only allows for terminating programs, while usages can describe infinite behaviour, we need the small-step semantics to describe infinite programs.

Definition 13 (Infinite Evaluations)**.** We say that a configuration is *infinitely evaluating* written $\langle env_S, h, e\rangle \Rightarrow^\omega$ if there exists a infinite transition sequence

$$\langle env_S, h, e\rangle \Rightarrow \langle env'_S, h', e'\rangle \Rightarrow \langle env''_S, h'', e''\rangle \Rightarrow \dots$$

Furthermore, we show some intuitive properties of infinitely evaluating programs. The first property states that a finite transition sequence followed by an infinite transition sequence is in itself an infinite sequence. The second property states, that an infinitely evaluating expression can be wrapped in an evaluation context, and remain infinitely evaluating.

Lemma 14. If $\langle env_S, h, e\rangle \Rightarrow^* \langle env'_S, h', e'\rangle$ and $\langle env'_S, h', e'\rangle \Rightarrow^\omega$ then $\langle env_S, h, e\rangle \Rightarrow^\omega$.

Proof. By induction in the length of the transition sequence $\langle env_S, h, e\rangle \Rightarrow^* \langle env'_S, h', e'\rangle$.
Case $k = 0$**:** With $k = 0$, no transition occurs, hence we must have $env'_S = env_S$, $h' = h$ and $e' = e$, hence we have directly that $\langle env_S, h, e\rangle \Rightarrow^\omega$.

Case $k = 1$**:** If $\langle env'_S, h', e'\rangle \Rightarrow^\omega$ then a transition sequence

$$\langle env'_S, h', e'\rangle \Rightarrow \langle env''_S, h'', e''\rangle \Rightarrow \langle env'''_S, h''', e'''\rangle \Rightarrow \dots$$

exists. Since $k = 1$ we have $\langle env_S, h, e\rangle \Rightarrow \langle env'_S, h', e'\rangle$. We must then have the transition sequence

$$\langle env_S, h, e\rangle \Rightarrow \langle env'_S, h', e'\rangle \Rightarrow \langle env''_S, h'', e''\rangle \Rightarrow \langle env'''_S, h''', e'''\rangle \Rightarrow \dots$$

hence by the definition of $\Rightarrow^\omega$ we have $\langle env_S, h, e\rangle \Rightarrow^\omega$.

Case Inductive argument**:** assume the lemma holds for k and show for $k+1$. If $\langle env_S, h, e\rangle \Rightarrow^{k+1} \langle env'_S, h', e'\rangle$ then by definition we have $\langle env_S, h, e\rangle \Rightarrow \langle env''_S, h'', e''\rangle \Rightarrow^k \langle env'_S, h', e'\rangle$. By IH we then have $\langle env''_S, h'', e''\rangle \Rightarrow^\omega$. Now we have $\langle env_S, h, e\rangle \Rightarrow \langle env''_S, h'', e''\rangle$ and $\langle env''_S, h'', e''\rangle \Rightarrow^\omega$, hence we can conclude by IH that $\langle env_S, h, e\rangle \Rightarrow^\omega$. □

Lemma 15. If $\langle env_S, h, e\rangle \Rightarrow^\omega$ then $\langle env_S, h, \mathcal{E}[e]\rangle \Rightarrow^\omega$.

Proof. If $\langle env_S, h, e\rangle \Rightarrow^\omega$ then by definition we have a transition sequence

$$\langle env_{S1}, h_1, e_1\rangle \Rightarrow \langle env_{S2}, h_2, e_2\rangle \Rightarrow \langle env_{S3}, h_3, e_3\rangle \Rightarrow \dots$$

Now let $i > 0$ be an arbitrary index in the transition sequence. We know from the transition sequence that $\langle env_{Si}, h_i, e'_i\rangle \Rightarrow \langle env_{Si+1}, h_{i+1}, e_{i+1}\rangle$. We can then conclude with (CTXS) that $\langle env_{Si}, h_i, \mathcal{E}[e'_i]\rangle \Rightarrow \langle env_{Si+1}, h_{i+1}, \mathcal{E}[e_{i+1}]\rangle$. As this holds for every index we have the transition sequence

$$\langle env_{S1}, h_1, \mathcal{E}[e_1]\rangle \Rightarrow \langle env_{S2}, h_2, \mathcal{E}[e_2]\rangle \Rightarrow \langle env_{S3}, h_3, \mathcal{E}[e_3]\rangle \Rightarrow \dots$$

□

Finally we can state the progress result that a well-typed program either terminates without errors or is infinitely evaluating.

Theorem 4 (Progress)**.** If $\Gamma \vdash e : t \triangleright \Gamma'$ and $\Gamma \vdash env_P, h, o$ then either $env_P, h \vdash \langle o, e\rangle \to v \dashv env'_P, h'$ or $\langle (o, env_P), h, e\rangle \Rightarrow^\omega$

Proof. Induction in the structure of e. We show the case for sequential composition. The full proof is in Appendix A.
Case (SEQ)**:** Assume $\Gamma \vdash e_1; e_2 : t \triangleright \Gamma'$ and $\Gamma \vdash env_P, h, o$. From (SEQ) we have $\Gamma \vdash e_1 : t' \triangleright \Gamma''$ and $\Gamma'' \vdash e_2 : t \triangleright \Gamma'$. By IH we have either $env_P, h \vdash \langle o, e_1\rangle \to v' env''_P, h''$ or $\langle (o, env_P), h, e_1\rangle \Rightarrow^\omega$ and either $env''_P, h'' \vdash \langle o, e_2\rangle \to v env'_P, h'$ or $\langle (o, env''_P), h'', e_2\rangle \Rightarrow^\omega$.

If $env_P, h \vdash \langle o, e_1\rangle \to v' \dashv env''_P, h''$ and $env''_P, h'' \vdash \langle o, e_2\rangle \to v \dashv env'_P, h'$, then using ($\text{SEQ}_\text{B}$) we conclude $env_P, h \vdash \langle o, e_1; e_2\rangle \to v \dashv env'_P, h'$.

If $\langle env_S \cdot (o, env_P), h, e_1 \rangle \Rightarrow^{\omega}$ then by Lemma 15 we have $\langle env_S \cdot (o, env_P), h, e_1; e_2 \rangle \Rightarrow^{\omega}$.

Finally if $env_P, h \vdash \langle o, e_1 \rangle \rightarrow v' \dashv env''_P, h''$ and $\langle env_S \cdot (o, env''_P), h'', e_2 \rangle \Rightarrow^{\omega}$. By Theorem 1 we have $\langle env_S \cdot (o, env_P), h, e_1 \rangle \Rightarrow^{*} \langle env_S \cdot (o, env''_P), h'', v' \rangle$. By Lemma 4 we have $\langle env_S \cdot (o, env_P), h, e_1; e_2 \rangle \Rightarrow^{*} \langle env_S \cdot (o, env''_P), h'', v'; e_2 \rangle$ and with (SEQ$_S$) we have $\langle env_S \cdot (o, env''_P), h'', v'; e_2 \rangle \Rightarrow \langle env_S \cdot (o, env''_P), h'', e_2 \rangle$. We now have $\langle env_S \cdot (o, env_P), h, e_1; e_2 \rangle \Rightarrow {}^{*}\langle env_S \cdot (o, env''_P), h'', e_2 \rangle \Rightarrow^{\omega}$ and by Lemma 14 conclude $\langle env_S \cdot (o, env_P), h, e_1; e_2 \rangle \Rightarrow^{\omega}$. □

With both progress and heap safety defined, we have shown that the type system is sound. From progress, we know that well-typed programs do not get stuck and from heap safety, we know that the types in the type system, correspond to the values in the semantics. Thus we can conclude that well-typed programs do not "go wrong", in the sense that the errors captured in the type system, for example, null dereferencing, cannot occur during the execution of a well-typed program.

Chapter 4

The Implemented Version of Mungo

In this chapter, we present our implementation of the Mungo programming language. Golovanov et al. presented a tool `mungoi`, which implemented a version of the Mungo language with support for usage inference [GJK20]. For the version of Mungo presented in this report, we created a tool `mungob` with support for behavioural separation. We display the functionality of the tool through examples in the Mungo language, that are type checked using the tool. The source code of `mungob`, written in Haskell, can be found at `https://mungotypesystem.github.io/MungoBehaviouralSeparation/`, which also includes full versions of the examples shown here.

The original work on Mungo by Kouzapas et. al resulted in the Mungo & StMungo toolchain [KDPG16], a custom Java-compiler with support for protocols written in the protocol language Scribble and translated into typestates with the tool StMungo (Scribble-To-Mungo). Only a subset of Java is supported by the toolchain, and unsupported program segments including external libraries are assumed to be well-typed. This is in contrast to `mungoi` which only implements the core calculus. The major advantage of the Mungo & StMungo toolchain is that Java is a production-ready language, while the core calculus of Mungo cannot be used for larger projects. The advantage of `mungoi` and `mungob` on the other hand, is that the whole language is formalised, and the programs are provably correct and are guaranteed to not exhibit the errors captured by the type system.

A major difference going from `mungoi` to `mungob` was the decision to include run-time semantics in the form of an interpreter. While `mungoi` only included the static phases of the compiler with parsing and type checking, `mungob` was extended with an interpreter and simple I/O operations. Including a runtime environment, allows us to write and run our example programs, motivating the addition of typestates to mainstream programming languages.

4.1 Implemented Mungo Language

The language expected by the implementation is similar to the syntax defined in Chapter 2, with minor changes to increase writability. In order to clearly express our examples, strings and integers have been added as base values in the language, along with the base types **`string`** and **`int`**, and operators for the new values. The typing rules for the operations on string and integers are trivial and are omitted from our presentation of the language.

Listing 4.1 shows the syntax of class declarations As can be seen on line 2, the syntax follows

the conventions of typestates $C[\mathcal{U}]$ rather than the class $C\{\mathcal{U}, \vec{F}, \vec{M}\}$ syntax defined in Section 2.2. Field and method declarations can be interleaved, and the $\bot$ type can be expressed with the **none** keyword.

```
1  // Class declaration with initial usage
2  class C [({setF2; {getF2; end}} | rec X.{ add; X, total; end}).end] {
3      // Field declaration
4      D f
5      // Method declarations
6      void setF(D[{use; end}] -> none x) { f = x }
7      D[{use; end}] getF() { f }
8
9      // Field declaration
10     int count
11     // Method declarations
12     void add(int x) { count = count + x }
13     int total() { count }
14 }
```

The example highlights some of the syntactic transformations the implementation employs, to heighten the writability of the language. In the method declarations, we see methods that specify zero or one parameter, rather than the two parameters from the formal syntax. One syntactic transformation is to assume parameters not specified are defined as void $\rightarrow$ void x. Another transformation to method parameters is that types $t\ x$ are transformed into $t \rightarrow t\ x$. This is useful for base types, whose types will not change during a method call.

4.2 Program Examples

In this section, we present small program examples written in the Mungo language and type checked and executed with `mungob`. We use the examples to describe features of the implementation, along with showcasing the kinds of behaviour that can be expressed with usages. A repository of program examples can be found on `https://mungotypesystem.github.io/MungoBehaviouralSeparation`, and includes the full versions of the examples presented below.

File Example

We return to a common example of protocols in programming language, namely the file example described in Chapter 1. On the website of the Mungo & StMungo toolchain, an implementation of the file protocol is shown [DKPG]. Here we present a version of the file example which reads a file one line at a time, while lines are available in the file. The protocol of a file object is illustrated in Figure 4.1.

```
1  class File [{open; rec X.{isEmpty; <{close; end}, {read; X}>}}] {
2      bool isEmpty() { ... }
3      string read() { ... }
4      void close() { ... }
5      void open() { ... }
6  }
```

As the implementation does not support interfacing with the file system, we only provide the skeleton of the `File` class, while providing the full implementation of the `main` class.

```
7  class Main [{main; end}] {
8      File f;
```

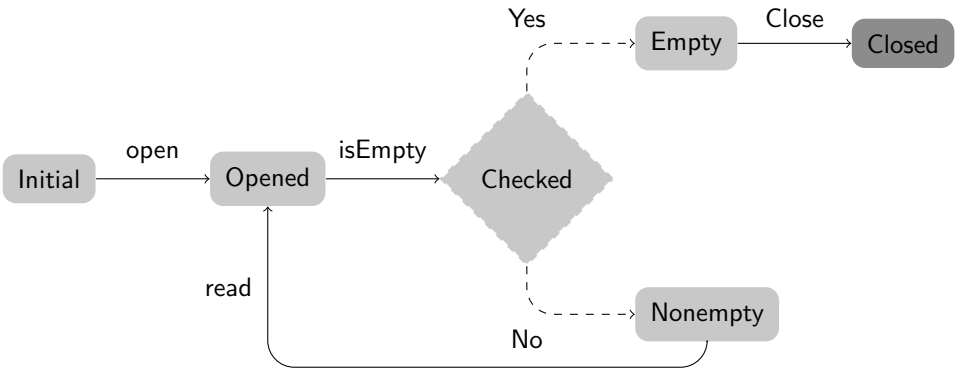

Figure 4.1: Illustration of File protocol

```
 9      string line;
10      void main() {
11          f = new File;
12          f.open();
13          loop: if (f.isEmpty()) {
14                  f.close();
15                } else {
16                   line = f.read();
17                   print(line);
18                   continue loop;
19                }
20      }
21  }
```

In the example repository, one can find an implementation of the `File` class that reads input from the terminal rather than from a file directly. This allows us to run the example above as follows:

```
$ mungob exampleprograms/file.mg < datafile.txt
  file
  with
  multiple
  lines
```

Any call to `isEmpty` before calling `open` would not be accepted by the type system. Similarly, calling `read` without calling `isEmpty` will also result in type errors.

Pair Example

We now present an example of the type of aliasing that is allowed in the presented type system. This example is based on a similar example presented by Militão et. al [MaAC10]. Consider a Pair-class with a left value and a right value. Both sides must be initialised before using the pair as a whole, but the two fields themselves are unrelated. We can model the initialisation of both sides as a parallel usage continued by the use of the combined pair. This is illustrated in the protocol in Figure 4.2 The shaded regions indicate parallel constituents and the shaded arrows indicate an implicit transition to multiple parallel states.

In Listing 4.1 the `Pair` class is shown. A pair contains two integers, that only after initialisation can be summed.

