João Daniel da Luz Mota

Bachelor in Computer Science and Engineering

Coping with the reality: adding crucial features to a typestate-oriented language

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: António Maria Lobo César Alarcão Ravara,
Associate Professor,
NOVA School of Science and Technology
Co-adviser: Marco Giunti,
Researcher,
NOVA School of Science and Technology

Examination Committee

Chair:	Hervé Miguel Cordeiro Paulino, Associate Professor, NOVA School of Science and Technology
Rapporteur:	Ornela Dardha, Lecturer, School of Computing Science, University of Glasgow
Members:	António Maria Lobo César Alarcão Ravara Marco Giunti

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2021

Coping with the reality: adding crucial features to a typestate-oriented language

Acknowledgements

Throughout the writing of this thesis I have received a great deal of support and assistance.

I would first like to thank my adviser, Professor António Ravara, and co-adviser, Researcher Marco Giunti, for the guidance and direction provided. Your feedback was crucial and allowed me to organize my work and write this thesis.

I would like to acknowledge all the professors who during these years have given me the skills and knowledge necessary to push me to become a good engineer in the future.

I would also like to thank my colleagues, which during these years, through the exchanging of ideas, have helped me grow.

Finally, I could not have completed this work without the support of my parents, Cidália and Mário, who were very kind and patient with me throughout these years.

"Unless the Lord builds the house, those who build it labor in vain." (Psalm 127:1; ESV)

Abstract

Detecting programming errors and vulnerabilities in software is increasingly important, and building tools that help with this task is an area of investigation, crucial for the industry these days. When programming in an object-oriented language, one naturally defines stateful objects that are non-uniform, i.e., their methods' availability depends on their internal state. One might represent their intended usage protocol with an automaton or a state machine. Behavioral types allow to statically check if all the code of a program respects the usage protocol of each object.

In this thesis we present a tool that extends Java with typestate definitions. These typestates are associated with Java classes and define the behavior of instances of those classes, specifying the sequences of method calls allowed. This tool checks statically that method calls happen in order, following the specified behavior.

The tool was implemented in Kotlin as a plugin for the Checker Framework. It is a new implementation of the Mungo tool and supports prevention of null pointer errors, state transitions depending on return values, assurance of protocol completion, droppable states, and association of protocols with classes from the standard Java library or from third-party libraries. Additionally, the tool integrates behavioral types with access permissions, allowing objects to be shared in a controlled way using a language of assertions. This language of assertions supports concepts like packing and unpacking, including unpacking of aliases objects, and transferring of permissions between aliases. To relieve the programmer from manually writing all the necessary assertions, the tool implements an inference algorithm which analyzes the code statically and, given the uses of objects, constructs all the required assertions.

Keywords: Behavioral types, object-oriented programming, typestates, session types, access permissions, inference

Resumo

A deteção de erros de programação e vulnerabilidades no *software* é cada vez mais importante, e a criação de ferramentas que ajudem nesta tarefa é uma área de investigação crucial para a indústria atualmente. Ao programar numa linguagem orientada a objetos, definem-se naturalmente objetos com estado que não são uniformes, ou seja, a disponibilidade dos seus métodos depende do seu estado interno. Pode-se representar o protocolo de uso pretendido com um autómato ou uma máquina de estados. Os tipos comportamentais permitem verificar estaticamente se todo o código de um programa respeita o protocolo de uso de cada objeto.

Nesta tese apresentamos uma ferramenta que estende o Java com definições de *typestates*. Esses estão associados às classes Java e definem o comportamento das instâncias dessas classes, especificando as sequências de chamadas de métodos permitidas. Esta ferramenta verifica estaticamente se as chamadas de métodos ocorrem pela ordem correta, seguindo o comportamento especificado.

A ferramenta foi implementada em Kotlin como um *plugin* para o Checker Framework. É uma implementação nova da ferramenta Mungo e suporta a prevenção de erros de ponteiro nulo, transições de estado dependendo de valores de retorno, asseguração da conclusão dos protocolos, objetos que podem ser «largados», e a associação de protocolos com classes da biblioteca padrão do Java ou de terceiros. Além disso, esta integra tipos comportamentais com permissões de acesso, permitindo que objetos possam ser partilhados por meio de uma linguagem de asserções. Esta linguagem de asserções oferece suporte para conceitos como *packing* e *unpacking*, incluindo *unpacking* de objetos partilhados, e transferência de permissões entre variáveis que apontam para o mesmo objeto. Para aliviar o programador de escrever manualmente todas as asserções necessárias, a ferramenta implementa um algoritmo de inferência que analisa o código estaticamente e, consoante os usos dos objetos, constrói todas as asserções necessárias.

Palavras-chave: Tipos comportamentais, programação orientada a objetos, *typestates*, *session types*, permissões de acesso, inferência

Contents

List of Figures

List of Tables

Glossary

access permission	An abstract capability that characterizes the way a shared resource can be accessed by multiple references [74].
aliasing	A situation in which a data location can be accessed through more than one reference in a program.
assertion	A logical proposition that should always hold at a given point in code execution.
behavioral types	Type disciplines that describe properties associated with the behavior of programs [43].
dynamic checking	Type-checking process that occurs in runtime [14].
explicitly typed language	A language where types appear in the syntax of program sources [14].
gradual typing	Process that combines both static and dynamic checking in the same program [75].
ill typed program	A program that does not pass the type-checker [14].
implicitly typed language	A language where types do not appear in the syntax of program sources [14].
safe language	A language in which all program fragments are safe [14].
safe program	A program that does not cause untrapped errors to occur [14].
session type	A notion of behavioral types where the interactions between different parties are described [43, 78].
static checking	Type-checking process that occurs in compilation time [14].

strongly checked language	A language in which all programs are well-behaved [14].
trapped error	An error that causes the computation to stop immediately. For example, division by zero or accessing an illegal memory address [14].
type	A range of values a program variable can assume during the execution of a program [14].
type-checker	Tool or algorithm that performs type-checking [14].
type-checking	The process of checking programs to ensure they are well-behaved [14].
type sound language	A language where the absence of wrong behavior is ensured for all possible runs expressed within that language [14].
typestate	A notion of behavioral types where the type of an entity depends on the operations that are permitted, when at a particular state [43, 32].
type system	A tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute [71].
typed language	A language where the variables can be given nontrivial types [14].
untrapped error	An error that does not cause the computation to stop immediately and that can go unnoticed. For example, accessing an array off bounds in a language that has no runtime bounds checks [14].
untyped language	A language that does not have types or that has a single universal type that contains all values [14].
usage type	A description of all the possible states and permitted operations for each state of an entity.
weakly typed language	A language where the set of forbidden errors does not include all untrapped errors [14].
well typed program	A program that passes the type-checker [14].
well-behaved program	A safe program where no forbidden errors can occur during the execution [14].

Chapter 1

Introduction

1.1 Context

Detecting programming errors and vulnerabilities in software is increasingly important, and building tools that help with this task is an area of investigation, crucial for the industry these days.

Programming errors result in programs that might malfunction in many ways. The most common mistakes involve mixing values of different types, calling non-existing functions, calling functions with the wrong number of parameters, accessing unauthorized parts of the memory, creating loops that might not terminate, de-referencing null pointers, dangling pointers, division by zero... These mistakes might cause computations to do not progress, producing unexpected behaviors or crashing due to runtime errors.

In modern programming languages, some of these common errors are detected thanks to type systems implemented in compilers. Type systems ensure that if a program is well-typed, nothing "goes wrong" [63]. By "wrong", we mean the kind of bad behavior that the type system was designed to rule out. Usually, type systems are used to detect errors statically, by analyzing the source code, preventing the errors from happening at runtime. Other techniques, usually not integrated with compilers, use deductive logics [1] and model checking [17]. Since we are interested in preventing errors before the code is run, we will focus on static error detection approaches.

1.2 Problem

Type systems nowadays are able to detect a lot of errors, but there are errors that are not prevented in some programming languages. For example, C and Java still do not prevent null de-referencing. Fortunately, languages like OCaml do prevent that, by considering

null not being a value of every type. Go and Rust are also examples of modern languages detecting more than data-errors: Go does dynamic deadlock detection; Rust controls resource interference statically.

Unfortunately, the type of errors detected in modern languages is still limited. Modern languages for instance, do not statically ensure that methods are called in the right moment and in the right order, which is a source of many errors, like accessing a variable that was not initialized [6]. Additionally, the language frameworks that do allow one to verify that methods are called in the right order either have limitations or require expert users, and not average programmers, to provide complex specifications.

One real example of a method being called out of order was a bug found[1] in Jedis. Jedis[2] is a Java client for Redis[3], an in-memory database that persists on disk. The error happened when there was an attempt to close a socket that timed out, in other words, there was an operation being available on a state that should not allow that.

1.3 Contributions

In this thesis, we present the implementation of a tool that type-checks a Java program where objects are associated with typestates. These typestates are associated with Java classes and define the behavior of instances of those classes, specifying the sequences of method calls allowed. It checks statically that method calls happen in the prescribed order, following the specified behavior.

This tool is a new implementation of Mungo [82, 54] which fixes issues and adds new features. It was implemented in Kotlin [49, 50] as a plugin for the Checker Framework [79, 70].

The major features supported by this tool are:

- checking the **absence of null pointer errors**, fixing some issues that Mungo currently has;
- checking that **protocols of objects are completed**, even in some corner cases that Mungo was not checking;
- a **language of assertions** that focuses on allowing a program that uses typestates to be type-checked **even in the presence of aliasing**;
- an **inference algorithm** which analyzes the code statically and infers all the required assertions.

The language of assertions employs the notion of **access permissions** [10] and supports concepts like **packing** and **unpacking** [25], **unpacking of aliases objects**, and

[1] https://github.com/xetorthio/jedis/issues/1747
[2] https://github.com/xetorthio/jedis
[3] https://github.com/antirez/redis

transferring of permissions between aliases. With this language, it is possible to **share objects between different threads**, having assurance that their use follows the specified behavior in the protocol, and that there is **no interference** between method calls that mutate the state of the object. Furthermore, the **inference algorithm** infers all the required assertions by building a constraints system and solving it with Z3 [22]. This relieves the programmer from manually writing all the necessary assertions.

The tool also includes other features that improve the developer experience:

- support for **protocols to be associated with classes** from the standard Java library or from third-party libraries, allowing the use of objects to be verified even when the source code of their classes is not available to be edited with a *Typestate* annotation (e.g. ensuring that *hasNext* is called before *next* in an iterator from the standard library);
- support for **"droppable" states**, which allow one to specify states in which an object may be "dropped" (i.e. stop being used) without having to reach the final state;
- support for **transitions of state to depend on boolean values** returned by methods, not just on enumeration values, as the current version of Mungo requires;
- invalid sequences of method calls are ignored when analyzing the use of objects stored inside other objects by taking into account that the methods of the outer object will only be called in the order specified by the corresponding protocol, thus **avoiding false positives**.

1.4 Thesis outline

The thesis is structured as follows:

- Chapter 2 presents a study on the concept of behavioral types;
- Chapter 3 presents a study on tools that employ the concept of behavioral types;
- Chapter 4 presents the first version of the tool, where linear use of objects is enforced;
- Chapter 5 presents a study on the concept of access permissions;
- Chapter 6 presents a study on tools that employ the concept of access permissions;
- Chapter 7 presents the second version of the tool, which allows objects to be shared;
- Chapter 8 presents a summary and discusses future work.

Chapter 2

Theoretical Work on Behavioral Types

When designing a programming language, it is important to forbid a set of errors from ever happening. This set should include all untrapped errors, and some trapped errors. If no forbidden error can occur during the execution of a program, we say that program is **well-behaved**. A well-behaved program is also safe. Safety prevents errors being unnoticed for too long, reducing debugging time and preventing arbitrary behavior from happening later [14].