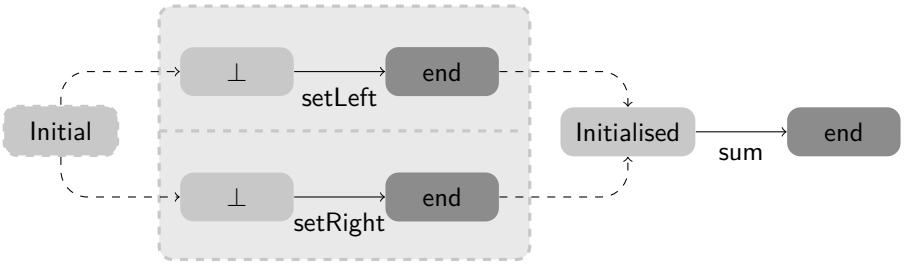

Figure 4.2: Protocol for `Pair` class

```
1  class Pair[({setLeft; end} | {setRight; end}).{sum; end}] {
2      int left
3      int right
4
5      void setLeft(int x) { left = x }
6      void setRight(int x) { right = x }
7      int sum() { left + right }
8  }
```

Listing 4.1: Pair class with parallel usage

Now consider the the class `PairUser` in Listing 4.2. the initialisation method takes as arguments two pairs, one where the left side can be initialised and one where the right side can be initialised.

```
1  class PairUser[{initialise; end}] {
2      void initialise(Pair[{setLeft; end}] -> Pair[end] l,
3                      Pair[{setRight; end}] -> Pair[end] r) {
4          l.setLeft(2);
5          r.setRight(5)
6      }
7  }
```

Listing 4.2: Class for initialising pairs

Finally, in the main class in Listing 4.3 we call the initialisation method with the same field as both parameters on line 12, and introduce aliases for f. The subtyping relation allows the uninitialised pair to be split into two objects, each matching the required type of the parameters of the initialisation method. After the method call, the type of the pair is updated to reflect the progression of both branches, and since both branches are end, the resulting usage is $\{\mathsf{sum}; \mathsf{end}\}$.

```
1  class main[{main; end}] {
2      Pair p
3      PairUser pu
4      int res
5
6      void main() {
7          //p : ⊥
8          p = new Pair;
9          //p : Pair[({setLeft; end} | {setRight; end}).{sum; end}]
10         pu = new PairUser;
```

```
11          // p : Pair[(◎ | ◎).{sum; end}], p: Pair[{setLeft; end}], p: Pair
                [{setRight; end}]
12          pu.initialise(p, p);
13          // p : Pair[(end | end).{sum; end}]
14          // p : Pair[{sum; end}]
15          res = p.sum();
16          // p : Pair[end]
17          print(res)
18      }
19  }
```

Listing 4.3: Main class that shows the use of aliasing

HouseController Example

We now return to the example of a house controller, as introduced in Chapter 1. We have discussed how in the previous work on Mungo, unrelated linear fields could lead to exponential size usages, and we have also discussed how this has been mitigated with behavioural separation. Here we present a `HouseController` class, type checked by `mungob`, making use of parallel usages to express unrelated operations. The language supports parallel usages with more than two usages in parallel, with the translation $(u_1 \mid u_2 \mid u_3).u_4 \triangleq (u_1 \mid (u_2 \mid u_3).\mathsf{end}).u_4$, generalised to any number of parallel usages. Figure 4.3 shows the protocol for the class.

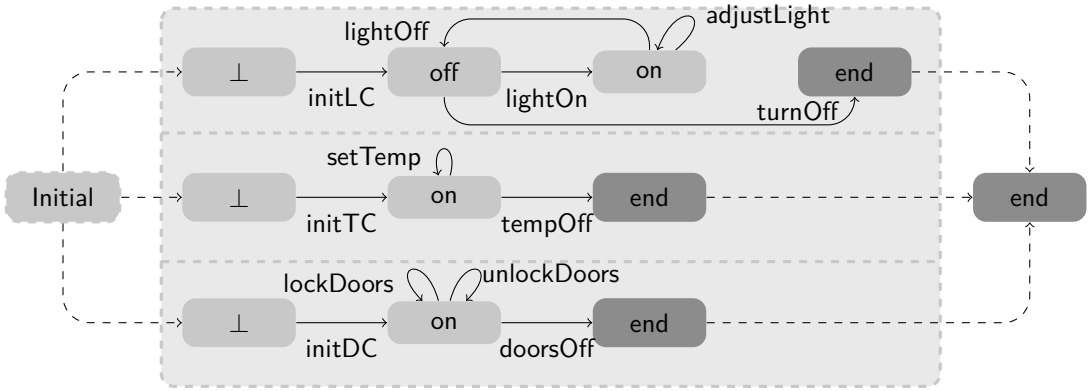

Figure 4.3: Protocol for `HouseController` class

The local protocol for each linear field can then be represented separately. Take for example the `LightController` class. The field is used in the methods `setTemp`, `initTC`, and `tempOff`, We can derive the protocol $\{\mathsf{initTempController}; \mu X.\{\mathsf{setTemperature}; X\ \mathsf{turnOff}; \mathsf{end}\}\}$. The same procedure is repeated for the remaining two fields, and combined into the parallel usage shown in Listing 4.4.

```
1  class HouseController [
2      ({initLightController;
3          rec X.{lightOn;
4              rec Y. {adjustLight; Y, lightOff; X}} turnOff; end}} |
5      {initTempController;
6          rec X. {setTemperature; X, turnOff; end}} |
7      {initDoorController;
```

```
8           rec X. {lockDoors; X, unlockDoors; X, turnOff; end}}).end
9       ] {
10
11      LightController lc
12      TempController tc
13      DoorController dc
14
15      void initLightController(){
16          lc = new LightController()
17      }
18
19      ...
20
21      void lightOn(){lc.on()}
22
23      void adjustLight(){lc.setIntensityHigh()}
24
25      void lightOff(){lc.off()}
26
27      ...
28  }
```

Listing 4.4: An example of the `HouseController` class in Mungo

Account Example

With this example, we introduce a syntactical transformation, that allows the programmer to specify sequential usages $u; u'^s$. Sequential behaviour is nothing new, the choice usage $\{m_i; w_i\}_{i \in I}$ describes a method call m *followed by* a protocol w. What makes the sequential usage special, is the fact that an object with type $C[u; u'^s]$ is treated as an object of type $C[u^s]$ and only after u^s has evolved to end, is the object treated as having type $C[u'^s]$.

Sequential usages are introduced with the syntactical transformation $u; u'^s \triangleq (u \mid \mathsf{end}).u'^s$. We see that only after u^s has evolved to end do we have $(\mathsf{end} \mid \mathsf{end}).u'^s \equiv u'^s$.

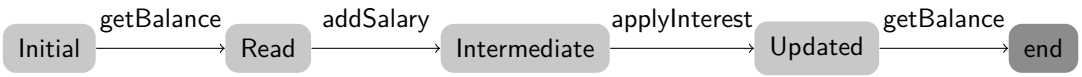

Figure 4.4: Protocol for `Account` class

Consider a class modelling a bank account with the methods getBalance, addSalary, and applyInterest. A simplified protocol for such a class is illustrated in Figure 4.4. To ensure that the balance is always consistent, getBalance cannot be called after addSalary but before applyInterest. The obvious way to model this protocol is {getBalance; {addSalary; {applyInterest; {getBalance; end}}}}. Instead we propose to use sequential usages, as illustrated in Listing 4.5.

```
1  class Account [
2           {getBalance; end};{addSalary; {applyInterest; end}};{getBalance;
                end}
3       ] {
4
5       int balance
6
```

```
7       int getBalance() { balance }
8       void addSalary() { balance = (balance + 16000) }
9       void applyInterest() { balance = (balance + 30) }
10  }
```

Listing 4.5: `Account` class with sequential usage

We introduce a second class, a class which performs operations on bank accounts. For simplicity, we illustrate this with a `Printer` class which simply prints the balance of a bank account. If we used the obvious usage to model the `Account` class, the `Printer` class would require two methods, as illustrated in Listing 4.6.

```
1   class Printer [rec X.{output; X, output2; X, finish; end}] {
2       int balance
3       void output(
4       Account[{getBalance; {addSalary; {applyInterest; {getBalance; end}}}}
            ->
5       Account[{addSalary; {applyInterest; {getBalance; end}}}] x]) {
6           balance = x.getBalance();
7           print(balance)
8       }
9
10      void output2(Account[{getBalance; end} -> Account[end] x]) {
11          balance = x.getBalance();
12          print(balance)
13      }
14  }
```

Listing 4.6: `Printer` class with duplicate methods

If we used the sequential usage instead we could use the `Printer` class illustrated in Listing 4.7, relying on the subtyping relation to update the usage of the `Account` object correctly.

```
1   class Printer [rec X.{output; X finish; end}] {
2       int balance
3       void output(Account[{getBalance; end}] -> Account[end] x) {
4           balance = x.getBalance();
5           print(balance)
6       }
7       void finish() { unit }
8   }
```

Listing 4.7: Simpler `Printer` class

Finally, we show the entry point for the program in Listing 4.8. Notice that on lines 10 and 16 we call the `output` method, even though the usage of `acc` is different for each call.

```
1   class main [{main; end}] {
2       Account acc
3       Printer p
4
5       void main() {
6           // acc: ⊥
7           acc = new Account;
```

```
8           // acc: {getBalance; end}; {addSalary; {applyInterest; end}}; {
                getBalance; end}
9           p = new Printer;
10          p.output(acc);
11          // acc: {addSalary; {applyInterest; end}}; {getBalance; end}
12          acc.addSalary();
13          // acc: {applyInterest; end}; {getBalance; end}
14          acc.applyInterest();
15          // acc: {getBalance; end}
16          p.output(acc);
17          // acc: end
18          p.finish()
19      }
20  }
```

Listing 4.8: `main` class of Account example

4.3 Runtime Complexity of Type Checking

In this section, we analyse the time complexity of the type checking algorithm. We present the algorithms for the interesting parts of the type checking process. The cases that are omitted are bounded by the same complexity class as the presented cases.

4.3.1 Type Checking Algorithm

The implementation follows the structure presented in Chapter 3 where classes are type checked by following the usage while keeping track of the current field bindings.

Function CheckUsage in Algorithm 1 implements the typing rules (TCBr) and (TCCh). It maintains a frontier queue of unexplored pairs of usages and Φ environments. It then checks all branch and choice usage transitions with respect to the associated Φ environment. The loop starting on line 7 corresponds to rule (TCBr) and the loop on line 16 implements rule (TCCh).

On line 9 in Algorithm 1 method bodies are type checked, and we turn to type checking of expressions instead of classes. An interesting case is that of type checking call expressions which are shown in Algorithm 2. It is interesting because it uses the subtyping relation to find the appropriate behavioural separation for the reference a method is called on and its arguments.

The main idea is to split parallel usages in order to find the usage that matches a parameter type or allows a particular method transition. For each value, we extract the necessary binding from the typing environment and continue the process for the remaining values in the remaining typing environment. The extraction process removes the binding from the original typing environment so that the same value cannot be extracted twice. With a parallel usage, this allows us to extract both sides of a parallel usage, but not the same side twice.

CheckCall constructs the environment split for typing the method call. It extracts environments for reference r, arguments v_1 and v_2 and the remaining environment containing the unused parts of the original Γ. This corresponds to the typing environment $\Gamma''' * \Gamma_{\mathsf{R}} * \Gamma_{\mathsf{v1}} * \Gamma_{\mathsf{v2}}$. CheckCall then proceeds to check that a method transition m is allowed on line 10 and that the types of the parameters and arguments match on line 10. Finally, the typing environment is updated with the resulting types and parallel usages are restored to their unsplit form. The function SplitReference is similar to SplitValue except it matches usages that allows a specified method transition instead of matching typestates.

Finally, we explore Algorithm 3 which implements the subtyping relation. The purpose of

Algorithm 1 Usage checking

```
1:  function CheckUsage(𝒰, C)
2:      explored ← ∅
3:      frontier ← {(initFields(C), 𝒰)}
4:      while frontier ≠ ∅ do
5:          (Φ,𝒰′) ← dequeue(frontier)
6:          explored ← explored ∪ (Φ,𝒰′)
7:          for all 𝒰′ —m→ 𝒰″ do
8:              if t m(t₁ → t′₁ x₁, t₂ → t′₂ x₂) ∈ C.methods then
9:                  if Φ, x₁ ↦ t₁, x₂ ↦ t₂ ⊢ e : t ⊣ Φ′, x₁ ↦ t′₁, x₂ ↦ t′₂ then
10:                     if (Φ′,𝒰″) ∉ explored then
11:                         frontier ← frontier ∪ (Φ′,𝒰″)
12:                 else
13:                     return false
14:             else
15:                 return false
16:         for all 𝒰′ —v→ 𝒰″ do
17:             if (Φ,𝒰″) ∉ explored then
18:                 frontier ← frontier ∪ (Φ,𝒰″)
19:     return true
```

the subtyping relation is to split parallel usages such that they match a particular target usage. For example, SplitValue takes the formal and actual parameter of a method and then checks that they match both in terms of class name and usage. If the actual parameter has a parallel usage then SplitValue splits the parallel usage until it finds a matching usage or returns an error if a correct usage cannot be found by splitting. Note, in the implementation SplitValue also considers base values.

4.3.2 Complexity Analysis

We now analyse the time complexity of the algorithm. In this section, we denote the size of usages with $|\mathcal{U}|$, the number of fields with $|F|$ and the size of an expressions with $|e|$.

We start by analysing environment splitting (Algorithm 3). In cases where $\mathcal{U}'$ is not parallel only line 5 is interesting which checks usage equality in $\mathcal{O}(|\mathcal{U}|)$ time. In cases where $\mathcal{U}'$ is a parallel usage but its constituents are not parallel, the recursive calls to SplitValue on line 9 and 10 both require in the worst case $\mathcal{O}(|\mathcal{U}|)$ time. Finally, when $\mathcal{U}'$ is a parallel usage and its constituents are also parallel, the worst case is when the parallel usage is skewed to one side such that we have the following input sizes to SplitValue $(1+|\mathcal{U}|-2)+(1+|\mathcal{U}|-4)+\cdots+(1+1)$. Since each usage comparison is linear in the size of the usages, we have that the time complexity of environment splitting is $\mathcal{O}(|\mathcal{U}|^2)$.

We now consider the time complexity of type checking expressions (Algorithm 2). The call SplitReference on line 4 is $\mathcal{O}(|\mathcal{U}|^2)$ and the calls to SplitValue on line 7 and 8 are similarly $\mathcal{O}(|\mathcal{U}|^2)$ in the worst case. Lastly, on line 17 we call UnSplit, which has a time complexity of $\mathcal{O}(|\mathcal{U}|^2)$ for each argument due to two nested loops over the usage constituents to rebuild a single usage. The time complexity of type checking call is bounded by environment split and unsplit, hence the time complexity of CheckCall is $\mathcal{O}(|\mathcal{U}|^2)$.