Traditional work on type systems has focused on the result of computations. With the growth of concurrent systems, there is a need to verify the behavior of computations, not just the result given. **Behavioral Types** are type disciplines that describe properties associated with the behavior of programs [43]. Type systems that include this notion, allow for the static verification of interactions and protocol compliance, like ensuring that methods on an object are called in the correct order.

In this chapter, we study the concept of behavioral types. Section 2.1 presents the notion of **typestates**. Section 2.2 presents the concept of **session types**. In section 2.3 we discuss the differences between these and **usage types**. Section 2.4 presents a motivating example for the use of behavioral types.

2.1 Typestates

Typestates are a notion of behavioral types where the type of an entity depends on the operations that are permitted, when at a particular state [43]. Each type has associated with it a set of typestates, and each typestate is the set of operations that can be safely executed in that state. Therefore, typestates are similar to finite-state machines and type-checking is then able to reject programs where there are sequences of method calls that are not admissible [43]. For example, in a Java iterator, the *next* method should only be

called after *hasNext* was called.

Garcia, Tanter, Wolff and Aldrich present in [32] the concept of typestate-oriented programming, where the language directly supports the expression of typestates, instead of having typestate checkers as an additional layer. One example of such language is Plaid [77, 37]. In the Plaid programming language, the class of an object represents its typestate, and that class can change dynamically during runtime. Not only the interface (i.e. available methods) depends on the typestate, the behavior also depends on the current state [32].

2.2 Session types

Session types describe the interactions between different partners. Originally, this concept had in mind only two parties running in parallel and communicating via message passing [41, 78, 86]. Takeuchi, Honda and Kubo were the first to present formally in [78] a small language and its typing system based on the concept of interaction between processes, important for concurrent systems. Interaction is seen as a chain of actions between two parties, sometimes interleaved with actions with other parties [78]. The language also provides a type inference system, where it can be proven that if the program is well-typed, there will be no inconsistent communication patterns [78].

Since then, session types have been the subject of great study, and have been integrated into some object-oriented programming languages, like Plaid [37], Mool [13] and Mungo [82].

2.3 Typestates/Session types/Usage types

In the study of behavioral types, terms like *typestate*, *session type* and *usage type* appear frequently and are sometimes used interchangeably. The term *typestate* is usually associated with a particular state, which can be seen as a set of available operations or the pre- and post-conditions related with each operation. *Session types* have their roots in typed π-calculus - a model of computation for concurrent systems that can represent processes, parallel composition of processes, communication between those through channels and creation of new channels [89]. *Session types* allow for protocol checking in channels, but they have been incorporated in object-oriented languages where method calls are the communication primitive, without the need for explicit channel creation. They can be seen as a set of typestates. A *usage type* is a description of all the possible states and permitted operations for each state.

Although originally *typestates* were in line with a contract-oriented, or assertion-based, approach, it is possible to easily transform the assertion-like typestates into usage-like ones [85]. In Mungo [82], a *typestate* is no different from a *usage type*.

The following is an example, from the language Bica [33], of a *usage type*, which describes the *session type* of an object, which has a set of possible states (typestates). In

this example, the type system ensures that method calls happen in order, for example, ensuring that the *eof* method is called before *read*, to make sure the end of the file was not reached yet.

Listing 2.1: Usage type example

```
enum Res {OK, ERROR}
class FileReadToEnd {
    session Init
    where Init = { open: <OK: Open, ERROR: end> }
          Open = { eof: <TRUE: Close, FALSE: Read> }
          Read = { read: Open }
          Close = { close: end }
    Res open() {...}
    Bool eof() {...}
    String read() {...}
    void close() {...}
}
```

2.4 Motivating example: Checker Framework

The Checker Framework is a tool that makes Java's type system more powerful and useful [79]. It includes plugins that verify the absence of many types of bugs: null pointer exceptions, unintended side effects, SQL injections, concurrency errors, mistaken equality tests. It also allows the programmer to add new typing rules and enforcing those by the creation of new plugins. Checker Framework has been successful at detecting and confirming the absence of errors in Java code [27, 70].

One of the most well known plugins is the Nullness Checker. It ensures that null pointer exceptions are never thrown[1]. By default, the type of objects is considered different from *null*. If one wants a nullable type, it must indicate so with the *@Nullable* annotation, like in the following example.

Listing 2.2: Checker Framework: Nullness Checker example

```
import org.checkerframework.checker.nullness.qual.Nullable;
public class ValueWrapper {
    private @Nullable Object obj = null;
    public boolean hasValue() {
        return obj != null;
    }
    public Object getValue() { return obj; }
    public void setValue(@Nullable Object obj) {
        this.obj = obj;
    }
}
```

[1] https://checkerframework.org/manual/#nullness-checker

This example presents a wrapper object which may store a reference to another object or the *null* value. The *hasNext* method checks if an object is stored. The *getValue* method retrieves the object (assuming it is not null). The *setItem* method stores a new object.

The Nullness Checker reports an error on line 7 indicating that the *obj* variable may be *null*. It is a real error that could happen if *hasValue* is not called before calling *getValue*. Unfortunately, there is no way to specify that we expect the nullness of *obj* to be checked before with the *hasValue* method, creating a scenario where we need to use defensive programming: always check that *obj* is not *null* inside *getValue*. This gives an example where detecting data-errors is not enough and motivates the inclusion of behavior information in types, which would enforce that methods are called in the correct order and would avoid false positives like this, by pruning execution paths that do not occur.

2.5 Conclusion

Type systems have been focused on the result of computations. Now, there is the need to also verify the behavior of computations and ensure that methods are called in the correct order. Although the need for *behavioral types* is increased by the growth of concurrent systems, this notion is also important for verifying properties of sequential programs. Like in the motivating example, ensuring that methods are called in the proper order increases safety and allows false positives to be avoided, by discarding execution paths that do not occur. In the following chapter, we analyze some tools and programming languages that incorporate the concept of *behavioral types*.

Chapter 3

Practical work on Behavioral Types

This chapter presents tools and object-oriented languages that incorporate the concept of behavioral types. In the following sections, we will present programming languages and tools that employ a typestate-oriented approach (section 3.1) and ones that employ a usage types approach (section 3.2). Section 3.3 summaries this chapter.

3.1 Typestate-oriented approaches

3.1.1 Plaid

Plaid [77, 37] is a programming language designed for component-based computing in concurrent software. One of Plaid's main characteristics is typestate-oriented programming [32], where the class of an object represents its typestate, and that class can change dynamically during runtime. Not only the interface (i.e. available methods) depends on the typestate, the behavior also depends on the current state. Plaid also supports gradual typing [75], which allows a programmer to mix dynamically and statically typed code. Additionally, Plaid programs are interoperable with Java programs [38].

In Plaid, each class is represented as a state and each state is represented as a sub-state of that. The main state should contain all the variables and methods available in all sub-states. Each sub-state should contain the variables and methods available only on those sub-states. Additionally, Plaid allows composition of multiple states in complex ways. It supports *hierarchical-states*, which are states that are composed of other states; *and-states*, which are states where both must be present, modeled using the *with* keyword; and *or-states*, which are states where only one can be present in an object - a state that is a *case of* another state [77].

The following is a simple example (from [77]) of a File with two sub-states: one where the file is opened, allowing the file to be read or closed, and another where the file is

closed and may be opened again. State transitions are declared with the use of *this <- NewState*.

Listing 3.1: Plaid example: File

```
state File {
    val filename;
}
state OpenFile case of File = {
    val filePtr;
    method read() { ... }
    method close() { this <- ClosedFile; }
}
state ClosedFile case of File {
    method open() { this <- OpenFile; }
}
```

Listing 3.2: Plaid example: File use

```
method readClosedFile(f) {
    f.open();
    val x = f.read();
    f.close();
    x; //return
}
```

The main features mentioned previously, constitute Plaid's main advantages. Unfortunately, it is not clear how one may specify, on a class declaration, what the initial state of an object after initialization should be. It seems that this is done in client code every time an object is initialized, using the @ operator, like in the following example:

Listing 3.3: Plaid example: File initialization

```
val file = new File @ ClosedFile;
```

That might compromise safety if an object starts in a state that was not designed to be the initial state. Furthermore, Plaid does not include a way to force an object to reach a certain final state, which could be useful for ensuring protocol termination.

3.1.2 Fugue

Fugue is a modular static checker for languages that compile to the Common Language Runtime[1], integrating typestates with an object-oriented programming language. Fugue allows the programmer to add declarative specifications on interfaces, marking methods that are used for allocating or releasing resources, limiting the order in which object's methods are called, or even providing preconditions and postconditions. Fugue then ensures that resources are not used before allocated or after being released, ensures that methods are called in correct order, and that preconditions are met before a method is

[1] https://docs.microsoft.com/en-us/dotnet/standard/clr

called. Fugue is modular because it only analyzes method declarations, allowing for faster checking [24].

The following is an example of a class for socket objects from [24].

Listing 3.4: Fugue example: A class for socket objects

```
[WithProtocol("raw","bound","connected","down")]
class Socket {
    [Creates("raw")]
    public Socket (...);
    [ChangesState("raw", "bound")]
    public void Bind (EndPoint localEP);
    [ChangesState("raw", "connected"), ChangesState("bound", "connected")]
    public void Connect (EndPoint remoteEP);
    [InState("connected")]
    public int Send (...);
    [InState("connected")]
    public int Receive (...);
    [ChangesState("connected", "down")]
    public void Shutdown (SocketShutdown how);
    [Disposes(State.Any)]
    public void Close ();
}
```

The *WithProtocol* annotation declares the possible states. The *Creates* annotation specifies the initial state of an object when created. The *ChangesState* annotation specifies that if the object is in the state indicated on the first argument, the method may be called and the object transits to the state indicated in the second argument. The *Dispose* annotation declares that a method is used for releasing resources. It is also possible to declare the availability of fields in classes, like in the following example from [24].

Listing 3.5: Fugue example: A web page fetcher using a socket object

```
[WithProtocol("open", "closed")]
class WegPageFetcher
{
    [InState("connected", WhenEnclosingState="open"),
     NotAliased(WhenEnclosingState="open")]
    [Unavailable(WhenEnclosingState="closed")]
    private Socket socket;
    ...
}
```

Fugue is very useful and has been used to check, for example, the implementation of an internal Microsoft Research web site, detecting multiple errors, including connections to a database not being properly disposed. Still, Fugue has some limitations. It does not allow protocol checking on static fields, it ignores the concurrency aspect of the language (i.e. does not control the use of shared variables) and ignores exception control flow during analysis [24].

3.1.3 Contractor.NET

Contractor.NET [91] is a Visual Studio extension that uses contract specifications with typestate information to verify client code. The typestate information is inferred from the class source code in a way that is *enabled preserving* [15]. This means that states are grouped if they have the same set of actions enabled. This level of abstraction has been useful to detect issues in various case studies including specifications of Microsoft Server protocols [91]. The following is an example (from [91]) of a class implementation with contract specifications and the corresponding inferred typestate (figure 3.1).

Listing 3.6: Contractor.NET example: Train door controller

```
public class Door {
    public bool danger, closed, moving;
    private void Invariant() {
        Contract.Invariant(danger ? !closed : true);
    }
    public Door() {
        closed = true; moving = false; danger = false;
    }
    public void Open() {
        Contract.Requires(closed && !moving); closed = false;
    }
    public void Close() {
        Contract.Requires(!closed && !danger); closed = true;
    }
    public void Start() {
        Contract.Requires(!moving);
        moving = true; if (!danger) closed = true;
    }
    public void Stop() {
        Contract.Requires(moving); moving = false;
    }
    public void Alarm() {
        Contract.Requires(!danger); danger = true; closed = false;
    }
    public void Safe() {
        Contract.Requires(danger); danger = false;
    }
}
```

The ability to infer typestates from the class source code is useful not only for later static verification of client code, but also to confirm that the contract specification was defined as intended: one only needs to look at the inferred typestate and check if it matches what was expected. The concept of *enabled preserving* provides a good compromise between size and precision of the inferred typestate [15], but since it is an over-approximation, non-determinism may exist and invalid client sequences might be accepted [91], like in the following example provided by the authors.

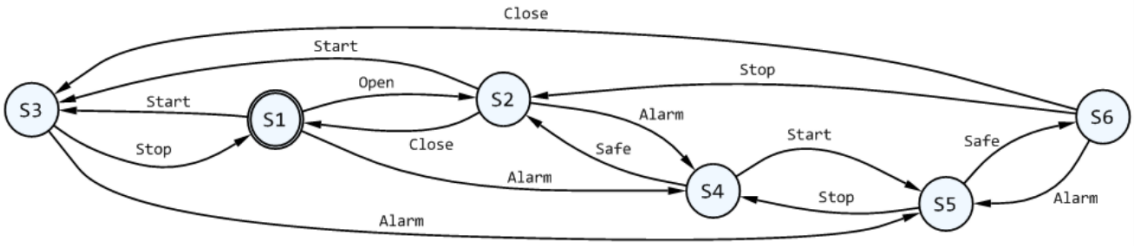

Figure 3.1: Contractor.NET example: Train door abstraction

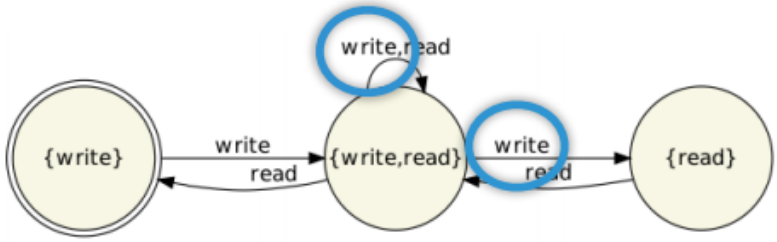

Figure 3.2: Contractor.NET example of non-determinism and over-approximation

3.2 Usage types approaches

3.2.1 Bica

Bica integrates channel session types with object-oriented programming by extending the Java language with session type annotations. These annotations are included in the class source code and describe the changes in object's interfaces. The interface of an object is the set of available methods, which changes over time dynamically. Such objects are called non-uniform. In contrast to other work on session types for object-oriented languages, channels are not required to be created and completely used within a single method. Several methods can operate on the same channel, thus allowing encapsulation of channels in objects. Bica verifies statically that clients use objects according to the specified session types [33].

Listing 3.7: Bica example: FileReadToEnd

```
enum Res {OK, ERROR}
class FileReadToEnd {
    session Init
    where Init = { open: <OK: Open, ERROR: end> }
          Open = { eof: <TRUE: Close, FALSE: Read> }
          Read = { read: Open }
          Close = { close: end }
    Res open() {...}
    Bool eof() {...}
    String read() {...}
    void close() {...}
}
```

In the previous example, the file has four states, where *open*, *eof*, *read*, *close* are available for each state respectively. The first state is *Init*. Upon calling the *open* method, the state transition depends on the return value. That return value is checked in a *switch* statement, and the state changes to *Open* or the final one. When the file is open, the client must check if the end of the file was reached. If it did, the state changes to *Close*, otherwise it changes to *Read*. On the *Read* state, one may call the *read* operation, returning to the *Open* state. On the *Close* state the *close* method is available, and if called, the file goes to the final state, where no operations are available.

In Bica, non-uniform objects must be strictly used in a linear way. That is enforced statically. It has the downside of not allowing these objects to be stored in shared data structures. Support for session types on interfaces is also lacking. If a class *C* implements interface *I*, the interface should be interpreted as the specification of minimum method availability [33].

3.2.2 Mool

Mool [13] is a object-oriented language, designed for concurrent systems, where protocols can be specified in the form of usage types, attached to class definitions. These usage types specify: the availability of methods; the tests clients must perform on the result of methods; and if the object must be used in a linear way or if it can be shared. Mool extends modular session types by eliminating channel operations, and by considering method calls as the only communication primitive, for both sequential and concurrent code. Furthermore, instead of making a distinction between linear and shared objects, Mool allows linear objects to evolve into shared ones. The status of an object that can only be referenced by a single thread is described by the *lin* (linear) qualifier. For shared objects, the *un* (unrestricted) qualifier is used. Shared objects do not evolve into linear ones since the number of references to an object is not tracked. To enable an operation to be shared, the *sync* method modifier is used [12].

The following is an example adapted from Mool's tutorial[2].

Listing 3.8: Mool example: File

```
1  class File {
2      usage lin{open; Read} where
3      Read = lin{canRead;
4                 <lin{read; Read} + lin{close; end}>};
5
6      int linesInFile; int linesRead;
7
8      unit open() {
9          linesRead = 0;
10         linesInFile = 5;
11     }
12     string read() {
13         linesRead = linesRead + 1;
14         "reading line... \n";
15     }
16     boolean canRead() {
17         linesInFile != linesRead;
18     }
19     unit close() {
20         unit;
21     }
22 }
23
24 class Main {
25     usage lin{main; end};
26     unit main() {
27         File f; f = new File();
28         f.open();
29         while(f.canRead()) {
30             printStr(f.read());
31         }
32         f.close();
33     }
34 }
```

Looking at the usage type declared on the *File* class, we observe that when a file is initialized, it starts on a state where only linear use is allowed, and only the *open* method is available. If *open* is called, the file moves to the *Read* state. The method's name and the name of the state the object transits to, upon that method being called, are separated by a semicolon. In the *Read* state, one must call *canRead* to make sure there is something to be read. The transition now depends on the return value of *canRead*. If it returns *true*, the *read* method is available otherwise, the *close* method is available and it makes a transition to the final state, where no operations can be done. This decision is represented by <> and a plus sign to signify the disjunction between the two choices.

[2]http://gloss.di.fc.ul.pt/tryit/tools/Mool

The plus sign may also be used to add more available methods on non decision states (between *{}*). An asterisk can be used to create a state where the object remains in it when its available methods are called. An example follows.

Listing 3.9: Mool example: File reader

```
class FileReader {
    usage lin{open; Next} where
        Next = lin{next; <Next + Done> + toString; Next}
        Done = *{toString + getCounter};
    ...
}
```

Channels as conceived in session type theory are special entities used to carry messages. Mool abstracts this notion by making method calls the communication primitive, which allows for more natural code in the context of object-oriented programming. Furthermore, allowing linear objects to evolve into shared ones gives more flexibility, instead of forcing a separation between linear and shared objects. Despite that, aliasing of linear types is forbidden, while aliasing of unrestricted types is completely allowed. Limited forms of aliasing without loosing track of an object state should be allowed, but this is still a topic of research. Additionally, Mool does not consider the treatment of exceptions [12].

3.2.3 Mungo

Mungo [82] is a tool that extends Java with typestate definitions [54]. These typestates are associated with Java classes and define the behavior of instances of those classes, specifying the sequences of method calls allowed. The Mungo tool checks statically that method calls happen in order, following the specified behavior.

Typestate definitions are written in *.protocol* files and associated with the respective Java class using an annotation. The following is an example of a file reader. It is adapted from an example shown in Mungo's web page [82].

Listing 3.10: Mungo example: File class

```
import mungo.lib.Typestate;
import mungo.lib.Boolean;

@Typestate("FileProtocol")
public class File {
    public File(String filename) { /* ... */ }
    public Status open() { /* ... */ }
    public Boolean hasNext() { /* ... */ }
    public byte read() { /* ... */ }
    public void close() { /* ... */ }
}
```

Listing 3.11: Mungo example: File protocol

```
typestate FileProtocol {
    Init = {
        Status open(): <OK: Open, ERROR: end>
    }
    Open = {
        Boolean hasNext(): <TRUE: Read, FALSE: Close>,
        void close(): end
    }
    Read = {
        byte read(): Open,
        void close(): end
    }
    Close = {
        void close(): end
    }
}
```

When an object is initialized, its state is the first one defined in the protocol. In this example, if we create a File, we are only allowed to call the *open* method. The *open* method returns *OK* or *ERROR*. If the return value is *OK*, we move to the *Open* state, if not, we reach the *end* state, and nothing can be done with the object later because it reached the final state. In this case, the transition depends on the return value of this method. In the *Open* state, we have the *hasNext* method to ensure that we actually have something to read. If it returns *TRUE*, we move to the *Read* state. If it returns *FALSE*, then we move to the *Close* state, and in that case, the only choice left is to close the file. On the *Read* state we can read, and then we move directly to the *Open* state again. The return value makes no difference to the transition in this case. From the *Open* or *Read* states, we can also close the file.

Other examples of Mungo's usage are available on a repository[3]. One such example[4] shows Mungo detecting a state, declared on a typestate that could never be reached, warning about a possible human error.

Listing 3.12: Mungo example: Store protocol

```
typestate StoreProtocol {
    Start = {
        BuyResult buy(String): <OK: end, KO: end>
    }
    Receipt = {
        String emitReceipt(): end
    }
}
```

[3]https://github.com/jdmota/behaviour-types-research

[4]https://github.com/jdmota/behaviour-types-research/tree/unreachable-state

Listing 3.13: Mungo example: State not reachable warning

```
1 StoreProtocol.protocol: 5-3: Warning
2     State not reachable: Receipt.
```

3.3 Summary

Much work has been done to produce tools that verify that behavior in programs is correct. Plaid, for instance, includes many great features, like typestate-oriented programming, gradual typing and control over aliasing and mutability. These concepts should be part of languages. Fugue reads annotations of methods and class fields to ensure correct use of objects. Unfortunately, adding annotations can be considered a burden, and since Fugue does not check the body of methods, some errors might escape unnoticed. Contractor.NET includes typestate inference, but since it overapproximates, invalid client sequences might be accepted. Bica is a small object-oriented language that supports session type annotations. Sadly, the project is old and no longer maintained. Mool is also an object-oriented language like Bica, and allows the programmer to specify for each state, if the object can be shared or not. Mungo is a more recent project, designed to verify Java code, but it lacks some features. In the following chapter, we discuss the implementation of a tool which is inspired by Mungo and discuss the features Mungo lacks that are present in our tool.

Chapter 4

Typestate-oriented tool: version 1

In this chapter, we are going to discuss the implementation of a tool that type-checks a Java program where objects are associated with typestates. These typestates are associated with Java classes and define the behavior of instances of those classes, specifying the sequences of method calls allowed. It checks statically that method calls happen in the prescribed order, following the specified behavior.

This tool is inspired by Mungo [82, 54], which was built in Java using the JastAdd framework [47, 28]. Initially, the objective was to extend Mungo, but the result ended up being a completely new implementation with support for new features like droppable states and association of protocols with classes from the standard Java library or from third-party libraries. Furthermore, it allows for state transitions to depend on boolean return values, not just on enumeration values, allowing for common Java code to be accepted and removing this artificial restriction that was not necessary for the verification of code. Finally, the new implementation fixes issues around the prevention of null pointer errors and analysis of the flow of execution.

This chapter presents the first version of the tool where objects must be used in a linear way (i.e. no aliases are allowed). In the following sections, we will discuss the framework and language used for the implementation (section 4.1), the features the tool provides (section 4.2), implementation details (section 4.3), and the differences between Mungo and our tool (section 4.4).

4.1 Design choices

To implement this tool, we have decided to use the Checker Framework [79, 70], instead of the JastAdd framework, and use the Kotlin language [49, 50], instead of Java. In the following subsections we will discuss the features that Checker and Kotlin give us over

JastAdd and Java, which were the tools used for the implementation of Mungo.