Lastly, we analyse the complexity of type checking a class (Algorithm 1). On line 4 we have

Algorithm 2 Type checking call

1: **function** CHECKCALL(Γ, r, m, v_1, v_2)
2: **if** $t_r\ (t_1 \to t_1'\ x_1, t_2 \to t_2'\ x_2)\ \{e\} \notin \Gamma(r).\mathsf{class.methods}$ **then**
3: **return** Error
4: $\Gamma_\mathsf{R} \leftarrow$ SPLITREFERENCE(r, m, t_r)
5: $\Gamma' \leftarrow$ FILTERGAMMA(r, Γ)
6:
7: $(\Gamma_{\mathsf{v1}}, \Gamma'') \leftarrow$ SPLITVALUE'(Γ', v_1, t_1)
8: $(\Gamma_{\mathsf{v2}}, \Gamma''') \leftarrow$ SPLITVALUE'(Γ'', v_2, t_2)
9:
10: **if** $\Gamma_\mathsf{R}(r).\mathsf{usage} \overset{m}{\not\to} \mathcal{U}'$ **then**
11: **return** Error
12:
13: **if** $\Gamma_{\mathsf{v1}}(v_1) \neq t_1$ or $\Gamma_{\mathsf{v2}}(v_2) \neq t_2$ **then**
14: **return** Error
15:
16: $\Gamma_\mathsf{final} \leftarrow\ \Gamma''' \cup \Gamma_\mathsf{R}\{r.\mathsf{usage} \mapsto \mathcal{U}'\} \cup \Gamma_{\mathsf{v1}}\{v_1 \mapsto t_1'\} \cup \Gamma_{\mathsf{v2}}\{v_2 \mapsto t_2'\}$
17: $\Gamma \leftarrow$ UNSPLITGAMMA(Γ_final, r, v_1, v_2)
18: **return** (Γ, t_r)
19: **function** FILTERGAMMA(r, Γ)
20: $\Gamma_\mathsf{new} \leftarrow \emptyset$
21: **for all** $(r', t) \in \Gamma$ **do**
22: **if** $r' \neq r$ **then**
23: $\Gamma_\mathsf{new} \leftarrow \Gamma_\mathsf{new} \cup (r', t)$
24: **return** Γ_new
25: **function** SPLITVALUE'(Γ, v, t)
26: $C[\mathcal{U}] \leftarrow t$
27: $C'[\mathcal{U}'] \leftarrow \Gamma(v)$
28: $\Gamma_\mathsf{new} \leftarrow$ FILTERGAMMA(v, Γ)
29: $(\Gamma_1, \Gamma_2) \leftarrow$ SPLITVALUE($C, \mathcal{U}$, v, $C', \mathcal{U}'$)
30: $(\Gamma_\mathsf{val}, \Gamma') \leftarrow (\Gamma_1, \Gamma_2 \cup \Gamma_\mathsf{new})$
31: **return** $(\Gamma_\mathsf{val}, \Gamma')$

Algorithm 3 Environment splitting

1: **function** SplitValue(C, $\mathcal{U}$, v, C', $\mathcal{U}'$)
2: **if** $C \neq C'$ **then**
3: **return** Error
4: **if** $\mathcal{U} \neq \mathcal{U}'$ **then**
5: **if** $\mathcal{U}' = (u_1 \mid u_2).u_3^s$ **then**
6: $\mathcal{U}_l \leftarrow u_1^{s \cdot \mathtt{l}}$
7: $\mathcal{U}_r \leftarrow u_2^{s \cdot \mathtt{r}}$
8: $\mathcal{U}_p \leftarrow \{(\circledcirc \mid \circledcirc).u_3^s\}$
9: $(\Gamma_1, \Gamma_1') \leftarrow$ SplitValue(C, $\mathcal{U}$, v, C', $\mathcal{U}_l$)
10: $(\Gamma_2, \Gamma_2') \leftarrow$ SplitValue(C, $\mathcal{U}$, v, C', $\mathcal{U}_r$)
11: **if** $\Gamma_1 \neq$ Error **then**
12: **return** $(\Gamma_1, \Gamma_1' \cup \{(v, C[\mathcal{U}_r]), (v, C[\mathcal{U}_p])\})$
13: **else if** $\Gamma_2 \neq$ Error **then**
14: **return** $(\Gamma_2, \Gamma_2' \cup \{(v, C[\mathcal{U}_l]), (v, C[\mathcal{U}_p])\})$
15: **else**
16: **return** Error
17: **else**
18: **return** Error
19: **else**
20: **return** $((v, C[\mathcal{U}]), \emptyset)$

a loop that is bounded by the unique typestate of all fields $\mathcal{O}(|F| \cdot |\mathcal{U}|)$. The loop on line 7 is bounded by the number of transitions for a usage which gives us a worst-case of $\mathcal{O}(|\mathcal{U}|)$ and line 9 is bounded by the time complexity of checking the expression $\mathcal{O}(|e| \cdot |\mathcal{U}|^2)$. Additionally; since we allow non-determinism in usages the final time complexity is $\mathcal{O}(2^{|F| \cdot |\mathcal{U}|^4 \cdot |e|})$ to check a class.

The time complexity can be improved by disallowing non-determinism in usages. Disallowing non-deterministic usages does not seem overly restrictive, as they were introduced in earlier work on Mungo mainly to deal with repeating protocols in inferred usages [GJK20], which is not a goal of this type system. By disallowing non-determinism in usages we get the time complexity $\mathcal{O}(|F| \cdot |\mathcal{U}|^4 \cdot |e|)$ for type checking a class.

Chapter 5

Conclusions

5.1 Results

In this project, we have presented a type system for Mungo that is capable of reasoning about object states with aliases while preserving modularity. The type system is based on the work done in the area of behavioural separation by Caires & Seco and Militão et al. where concepts from separation logic and behavioural types are combined to reason about interference from aliasing or concurrency. We extended the syntax of usages in Mungo with a parallel construct that describes when an object can be safely decomposed and aliased.

Furthermore, by studying alternative approaches to typestates, a significant disadvantage in the protocol specification of Mungo was described; specifically, concerning protocols involving independent field variables that result in large usages which Fugue and Plaid handle appropriately. This problem with protocol specification in Mungo is solved by extending usage syntax which now allows usages for independent fields to be specified as parallel usages, instead of enumerating all combinations of field variable states and their transitions as described in Section 1.1.

In addition to the type system, we defined both a big-step and small-step semantics for Mungo, and an equivalence proof showing the two semantics are equivalent for terminating expressions. By including both a big-step and a small-step semantics certain proofs, that were previously large and complicated, became more straightforward. For example, proving soundness using a big-step semantics allows the properties of the type system to be expressed more naturally, compared with using a small-step semantics. We have shown that key properties hold for the type system, the most important of which is soundness. Soundness and protocol fidelity together guarantees that null-dereferencing and protocol errors do not occur for well-typed programs; furthermore, with the inclusion of behavioural separation, we ensure that aliasing does not result in unsafe interference. Finally, we developed an implementation of the type system along with an interpreter. The interpreter allows us to execute Mungo programs with simple I/O operations and some primitive types.

By extending the Mungo calculus with parallel usages and the type system with behavioural separation, along with proofs showing key properties concerning protocol fidelity and soundness; we have demonstrated that it is possible to use behavioural separation to reason about some forms of aliasing and solve the exponential state-space problem related to usage specification in Mungo. Moreover, by implementing the type system and an interpreter, we have shown that it is also practically feasible to use behavioural separation to reason about aliasing and exponential usage state-spaces in Mungo.

5.2 Discussion

In previous versions of Mungo [BFG+20, GJK20] a small-step operational semantics was defined for the language and in many ways, it makes sense to do so, since following usages and ensuring transitions are conforming to the specified protocols are inherently small-step properties. In this report, we defined the operational semantics using a big-step semantics and included a small-step version. While both semantics are equivalent for terminating programs and fairly simple on their own they ultimately serve different purposes. The big-step semantics lets us naturally express proofs and properties concerning the soundness of the type system since the type system and the big-step semantics employ similar transitions. Using the small-step semantics to prove the type system is sound would result in large and overly complex proofs since they involve showing that type system is an approximation of the small-step semantics. This can be seen in the subject reduction proof of [BFG+20].

Furthermore, by using a big-step semantics to prove properties about the type system we avoid convoluted type rule definitions that are required when modelling the counterparts of a small-step semantics. For example, the small-step semantics uses a stack hence a stack would have to be modelled in the type system as well. The environments in the type system are therefore relatively simple since we do not have to model the small-step environment counterparts. A clear downside of using a big-step semantics to prove soundness is the fact that it cannot express non-terminating execution since a big-step transition always results in a terminal configuration and programs written Mungo can, in fact, be non-terminating. As a result, to prove properties where non-terminating execution is relevant, for example, the progress property, both the small-step and the big-step semantics are used to deal with possibly non-terminating execution.

Employing behavioural separation with parallel usages as described in this project does allow some forms of aliasing in Mungo while guaranteeing only safe interference; however, we are currently unable to express *unlimited* and *direct aliasing*, where values can be aliased any number of times, or aliased through assignment respectively. It begs the question of whether aliasing through method parameters is enough and if behavioural separation is the most appropriate approach. Especially since we have seen other successful ways to reason about aliasing some of which are described in Section 1.2. Our limited aliasing does not represent the final goal or entire potential of behavioural separation in terms of reasoning about aliasing; furthermore, behavioural separation is an approach to reason about interference in general and therefore extends beyond borrowing approaches that reason about aliasing by exchanging a linear resource. Behavioural separation let us naturally extend the existing language whereas introducing permissions or capabilities would introduce significant changes where explicit permissions would be part of type definitions.

Due to non-deterministic usages, the time-complexity of the implementation is exponential, and as usages become more complex, type checking becomes slow or even infeasible. In previous work on Mungo [BFG+20] non-determinism in usages made it possible to infer usages and facilitated usages that could either continue or terminate. The main problem with non-determinism is its adverse effect on the time complexity of expression checking since we would have to consider expressions in the context of whether a usage continues or terminates. Disallowing non-determinism primarily affects usage inference and it is unclear whether usage inference is even possible with parallel usages, hence restricting non-determinism in usages would not be a considerable change at this stage. If non-deterministic usages are not specified in a program, the time complexity of the implementation changes from exponential to polynomial.

5.3 Future Work

In this section we outline some suggestions for future work in the context of Mungo and behavioural separation. Most notably, we identify three interesting directions: usage inference with parallel usages, proving protocol termination, and extending the type system of Mungo to support direct and unrestricted aliasing.

Usage Inference

In previous work on Mungo, Golovanov et. al presented usage inference for classes [GJK20]. The inference algorithm guarantees that the inferred usage is the largest usage up to a simulation ordering $\sqsubseteq$, meaning that the usage covers all well-typed behaviour from the class. It is then natural to ask whether or not usage inference remains possible in our new type system, with the addition of parallel usages. We divide this question into two others: (i) can we infer the largest usage of a class, and (ii) can we infer parallel usages?

The answer to (i) seems to be *yes*. The approach presented in [GJK20] defines a transition system between field typing environments, The transition system represents the finite state automata recognising the regular and ω-regular languages defined by the most permissive usage for the class. This approach still works for the updated type system, with the exception that the usages generated are still exponential in size of the number of linear fields, and do not contain parallel usages. All method call sequences described by parallel usages will still be covered by the exponential size usage, hence the inferred usage can still simulate parallel usages.

The answer to (ii) is a question for further research. Usages extended with parallel behaviour describe behaviour modelled by parallel finite automata introduced by Stotts & Pugh [SP94]. Beek et. al [TBK07] define a *shuffle* operation as: "*[...] a shuffle of two words as an interleaving of consecutive finite subwords of these words which stops (is finite) only if both words have been used completely. Furthermore, one (infinite) word may prevail when the other word, from some point onwards, contributes nothing but the trivial subword* λ*. [...] The shuffling of two languages is defined element-wise.*" To convert the exponential size finite state automata from the inference algorithm into a parallel usage would require an *un-shuffle* operation on the language recognised by the finite state automata. However, as noted by Estrada et. al [EPH06], this un-shuffle operation is not trivial, and would require further investigation. This un-shuffling is further complicated by the fact that usages also describe infinite behaviour hence requiring the un-shuffle operation to work for both regular languages and ω-regular languages.

Protocol Termination Result

In our type system we require protocol completion. This is enforced first by well-formed usages which requires that usages can never reach a point where they cannot terminate. Furthermore, in the rule (TClass) we require that upon reaching the end usage, all fields must be terminated as well, meaning that objects stored as a field of a terminated object, are terminated themselves. Finally, we ensure that we never lose a linear reference in field assignment or sequential expressions.

However, it is not trivial to prove that the protocol termination is achieved in well-typed programs. A result of protocol completion would state, that after a big-step transition of a well-typed initial configuration, all objects in the heap are terminated. While we can reason about the fields of the main object being terminated, using our knowledge of the initial configuration, it is not trivial to lift this property to *all* objects in the heap. This seems to require some additional annotation to the semantics, keeping track of when object become terminated, and then in the

proof showing that this sequence of terminated objects are ordered such that all fields of an object become terminated before the object itself.

Support for Additional Forms of Aliasing

As mentioned in Section 5.2 the current version of the type system only supports a limited form of aliasing where aliases can only be created through parameters during method calls. In mainstream object-oriented languages, we often encounter aliasing in the form of direct and unrestricted aliasing. For example, injecting an object dependency through a setter method or iterating a list of objects. Unrestricted aliasing refers to aliases of objects that can be created an unbounded number of times. In a modular type system, the types of aliasing possible will always be limited in some capacity since we cannot reason about global aliases between classes. The type systems presented by Caires & Seco [CS13] and Militão et al. [MaAC10] incorporate a limited form of unrestricted aliasing where the type system is still modular. Fields are associated with a permission specifying whether the field is read-only or not. In an unrestricted view, the object can only access read-only fields, so that aliases do not interfere. Additionally, in the work by Militão et al. aliased objects are tracked to collect all splits of an aliased object before regaining access to writable fields. Implementing a form of unrestricted aliasing in Mungo has proved to be nontrivial especially considering the interaction between parallel and unrestricted usages. For example, since context-split ensures that fields do not overlap when splitting parallel usages, it becomes difficult to negotiate when to allow overlapping fields in the presence of unrestricted aliasing.

A potential solution is to add read-only values to Mungo on which context-split is not performed since read-only values do not interfere with the state of an object. This approach would permit a form of unrestricted aliasing where the aliases only access read-only fields. However, this approach is still rather limited in terms of matching the aliasing allowed in mainstream languages, since it does not allow direct aliasing. If we were to allow direct aliasing then we would still experience issues concerning protocol termination since we cannot keep track of aliases in multiple objects, while the type system remains modular and only checks one class at a time.

Bibliography

[BBA09] 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.

[BFG+20] Mario Bravetti, Adrian Francalanza, Iaroslav Golovanov, Hans Hüttel, Mathias Steen Jakobsen, Mikkel Klinke Kettunen, and António Ravara. Behavioural types for memory and method safety in a core object-oriented language, 2020.

[BFHR19] Mario Bravetti, Adrian Francalanza, Hans Hüttel, and António Ravara. A type system for mungo, 2019. Unpublished worksheet.

[CS13] Luís Caires and João C. Seco. The type discipline of behavioral separation. *SIGPLAN Not.*, 48(1):275–286, January 2013.

[CV11] Joana Campos and Vasco Vasconcelos. Channels as objects in concurrent object-oriented programming. *Electronic Proceedings in Theoretical Computer Science*, 69, 10 2011.