4.1.1 Checker Framework

The Checker Framework is a tool that enhances Java's type system to make it more powerful and useful by letting software developers detect and prevent errors in their Java programs [79]. This framework includes many plugins that help finding bugs. For example, the *Nullness Checker* is a plugin that avoids the presence of *NullPointerExceptions* that are very common is Java.

Additionally, the Checker Framework allows one to write their own plugins, including declarative and procedural mechanisms for writing type-checking rules and support for flow-sensitive local inference [70]. Plugins can be written in Java or any other Java interoperable language because Checker is well-integrated with the Java toolset [70]. Since it also provides basic checking functionalities from which we can extend, we could focus on implementing the concrete aspects of our type system.

We believe the change from JastAdd to the Checker Framework is helpful. JastAdd includes its own language that is used to define the semantics of the implemented type system. This may be useful but editor support for JastAdd is lacking (except for syntax highlighting), which can hurt productivity. Besides that, it seems JastAdd is not actively maintained while the Checker Framework, on the other hand, it is. As an example, while developing the tool, we found an issue in Checker, which we reported[1], and it took less than one hour to get a response.

Additionally, the Checker Framework has been heavily tested and used by others. For example, the Checker's team evaluated the framework by writing pluggable type-checkers and running them on over two million lines of existing code, and found hundreds of bugs in the process, including a potential null-pointer error in *Guava* [35] (which at the time was called *Google Collections*), even though it was already heavily tested [27]. They also observed the use of the framework by computer science students in their projects to eliminate null pointer errors [27]. Currently, the Checker Framework is used, for example, by *Guava*, to check the absence of null-pointer errors in its code, and used in the *KMS Compliance Checker* [53] by Amazon. Checker was also used as a foundation for *ReImInfer*, a type inference tool for *ReIm*, a type system for reference immutability, implemented for Java [42]. Finally, we note that the Checker Framework was presented in the Google Summer of Code of 2019 [16] and mentioned in an article by Oracle [52], which shows that Checker is not unknown to the Java community.

4.1.2 Kotlin

Kotlin [49, 50] is a new modern language developed by JetBrains. It is interoperable with Java which means we can use Java libraries from Kotlin code, and it provides many

[1] https://github.com/typetools/checker-framework/issues/3267

features that make the code more concise and safe, in comparison with Java, increasing productivity and safety [50].

One of those features is the fact that the *null* value is not assignable to variables unless they are marked as nullable and uses of nullable values are checked to make sure they are not *null*. This limits the existence of *NullPointerExceptions*, which may still happen when interacting with Java code that was not annotated, but the error surface is greatly reduced.

Listing 4.1: Null checking in Kotlin

```
var output: String
output = null // Compilation error

val name: String? = null // Nullable type
println(name.length()) // Compilation error

if (name != null) {
    println(name.length()) // OK
}
```

Another feature is called "smart casts". If one checks the type of an object in an *if* statement, Kotlin is able to automatically cast the type of that value inside the *if* body. Notice how in Java, one would need to use the *instanceof* keyword and cast the value inside the *if*. In Kotlin, the cast is automatic and instead of *instanceof*, one just needs to write *is*. The code gets more concise and readable.

Listing 4.2: Smart casting in Kotlin

```
fun calculateTotal(obj: Any) {
    if (obj is Invoice)
        obj.calculateTotal()
}
```

Additionally, Kotlin supports lambdas, functions declarations, which unlike in Java, do not need to be declared inside a class, and extends data structures with many utility functions, like *map* and *filter*. This allows code to be written in a more functional style and reduces the amount of code that needs to be written to process data in collections.

Listing 4.3: Lambdas and collection processing in Kotlin

```
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3)
```

Many other features could be mentioned but these were the ones that provided a greater increase in productivity in the development of this tool.

Kotlin is used, for example, in Android development [83] and is supported by the Spring Framework [44]. Kotlin is therefore stable, and we believe it creates a good foundation for the future of this tool as well.

4.2 Type-checker features

In this section, we will be discussing how programmers can use this tool and which features are available for use. We start by explaining how protocols may be defined and associated with each class, then we present useful annotations that may be used, we discuss the fact that our tool prevents null pointer errors, it ensures that protocols reach completion, it supports "droppable" states, and that protocol specifications may be associated with classes from libraries which source code is not available.

4.2.1 Protocols

This tool requires that protocol specifications be associated with each Java class if one desires to check the behavior of instances of those classes. Protocol specifications are written in text files usually with the *.protocol* extension. Since these protocols resemble finite-state machines, they declare the states and the available transitions between those. Each specification must follow the grammar in listing 4.4. Note that *id* is a meta-variable ranging over values of the set of all the valid Java identifiers.

Listing 4.4: The grammar of protocol specifications

```
1  Select := id | id "." Select
2
3  Package := "package" Select ";"
4
5  Import := "import" "static"? Select ("." "*")? ";"
6
7  Destination := Select | State | DecisionState
8
9  Decision := id ":" ( id | State )
10
11 DecisionState := "<" Decision ( "," Decision )* ">"
12
13 Arguments := "(" Select ( "," Select )* ")"
14
15 Method := Select id Arguments ":" Destination
16
17 Methods := Method ( "," Method )*
18
19 State := "{" Methods ( "," "drop" ":" "end" )? "}"
20
21 NamedState := id "=" State
22
23 Protocol := "typestate" id "{" NamedState* "}"
24
25 Start := Package? Import* Protocol
```

Protocols may start with a *package* statement and zero or more *import* statements. Just like in Java classes, these statements indicate to which package the protocol belongs and

which classes it wants to import. This is important for the resolution of Java types used in the protocol.

Following that, the keyword *typestate* must be used, followed by the name of the protocol. The name is used only for presentation purposes when reporting errors and does not need to match the file name. After the name, between curly brackets, zero or more named states may be declared. No state may be called *end*. The *end* state is the final state which is always implicit in the protocols and allows for no method calls. The initial state in the protocol is the first declared. If no state is declared, *end* is simultaneously the first and final state.

For each state, a set of methods which are allowed on that state may be declared. Each method in the protocol is composed by a return type, a list between parentheses of the types of the parameters separated by commas, and the state to which the method transits to in that given state. The destination state may be a name of another declared state, an anonymous state (i.e. a state declaration starting and ending with curly brackets), or a decision state. Each decision state is composed of pairs. For each pair, the first component is the name of a value from an enumeration or a boolean literal, depending on if the method returns an enumeration or a boolean value. The second component indicates to which state the method transits to depending on the return value of the method call, which may be the name of a state or an anonymous state.

Additionally, each state may optionally include a "droppable" transition. This is a special transition that happens implicitly when an object is no longer used, which moves the object's state to the final state. The lack of return type and parentheses avoids ambiguity with methods, allowing the parser to easily understand that *drop: end* is a special transition. This feature will be further explained later.

Listing 4.5: Example of an iterator protocol

```
1 typestate Iterator {
2   HasNext = {
3     boolean hasNext(): <true: Next, false: end>
4   }
5   Next = {
6     boolean hasNext(): <true: Next, false: end>,
7     Object next(): HasNext
8   }
9 }
```

In this example is presented a protocol for a Java iterator. This protocol has two declared states, *HasNext* and *Next* and the implicit *end* state. In the *HasNext* state, only the *hasNext* method is available to be called (line 3). In the *Next* state, both *hasNext* and *next* methods are available (lines 6 and 7). When the *hasNext* method is called, and if it returns *true*, the iterator transits to the *Next* state. If the method returns *false*, the iterator transits to the *end* state. When the *next* is called, the iterator transits to the *HasNext* state.

To associate a protocol with a Java class, the *Typestate* annotation should be placed

on top of the class declaration. The annotation accepts one string argument which is the path, relative to the class file, to the protocol file. If the extension is not mentioned and the file is not found, a file with the same name but ending with the *.protocol* extension is searched. If a protocol file is still not found, an error is reported.

Listing 4.6: *Typestate* annotation

```
@Typestate("Iterator.protocol")
public class Iterator {
  ...
}
```

With the protocol associated with the class, the type-checker will ensure that instances of that class follow the respective protocol.

Listing 4.7: Correct iterator example

```
1 Iterator it = new Iterator();
2 while (it.hasNext()) {
3   it.next();
4 }
```

This example would be accepted by the type-checker. When the iterator is instantiated (line 1), it is in the *HasNext* state, the first one declared in the protocol. The *hasNext* method is allowed because it is available on that state (line 2). If the method returns *true*, the flow of execution goes into the loop body, where the *next* method is called (line 3). That call is allowed because now the iterator is in the *Next* state. After the call, the iterator returns to the *HasNext* state. When *hasNext* returns *false*, the loop finishes, and the iterator is left in the *end* state, and no other method is called on the iterator.

Listing 4.8: Incorrect iterator example

```
1 Iterator it = new Iterator();
2 do {
3   it.next();
4 } while (it.hasNext())
```

This example would not be accepted by the type-checker. When the iterator is instantiated, it is in the *HasNext* state (line 1). Immediately after that, the *next* method is called before any *hasNext* call is performed (line 3). This goes against what is defined in the protocol because the *next* method is not available in the *HasNext* state.

4.2.2 State refinement

Besides the *Typestate* annotation, there are some other useful annotations that may be used. One example is the *Requires* annotation. This annotation may be used in the parameters of method declarations to indicate in which states the object pointed by the parameter is expected to be in. If this annotation is not used, the type-checker assumes

that the object may be in any state. The annotation expects a string argument or an array of strings with the names of the required states.

Listing 4.9: *Require* annotation

```
1 void readFile(@Requires("Open") File file) {
2   file.read();
3   file.close();
4 }
```

Imagine an example where there is a file object that must be opened first, then it can be read, and then it must be closed, and suppose there is a *readFile* method which expects the file to be in the *Open* state so that the method can immediately read it and close it. To allow the code to be type-checked, one can use the *Requires* annotation before the type of the parameter with a string argument with the name of the *Open* state.

Another useful annotation that the tool offers is the *Ensures* annotation. This annotation may be used in the parameters of method declarations to indicate in which states the object pointed by the parameter is left in after the method call. The annotation expects a string argument or an array of strings with the names of the states.

Listing 4.10: *Ensures* annotation

```
1 void readFile(
2   @Requires("Open") @Ensures("Read") File file
3 ) {
4   file.read();
5 }
```

Now suppose that the *readFile* method expects a file which is opened, reads it and then leaves the file in a state in which it can be closed, leaving the responsibility of closing the file to the caller. To allow the code to be type-checked, one can use the *Ensures* annotation before the type of the parameter with a string argument with the name of the *Close* state. Notice how different annotations can be combined to specify what the method does.

Finally, the tool provides the *State* annotation. This annotation may be used in the return types of method declarations. It is similar to the *Ensures* annotation in that it indicates the states in which an object is in after the method call but instead of referring to an object passed in a parameter, it refers to the object returned by the method. If this annotation is not used, the type-checker assumes that the object may be in any state. The annotation expects a string argument or an array of strings with the names of states.

Listing 4.11: *State* annotation

```
1 @State("Open") File newFile() {
2   File file = new File();
3   file.open();
4   return file;
5 }
```

Imagine there is a *newFile* method which returns a new file already opened, ready to

be read. To allow the code to be type-checked, one can use the *State* annotation before the return type of the method with a string argument with the name of the *Open* state.

4.2.3 Nullness checking

Null pointer errors are very common in Java. Because of this, it is crucial for the type-checker to be able to detect these errors before runtime. For every object type used in a program, the type-checker assumes the type is not nullable (contrary to the default type system of Java). If the programmer desires that a type be nullable, it can use the *Nullable* annotation. The annotation may be used before any type, in variable declarations, in parameter declarations and in return types of methods. The type-checker then ensures that no operations that could raise a null pointer error are performed. Additionally, the type-checker is able to refine the type of nullable values when a comparison with *null* is made in *if* statements or in the conditions of loops.