[DF04] Rob DeLine and Manuel Fahndrich. The fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, January 2004.

[DKPG] Ornela Dardha, Dimitrios Kouzapas, Roly Perera, and Simon Gay. Mungo. http://www.dcs.gla.ac.uk/research/mungo/index.html.

[EPH06] B. D. Estrade, A. L. Perkins, and J. M. Harris. Explicitly parallel regular expressions. In *First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06)*, volume 1, pages 402–409, 2006.

[FD02] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. *SIGPLAN Not.*, 37(5):13–24, May 2002.

[GJK19] Iaroslav Golovanov, Mathias Steen Jakobsen, and Mikkel Klinke Kettunen. Soundness of the Mungo Type System with Generics, 2019.

[GJK20] Iaroslav Golovanov, Mathias Steen Jakobsen, and Mikkel Klinke Kettunen. Typestate Inference for Mungo: Algorithm and Implementation, 2020.

[KDPG16] 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.

[MaAC10] Filipe Militão, Jonathan Aldrich, and Luís Caires. Aliasing control with view-based typestate. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, FTFJP '10, New York, NY, USA, 2010. Association for Computing Machinery.

[NBAB12] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. *SIGPLAN Not.*, 47(1):557–570, January 2012.

[O'H12] Peter W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 286–318. IOS Press, 2012.

[Rey00] John Reynolds. Intuitionistic reasoning about shared mutable data structure. *Millennial Perspectives in Computer Science*, 08 2000.

[SNS+11] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. *SIGPLAN Not.*, 46(10):713–732, October 2011.

[SP94] P.David Stotts and William Pugh. Parallel finite automata for modeling concurrent software systems. *The Journal of Systems & Software*, 27(1):27–43, 1994.

[SY86] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, Jan 1986.

[TBK07] Maurice H Ter Beek and Jetty Kleijn. Infinite unfair shuffles and associativity. *Theoretical Computer Science*, 380(3):401–410, 2007.

[Tho91] Wolfgang Thomas. *Automata on Infinite Objects*, page 133–191. MIT Press, Cambridge, MA, USA, 1991.

[TV16] Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. *SIGPLAN Not.*, 51(9):462–475, September 2016.

[VDG19] A. Laura Voinea, Ornela Dardha, and Simon J. Gay. Resource sharing via capability-based multiparty session types. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, *Integrated Formal Methods*, pages 437–455, Cham, 2019. Springer International Publishing.

Appendix A

Proofs

Proof of Lemma 3

Proof. Induction in the height h of the derivation tree. The base cases where $h = 1$ are (BVAL_B), (LINREFP_B), (LINREFF_B), (UNREFP_B), (UNREFF_B), and (NEW_B).

Case (BVAL_B)**:** This case is trivially shown, as zero transitions are needed.

Case (LINREFP_B)**:** Assume $env_P, h \vdash \langle o, x\rangle \to v \dashv env_P[x \mapsto \mathsf{null}], h$. $env_P(x) = v$ as well as $\neg\mathsf{term}(\mathsf{getType}(v, h))$ follows from (LINREFP_B) hence with (PAR_S) we conclude $\langle env_S \cdot (o, env_P), h, x\rangle \Rightarrow \langle env_S \cdot (o, env_P[x \mapsto \mathsf{null}]), h, v\rangle$.

Case (LINREFF_B)**:** Assume $env_P, h \vdash \langle o, f\rangle \to v \dashv env_P, h[o.f \mapsto \mathsf{null}]$. From ($\text{LINREFF}_\text{B}$) we know $h(o).f = v$ and $\neg\mathsf{term}(\mathsf{getType}(v, h))$, hence we conclude using (LINFLD_S) $\langle env_S \cdot (o, env_P), h, f\rangle \Rightarrow \langle env_S \cdot (o, env_P), h[o.f \mapsto \mathsf{null}], v\rangle$.

Case (UNREFP_B)**:** Assume $env_P, h \vdash \langle o, x\rangle \to v \dashv env_P, h$. $env_P(x) = v$ follows from (UNREFP_B) as well as $\mathsf{term}(\mathsf{getType}(v, h))$ hence with (UNPAR_S) we conclude $\langle env_S \cdot (o, env_P), h, x\rangle \Rightarrow \langle env_S \cdot (o, env_P), h, v\rangle$.

Case (UNREFF_B)**:** Assume $env_P, h \vdash \langle o, f\rangle \to v \dashv env_P, h$. From ($\text{UNREFF}_\text{B}$) we know $h(o).f = v$ and $\mathsf{term}(\mathsf{getType}(v, h))$, hence we conclude using (UNFLD_S) $\langle env_S \cdot (o, env_P), h, f\rangle \Rightarrow \langle env_S \cdot (o, env_P), h, v\rangle$.

Case (NEW_B)**:** Assume $env_P, h \vdash \langle o, \mathsf{new}\ C\rangle \to o' \dashv env_P, h[o' \mapsto (C[C.\mathsf{usage}], C.\mathsf{initvals})]$. From ($\text{NEW}_\text{B}$) we know that o' is fresh for h, hence we conclude with (NEW_S) that $\langle env_S \cdot (o, env_P), h, \mathsf{new}\ C\rangle \Rightarrow \langle env_S \cdot (o, env_P), h[o' \mapsto (C[C.\mathsf{usage}], C.\mathsf{initvals})], o'\rangle$.

The inductive cases where $h > 1$ are (SEQ_B), (IFTRUE_B), (IFFALSE_B), (LAB_B), (CALL_B), and (SUB_B).

Case (SEQ_B)**:** Assume $env_P, h \vdash \langle o, e_1; e_2\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$. By ($\text{SEQ}_\text{B}$) we have $env_P, h \vdash \langle o, e_1\rangle \xrightarrow{\alpha_1} v' \dashv env''_P, h''$ and $env''_P, h'' \vdash \langle o, e_2\rangle \xrightarrow{\alpha_2} v \dashv env'_P, h'$ where $\alpha = \alpha_1 \cdot \alpha_2$. By IH we have $\langle env_S \cdot (o, env_P), h, e_1\rangle \overset{\alpha_1}{\Rightarrow}{}^*\langle env_S \cdot (o, env''_P), h'', v'\rangle$ and $\langle env_S \cdot (o, env''_P), h, e_2\rangle \overset{\alpha_2}{\Rightarrow}{}^*\langle env_S \cdot$

$(o, env'_P), h', v\rangle$. From Lemma 2 we get $\langle env_S \cdot (o, env_P), h, e_1; e_2\rangle \overset{\alpha_1}{\Rightarrow}{}^* \langle env_S \cdot (o, env''_P), h'', v'; e_2\rangle$. From ($\text{SEQ}_\text{S}$) we have $\langle env_S \cdot (o, env''_P), h'', v'; e_2\rangle \Rightarrow \langle env_S \cdot (o, env''_P), h'', e_2\rangle$. In total we have $\langle env_S \cdot (o, env_P), h, e_1; e_2\rangle \overset{\alpha_1 \cdot \alpha_2}{\Longrightarrow}{}^* \langle env_S \cdot (o, env'_P), h', v\rangle$.

Case (IFTRUE_B)**:** Assume $env_P, h \vdash \langle o, \mathsf{if}_r\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \xrightarrow{\alpha_1 \cdot o'.\mathsf{true} \cdot \alpha_2} v \dashv env'_P, h'$. Let $o' = \mathsf{extract}(r, h, env_P, o)$, by ($\text{IFTRUE}_\text{B}$) $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \xrightarrow{\alpha_1} \mathsf{true} \dashv env''_P, h''$, $h''' <: h''$,$h'''(o').\mathsf{usage} \xrightarrow{\mathsf{true}} \mathcal{U}$, $h'''' <: h'''[o'.\mathsf{usage} \mapsto \mathcal{U}]$, and $env''_P, h'''' \vdash \langle o, e_1\rangle \xrightarrow{\alpha_2} v \dashv env'_P, h'$. From IH we get $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \overset{\alpha_1}{\Rightarrow}{}^* \langle env_S \cdot (o, env''_P), h'', \mathsf{true}\rangle$ and $\langle env_S \cdot (o, env''_P), h'''', e_1\rangle \overset{\alpha_2}{\Rightarrow} *\langle env_S \cdot (o, env'_P), h', v\rangle$. By Lemma 4 we have $\langle env_S \cdot (o, env_P), h, \mathsf{if}_r\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \overset{\alpha_1}{\Rightarrow}{}^* \langle env_S \cdot (o, env''_P), h'', \mathsf{if}_r\ (\mathsf{true})\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle$. Using ($\text{IFTRUE}_\text{S}$) we have $\langle env_S \cdot (o, env''_P), h'', \mathsf{if}_r\ (\mathsf{true})\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \overset{o'.\mathsf{true}}{\Longrightarrow} \langle env_S \cdot (o, env''_P), h'''', e_1\rangle$, hence in total we have $\langle env_S \cdot (o, env_P), h, \mathsf{if}_r\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \overset{\alpha_1 \cdot o'.\mathsf{true} \cdot \alpha_2}{\Longrightarrow}{}^* \langle env_S \cdot (o, env'_P), h', v\rangle$.

Case (IFFALSE_B)**:** Similar to the previous case.

Case (LAB_B)**:** Assume $env_P, h \vdash \langle o, k : e\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$. By ($\text{LAB}_\text{B}$) we have $env_P, h \vdash \langle o, e\{\mathsf{continue}\ k/k : e\}\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$ and by IH we have $\langle env_S \cdot (o, env_P), h, e\{\mathsf{continue}\ k/k : e\}\rangle \overset{\alpha}{\Rightarrow}{}^* \langle env_S \cdot (o, env'_P), h', v\rangle$. From ($\text{LBL}_\text{S}$) we have $\langle env_S \cdot (o, env_P), h, k : e\rangle \Rightarrow \langle env_S \cdot (o, env_P), h, e\{\mathsf{continue}\ k/k : e\}\rangle$, hence in total we have $\langle env_S \cdot (o, env_P), h, k : e\rangle \overset{\alpha}{\Rightarrow}{}^* \langle env_S \cdot (o, env'_P), h', v\rangle$.

Case (CALL_B)**:** Assume $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \xrightarrow{o_r.m \cdot \alpha} v \dashv env'_P, h'$. By ($\text{CALL}_\text{B}$) we have $h'' <: h$. Now let $o_r = \mathsf{extract}(r, h'', env_P, o)$, $v' = \mathsf{extract}(v_1, h'', env_P, o)$, and $v'' = \mathsf{extract}(v_2, h'', env_P, o)$. By ($\text{CALL}_\text{B}$) we also have $\{x_1 \mapsto v', x_2 \mapsto v''\}, h''[o_r.\mathsf{usage} \mapsto \mathcal{U}] \vdash \langle o_r, e\rangle \xrightarrow{\alpha} v \dashv \{x_1 \mapsto v^{(3)}, x_2 \mapsto v^{(4)}\}, h''''$, where $h''', env''_P = \mathsf{update}(v^{(3)}, h'''', env_P, o)$, and $h''''', env'_P = \mathsf{update}(v^{(4)}, h''', env''_P, o)$. Finally from ($\text{CALL}_\text{B}$) we also know $h' <: h'''''$. By IH we have $\langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v', x_2 \mapsto v''\}), h''[o_r.\mathsf{usage} \mapsto \mathcal{U}], e\rangle \overset{\alpha}{\Rightarrow}{}^* \langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v^{(3)}, x_2 \mapsto v^{(4)}\}), h'''', v\rangle$. By Lemma 4 we have $\langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v', x_2 \mapsto v''\}), h''[o_r.\mathsf{usage} \mapsto \mathcal{U}], \mathsf{return}_{r.m(v_1,v_2)}\{e\}\rangle \overset{\alpha}{\Rightarrow}{}^* \langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v^{(3)}, x_2 \mapsto v^{(4)}\}), h'''', \mathsf{return}_{r.m(v_1,v_2)}\{v\}\rangle$. With ($\text{CALL}_\text{S}$) we can conclude $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \overset{o'.m}{\Longrightarrow} \langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v', x_2 \mapsto v''\}), h''[o_r.\mathsf{usage} \mapsto \mathcal{U}], \mathsf{return}_{r.m(v_1,v_2)}\{e\}\rangle$. Finally, using ($\text{RET-B}_\text{S}$), we can conclude that $\langle env_S \cdot (o, env_P) \cdot (o_r, \{x_1 \mapsto v^{(3)}, x_2 \mapsto v^{(4)}\}), h''', \mathsf{return}_{r.m(v_1,v_2)}\{v\}\rangle \Rightarrow \langle env_S \cdot (o, env'_P), h', v\rangle$, hence in total we can conclude $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \overset{\alpha}{\Rightarrow}{}^* \langle env_S \cdot (o, env'_P), h', v\rangle$.

□

Proof of Lemma 6

Proof. If $\langle env_S \cdot (o, env_P), h, e\rangle \overset{\alpha}{\Rightarrow}{}^* \langle env_S \cdot (o, env'_P), h', v\rangle$ then by definition there exists a k such that $\langle env_S \cdot (o, env_P), h, e\rangle \overset{\alpha}{\Rightarrow}{}^k \langle env_S \cdot (o, env'_P), h', v\rangle$. We prove this with induction in the length of the transition sequence k. We use the strong induction principle, and assume that the theorem holds for all $k' \leq k$ and show that it holds for $k + 1$.
Base step $k = 0$: No transition sequences of length $k = 0$ exists, hence this case is trivially satisfied.

Assume for $k' \leq k$ and show for $k+1$: we now have a transition sequence $\langle env_S \cdot (o, env_P), h, e\rangle \overset{\alpha}{\Rightarrow}^{k+1} \langle env_S \cdot (o, env'_P), h', v\rangle$ which we can write as

$$\begin{aligned}\langle env_S \cdot (o, env_P), h, e\rangle &\overset{\alpha'}{\Longrightarrow} \langle env_S \cdot (o, env''_P), h'', e'\rangle \\ &\overset{\alpha''}{\Longrightarrow}^{k+1} \langle env_S \cdot (o, env'_P), h', v\rangle\end{aligned}$$

The continuation of the proof depends on the first transition in the sequence, and will continue as a case-analysis on the structure of e.

Case (SEQ-B_S)**:** Assume $\langle env_S \cdot (o, env_P), h, v'; e'\rangle \overset{\alpha}{\Rightarrow}^{k+1} \langle env_S \cdot (o, env'_P), h', v\rangle$. We can rewrite this as $\langle env_S \cdot (o, env_P), h, v'; e'\rangle \Rightarrow \langle env_S \cdot (o, env_P), h, e'\rangle \overset{\alpha}{\Rightarrow}^{k} \langle env_S \cdot (o, env'_P), h', v\rangle$. Per our IH we know $env_P, h \vdash \langle o, e'\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$ and from (BVAL_B) we can conclude $env_P, h \vdash \langle o, v\rangle \rightarrow v \dashv env_P, h$. Finally using ($\text{SEQ}_\text{B}$) we can conclude $env_P, h, \langle o, v'; e'\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$.