Listing 4.12: Nullness checking example

```
1  @Nullable @State("Open") File tryOpening() {
2    File file = new File();
3    return file.open() ? file : null;
4  }
5
6  void main1() {
7    @Nullable file = tryOpening();
8    file.read(); // Error
9    file.close();
10 }
11
12 void main2() {
13   @Nullable File file = tryOpening();
14   if (file != null) {
15     file.read();
16     file.close();
17   }
18 }
```

In this example there is a *tryOpening* method which creates a new file, attempts to open the file, and if it succeeds, it returns the newly created file otherwise, it returns *null* (line 3). The type-checker would report an error in the *main1* method because there is an attempt to call the *read* method in a potentially *null* value (line 8). In the *main2* method, no error would be reported since before calling methods on the file, it is checked that the *file* variable is not *null* (line 14).

4.2.4 Protocol completion

When working with objects whose use is expected to follow a protocol, it is important not only to ensure that method calls are performed in valid sequences, but that objects

are used to completion (i.e. until they reach the final state). This ensures that the implementation is correct by preventing method calls from being forgotten and by freeing resources that are no longer necessary. For example, this can be used to ensure that the *close* method on a socket is called after the socket is no longer in use.

Listing 4.13: Socket protocol example

```
1 typestate Socket {
2   NotConnected = {
3     void connect(): Connected
4   }
5   Connected = {
6     void send(String): Connected,
7     void close(): end
8   }
9 }
```

Imagine for example a protocol for a socket which may be in the *NotConnected*, *Connected* or *end* states. In the initial state, the socket is not connected and only the *connect* method is allowed (line 3). When the socket is connected, messages can be sent via the *send* method (line 6). Finally, the socket can be closed via the *close* method (line 7). Closing the socket makes it reach the *end* state, where no other operations are allowed.

Listing 4.14: Socket usage: example 1

```
1 void main1() {
2   Socket s = new Socket();
3   s.connect();
4   s.send("Hello World!");
5   // Error
6 }
```

In this first example, the type-checker would report an error since the socket is not used until its protocol is completed. Notice how the socket was created, connected, a message was sent, but the socket was not closed.

Listing 4.15: Socket usage: example 2

```
1 void main2() {
2   Socket s = new Socket();
3   s.connect();
4   s.send("Hello World!");
5   s.close();
6 }
```

In this second example, the type-checker would report no errors, since the use of the socket follows the protocol and the socket is properly closed when it is no longer necessary.

Listing 4.16: Socket usage: example 3

```
void main3() {
  Socket s = new Socket();
  s.connect();
  s.send("Hello World!");
  s.close();
  s.send("Hello World!"); // Error
}
```

In this third example, the type-checker would report an error. Even though the socket was closed, there was an attempt to send another message after the fact. That is not allowed since when an object is in the *end* state, no method calls are allowed.

4.2.5 Droppable states

It makes sense to force a protocol to reach the final state, for example, if we want to make sure that some resources are freed or to make sure a protocol finishes. On the other hand, there are cases where that is not strictly necessary. For example, there are scenarios where one does not need an iterator to iterate for all elements of a collection, but just a few.

To support this kind of scenario, protocol specifications support a special kind of transition. This transition represents the "dropping" of an object and transits it to the *end* state. It happens implicitly when an object is no longer used. This is similar to how the *drop* method is automatically called in Rust when an object goes out of scope[2].

One key feature of this special transition is that it does not need to be defined in all states. This allows the programmer to indicate in which states the object may be "dropped". States that do not include this transition are states where the object cannot be "dropped". This notion of "droppability" was also proposed in [23].

Listing 4.17: Example of an iterator protocol with "droppable" transitions

```
typestate Iterator {
  HasNext = {
    boolean hasNext(): <true: Next, false: end>,
    drop: end
  }
  Next = {
    boolean hasNext(): <true: Next, false: end>,
    Object next(): HasNext,
    drop: end
  }
}
```

In this example is defined a protocol for an iterator which may be "dropped" in any state. That is specified by the *drop: end* transition defined in the *HasNext* and *Next* states.

[2] https://doc.rust-lang.org/book/ch15-03-drop.html

This means that one may stop using the iterator if it is either in the *HasNext* state or in the *Next* state.

4.2.6 Protocols for classes of libraries

For the type-checker to ensure that method calls are performed in correct order, each class needs to be associated with a protocol. That is easily done for classes which the programmer owns: by adding the *Typestate* annotation. But not very class used in a project is owned by the programmer of that project. It is very common to use third-party libraries and specially the standard library of Java. Since classes belonging to those libraries might not be associated with protocol specifications, the tool allows the programmer to associate protocol files with classes or interfaces in a configuration file.

The content of this configuration file follows the same syntax of *.properties* files, commonly used in the Java community. Each line in the file is a mapping from a *key* to a *value*. In this instance, the *key* is the full qualified name of a class or interface and the *value* is the path (relative to the configuration file) of the protocol file. An example is presented in listing 4.18, where the standard *Iterator* class of Java is associated with a protocol.

Listing 4.18: Example of a configuration file

```
1 java.util.Iterator=Iterator.protocol
```

To allow for common code in Java to be type-checked with our tool, it is not enough to associate other classes or interfaces with protocols. For example, the type-checker must know that when the *iterator* method is called on a list, a new iterator in the *HasNext* state is returned. To account for this situation, we make use of a feature that is already available through the Checker Framework: *stub files*[3]. A *stub file* is Java source code that omits method bodies and allows one to write annotations for a library when the code is not available to be edited.

Listing 4.19: Example of a *stub file*

```
1 package java.util;
2
3 public interface List<E> {
4   @State("HasNext") Iterator<E> iterator();
5 }
```

In this example is shown the content of a *stub file* which provides an annotation for the *iterator* method in the *List* interface of the standard Java library. With the use of the *State* annotation, we inform the type-checker that the returned iterator is in the *HasNext* state, allowing the code presented in listing 4.20 to be type-checked[4].

[3] https://checkerframework.org/manual/#stub

[4] With the exception that generics are not yet fully supported.

Listing 4.20: Example of the use of the standard iterator

```
1  import java.util.*;
2
3  public class Main {
4    public static void main(String[] args) {
5      Iterator<String> it = Arrays.asList(args).iterator();
6      while (it.hasNext()) {
7        it.next();
8      }
9    }
10 }
```

4.3 Type-checker implementation

The type-checker is implemented as a plugin for the Checker Framework [79]. It works by visiting each class and analyzing each method in two phases. In the first phase, the content of methods is analyzed to infer the types of variables and fields. In the second phase, using the inferred types, errors are reported when type incompatibility exists or when invalid operations are performed. Additionally, it is ensured that the protocol of objects is completed and that objects are used in a linear way.

Forcing the linear use of objects associated with a protocol is important to ensure that there was no other piece of code that could have changed the state of an object, breaking the assumptions of the static analysis, and compromising the checking of protocol compliance. There are techniques that can be employed to relax this restriction without compromising the checking process. These will be used in the second version of the tool, which will be presented later. For the first version, linearity is enforced.

In the following sections, we will discuss the type system employed and implementation details.

4.3.1 Type system

Each variable or field declaration and each expression in the code is associated with a Java type, which is statically known. To be able to track the state of each object, we need a parallel type system with other types that will represent the information the type-checker needs. To that end, we introduce a type system where every value has a type from the lattice in figure 4.1.

Unknown is the top type. It includes all possible values. *Primitive* is the type of all primitive values, like integers and booleans. *Object* contains all objects, not including the *null* value. This detail is important because that allows us to ensure statically that null pointer errors do not occur. *Null* is a type that only includes the *null* value.

Moved is a type applied to variables that point to an object that was passed as a parameter to a method call or delegated to another variable. Variables with the *Moved*

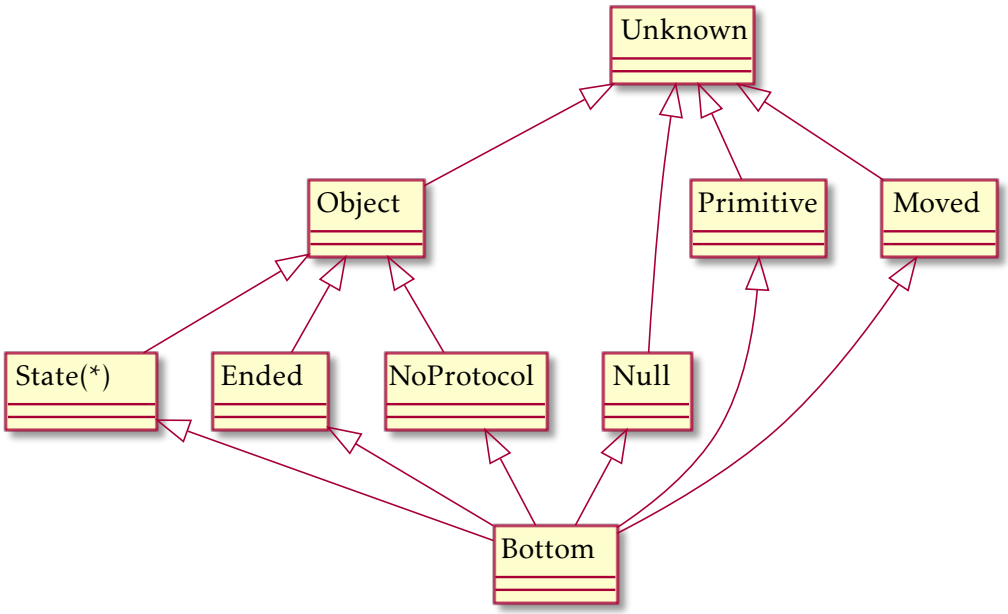

Figure 4.1: Type System Lattice

type cannot be used, because they no longer own the data. This ensures that objects are used linearly.

This draws inspiration from the ownership concept of the Rust language: if something takes ownership of some data, such data is considered to have been "moved", and the previous reference cannot be used [88]. This also avoids the need to nullify variables after a value is obtained, as also proposed in [9].

The *NoProtocol*, *Ended* and *State* types are subtypes of *Object*. *NoProtocol* applies to all objects that do not have a protocol specification associated with it. The *Ended* type is applied to objects which protocol has completed. The *State* type represents objects which are in a specific state. In practice, each *State* type is distinguished by the name of the state it refers to and the protocol that declared that state.

Finally, the *Bottom* type is the subtype of all types. All operations are allowed on it. Conceptually, it is an empty set, which means no value introduced in the language has the *Bottom* type. We attribute this type to variables in contexts where the code that uses them is never reached, to computations that might generate an error or just as a way to avoid propagating errors. For example, imagine we call a method on an object which is in a state that does not allow for that, the type of the object after that invocation will be the *Bottom* type.

Every type is conceptually a set of all the values that belong to that type. In practice, a type may be one of the singleton types presented in figure 4.1 or an union type comprising of a set of those types. For example, union types are useful so that we can specify that an object is in a state from a set of states or that a variable may point to an object in a certain state or be *null*.

In the implementation, the representation of each type follows these properties:

- union types do not include unions in their structure;
- a set with the *Unknown* type is just the *Unknown* type;
- a set with only one type is just that type;
- an empty set is just the *Bottom* type;
- the *Bottom* type does not appear in union types since it represents the empty set;
- if the *Object* type is present in an union type, *NoProtocol*, *Ended* and *State* are not since they are already subtypes of *Object*.

In the following listings, we define the semantics concerning the creation of the types presented and their subtyping relationship in ML. Additionally, we define two functions to compute the union of two types and the intersection of two types. Assume that the ML lists do not include repeated values.

Listing 4.21: Types definition

```
1  type ttype =
2    | Unknown | Object
3    | State of string | Ended | NoProtocol
4    | Null | Primitive | Moved
5    | Bottom | Union of ttype list
```

The type definition includes all the singleton types in the type system and an union type with a list of types.

Listing 4.22: *createType* function

```
1  let createType (types:ttype list) : ttype =
2    let types = flatTypes types in
3    let types = if contains Object types
4      then filter_not inObject types
5      else types
6    in match types with
7    | [] -> Bottom
8    | [t] -> t
9    | types -> if contains Unknown types
10     then Unknown
11     else Union types
```

The *createType* functions takes a list of types. If that list contains union types, they are removed and the types in the union are directly included in the list (using the *flatTypes* function in line 2). If the list contains the *Object* type, subtypes of it are removed (line 4). If the list is then empty, the *Bottom* type is returned (line 7). If the list contains only one type, such type is returned (line 8). If the list contains the *Unknown* type, *Unknown* is

returned (line 10). Otherwise, an union type is returned containing the types in the list (line 11).

Listing 4.23: *isSubtype* function

```
1  let rec isSubtype (a:ttype) (b:ttype) : bool =
2    match a with
3    | Unknown -> b = Unknown
4    | Object ->
5      begin match b with
6      | Unknown | Object -> true
7      | Union bTypes -> contains Object bTypes
8      | _ -> false
9      end
10   | State _ | Ended | NoProtocol ->
11     begin match b with
12     | Unknown | Object -> true
13     | Union bTypes ->
14       (contains a bTypes) || (contains Object bTypes)
15     | _ -> a = b
16     end
17   | Moved | Null | Primitive ->
18     begin match b with
19     | Unknown -> true
20     | Union bTypes -> contains a bTypes
21     | _ -> a = b
22     end
23   | Bottom -> true
24   | Union aTypes -> for_all (fun a -> isSubtype a b) aTypes
```

The *isSubtype* function returns *true* if and only if the first type is a subtype of the second according to the lattice provided in figure 4.1. *Unknown* is only a subtype of *Unknown* (line 3). *Object* is subtype of *Unknown*, *Object* and an union that includes *Object* (lines 4 to 9). Remember that the representation of union types does not include *Unknown*, so it is enough to check that *Object* is in the union. A type representing a state is subtype of *Unknown*, *Object*, itself and any union containing itself or *Object* (lines 10 to 16). The same goes for *Ended* and *Protocol*. *Moved*, *Null* and *Primitive* are subtypes of *Unknown*, themselves and any union containing one of them (lines 17 to 22). *Bottom* is a subtype of any type (line 23). An union type is a subtype of another type if the types in the union are all subtypes of the other type (line 24).

Listing 4.24: *union* function

```
1  let union (a:ttype) (b:ttype) : ttype = createType [a; b]
```

The *union* function accepts two types and returns the type that corresponds to the union of the two. The implementation of this function is very simple because it simply reuses the *createType* presented previously.

Listing 4.25: *intersection* function

```
1  let rec intersection (a:ttype) (b:ttype) : ttype =
2    match a with
3    | Unknown -> b
4    | Object ->
5    begin match b with
6      | Unknown -> a
7      | Object -> a
8      | Union bTypes -> createType (
9          map (fun b -> intersection a b) bTypes)
10     | _ -> if isSubtype b Object then b else Bottom
11   end
12   | State _ | Ended | NoProtocol
13   | Moved | Null | Primitive -> if isSubtype a b then a else Bottom
14   | Bottom -> a
15   | Union aTypes -> createType (
16       map (fun a -> intersection a b) aTypes)
```

The *intersection* function accepts two types and returns the type that corresponds to the intersection of the two. The intersection of any type with *Unknown* is that type (line 3). The intersection of *Unknown* or *Object* with *Object* is *Object* (lines 6 and 7). The intersection of *Object* with an union type is computed by intersecting the types of the union with *Object* (lines 8 and 9). The intersection of a singleton type which is subtype of *Object* with *Object* is that type (line 10). The intersection of any other type with *Object* is *Bottom* (line 10). The intersection of a singleton type with another type is that singleton type if it is subtype of the other otherwise, it is *Bottom* (line 13). An intersection with *Bottom* is always *Bottom* (line 14). The intersection of an union type with another type is the result of intersecting the types of the union with the other type (lines 15 and 16).

4.3.2 Architecture

In the first phase of the type-checking process, the types of variables and fields are inferred. To do that, each class is visited independently. If the class is associated with a protocol (i.e. has a *Typestate* annotation), the non-static methods of that class are analyzed in a order that follows the protocol, a process that we are going to call *class analysis*. If the class is not associated with a protocol, then we assume a trivial one for it. That trivial protocol has only one state. In that state, the object may be "dropped", all its methods are available to be called, and any method calls leave the object in the same state.

When analyzing each method of a class, each expression in the method is analyzed independently. For each expression, a *transfer* function is called. This function accepts a pair of *stores* and returns another pair of *stores*. Each *store* is a mapping between variables or fields and their respective types. The first element of the pair is called the *then store*, and the second is called the *else store*. The *then store* refers to the information that is true when a given expression evaluates to *true*. The *else store* refers to the information that

is true when a given expression evaluates to *false*. If the expression does not evaluate to a boolean, then both stores should be the same. The input that the *transfer* function receives is the result of analyzing the previous expressions. The return value of the *transfer* function is the result of analyzing that expression.

The existence of these two stores is important, for example, when dealing with an *if* statement. The *then store* will correspond to the information that is true in the *then* branch, while the *else store* will correspond to the information that is true in the *else* branch.

Listing 4.26: Two stores example

```
1 Iterator it = new Iterator();
2 if (it.hasNext()) {
3   // it: Next
4   it.next();
5 } else {
6   // it: end
7 }
```

Consider for example the *hasNext* method call on an iterator object in the condition of an *if* statement. When the call returns *true*, then the iterator is in the *Next* state otherwise, the iterator is in the *end* state. With the distinction between the two stores, we can distinguish between what is true in one branch and what is true in the other branch.

For the first expression in the method, the input to the *transfer* function corresponds to the *entry store* of the method. For the last expressions in the method, the results of their *transfer* functions are merged to produce the *exit store* of the method. This is done because the method might have multiple *return* statements. These *entry* and *exit* stores are important to track what is true at the beginning of the method call and what is true after the method call. We can think of these as the pre-condition and the post-condition of the method, respectively.

During the analysis, there might be the need to merge stores. The result of merging two stores is a store that includes all the variables or fields in the two stores, and when the same variable or field is present in both, the union of the corresponding types is computed.

In the second phase of the type-checking process, checks are performed and potential errors are reported. Each statement and expression in the code is visited to ensure that types are compatible. For example, the type of an expression to be passed to a method call needs to be a subtype of the type of the parameter. Additionally, we ensure that operations are only performed when it is safe to do so. For example, operations on *null* that could raise a null pointer error are reported and method calls are only accepted if the object is in a state that allows for that call. Finally, protocol completion is ensured by, in general, checking the end of methods.

4.3.3 Class analysis

To allow for the precise inference of the types of fields of objects, we analyze each method of a class in an order that follows the protocol. Since we only consider as input, for the analysis of a method, the results obtained from analyzing methods that lead to that method, we avoid false positives. In other words, we exclude invalid sequences of method calls from our consideration. What follows is an explanation of the algorithm with line references pointing to the relevant lines of pseudocode in listing 4.27.

Listing 4.27: Class analysis in pseudocode

```
1  let stateToStore = Map<State, Store>()
2  let methodToStore = Map<Method, Store>()
3  let stateQueue = Queue<State>()
4
5  fun mergeStateStore(state, store) {
6    let current = stateToStore[state]
7    let new = merge(current, store)
8    if (current != new) {
9      stateToStore[state] = new
10     stateQueue.add(state)
11   }
12 }
13
14 fun mergeMethodStore(method, store) {
15   let current = methodToStore[method]
16   let new = merge(current, store)
17   if (current != new) {
18     methodToStore[method] = new
19     return new
20   }
21   return null
22 }
23
24 mergeStateStore(initialState, initialStore)
25
26 while (stateQueue.size > 0) {
27   let state = stateQueue.take()
28   let store = stateToStore[state]
29   for ((method, destination) in state.transitions) {
30     let entryStore = mergeMethodStore(method, store)
31     if (entryStore == null) continue
32
33     let result = analyze(entryStore, method)
34
35     mergeStateStore(destination, result)
36   }
37 }
```

The algorithm in the class analysis starts by analyzing the constructors and inferring

what are the types of the fields after the initialization. After analyzing the constructors, we merge the resulting stores, save the merged store and associate it with the initial state (line 24). This store contains the facts that are true in that state. The initial state is now added to a queue (line 10).

For each state in the queue we analyze each method available in that state. The method analysis needs an initial store to start with, so that it can then output an inferred store with the facts that are true after the method call. Because of that, each method is also associated with a store (line 2), which might be updated during the algorithm. We need the mappings from state to store (line 1) and method to store (line 2) to be different because the same method might be reachable from different states with different stores.

The initial store for each method is computed by merging the store associated with the state we are processing and the previous store associated with that method (line 30). If no store was previously associated with a method, we default to the *empty store*, where all the fields have the *Bottom* type. If the initial store is not different from the previous store associated with the method, the method is skipped (line 31). Otherwise, the store associated with the method is updated with the new merged store and the method is analyzed (line 33).

After analyzing each method, the inferred stores are merged with the stores of the respective destinations states (line 35). If no store was previously associated with a state, we default to the *empty store*. If the new merged store is different from the previous store associated with the state, the state is added to the queue (line 10).

The same states or the same methods might be analyzed more than once, but there is a fixed point. The algorithm stops when the queue is empty (line 26). We know that the queue will become empty because states are not requeued if their respective stores did not change and because the number of states in protocols is finite.

4.3.4 Inference and checking

In the following sections, we will discuss how types are introduced, how types are inferred when analyzing each type of expression or statement, what properties need to be checked for each expression or statement, and how that is done.

4.3.4.1 Type introduction

Each Java type appearing in declarations has a corresponding type from our type system. In general, the Java type *java.lang.Object* corresponds to the *Object* type in our type system. Any primitive type, like an integer or a boolean, corresponds to the *Primitive* type. The *null* value has the *Null* type. Any Java type associated with a class that has no protocol, corresponds to the *NoProtocol*. For Java types of objects with protocol, the associated type depends on the location of the declaration.

For return types of methods that can be analyzed (i.e. which source code is available), the type is the union of all the states in the protocol expect *end* or the union of the states

specified in a *State* annotation. For return types of methods of other libraries, the type is *Unknown* unless a *State* annotation is provided for that method in a *stub file*. This is because we cannot be sure of the state of an object and if there are aliases to it, if it came from a method that we cannot analyze.

For parameters, the initial type is the union of all the states in the protocol except *end* or the union of the states specified in a *Requires* annotation. The *end* state is not considered because there is no reason to return or pass to a method objects that can no longer be used.

For public field declarations, the type corresponds to *Unknown*. Since public fields may be modified without calling any of the methods of the class, it is difficult to track in what ways those fields may be altered. By applying the *Unknown* type, we preserve correctness, since no operations can be performed on *Unknown*, and we help the programmer to use good practices such as not exposing the fields of an object.

For local variable declarations or private field declarations, the initial type is the union of all the states in the protocol. Since the type may change during the execution of the code, such changes are tracked during the inference process.

For any of the cases previously mentioned, if the *Nullable* annotation is used before any Java type, then the corresponding type will initially include the *Null* type, which can be potentially refined during the inference process.

4.3.4.2 Method parameters

When analyzing a method, an initial store is needed as input to the analysis of the first expression in the method. The initial store for a non-static method is the result of merging the type information provided by the *class analysis*, which includes information about the fields of the object, with the type information of the parameters just introduced. The initial store for a static method is only composed by the type information of the parameters. The initial type of the parameters is computed using the reasoning explained in section 4.3.4.1.