Case (ASGN-B_S)**:** Assume $\langle env_S \cdot (o, env_P), h, f = v'\rangle \overset{\alpha}{\Rightarrow}^{k+1} \langle env_S \cdot (o, env'_P), h', v\rangle$. From (ASGN-$\text{B}_\text{S}$) we can directly see that $k = 0$ and that the transition must be $\langle env_S \cdot (o, env_P), h, f = v'\rangle \Rightarrow \langle env_S \cdot (o, env_P), h[o.f \mapsto v'], \mathsf{unit}\rangle$. From ($\text{BVAL}_\text{B}$) we know $env_P, h \vdash \langle o, v'\rangle \rightarrow v' \dashv env_P, h$ hence we can conclude using (ASGN_B) that $env_P, h \vdash \langle o, f = v'\rangle \rightarrow \mathsf{unit} \dashv env_P, h[o.f \mapsto v']$.

Case (LINPAR_S)**:** Assume $\langle env_S \cdot (o, env_P), h, x\rangle \Rightarrow^{k+1} \langle env_S \cdot (o, env'_P), h' v_i\rangle$. We know from ($\text{LINPAR}_\text{S}$) that $\alpha = \varepsilon$, $k = 0$ $env_P = \{x_1 \mapsto v_1, x_2 \mapsto v_2\}$, $h' = h$ and $env'_P = env_P[x \mapsto \mathsf{null}]$. We can directly conclude with (LINREFP_B) conclude $env_P, h \vdash \langle o, x_i\rangle \rightarrow v_i \dashv env'_P, h$.

Case (LINFLD_S)**:** Assume $\langle env_S \cdot (o, env_P), h, f\rangle \Rightarrow^{k+1} \langle env_S \cdot (o, env'_P), h', v\rangle$. We know from the premise of (LINFLD_S) that $\alpha = \varepsilon$, $k = 0$, $h(o).f = v$, $h' = h[o.f \mapsto \mathsf{null}]$ and $env'_P = env_P$. We can then conclude with (LINREFF_B) that $env_P, h, \vdash \langle o, f\rangle \rightarrow v \dashv env'_P, h'$.

Case (UNPAR_S)**:** Assume $\langle env_S \cdot (o, env_P), h, x\rangle \Rightarrow^{k+1} \langle env_S \cdot (o, env'_P), h' v\rangle$. We know from ($\text{UNPAR}_\text{S}$) that $\alpha = \varepsilon$, $k = 0$, $env_P = \{x_1 \mapsto v_1, x_2 \mapsto v_2\}$, $h' = h$, $env'_P = env_P$, $env_P(x) = v$ and $\mathsf{term}(\mathsf{getType}(v, h))$. We can directly conclude with (UNREFP_B) conclude $env_P, h \vdash \langle o, x_i\rangle \rightarrow v_i \dashv env'_P, h$.

Case (UNFLD_S)**:** Assume $\langle env_S \cdot (o, env_P), h, f\rangle \Rightarrow^{k+1} \langle env_S \cdot (o, env'_P), h', v\rangle$. We know from the premise of (LINFLD_S) that $\alpha = \varepsilon$, $k = 0$, $h(o).f = v$, $h' = h$, $\mathsf{term}(\mathsf{getType}(v, h))$ and $env'_P = env_P$. We can then conclude with (UNREFF_B) that $env_P, h, \vdash \langle o, f\rangle \rightarrow v \dashv env'_P, h'$.

Case (NEW_S)**:** Assume $\langle env_S \cdot (o, env_P), h, \mathsf{new}\ C\rangle \overset{\alpha}{\Rightarrow}^{k+1} \langle env_S \cdot (o, env'_P), h, v\rangle$. From ($\text{NEW}_\text{S}$) we know that $k = 0, \alpha = \varepsilon$, $env'_P = env_P$, $v = o'$ and $h' = h[o' \mapsto (C, C.\mathsf{initvals})]$ where o' is a fresh location in h. Then we can directly conclude with (NEW_B) $env_P, h \vdash \langle o, \mathsf{new}\ C\rangle \rightarrow v \dashv env'_P, h'$.

Case (LBL_S)**:** Assume $\langle env_S \cdot (o, env_P), h, k : e\rangle \overset{\alpha}{\Rightarrow}^{k+1} \langle env_S \cdot (o, env'_P), h', v\rangle$. We can split this into $\langle env_S \cdot (o, env_P), h, k : e\rangle \Rightarrow \langle env_S \cdot (o, env_P), h, e\{\mathsf{continue}\ k/k : e\}\rangle \overset{\alpha}{\Rightarrow}^{k} \langle env_S \cdot (o, env'_P), h', v\rangle$. Per the IH we have $env_P, h \vdash \langle o, e\{\mathsf{continue}\ k/k : e\}\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$, hence we can directly conclude using (LBL_B) $env_P, h \vdash \langle o, k : e\rangle \xrightarrow{\alpha} v \dashv env'_P, h'$.

Case (CALL_S)**:** Assume $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \overset{\alpha}{\Rightarrow}^{k+1} \langle env_S \cdot (o, env'_P), h', v\rangle$. We can

rewrite this as

$$\begin{aligned}\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle &\xRightarrow{o'.m} \langle env_S \cdot (o, env_P) \cdot (o', env''_P), h''', \mathsf{return}\{e\}\rangle \\ &\overset{\alpha'}{\Rightarrow}{}^{k-1}\langle env_S \cdot (o, env_P) \cdot (o', env'''_P), h'''', \mathsf{return}\{v\}\rangle \\ &\Rightarrow \langle env_S \cdot (o, env'_P), h', v\rangle\end{aligned}$$

where $h'' <: h$, $env''_P = \{x_1 \mapsto v', x_2 \mapsto v''\}$, $v' = \mathsf{extract}(v_1, h'', env_P, o)$, $v'' = \mathsf{extract}(v_2, h'', env_P, o)$, $o' = \mathsf{extract}(r, h'', env_P, o)$, $h''' = h''[o'.\mathsf{usage} \mapsto \mathcal{U}]$, $h^{(5)}, env_P^{(4)} = \mathsf{update}(env'''_P(x_1), h'''', env'''_P, o)$, and $h', env'_P = \mathsf{update}(env_P^{(4)}(x_2), h^{(5)}, env_P^{(4)}, o)$. From Lemma 4 we get that $\langle env_S \cdot (o, env_P) \cdot (o', env''_P), h''', e\rangle \overset{\alpha'}{\Rightarrow}{}^{k-1}\langle env_S \cdot (o, env_P) \cdot (o', env'''_P), h'''', v\rangle$. From our IH we then have $env''_P, h'' \vdash \langle o', e\rangle \xrightarrow{\alpha'} v \dashv env'''_P, h''''$. With (CALL$_\mathrm{B}$) we can now conclude $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \xrightarrow{o'.m.\alpha'} v \dashv env_P, h'$.

Case (CTX$_\mathrm{S}$) - Sequential composition**:**
Assume $\langle env_S \cdot (o, env_P), h, e; e'\rangle \overset{\alpha}{\Rightarrow}{}^{k+1}\langle env_S \cdot (o, env'_P), h', v\rangle$. We can rewrite this as:

$$\begin{aligned}\langle env_S \cdot (o, env_P), h, e; e'\rangle &\overset{\alpha'}{\Rightarrow}{}^{k_1}\langle env_S \cdot (o, env''_P), h'', v'; e'\rangle \\ &\Rightarrow \langle env_S \cdot (o, env''_P), h'', e'\rangle \\ &\overset{\alpha''}{\Rightarrow}{}^{k_2}\langle env_S \cdot (o, env'_P), h', v\rangle\end{aligned}$$

where $k_1 + 1 + k_2 = k + 1$. From Lemma 4 we know that $\langle env_S \cdot (o, env_P), h, e\rangle \overset{\alpha'}{\Rightarrow}{}^{k_1}\langle env_S \cdot (o, env''_P), h'', v'\rangle$, hence per IH we can conclude both $env_P, h \vdash \langle o, e\rangle \xrightarrow{\alpha'} v' \dashv env''_P, h''$ and $env''_P, h'' \vdash \langle o, e'\rangle \xrightarrow{\alpha''} v \dashv env'_P, h'$. Now finally we can use (SEQ$_\mathrm{B}$) to conclude $env_P, h \vdash \langle o, e; e'\rangle \xrightarrow{\alpha'.\alpha''} v \dashv env'_P, h'$.

Case (CTX$_\mathrm{S}$) - Field assignment**:** Assume $\langle env_S \cdot (o, env_P), h, f = e\rangle \overset{\alpha}{\Rightarrow}{}^{k+1}\langle env_S \cdot (o, env'_P), h', v\rangle$. We can rewrite this as

$$\begin{aligned}\langle env_S \cdot (o, env_P), h, f = e\rangle &\overset{\alpha}{\Rightarrow}{}^{k}\langle env_S \cdot (o, env'_P), h', f = v'\rangle \\ &\Rightarrow \langle env_S \cdot (o, env'_P), h'[o.f \mapsto v'], \mathsf{unit}\rangle\end{aligned}$$

With Lemma 4 we can conclude $\langle env_S \cdot (o, env_P), h, e\rangle \overset{\alpha}{\Rightarrow}{}^{k}\langle env_S \cdot (o, env'_P), h', v'\rangle$, hence by our IH we know $env_P, h \vdash \langle o, e\rangle \xrightarrow{\alpha} v' \dashv env'_P, h'$. With (ASGN$_\mathrm{B}$) we can then conclude $env_P, h \vdash \langle o, f = e\rangle \xrightarrow{\alpha} \mathsf{unit} \dashv env'_P, h'[o.f \mapsto v']$.

Case (CTX$_\mathrm{S}$) - If**:**
Assume $\langle env_S \cdot (o, env_P), h, \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \overset{\alpha}{\Rightarrow}{}^{k}\langle env_S \cdot (o, env'_P), h', v\rangle$. We can rewrite this as follows

$$\begin{aligned}&\langle env_S \cdot (o, env_P), h, \mathsf{if}_r\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \\ &\quad\overset{\alpha'}{\Rightarrow}{}^{k_1}\langle env_S \cdot (o, env''_P), h'', \mathsf{if}_r\ (v')\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \\ &\quad\overset{\alpha''}{\Rightarrow}{}^{k_2}\langle env_S \cdot (o, env'_P), h', v\rangle\end{aligned}$$

Lemma 4 tell us that $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \stackrel{\alpha'}{\Longrightarrow}^{k_1} \langle env_S \cdot (o, env''_P), h'', v'\rangle$ where $v' \in \{\mathsf{true}, \mathsf{false}\}$. From the IH we have $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \to v' \dashv env''_P, h''$. We now consider the case where $v' = \mathsf{true}$. The case for false is similar hence it will not be presented. When $v' = \mathsf{true}$ we know the following:

$$\langle env_S \cdot (o, env''_P), h'', \mathsf{if}\ (\mathsf{true})\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \stackrel{o'.\mathsf{true}}{\Longrightarrow} \langle env_S \cdot (o, env''_P), h''', e_1\rangle \stackrel{\alpha'''}{\Longrightarrow}^{k_2-1} \langle env_S \cdot (o, env'_P), h', v\rangle$$

where $h'''' <: h''$, $h''''(o'.\mathsf{usage}) \xrightarrow{\mathsf{true}} \mathcal{U}$, and $h''' <: h'''[o'.\mathsf{usage} \mapsto \mathcal{U}]$. From the IH we have that $env''_P, h''' \vdash \langle o, e_1\rangle \xrightarrow{\alpha'''} v \dashv env'_P, h'$ and with rule (IfTrueB) we can now conclude $env_P, h \vdash \langle o, \mathsf{if}\ (e)\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \xrightarrow{\alpha' \cdot o'.\mathsf{true} \cdot \alpha'''} v \dashv env'_P, h'$.

□

Proof of Lemma 9

Proof. Induction in the height of the derivation of $\Gamma \vdash e\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$.

Base cases where $h = 0$

If the derivation tree has height 0, then it must have been concluded using one of the following rules: (Val), (Con), (New) and (Call), depending on the structure of e. The proofs for (Val), (New) and (Call) are similar.

Case (Val), (New), (Call)**:** For these cases, no substitution happens, and we have that $e = e\{\mathsf{continue}\ k/k : e\}$. From (Lab) we know $\Gamma \vdash^{\Omega'} e : \mathsf{void} \triangleright \Gamma'$ and for each rule we can see that the value of Ω is not used, hence we also have that $\Gamma \vdash^{\Omega} e : \mathsf{void} \triangleright \Gamma'$. Finally, since no substitution happens, we can conclude $\Gamma \vdash^{\Omega} e\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$.

Case (Con)**:** We consider the case where $e = \mathsf{continue}\ k'$. There are two sub-cases. First we treat the case where $k' = k$. Assume $\Gamma \vdash^{\Omega} k : e : \mathsf{void} \triangleright \Gamma'$. Here we have that $e\{\mathsf{continue}\ k/k : e\} = k : e$, which is is well-typed due to our assumption. For the case where $k' \neq k$ it follows the same structure as the other base cases. From (Lab) we know $\Gamma \vdash^{\Omega'} \mathsf{continue}\ k' : \mathsf{void} \triangleright \Gamma'$. From (Con) we know that $\Omega'(k') = \Gamma$, but since $k' \neq k$ we must have that $\Omega(k') = \Gamma$. As no substitution happens, we can conclude $\Gamma \vdash^{\Omega} e\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$.

Inductive step where $h > 0$

Case (Seq)**:** Assume $\Gamma \vdash k : (e_1; e_2) : \mathsf{void} \triangleright \Gamma'$ By IH we have $\Gamma \vdash e_1\{\mathsf{continue}\ k/k : e\} : t \triangleright \Gamma''$ and $\Gamma'' \vdash e_2\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$. From our assumption we know that $\mathsf{term(t)}$ hence we can conclude $\Gamma \vdash (e_1; e_2)\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$.

Case (Fld)**:** Assume $\Gamma \vdash k : f = e' : \mathsf{void} \triangleright \Gamma'$ By well-formedness $\mathsf{continue}\ k$ cannot appear in e', hence no substitution happens, and by our assumption we know that $\Gamma \vdash f = e'\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$.

Case If**:** Assume $\Gamma \vdash k : \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\} : \mathsf{void} \triangleright \Gamma'$. From (If) we know $\Gamma'''' \vdash e_1 : \mathsf{void} \triangleright \Gamma'$ and $\Gamma''''' \vdash e_2 : \mathsf{void} \triangleright \Gamma'$. Then from IH we have $\Gamma'''' \vdash e_1\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$

and $\Gamma'''' \vdash e_2\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$. Now we can directly conclude $\Gamma \vdash k : \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$

Case (LAB)**:** Assume $\Gamma \vdash^{\Omega} k : (k' : e') : \mathsf{void} \triangleright \Gamma'$. For e to be well-formed, we must have that $k' \neq k$. From (LAB) we know that $\Gamma \vdash^{\Omega'} k' : e' : \mathsf{void} \triangleright \Gamma'$ and by applying (LAB) again $\Gamma \vdash^{\Omega''} e' : \mathsf{void} \triangleright \Gamma'$. By the IH we have $\Gamma \vdash^{\Omega''} e'\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$ hence we can conclude $\Gamma \vdash^{\Omega} k : (k' : e')\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$.

□

Proof of Lemma 10

Proof. **Case** (ASGN_B)**:** Assume $env_P, h \vdash \langle o, f = e\rangle \to \mathsf{unit} \dashv env'_P, h'[o.f \mapsto v]$. From premise of ($\text{ASGN}_\text{B}$) we have $env_P, h \vdash \langle o, e\rangle \to v \dashv env'_P, h'$. By applying the IH we know $\forall o' \in h.o' \notin \mathsf{reach}(env_P, h, o) \implies h(o') = h'(o')$. By definition of reach we know that $o.f \in \mathsf{reach}(o, h)$, hence we can conclude (ASGN_B).

Case (BVAL_B)**:** Trivially true.

Case (LINREFP_B)**:** Assume $env_P, h \vdash \langle o, x\rangle \to v \dashv env_P[x \mapsto \mathsf{null}], h$. By definition of reach we know that $x \in \mathsf{reach}(env_p, h, o)$, hence we can conclude (LINREFP_B).

Case (LINREFF_B)**:** Assume $env_P, h \vdash \langle o, f\rangle \to v \dashv env_P, h[o.f \mapsto \mathsf{null}]$. By definition of reach we know that $o.f \in \mathsf{reach}(env_p, h, o)$, hence we can conclude (LINREFP_B).

Case (UNREFP_B)**:** Trivially true.

Case (UNREFP_B)**:** Trivially true.

Case (NEW_B)**:** Assume $env_P, h \vdash \langle o, \mathsf{new}C\rangle \to o' \dashv env_P, h[o' \mapsto (C[C.\mathsf{usage}], C.\mathsf{initvals})]$. From ($\text{NEW}_\text{B}$) we know that o' is fresh, hence we know that $o' \notin \mathsf{dom}(h)$. We can now conclude ($\text{NEW}_\text{B}$).

Case (SEQ_B)**:** Assume $env_P, h \vdash \langle o, e; e'\rangle \to v' env'_P, h'$. From premise of ($\text{CALL}_\text{B}$) we know $env_P, h \vdash \langle o, e\rangle \to v'' \dashv env''_P, h''$ and $env''_P, h'' \vdash \langle o, e'\rangle \to v' \dashv env'_P, h'$. By IH on $env_P, h \vdash \langle o, e\rangle \to v'' \dashv env''_P, h''$ we know $\forall o' \in \mathrm{dom}(h).o' \notin \mathsf{reach}(env_P, h, o) \implies h(o') = h''(o')$. By IH on $env''_P, h'' \vdash \langle o, e'\rangle \to v' \dashv env'_P, h'$ we know $\forall o' \in \mathrm{dom}(h'').o' \notin \mathsf{reach}(env''_P, h'', o) \implies h''(o') = h'(o')$. Hence we can conclude (SEQ_B).

Case (IFTRUE_B)**:** Assume that $env_P, h \vdash \langle o, \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \to v \dashv env'_P, h'$. From premise of assumption we know $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \to \mathsf{true} \dashv env''_P, h''$. By applying the IH we know $\forall o' \in \mathsf{dom}(h).o' \notin \mathsf{reach}(env_P, h, o) \implies h(o') = h''(o')$. From definition of extract we know that $o' \in \mathsf{reach}(env_P, h, o)$. From assumption we know $env''_P, h''[o' \mapsto \mathcal{U}] \vdash \langle o, e_1\rangle \to v \dashv env'_P, h'$. By applying the IH we know $\forall o'' \in \mathsf{dom}(h''[o' \mapsto \mathcal{U}]).o'' \notin \mathsf{reach}(env''_P, h''[o' \mapsto \mathcal{U}], o) \implies h''[o' \mapsto \mathcal{U}](o'') = h'(o'')$. We can now conclude $\forall o' \mathsf{dom}(h).o' \notin \mathsf{reach}(env_P, h, o) \implies h(o') = h'(o')$. Hence we can conclude (IFTRUE_B)

Case (IFFALSE_B)**:** Follows the same reasoning as (IFTRUE_B)

Case (LAB_B)**:** Assume $env_P, h \vdash \langle o, k : e\rangle \to v \dashv env'_P, h'$. From premise of assumption

we know $env_P, h \vdash \langle o, e\{\mathsf{continue}\ k/k : e\}\rangle \to v \dashv env'_P, h'$. By applying the IH we know $\forall o' \in \mathsf{dom}(h).o' \notin \mathsf{reach}(env_P, h, o) \implies h(o') = h'(o')$. Hence we can conclude ($\textsc{Lab}_\textsc{B}$).

Case ($\textsc{Call}_\textsc{B}$)**:** Assume $env_P, h \vdash \langle o, r.m(v_1, v_2) \to v \dashv env'_P, h'$. We have that $h'' <: h$ does not touch unreachable objects. From the definition of extract, we know that v', v'' and r does not change the heap and that they are in reach. From the definition of reach on o_r, we know that $\mathsf{reach}(\{x_1 \mapsto v', x_2 \mapsto v''\}, h[o_r.\mathsf{usage} \mapsto \mathcal{U}], o_r) \subseteq \mathsf{reach}(env''_P, h[o_r.\mathsf{usage} \mapsto \mathcal{U}], o)$. From IH we on $\{x_1 \mapsto v', x_2 \mapsto v''\}, h''[o_r.\mathsf{usage} \mapsto \mathcal{U}] \vdash \langle o_r, e\rangle \to v \dashv \{x_1 \mapsto v^3, x_2 \mapsto v^4\}, h''''$ we know $\forall o' \in \mathsf{dom}(h''[o_r.\mathsf{usage} \mapsto \mathcal{U}]).o' \notin \mathsf{reach}(\{x_1 \mapsto v', x_2 \mapsto v''\}, h''[o_r.\mathsf{usage} \mapsto \mathcal{U}], o_r) \implies h''[o_r.\mathsf{usage} \mapsto \mathcal{U}](o') = h''''(o')$. By definition of update, we know that it can not change objects which are not in h, when input is in $\mathsf{dom}(h)$. Hence we can conclude ($\textsc{Call}_\textsc{B}$).

□

Proof of Lemma 12

We first show one direction of the proof: *If* $\Gamma \vdash env_P, h, o$ *and* $\exists h'$ *s.t.* $h' <: h$ *then* $\exists \Gamma'$ *s.t.* $\Gamma' <: \Gamma$ *and* $\Gamma' \vdash env_P, h', o$.

Proof. By induction in the height of the judgment $h' <: h$.
Case ($\textsc{Trans}$)**:** Assume $\Gamma \vdash env_P, h, o$ and that $h' <: h$ has been concluded with ($\textsc{Trans}$). From ($\textsc{Trans}$) we then know that $\exists h''$ s.t. $h' <: h''$ and $h'' <: h$. From the IH we know $\exists \Gamma''$ s.t. $\Gamma'' <: \Gamma$ and $\Gamma'' \vdash env_P, h'', o$. Then by applying the IH again we have $\exists \Gamma'$ s.t. $\Gamma' <: \Gamma''$ and $\Gamma' \vdash env_P, h, o$. Finally since both $\Gamma' <: \Gamma''$ and $\Gamma'' <: \Gamma$ we can conclude using ($\textsc{Trans}$) that $\Gamma' <: \Gamma$.

Case ($\textsc{ParL}$)**:** We prove both directions of the rule. Assume $\Gamma \vdash env_P, h, o$ and $h' <: h$ was concluded with ($\textsc{ParL}$), hence $h' = h'', o' \mapsto (C[(\circledcirc \mid u_2).u_3^s], env'_f) * \{o' \mapsto (C[u_1^{s\cdot 1}], env''_f)\}$, $h = h'', o' \mapsto (C[(u_1 \mid u_2).u_3^s], env_f)$, and $env_f = env'_f \cdot env''_f$. If no f exists s.t. $h(o).f = o'$ then $\Gamma * \emptyset \vdash env_P, h', o$, and with ($\textsc{Id}$) we conclude $\Gamma * \emptyset <: \Gamma$. If f does exist, then we know that $\Gamma(f) = \mathsf{getType}(o', h) = C[(u_1 \mid u_2).u_3^s]$, hence $\Gamma = \Gamma'', f \mapsto C[(u_1 \mid u_2).u_3^s]$. By ($\textsc{ParL}$) we conclude $\Gamma' < \Gamma$ where $\Gamma' = \Gamma'', f \mapsto C[(\circledcirc \mid u_2).u_3^s] * \{f \mapsto C[u_1^{s\cdot 1}]\}$. Using (WTC-S) we conclude $\Gamma' \vdash env_P, h', o$. We now prove the opposite direction. Assume $\Gamma \vdash env_P, h, o$ and $h' <: h$ was concluded with ($\textsc{ParL}$), hence $h' = h'', o' \mapsto (C[(u_1 \mid u_2).u_3^s], env_f)$, $h = h'', o' \mapsto (C[(\circledcirc \mid u_2).u_3^s], env'_f) * \{o' \mapsto (C[u_1^{s\cdot 1}], env''_f)\}$, and $env_f = env'_f \cdot env''_f$. If $\Gamma \vdash env_P, h, o$ then by (WTC-S) we have $\Gamma = \Gamma_1 * \Gamma_2$ where $\Gamma_1 \vdash env_P, h'', o' \mapsto (C[(\circledcirc \mid u_2).u_3^s], env'_f), o$ and $\Gamma_2 \vdash env_P, \{o' \mapsto (C[u_1^{s\cdot 1}], env''_f)\}, o$. From (WTC-B) we have $\mathrm{dom}(\Gamma_2) \setminus \{x_1, x_2\} = \mathrm{dom}(h(o).\mathsf{fields})$, hence if no f exists s.t. $h(o).f = o'$ then $\mathrm{dom}(\Gamma_2) = \emptyset$, hence trivially we have $\Gamma_1 \vdash env_P, h', o$. Otherwise, if f exists, then we know $\mathsf{getType}(o', \{o' \mapsto C[(u_1^{s\cdot 1})], env''_f\}) = C[u_1^{s\cdot 1}] = \Gamma_2(f)$ and $\mathsf{getType}(o', h'', o' \mapsto C[(\circledcirc \mid u_2).u_3^s], env'_f) = C[(\circledcirc \mid u_2).u_3^s] = \Gamma_1(f)$, hence $\Gamma_2 = \{f \mapsto C[u_1^{s\cdot 1}]\}$ and $\Gamma_1 = \Gamma'', f \mapsto C[(\circledcirc \mid u_2).u_3^s]$. With ($\textsc{ParL}$) we can conclude $\Gamma' = \Gamma'', f \mapsto C[(u_1 \mid u_2).u_3^s]$ and finally $\Gamma' \vdash env_P, h', o$. The situation where o' is a parameter is similar, and will be omitted.

Case ($\textsc{ParR}$)**:** similar to ($\textsc{ParL}$).

Case ($\textsc{Id}$)**:** Assume $\Gamma \vdash env_P, h, o$ and $h' <: h$ was concluded using ($\textsc{Id}$) with the rule $\emptyset * h <: h$, hence $h' = \emptyset * h$. Now let $\Gamma' = \emptyset * \Gamma$. $\emptyset \vdash env_P, \emptyset, o$ is trivial, and by our assumption we know $\Gamma \vdash env_P, h, o$, hence we can conclude $\emptyset * \Gamma \vdash env_P, \emptyset * h, o$.

Assume $\Gamma \vdash env_P, h, o$ and $h' <: h$ was concluded using (ID) with the rule $h <: \emptyset * h$, hence $h = \emptyset * h''$ and $h' = h''$. From (WTC-S) we know that Γ must be $\Gamma = \Gamma_1 * \Gamma_2$ s.t. $\Gamma_1 \vdash env_P, \emptyset, o$ and $\Gamma_2 \vdash env_P, h'', o$. Now let $\Gamma' = \Gamma_2$, then we have $\Gamma' \vdash env_P, h', o$.

The two remaining cases for (ID) are similar.

Case (CONCAT)**:** Assume $\Gamma \vdash env_P, h, o$ and $h' <: h$ was concluded using (CONCAT) hence $h' = h_1, h_2$ and $h = h_1 * h_2$ where $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$. From (WTC-S) we know that $\Gamma = \Gamma_1 * \Gamma_2$ s.t. $\Gamma_1 \vdash env_P, h_1, o$ and $\Gamma_2 \vdash env_P, h_2, o$. From (WTC-B) we know that $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. Now let $\Gamma' = \Gamma_1, \Gamma_2$. We know that $\Gamma' \vdash env_P, h', o$ and by (CONCAT) we have $\Gamma' <: \Gamma$.
Assume $\Gamma \vdash env_P, h, o$ and $h' <: h$ was concluded using (CONCAT), hence $h' = h_1 * h_2$ and $h = h_1, h_2$ where $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$. Since $\Gamma \vdash env_P, (h_1, h_2), o$ there must be an ordering of $\Gamma = \Gamma_1, \Gamma_2$ s.t. the bindings of Γ_1 corresponds to the bindings of h_1 and vice versa. Furthermore, from (WTC-B) we know that $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2)$. Now let $\Gamma' = \Gamma_1 * \Gamma_2$. Using (WTC-S) we see that $\Gamma' \vdash env_P, h', o$ and from (CONCAT) we know that $\Gamma' <: \Gamma$. □

We now show the opposite direction: *If* $\Gamma \vdash env_P, h, o$ *and* $\exists \Gamma'$ *s.t.* $\Gamma' <: \Gamma$ *then* $\exists h'$ *s.t.* $h' <: h$ *and* $\Gamma' \vdash env_P, h', o$.

Proof. By induction in the height of the judgment $\Gamma' <: \Gamma$.
Case (TRANS)**:** Assume $\Gamma \vdash env_P, h, o$ and $\Gamma' <: \Gamma$ which have been conclude with (TRANS). From (TRANS) we know $\exists \Gamma''$ s.t. $\Gamma'' <: \Gamma$ and $\Gamma' <: \Gamma''$. From IH we know $\exists h''$ s.t. $h'' <: h$ and $\Gamma'' \vdash env_P, h'', o$. By appling IH again we know $\exists h'$ s.t. $h' <: h''$ and $\Gamma' \vdash env_P, h', o$. Using (TRANS) we can conclude (TRANS).