Listing 4.28: Types of parameters (1)

```
1 void useIterator(Iterator it) {
2   // it: State "HasNext" | State "Next"
3   ...
4 }
```

In this example, the initial store of the method only contains information about the *it* variable which has initially the *State "HasNext"*| *State "Next"* type, which is the union of all the possible states of the iterator except *end*.

Listing 4.29: Types of parameters (2)

```
1 void useIterator(@Requires("HasNext") Iterator it) {
2   // it: State "HasNext"
3   ...
4 }
```

In this example, the initial store of the method only contains information about the *it* variable which has initially the *State "HasNext"* type. Unlike the previous example, since the *Requires* annotation was used, the set of states considered was refined, allowing the method to assume that it receives an iterator only in the *HasNext* state.

4.3.4.3 Object instantiations

When a new object without protocol is initialized, its type is *NoProtocol*. When a new object with protocol is initialized, its type corresponds to the first state in its protocol.

Listing 4.30: Object instantiation example

```
1 Iterator iterator = new Iterator();
2 // iterator: State "HasNext"
```

4.3.4.4 Assignments

When an assignment statement is analyzed, the type of the variable or field becomes the type of the assigned expression, and the type of the expression becomes *Moved* after the statement (when dealing with object types). By changing the type of the assigned expression to *Moved*, we transfer the ownership of that object to the assignee, and since no operations are allowed on the *Moved* type, we ensure that each object is used linearity.

Listing 4.31: Assignments example (1)

```
1 Iterator it1 = new Iterator();
2 // it1: State "HasNext"
3 Iterator it2 = it1;
4 // it1: Moved
5 // it2: State "HasNext"
```

In this example, a new iterator is created and assigned to the variable *it1*, which now has the *State "HasNext"* type. When *it1* is later assigned to *it2*, the *it1* variable gets the *Moved* type and *it2* gets the type *State "HasNext"*, which corresponds to the type of *it1* before the assignment. This means that now *it2* has ownership of the iterator while *it1* lost ownership of it. If one wants to use the iterator, it must now do it via the *it2* variable.

In the checking phase, assignments are analyzed to ensure that the type of the assigned expression is a subtype of the expected type of the assignee. This expected type corresponds to what was explained in section 4.3.4.1. For example, if we try to assign *null* to a variable or field, an error will be reported unless the *Nullable* annotation is provided.

Listing 4.32: Assignments example (2)

```
1 Iterator it1 = null; // Error
2
3 @Nullable Iterator it2 = null; // Ok
```

Note that assignments on parameters with the *Ensures* annotation cannot be performed. This is important so that we can ensure that when the ownership of the object is returned to the caller, the object is in the specified state.

4.3.4.5 Return statements

When a return statement is analyzed, the type of the returned expression becomes *Moved* after the statement (when dealing with object types). Although return statements immediately return control to the caller, it is important to track this "move" of data, for example, when returning an object referenced from a field. By updating the type information of this field, other methods reachable by that method will have information about the fact that the data on that field is no longer owned by the object.

After the inference phase, the checking phase ensures that the returned expression is subtype of the expected type to be returned. The expected return type is computed as explained in section 4.3.4.1.

Listing 4.33: Return example (1)

```
1 class Wrapper {
2   private Iterator iterator = new Iterator();
3   public @State("HasNext") Iterator get() {
4     return this.iterator;
5     // this.iterator: Moved
6   }
7 }
```

In this example, there is a wrapper object that stores an iterator object that can be retrieved via the *get* method, which is expected to return an iterator in the *HasNext* state. When analyzing the return statement, the type of *this.iterator* becomes *Moved* after that statement. This information will be provided to the analysis of methods that are reachable after the *get* call, ensuring that it is known that the ownership of the iterator was lost.

Listing 4.34: Return example (2)

```
1 class Wrapper {
2   private Iterator iterator = new Iterator();
3   public @State("HasNext") Iterator get() {
4     // this.iterator: State "HasNext" | Moved
5     return this.iterator; // Error!
6   }
7 }
```

If the protocol of the wrapper were to allow for multiple *get* calls, then when *get* is called, *this.iterator* might be in the *HasNext* state or it might have been "moved" by a previous *get* call. And since *State "HasNext"| Moved* is not a subtype of *State "HasNext"*, an error would be reported in the checking phase.

4.3.4.6 Method calls

To be able to call a method, the receiver (i.e. the object on which the call is performed) needs to be in a state that allows for that method call, and the types of the argument expressions need to be subtypes of the types of the corresponding parameters. These type compatibility checks are performed in the checking phase, while the types are computed in the inference phase.

When analyzing a method call on an object, the current type of the receiver is retrieved from the stores given as input to the analysis of this call. Knowing the current type and the method call, the inference phase computes the type of the object after that method call and saves that information in the stores that result from this analysis step.

The following listings present the ML representation of a protocol, the ML implementation of a function that given the current type, the name of the method, and the protocol, returns *true* if the method may be safely called on that type, and the ML implementation of a function that returns the new type of the object after the method call.

Listing 4.35: Protocol representation

```
type
  label = string and
  destination =
    ExternalState of string |
    DecisionState of (label * string) list and
  method_name = string and
  transition = method_name * destination and
  typestate = string * (transition list) and
  protocol = typestate list
```

The protocol is represented with a list of states. Each state is a pair with its name and a list of transitions. Each transition is a pair with the name of the method and the destination state, which may be the name of another state or a decision state. Decision states are represented with a list of pairs, where each pair contains the label and the corresponding name of the destination state given that label.

Listing 4.36: *available* function

```
let rec available (t:ttype) (m:string) (p:protocol) : bool =
  match t with
  | Unknown | Object
  | Ended | Moved | Null | Primitive -> false
  | NoProtocol | Bottom -> true
  | State state ->
    let (_,transitions) =
      find (fun ((name,_)) -> state = name) p in
      exists (fun ((method,_)) -> method = m) transitions
  | Union types -> for_all (fun it -> available it m p) types
```

The *available* function returns *false* for the *Unknown* and *Object* types since there is no guarantee that methods may be safely called on objects with these types. The function also returns *false* for the *Ended*, *Moved*, *Null* and *Primitive* types, since method calls are not allowed on these. For objects with the *NoProtocol* type, any method call is allowed. If an object is in a given state, the method call is allowed if it is available in that state according to the protocol. Any method call is allowed in the *Bottom* type. Finally, for union types, the method is available if it is available for all the types in the union.

Listing 4.37: *transition* function

```
let rec transition (t:ttype) (m:string) (p:protocol) : ttype =
  match t with
  | Unknown | Object | NoProtocol | Bottom -> t
  | Ended | Moved | Null | Primitive -> Bottom
  | State state ->
    let (_,transitions) = find (fun ((name,_)) -> state = name) p in
    let transition = find_opt (fun ((method_name,_)) -> method_name = m) transitions in
    begin match transition with
    | None -> Bottom
    | Some(_,ExternalState name) ->
      if name = "end" then Ended else State name
    | Some(_,DecisionState list) ->
      createType (map (fun ((_, name)) -> if name = "end" then Ended else State name) list
          ↪ )
    end
  | Union types -> createType (map (fun it -> transition it m p) types)
```

If an object has the *Unknown* or the *Object* types, the type after the method call remains the same, since it is unknown what the new type may be after the method call. If an object has the *Ended*, *Moved*, *Null* or *Primitive* types, the new type is *Bottom*, since method calls are not allowed on these types and any attempt to call a method on these would produce an error. Giving the object the *Bottom* type avoids the propagation of errors. Objects with the *NoProtocol* type remain with the same type. If an object is in a given state, the new type corresponds to the destination state, given that method call. If the method is not available on that state, the new type is *Bottom*. Objects with the *Bottom* type remain with

the same type. Finally, for union types, the new type is the union of all the types that result from taking each type and applying the *transition* function to them.

Listing 4.38: Method calls example (1)

```
1  void main() {
2    File file = new File();
3    // file: State "Init"
4    file.open();
5    // file: State "Open"
6    file.read();
7    // file: State "Read"
8    file.close();
9    // file: Ended
10 }
```

In this example, there is a file with a simple protocol: in the first state it must be opened, then it must be read, and then it must be closed. After the file is closed, no operations may be performed on it. In the *main* method, the object file is created, and then all the methods are called in the correct order so, this example type-checks.

Listing 4.39: Method calls example (2)

```
1  void main() {
2    File file = new File();
3    // file: State "Init"
4    file.open();
5    // file: State "Read"
6    file.close(); // Error
7    // file: Bottom
8  }
```

Now imagine that after the file object being instantiated and opened, there is an attempt to close the file before reading from it. In the inference phase, when analyzing the *close* method, the current state of the object would be *Read* and the type after the method call would be *Bottom*, because the method is not available in the *Read* state. This technique is used to avoid propagating errors. In the checking phase, an error would be reported since we cannot call the *close* method while the file is in the *Read* state.

One important aspect of analyzing method calls is making sure that objects are still used in a linear way. To that end, when analyzing a method call, the receiver and the argument expressions are all marked with the *Moved* type after the program point where they are evaluated, following the order they appear in the code (i.e. the order in which they would be evaluated). This step is important to ensure that the receiver object is not passed into the parameters (which would create an alias between *this* and one of the parameters), and to ensure that there is no aliasing between the parameters. This works because the parameter types never include the *Moved* type, which means that if one of them was previously "moved", a type incompatibility with the parameter type would be reported.

Listing 4.40: Method calls example (3)

```
1 object.compareTo(object); // Error
```

Consider for example an object that implements a *compareTo* method which allows the object to be compared with another object of the same type. In this example, there is an attempt to compare the object to itself. Immediately after the receiver expression is analyzed, it is marked with the *Moved* type, which means that when the argument expression is analyzed, it already has the *Moved* type, which will be incompatible with the type of parameter, causing an error to be reported.

Note that the receiver object having the *Moved* type is only true when the argument expressions are evaluated. Immediately after the method call, the receiver object will have the type corresponding to calling the given method in the given state.

Finally, we need to know the type of the method call expression itself and the types of the arguments after the method call is performed. The type of the method call corresponds directly to the return type. The types of the arguments are in general *Moved*, because they were delegated to a different method. The exception is when the *Ensures* annotation is used. This annotation informs us that the ownership of the object passed in the parameter is returned to the caller in a given set of states. Therefore, if this annotation is provided in a parameter, then the type of the corresponding argument expression will be the union of the states indicated in that annotation.

Listing 4.41: Method calls example (4)

```
1  void openFile(
2    @Requires("Init") @Ensures("Open") File file
3  ) {
4    file.open();
5  }
6  void main() {
7    File file = new File();
8    // file: State "Init"
9    openFile(file);
10   // file: State "Open"
11   ...
12 }
```

Consider for example an *openFile* method which expects to receive a file in the *Init* state and then returns it to the caller in the *Open* state. In this example, although *file* is temporally set with the *Moved* type, after the method call is performed, *file* does not remain with the *Moved* type and instead gets the *State "Open"* type, according to the *Ensures* annotation provided in the declaration of the method.

Note that the analysis of static method calls is performed in a similar way as non-static method calls, the only difference is that there is no receiver object to be concerned about.

4.3.4.7 Control flow statements

In Java, statements are generally executed from top to bottom, in the order that they appear in the code. However, control flow statements are used to break up the flow of execution. These include decision-making statements (*if-else*, *switch*), looping statements (*for*, *while*, *do-while*), and branching statements (*break*, *continue*, *return*) [18].

To ease the reasoning about the flow of execution of programs, each method declaration is analyzed not by visiting each node in the abstract syntax tree, but by visiting each node in a *control flow graph* [3], which is built by the Checker Framework.

Additionally, as explained before, the result of analyzing each expression in the code is composed by a *then store* and by an *else store*, with information about what are the types of variables when such expression evaluates to *true* or *false* (respectively). If the expression does not evaluate to a boolean, both stores should be the same.