Case (SPLITL)**:** We prove both directions of this rule. Assume $\Gamma \vdash env_P, h, o$ and $\Gamma' <: \Gamma$ which have been concluded with (PARL), hence $\Gamma = \Gamma'', v \mapsto C[(u_1|u_2).u_3^s]$ and $\Gamma' = (\Gamma'', v \mapsto C[u_1^{l*s}]) * \{v \mapsto C[(\circledcirc|u_2).u_3^s\}$. if $v = f$ then we know that $\Gamma(v) = \mathsf{getType}(h(o).f, h) = C[(u_1|u_2).u_3^s]$ and $h(o).f = o'$, hence $h = h'', < o' \mapsto C[(u_1|u_2).u_3^s], env_f >$ by (PARL) we conclude $h' <: h$ with $h', o \mapsto < C[(\circledcirc|u_2).u_3^s], env'_f > * \{o \mapsto < C[u_1^{l*s}], env''_f >\}$ where $env_f = env'_f * env''_f$ using (WTC-S) we conclude $\Gamma' \vdash env_P, h', o$ if $v = x$ then we know that $\Gamma(v) = \mathsf{getType}(env_P(v), h) = C[(u_1|u_2).u_3^s]$ and $env_P(v) = o'$ and $o' \in \mathsf{dom}(h)$ hence $h = h'', < o' \mapsto C[(u_1|u_2).u_3^s], env_f >$ by (PARL) we conclude $h' <: h$ with $h', o \mapsto < C[(\circledcirc|u_2).u_3^s], env'_f > * \{o \mapsto < C[u_1^{l*s}], env''_f >\}$ where $env_f = env'_f * env''_f$ using (WTC-S) we conclude $\Gamma' \vdash env_P, h', o$

We now prove the other direction Assume $\Gamma \vdash env_P, h, o$ and $\Gamma' <: \Gamma$ which have been concluded with (PARL), hence $\Gamma' = \Gamma'', v \mapsto C[(u_1|u_2).u_3^s]$ and $\Gamma = (\Gamma'', v \mapsto C[u_1^{l*s}]) * \{v \mapsto C[(\circledcirc|u_2).u_3^s]\}$. We know that from (WTS) we know that $\exists h = h_1 * h_2$ s.t. $(\Gamma'', v \mapsto [u_1^{r*s}]) \vdash env_P, h_1, o$ and $\{v \mapsto C[(\circledcirc|u_2).u_3^s]\} \vdash env_P, h_2, o$. and we know $\exists h''$ s.t $\Gamma'' \vdash env_P, h'', o$ from (PARL) we know that $h' = h'', o' \mapsto < C[(u_1|u_2).u_3^s], env_f >$. we get that $\mathsf{getType}(o', h') = \Gamma'(v) = C[(u_1|u_2).u_3^s]$, hence we can conclude $\Gamma' \vdash env_P, h', o$.

Case (PARR)**:** similar to (PARL).

Case (ID)**:** Assume $\Gamma \vdash env_P, h, o$ and $\Gamma' <: \Gamma$ which have been concluded with (ID) with the rule $\Gamma <: \Gamma * \emptyset$ hence $\Gamma' = \Gamma * \emptyset$. From (WTC-S) we know that $\Gamma * \emptyset \vdash env_P, h, o$ and from (WTC-B) we know $\Gamma \vdash env_P, h_1, o$ and $\emptyset \vdash env_P, h_2, o$. Now let $h_1 = h$ and $h_2 = \emptyset$ we can conclude $\Gamma' \vdash env_P, h, o$. The two remaining cases for (ID) are similar.

Case (CONCAT)**:** Assume $\Gamma \vdash env_P, h, o$ and $\Gamma' <: \Gamma$ which have been concluded with (CONCAT), hence $\Gamma = \Gamma_1 * \Gamma_2$ and $\Gamma' = \Gamma_1, \Gamma_2$ where $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$. From (WTC-S) we know that $h = h_1 * h_2$ s.t. $\Gamma_1 \vdash env_P, h_1, o$ and $\Gamma_2 \vdash env_P, h_2, o$. From (WTC-B) we know that

$\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$. Now let $h' = (h_1, h_2)$, We know from (CONCAT) that $h' <: h$, hence we can conclude $\Gamma' \vdash env_P, h', o$

Assume $\Gamma \vdash env_P, h, o$ and $\Gamma' <: \Gamma$ which have been concluded with (CONCAT), hence $\Gamma' = \Gamma_1 * \Gamma_2$ and $\Gamma = \Gamma_1, \Gamma_2$ where $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$. From (WTC-S) we know that $h = (h_1, h_2)$ s.t. $\Gamma_1 \vdash env_P, h_1, o$ and $\Gamma_2 \vdash env_P, h_2, o$. From (WTC-B) we know that $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$. Now let $h' = h_1 * h_2$. We know from (CONCAT) that $h' <: h$. We can now conclude (CONCAT) for $\Gamma' \vdash env_P, h', o$ □

Proof of Theorem 3

Proof. **Case** (ASGN_B)**:** Assume $env_P, h \vdash \langle o, f = e \rangle \xrightarrow{\alpha} unit \dashv env'_P, h'[o.f \mapsto v]$, $\Gamma \vdash f = e : \mathsf{void} \triangleright \Gamma', f \mapsto t$ and $\Gamma \vdash env_P, h, o$ then from (FLD) we have $\Gamma \vdash e : t \triangleright \Gamma', f \mapsto t'$. From premise of ($\text{ASSGN}_\text{B}$) we have $env_P, h \vdash \langle o, e \rangle \xrightarrow{\alpha} v \dashv env'_P, h'$. From IH we have $\Gamma', f \mapsto t' \vdash env'_P, h', o$ and $\mathsf{getType}(v, h') = t$. By only updating f and by IH we know that $\Gamma, f \mapsto t \vdash env'_P, h[o.f \mapsto v], o$ Hence we can conclude (ASGN_B).

Case (BVAL_B)**:** Assume $env_P, h \vdash \langle o, v \rangle \to v \dashv env_P, h$, $\Gamma \vdash v : t \vdash \Gamma$ and $\Gamma \vdash env_P, h, o$ From definition of $\mathsf{getType}$ we know $\mathsf{getType}(v, h) = t$ for base types. Hence we can conclude (BVAL_B).

Case (LINREFP_B)**:** Assume $env_P, h \vdash \langle o, x \rangle \to v \dashv env_P[x \mapsto \mathsf{null}], h$, $\Gamma, x \mapsto t \vdash x : t \triangleright \Gamma, x \mapsto \bot$ and $\Gamma, x \mapsto t \vdash env_P, h, o$. We can conclude $\Gamma[x \mapsto \bot] \vdash env_P[x \mapsto \mathsf{null}], h, o$ as type of null is $\bot$ and the rest follows from WTC. From WTC we know $\mathsf{getType}(v, h[x \mapsto \mathsf{null}]) = t$. Hence we can conclude (LINREFP_B).

Case (LINREFF_B)**:** Assume $env_P, h \vdash \langle o, f \rangle \to v \dashv env_P, h[o.f \mapsto \mathsf{null}]$, $\Gamma, f \mapsto t \vdash f : t \triangleright \Gamma, f \mapsto \bot$ and $\Gamma, f \mapsto t \vdash env_P, h, o$. We can conclude that $\Gamma[f \mapsto \bot](f) \vdash env_P, h[o.f \mapsto \mathsf{null}], o$ as the type of $\bot$ is null and the rest follows from WTC. From WTC we know $\mathsf{getType}(v, h[o.f \mapsto \mathsf{null}]) = t$. Hence we can conclude (LINREFF_B).

Case (UNREFP_B)**:** Assume $env_P, h \vdash \langle o, x \rangle \to v \dashv env_P, h$, $\Gamma, x \mapsto t \vdash x : t \triangleright \Gamma, x \mapsto t$ and $\Gamma, x \mapsto t \vdash env_P, h, o$. From assumption we know $\Gamma, x \mapsto t \vdash env_P, h, o$ and $\mathsf{getType}(v, h) = t$. Hence we can conclude (UNREFP_B).

Case (UNREFF_B)**:** Assume $env_P, h \vdash \langle o, f \rangle \to v \dashv env_P, h$, $\Gamma, f \mapsto t \vdash f : t \triangleright \Gamma, f \mapsto t$ and $\Gamma, f \mapsto t \vdash env_P, h, o$. From assumption we know $\Gamma, f \mapsto t \vdash env_P, h, o$ and $\mathsf{getType}(v, h) = t$. Hence we can conclude (UNREFF_B).

Case (NEW_B)**:** Assume $env_P, h \vdash \langle o, \mathsf{new}\ C \rangle \to o' \dashv env_P, h[o' \mapsto (C[C.\mathsf{usage}], C.\mathsf{initvals})]$, $\Gamma \vdash \mathsf{new}\ C : C[C.\mathsf{usage}] \triangleright \Gamma$ and $\Gamma \vdash env_P, h, o$. By assumption we know $\Gamma \vdash env_P, h, o$ and from $\mathsf{getType}(o', h[o' \mapsto (C[C.\mathsf{usage}], C.\mathsf{initvals})]) = C[C.\mathsf{usage}]$, hence we can conclude (NEW_B).

Case (SEQ_B)**:** Assume $env_P, h \vdash \langle o, e; e' \rangle \to v' \dashv env'_P, h'$, $\Gamma \vdash e; e' : t' \triangleright \Gamma'$ and $\Gamma \vdash env_P, h, o$. From premise of ($\text{SEQ}_\text{B}$) we know $env_P, h \vdash \langle o, e \rangle \to v'' \dashv env''_P, h''$ and $env''_P, h'' \vdash \langle o, e' \rangle \to v' \vdash env'_P, h'$. From premise of (SEQ) we know $\Gamma \vdash e : t \triangleright \Gamma''$ and $\Gamma'' \vdash e' : t' \triangleright \Gamma'$. By applying IH we get $\Gamma'' \vdash env''_P, h'', o$ and $\mathsf{getType}(v, h'') = t$. By applying IH again we get $\Gamma' \vdash env'_P, h', o$ and $\mathsf{getType}(v, h') = t'$. Hence we can conclude (SEQ_B).

Case (IFTRUE_B)**:** Assume $env_P, h \vdash \langle o, \mathsf{if}\ (r.m(v_1, v_2))\ \{e\}\ \mathsf{else}\ \{e'\} \rangle \to v \dashv env'_P, h'$, $\Gamma \vdash \mathsf{if}$

$(r.m(v_1, v_2))$ $\{e\}$ else $\{e'\}$: $t \triangleright \Gamma'$ and $\Gamma \vdash env_P, h, o$. From the premise of (IfTrue$_B$) we know $env_P, h \vdash \langle o, r.m(v_1, v_2) \rightarrow \mathsf{true}\rangle \dashv env''_P, h''$ and from the premise of (If) we know $\Gamma \vdash r.m(v_1, v_2) : \mathsf{bool} \triangleright \Gamma''$. By applying the IH we get that $\Gamma'' \vdash env''_P, h'', o$ and $\mathsf{getType}(\mathsf{true}, h'') = \mathsf{bool}$. From Lemma 12 and $\Gamma'' \vdash env''_P, h'', o$ and $h''' <: h''$ we know $\exists \Gamma''' * \{r : C[\langle \mathcal{U}_1, \mathcal{U}_2\rangle]\}$ such that $\Gamma''' * \{r : C[\langle \mathcal{U}_1, \mathcal{U}_2\rangle]\} \vdash env''_P, h''', o$ and $\Gamma''' * \{r : C[\langle \mathcal{U}_1, \mathcal{U}_2\rangle]\} <: \Gamma$. From (SelTrue) we know $\mathcal{U}$ in (IfTrueB$_B$) is the same as $\mathcal{U}_1$ in (If), hence we know $\Gamma''' * \{r : C[\mathcal{U}_1]\} \vdash env''_P, h'''[o' \mapsto \mathcal{U}], o$. From Lemma 12 and $h'''' <: h'''[o' \mapsto \mathcal{U}]$ we know $\exists \Gamma''''$ such that $\Gamma'''' <: \Gamma''' * \{r : C[\mathcal{U}_1]\}$ and $\Gamma'''' \vdash env''_P, h'''', o$. We know that $\Gamma'''' \vdash e : t \triangleright \Gamma'$, $env''_P, h'''' \vdash \langle o, e\rangle \rightarrow v \dashv env'_P, h'$ and $\Gamma'''' \vdash env''_P, h'''', o$. Then by the IH we can conclude (IfTrue$_B$).

Case (IfFalse$_B$)**:** Follows the same reasoning as (IfTrue$_B$).

Case (Lab$_B$)**:** Assume that $env_P, h \vdash \langle o, k : e\rangle \rightarrow v \dashv env'_P, h'$, $\Gamma \vdash^{\Omega} k : e : \mathsf{void} \triangleright \Gamma'$ and $\Gamma \vdash env_P, h, o$. From premise of (Lab$_B$) we know $env_P, h \vdash \langle o, e\{\mathsf{continue}\ k/k : e\}\rangle \rightarrow v \dashv env'_P, h'$ and from premise of (Lab) we know $\Gamma \vdash^{\Omega'} e : \mathsf{void} \triangleright \Gamma'$. From the IH we get $\Gamma' \vdash env'_P, h', o$ and $\mathsf{getType}(v, h') = \mathsf{void}$ Hence we can conclude (Lab$_B$).

Case (Call$_B$)**:** Assume $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \rightarrow v \dashv env'_P, h'$, $\Gamma \vdash r.m(v_1, v_2) : t\Gamma'$ and $\Gamma \vdash env_P, h, o$. From (Call$_B$) we know $h'' <: h$, $h''(o_r).\mathsf{usage} \xrightarrow{m} \mathcal{U}$, $t\ m(t_1 \rightarrow t'_1\ x_1, t_2 \rightarrow t'_2\ x_2)$, $h' <: h''''$ and $\{x \mapsto v', x_2 \mapsto v''\}, h''[o_r.\mathsf{usage} \mapsto \mathcal{U}] \vdash \langle o_r, e\rangle \rightarrow v \dashv \{x_1 \mapsto v^3, x_2 \mapsto v^4\}$. Using Lemma 12 on assumption and $h'' <: h$ we get $\exists \Gamma'' * (r : C[\mathcal{U}], v_i : t_i)$ such that $\Gamma'' * (r : C[\mathcal{U}], v_i : t_i) <: \Gamma$ and $\Gamma'' * (r : C[\mathcal{U}], v_i : t_i) \vdash env_P, h'', o$. From transitions we know that $\mathcal{U}$ from (Call$_B$) is the same as $\mathcal{U}_1$ from (Call). We know from (WTC) that v' and v'' matches the expected type. From TCBR we know that there $\exists \Gamma^3$ such that $\Gamma^3 \vdash e : t \triangleright \Gamma^4$. We know that all transitions that happens in e can be done in the heap $h''[o_r.\mathsf{usage} \mapsto \mathcal{U}]$ and Γ^3, hence we must have $\Gamma^3 \vdash \{x_1 \mapsto v', x_2 \mapsto v''\}, h''[o_r.\mathsf{usage} \mapsto \mathcal{U}], o_r$. From IH we now get $\Gamma^4 \vdash \{x_1 \mapsto v^3, x_2 \mapsto v^4\}, h'''', o_r$ and $\mathsf{getType}(v, h'''') = t$. From Lemma 10, we know the only objects that can change in a call is o_r, v' and v'' which are updated using update, hence we can conclude $\Gamma'' * (r : C[\mathcal{U}'], v_i : t'_i) \vdash h'''', o$ and $h' <: h''''$. Then by Lemma 12 and assumption we know. $\exists \Gamma'$ such that $\Gamma' <: \Gamma'' * (r : C[\mathcal{U}'], v_i : t'_i)$ and $\Gamma' \vdash env'_P, h', o$ hence we can conclude (Call$_B$).

□

Proof of Theorem 4

Proof. Induction in the structure of e.
Case (Seq)**:** Assume $\Gamma \vdash e_1; e_2 : t \triangleright \Gamma'$ and $\Gamma \vdash env_P, h, o$. From (Seq) we have $\Gamma \vdash e_1 : t' \triangleright \Gamma''$ and $\Gamma'' \vdash e_2 : t \triangleright \Gamma'$. By IH we have either $env_P, h \vdash \langle o, e_1\rangle \rightarrow v' \dashv env''_P, h''$ or $\langle (o, env_P), h, e_1\rangle \Rightarrow^{\omega}$ and either $env''_P, h'' \vdash \langle o, e_2\rangle \rightarrow v \dashv env'_P, h'$ or $\langle (o, env''_P), h'', e_2\rangle \Rightarrow^{\omega}$.

If $env_P, h \vdash \langle o, e_1\rangle \rightarrow v' \dashv env''_P, h''$ and $env''_P, h'' \vdash \langle o, e_2\rangle \rightarrow v \dashv env'_P, h'$, then using (Seq$_B$) we conclude $env_P, h \vdash \langle o, e_1; e_2\rangle \rightarrow v \dashv env'_P, h'$.

If $\langle env_S \cdot (o, env_P), h, e_1\rangle \Rightarrow^{\omega}$ then by Lemma 15 we have $\langle env_S \cdot (o, env_P), h, e_1; e_2\rangle \Rightarrow^{\omega}$.

Finally if $env_P, h \vdash \langle o, e_1\rangle \rightarrow v' \dashv env''_P, h''$ and $\langle env_S \cdot (o, env''_P), h'', e_2\rangle \Rightarrow^{\omega}$. By Theorem 1 we have $\langle env_S \cdot (o, env_P), h, e_1\rangle \Rightarrow^* \langle env_S \cdot (o, env''_P), h'', v'\rangle$. By Lemma 4 we have $\langle env_S \cdot (o, env_P), h, e_1; e_2\rangle \Rightarrow^* \langle env_S \cdot (o, env''_P), h'', v'; e_2\rangle$ and with (Seq$_S$) we have $\langle env_S \cdot (o, env''_P), h'', v'; e_2\rangle \Rightarrow \langle env_S \cdot (o, env''_P), h'', e_2\rangle$. We now have $\langle env_S \cdot (o, env_P), h, e_1; e_2\rangle \Rightarrow {}^*\langle env_S \cdot (o, env''_P), h'', e_2\rangle \Rightarrow^{\omega}$ and by Lemma 14 conclude $\langle env_S \cdot (o, env_P), h, e_1; e_2\rangle \Rightarrow^{\omega}$.

Case (FLD)**:** Assume $\Gamma \vdash f = e : \mathsf{void} \triangleright \Gamma'$. By (FLD) we have $\Gamma \vdash e : t \triangleright \Gamma', f \mapsto t'$. By IH we have either $env_P, h \vdash \langle o, e\rangle \to v \dashv env'_P, h'$ or $\langle env_S \cdot (o, env_P), h, e\rangle \Rightarrow^{\omega}$.

If $env_P, h \vdash \langle o, e\rangle \to v \dashv env'_P, h'$ then we conclude with (ASGN$_\mathrm{B}$) $env_P, h \vdash \langle o, f = e\rangle \to \mathsf{unit} \dashv env'_P, h'[o.f \mapsto v]$.

Otherwise if $\langle env_S \cdot (o, env_P), h, e\rangle \Rightarrow^{\omega}$ then by Lemma 15 we have $\langle env_S \cdot (o, env_P), h, f = e\rangle \Rightarrow^{\omega}$.

Case (CALL)**:** Assume $\Gamma \vdash e : t \triangleright \Gamma'$. From we have $\Gamma'' <: \Gamma$ where $\Gamma'' = \Gamma''' * \{r : C[\mathcal{U}], v_i : t_i\}$ as well as $t\ m(t_1 \to t'_1\ x_1, t_2 \to t'_2\ x_2)\{e\} \in C.\mathsf{methods}$. Furthermore we have $\mathcal{U} \xrightarrow{m} \mathcal{U}'$ and $\Gamma' <: \Gamma''' * \{r : C[\mathcal{U}'], v_i : t'_i\}$.

From Lemma 12 we know $\exists h'' <: h$ s.t. $\Gamma'' \dashv env_P, h'', o$. Let $o_r = \mathsf{extract}(r, h'', env_P, o)$, $v' = \mathsf{extract}(v_1, h'', env_P, o)$, $v'' = \mathsf{extract}(v_2, h'', env_P, o)$. From (WTC) we know that $h''(o_r).\mathsf{usage} = \mathcal{U}$ and hence $h''(o_r).\mathsf{usage} \xrightarrow{m} \mathcal{U}'$. From (WTC) we also have that $\vdash h'', o$, and since $o_r \in \mathsf{reach}(h'', o)$ we know $\exists \Phi, \Phi'$ s.t. $\Phi \vdash h''(o_r).\mathsf{class}[h''(o_r).\mathsf{usage}] \dashv \Phi'$. Since we know $\mathcal{U} \xrightarrow{m} \mathcal{U}'$ we must have concluded using (TCBR) that $\Phi, x_1 \mapsto t_1, x_2 \mapsto t_2 \vdash e : t \triangleright \Phi', x_1 \mapsto t'_1, x_2 \mapsto t'_2$.

We must show that $\Phi, \{x_1 \mapsto t_1, x_2 \mapsto t_2\} \vdash \{x_1 \mapsto v', x_2 \mapsto v''\}, h'', o_r.\ \Phi, \{x_1 \mapsto t_1, x_2 \mapsto t_2\} \vdash \{x_1 \mapsto v', x_2 \mapsto v''\}$, since we know that $t_i = \Gamma(v_i)$, hence $\mathsf{getType}(v', h'') = t_1$ and $\mathsf{getType}(v'', h'') = t_2$. The remaining premises follows from (WTH).

By IH we then conclude that $\{x_1 \mapsto v', x_2 \mapsto v''\}, h''[o_r.\mathsf{usage} \mapsto \mathcal{U}'] \vdash \langle o_r, e\rangle v \dashv \{x_1 \mapsto v^{(3)}, x_2 \mapsto v^{(4)}\}, h^{(5)}$. Now let $h^{(4)}, env''_P = \mathsf{update}(v^{(3)}, v_1, h^{(5)}, env_P, o)$, $h^{(3)}, env'_P = \mathsf{update}(v^{(4)}, v_2, h^{(4)}, env''_P, o)$ and $h' <: h^{(3)}$. Then we can conclude with (CALL$_\mathrm{B}$) that $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \to v \dashv env'_P, h'\rangle$.

Case (LINREF)**:** Assume $\Gamma, r \mapsto t \vdash r : t \triangleright \Gamma, r \mapsto \bot$. From (LINREF) we know that $\neg\mathsf{term}(t)$. First assume $r = f$, and let $o_r = \mathsf{extract}(f, h, env_P, o)$. From $\Gamma, r \mapsto t \vdash env_P, h, o$ we know that $\mathsf{getType}(o_r, h) = t$. Then we can conclude using (LINREFF$_\mathrm{B}$) that $env_P, h \vdash \langle o, f\rangle \to h(o).f \dashv env_P, h[o.f \mapsto \mathsf{null}]$. Now assume $r = x$, and let $o_r = \mathsf{extract}(x, h, env_P, o)$. From $\Gamma, r \mapsto t \vdash env_P, h, o$ we know that $\mathsf{getType}(o_r, h) = t$. Then we can conclude using (LINREFP$_\mathrm{B}$) that $env_P, h \vdash \langle o, x\rangle \to env_P(x) \dashv env_P[x \mapsto \mathsf{null}], h$.

Case (UNREF)**:** Assume $\Gamma, r \mapsto t \vdash r : t \triangleright \Gamma, r \mapsto t$. From (UNREF) we know that $\mathsf{term}(t)$. First assume $r = f$, and let $o_r = \mathsf{extract}(f, h, env_P, o)$. From $\Gamma, r \mapsto t \vdash env_P, h, o$ we know that $\mathsf{getType}(o_r, h) = t$. Then we can conclude using (UNREFF$_\mathrm{B}$) that $env_P, h \vdash \langle o, f\rangle \to h(o).f \dashv env_P, h$. Now assume $r = x$, and let $o_r = \mathsf{extract}(x, h, env_P, o)$. From $\Gamma, r \mapsto t \vdash env_P, h, o$ we know that $\mathsf{getType}(o_r, h) = t$. Then we can conclude using (UNREFP$_\mathrm{B}$) that $env_P, h \vdash \langle o, x\rangle \to env_P(x) \dashv env_P, h$.

Case (VAL)**:** Assume $\Gamma \vdash v : t \triangleright \Gamma$ and $\Gamma \vdash env_P, h, o$. From (VAL) we know $v \in \{\mathsf{unit}, \mathsf{null}, \mathsf{false}, \mathsf{true}\}$, hence we can directly conclude with (BVAL$_\mathrm{B}$) $env_P, h \vdash \langle o, v\rangle \to v \dashv env_P, h$.

Case (NEW)**:** Assume $\Gamma \vdash \mathsf{new}\ C : C[C.\mathsf{usage}] \triangleright \Gamma'$ and $\Gamma \vdash env_P, h, o$. Then we can directly conclude with (NEW$_\mathrm{B}$) $env_P, h \vdash \langle o, \mathsf{new}\ C\rangle \to o' \dashv env_P, h[o' \mapsto \langle C[C.\mathsf{usage}], C.\mathsf{initvals}\rangle]$ where o' is fresh in h.

Case (IF)**:** Assume $\Gamma \vdash \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\} \triangleright \Gamma'$ and $\Gamma \vdash env_P, h, o$. From (IF) we know $\Gamma \vdash r.m(v_1, v_2) : \mathsf{bool} \triangleright \Gamma'$ hence by IH we have either $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \triangleright v' \dashv env''_P, h''$ or $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \Rightarrow^{\omega}$.

If $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \Rightarrow^{\omega}$ then by Lemma 15 we have $\langle env_S \cdot (o, env_P), h, \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \Rightarrow^{\omega}$.

Otherwise if $env_P, h \vdash \langle o, r.m(v_1, v_2)\rangle \triangleright v' \dashv env''_P, h''$, then by Theorem 3 we have $\Gamma'' \vdash env''_P, h'', o$ and that $v' \in \{\mathsf{true}, \mathsf{false}\}$. We assume $v' = \mathsf{true}$ but the case for $v' = \mathsf{false}$ is similar. From (IF) we have $\Gamma''' * \{r : C[\langle u_1, u_2\rangle]\} <: \Gamma''$. By Lemma 12 we have $\exists h'''$ s.t. $h''' <: h''$ and $\Gamma''' * \{r : C[\langle u_1, u_2\rangle]\} \vdash env''_P, h''', o$. From (WTC-S) we have $h''' = h^{(4)} * \{o' \mapsto (C[\langle u_1, u_2\rangle], env_f)\}$ where $o' = \mathsf{extract}(r, env_P, h, o)$. Directly we see that $h'''(o').\mathsf{usage} \xrightarrow{\mathsf{true}} u_1$. Now let $h^{(5)} = h^{(4)} * \{o' \mapsto (C[u_1], env_f)\}$. We see that $\Gamma''' * \{r : C[u_1]\} \vdash env''_P, h^{(5)}, o$. Now let $\Gamma^{(4)} <: \Gamma''' * \{r : C[u_1]\}$. By Lemma 12 we know $\exists h^{(6)}$ s.t. $h^{(6)} <: h^{(5)}$ and $\Gamma^{(4)} \vdash env''_P, h^{(6)}, o$. By IH we then have either $env''_P, h^{(6)} \vdash \langle o, e_1\rangle \to v \dashv env'_P, h'$ or $\langle env''_P, h^{(6)}, e_1\rangle \Rightarrow^{\omega}$.

If $env''_P, h^{(6)} \vdash \langle o, e_1\rangle \to v \dashv env'_P, h'$ then we conclude with (IF_B) that $env_P, h \vdash \langle o, \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \to v \dashv env'_P, h'$.

Otherwise if $\langle env''_P, h^{(6)}, e_1\rangle \Rightarrow^{\omega}$. By Theorem 1 we have $\langle env_S \cdot (o, env_P), h, r.m(v_1, v_2)\rangle \Rightarrow^* \langle env_S \cdot (o, env''_P), h'', v'\rangle$. By Lemma 4 we have $\langle env_S \cdot (o, env_P), h, \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \Rightarrow^* \langle env_S \cdot (o, env''_P), h'', \mathsf{if}\ (v')\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle$. By ($\text{IF-TRUE}_\text{S}$) we have $\langle env_S \cdot (o, env''_P), h'', \mathsf{if}\ (v')\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \Rightarrow \langle env_S \cdot (o, env''_P), h^{(6)}, e_1\rangle$. Hence by Lemma 14 we conclude $\langle env_S \cdot (o, env_P), h, \mathsf{if}\ (r.m(v_1, v_2))\ \{e_1\}\ \mathsf{else}\ \{e_2\}\rangle \Rightarrow^{\omega}$.

Case (LAB)**:** Assume $\Gamma \vdash k : e : \mathsf{void} \triangleright \Gamma'$. From Lemma 9 we know $\Gamma \vdash e\{\mathsf{continue}\ k/k : e\} : \mathsf{void} \triangleright \Gamma'$. By our IH we know that either $env_P, h \vdash \langle o, e\{\mathsf{continue}\ k/k : e\}\rangle \to v \dashv env'_P, h'$ or $\langle env_S \cdot (o, env_P), h, e\{\mathsf{continue}\ k/k : e\}\rangle \Rightarrow^{\omega}$. If $env_P, h \vdash \langle o, e\{\mathsf{continue}\ k/k : e\}\rangle \to v \dashv env'_P, h'$ then using (LAB_B) we conclude $env_P, h \vdash \langle o, k : e\rangle \to v \dashv env'_P, h'$.

Otherwise if $\langle env_S \cdot (o, env_P), h, e\{\mathsf{continue}\ k/k : e\}\rangle \Rightarrow^{\omega}$ then by (LAB_S) we have $\langle env_S \cdot (o, env_P), h, k : e\rangle \Rightarrow \langle env_S \cdot (o, env_P), h, e\{\mathsf{continue}\ k/k : e\}\rangle \Rightarrow^{\omega}$ hence by Lemma 14 we have $\langle env_S \cdot (o, env_P), h, k : e\rangle \Rightarrow^{\omega}$.

□