Listing 4.42: *If-else* statement example

```
1 Iterator it = new Iterator();
2 // it: State "HasNext"
3 if (it.hasNext()) {
4   // it: State "Next"
5   it.next();
6 } else {
7   // it: Ended
8 }
```

Imagine for example that the *hasNext* method of an iterator is called on the condition of an *if-else* statement. After the call, the iterator may be in the *Next* state or in the *end* state, depending on if the method returned *true* or *false*, respectively. Because the result of analyzing the method call produces a *then store*, where the iterator is in the *Next* state, and an *else store*, where the iterator is in the *end* state, we can propagate the information of each store to each respective branch, allowing us to keep track of the precise state of the iterator.

Listing 4.43: *While* statement example

```
1 Iterator it = new Iterator();
2 // it: State "HasNext"
3 while (it.hasNext()) {
4   // it: State "Next"
5   it.next();
6   // it: State "HasNext"
7 }
8 // it: Ended
```

The same reasoning is applied when the method call is performed on the condition of a *while* statement. The only differences are that the information in the *then store* propagates to the body of the loop and the information in the *else store* propagates to the program point reached when the loop exits, as seen in this example.

In protocol specifications, the state to which an object transits to, when a method call is executed, may also depend on an enumeration value, not just a boolean value. If this enumeration value is checked in a *switch* statement, then we can refine the state of the object for each *case*.

Listing 4.44: *Switch* statement example

```
1  Iterator it = new Iterator();
2  switch(it.hasNext()) {
3    case TRUE:
4      // it: State "Next"
5      it.next();
6      break;
7    case FALSE:
8      // it: Ended
9      break;
10   default:
11     // it: Bottom
12 }
```

Imagine a different implementation of an iterator where the *hasNext* method returns an enumeration value, *TRUE* or *FALSE*, instead of a boolean value. When the method returns *TRUE*, the state of the iterator is *Next*, and when it returns *FALSE*, the state of the iterator is *end*. For the default case of the *switch* statement, the type of the iterator is *Bottom* since that case is not reachable.

Listing 4.45: *Switch* statement example (2)

```
1  Iterator it = new Iterator();
2  BooleanEnum hasNext = it.hasNext();
3  if (hasNext == TRUE) {
4    // it: State "Next"
5    it.next();
6  } else {
7    // it: Ended
8    if (hasNext == FALSE) {
9      // it: Ended
10   } else {
11     // it: Bottom
12   }
13 }
```

Since the result of analyzing the *hasNext* method call is only composed of two stores, the *then store* and the *else store*, and since the method call does not return a boolean value, we look at a *switch* statement as if it was composed by multiple *if* statements, where each one of the *if* statements compares the returned value with each of the enumeration values. Since a comparison evaluates to a boolean value, we can make use of the resulting stores

to refine the type information for each *case*. This is exemplified in listing 4.45.[5]

In our type system, values might be nullable and operations on *null* that would raise a null pointer error are disallowed. Because of this, it is important to refine the type information of a value if one checks that it is not *null*, to avoid reporting unnecessary errors. To that end, when analyzing a comparison expression that sees if a value is not *null*, the resulting *then store* is such that the *Null* type is excluded from the type of the value.

Listing 4.46: Not null comparison example

```
1 void use(
2   @Nullable @Requires("HasNext") Iterator iterator
3 ) {
4   // iterator: State "HasNext" | Null
5   if (iterator != null) {
6     // iterator: State "HasNext"
7   }
8 }
```

Imagine a method which receives an iterator in the *HasNext* state or the *null* value. If the variable, which potentially holds the iterator, is checked to see if it is not *null*, then we can be sure that the value is not *null* and we can safely perform operations on the iterator.

4.3.4.8 Protocol completion

Protocol completion is verified in the checking phase. To ensure that the protocol of all objects reaches completion, we need to consider all the places on the code where an object might no longer be used. For example, if an object is used inside a method and is neither returned nor delegated to another method, then the object is no longer used.

To handle such cases, the *exit stores* of each method, which contain all the type information about the variables and fields at the end of the method, are read and the type of each variable and field in those *stores* is verified. If the variable corresponds to a parameter with an *Ensures* annotation, the type of the variable needs to match what is specified in the annotation. If that is not the case, the corresponding type needs to be either *Ended* (the protocol has completed), *Moved* (the object was delegated) or the object needs to be in a "droppable" state (i.e. any state with the *drop: end* transition). Otherwise, an error is reported.

Listing 4.47: Protocol completion example (1)

```
1 void main() {
2   File file = new File();
3   // file: State "Init"
4   // Error: protocol of file was not completed
5 }
```

[5] Currently, the example as is would not work because the method call is disconnected from the *if* statements. It is only used to exemplify the reasoning employed for *switch* statements.

In this example, a file object was created but not used. Since the object was not delegated, its protocol was not finished, and the file was not in a "droppable" state, an error was reported.

Additionally, each assignment in the code is verified to ensure that the type of the assignee, before the statement, was either *Ended*, *Moved* or the object was in a state in which it could be "dropped". This is important because assignments might makes us lose the reference to the object that was previously pointed by the variable or field. Remember that assignments on parameters with the *Ensures* annotation are disallowed.

Listing 4.48: Protocol completion example (2)

```
1 void main() {
2   @Nullable File file = new File();
3   // file: State "Init"
4   file = null;
5   // file: Null
6   // Error: protocol of file was not completed
7 }
```

Imagine now that a file object is created, assigned to a variable, and then that variable is overwritten with the *null* value. Since the assignment was performed when the file object was not ready to be "dropped", an error is reported.

Furthermore, objects returned from method calls must be assigned to a variable so that they can be used until the protocol is completed, unless the returned object is already in a state in which it can be "dropped".

Listing 4.49: Protocol completion example (3)

```
1 @State("Init") File createFile() {
2   return new File();
3 }
4 void main() {
5   createFile();
6   // Error: returned object not used
7 }
```

In this example, the file object is instantiated with a helper method, *createFile*. Notice how that method is used but the returned file is not used. In this instance, an error would be reported.

Moreover, if an object associated with a protocol is passed to a method that cannot be analyzed, an error is reported, since it is unknown if the method will properly finish the protocol of the object.

Finally, the *stores* resulting from the *class analysis*, associated with the *end* states and each "droppable" state, are checked to ensure that each field has type *Ended*, *Moved* or that the object pointed by the field is in a state in which it can be "dropped". This is important to ensure that objects that are referenced from other objects have their protocol completed when their owner also finishes its protocol.

Listing 4.50: Protocol completion example (4)

```
1  @Typestate("File.protocol")
2  public class FileWrapper {
3    // Error: protocol of file was not completed
4    private File file = new File();
5    public void open() {
6      // this.file: State "Init"
7      file.open();
8      // this.file: State "Open"
9    }
10   public String read() {
11     // this.file: State "Open"
12     return file.read();
13     // this.file: State "Read"
14   }
15   public void close() {
16     // this.file: State "Read"
17   }
18 }
```

Consider for example a scenario (listing 4.50) where a file object is used through a wrapper object which implements the same interface and protocol as the file. The *open* method calls the *open* method of the file (line 7) and the *read* method calls the *read* method of the file (line 12). However, the *close* method does not close the file (line 16). This implies that when the wrapper object reaches the *end* state (when its *close* method is called), the file, which is stored inside of it, is still in the *Read* state, which is not a "droppable" state. In this example, an error would be reported because the protocol completion of the file was compromised.

4.4 Comparison with Mungo

In this section, we will present examples that highlight the features supported by the tool we implemented and that compare it with the current (as of this writing) version of Mungo[6].

With the examples, we illustrate that our tool is able to check what Mungo checks, find errors that the Mungo could not find, and that it includes additional features. Note that all examples presented in the Mungo's repository are correctly handled by our implementation, only requiring the addition of some annotations.

In the following examples, files with *Ok* in their name are files where no errors are expected, and files with *NotOk* in their name are files where errors are expected. The examples are simple in nature as to present specifically the relevant features and issues found in Mungo. Although they might not represent real code, they show common coding patterns.

[6] https://bitbucket.org/abcd-glasgow/mungo/src/73dd8aeb/

Table 4.1 summarizes and compares the features supported by Mungo and our tool. Square symbols show where only partial support exists for a given feature or where issues exist.

Features	Mungo	Our tool
Basic checking	✓	✓
Decisions on enumeration values	✓	✓
Decisions on boolean values	×	✓
Nullness checking	□[1,2]	✓
Linearity checking	□[1,3]	✓
Force protocol completion	□[3]	✓
Class analysis	×	✓
State refinement via annotations	×	✓
Droppable transitions	×	✓
Protocols for classes of libraries	×	✓
Improved flow analysis	×	✓
Decisions based on equality checks in conditions	×	✓

Table 4.1: Comparison between Mungo and our tool

4.4.1 Basic checking

The first example presents a very simple file protocol. This example is used to show that both tools are able to verify the correct use of objects, associated with a protocol, in basic, but common, cases.

Listing 4.51: *FileProtocol.protocol*

```
typestate FileProtocol {
  Init = {
    FileStatus open(): <OK: Read, ERROR: end>
  }
  Read = {
    String read(): Close
  }
  Close = {
    void close(): end
  }
}
```

On the initial state, the first declared in the protocol, which is the *Init* state, one can only open the file, since it is a precondition for the read operation. The *open* method returns an enumeration value which indicates if the operation succeeded. If the *open* call returns *OK*, the operation has succeeded and the state transits to *Read*, where one can read the file. If the *open* call returns *ERROR*, it means that the file could not be opened,

[1] Some errors are reported but they are a bit cryptic.
[2] *obj != null* refinement does not exist.
[3] Some corner cases are not handled.

and the protocol finishes. After reading the file, the state changes to *Close*, where one must free resources by calling the *close* method. After calling *close*, the protocol ends, and no other operations are allowed.

Listing 4.52: *Ok.java*

```
1  public class Ok {
2    public static void main(String args[]) {
3      File f = new File();
4
5      switch (f.open()) {
6        case OK:
7          System.out.println(f.read());
8          f.close();
9          break;
10       case ERROR:
11         break;
12     }
13   }
14 }
```

Both tools can verify the correct use of the file in the *Ok.java* file and report no errors, which is correct. They ensure that method calls are called in the correct order according to the specified protocol. Both are also able to understand that if the *open* call (line 5) returns *OK*, the file is in the *Open* state, allowing a read (line 7) and then a closing (line 8), and if it returns *ERROR*, the protocol has ended.

Listing 4.53: *NotOk.java*

```
1  public class NotOk {
2    public static void main(String args[]) {
3      File f = new File();
4
5      System.out.println(f.read());
6      f.close();
7    }
8  }
```

In the *NotOk.java* file, both tools can detect the error. They detect that the *read* method (line 5) is called without the *open* being called first. The Mungo reports that it found the use of the *read* method while it expected to find a state in which the *open* operation is allowed. Our tool reports the same issue, but in different words: the *read* method cannot be called in the *Init* state.

Listing 4.54: Mungo's output

```
1  NotOk.java: 3-14: Semantic Error
2    Object created at NotOk.java: 3. Typestate mismatch. Found: String read(). Expected:
         ↪ FileStatus open().
```

Listing 4.55: Our tool's output

```
NotOk.java:5: error: Cannot call read on state Init (got: Init)
   System.out.println(f.read());
                      ^
1 error
```

4.4.2 Decisions on boolean values

The following example presents an iterator protocol. In the first state, one can only call the *hasNext* method, to ensure that there are items in the iterator. If *hasNext* returns *true*, the state changes to *Next* and the next item can be obtained by calling the *next* method. The protocol then returns to the *HasNext* state. If *hasNext* returns *false*, there are no more items and the protocol finishes.

Listing 4.56: *JavaIteratorProtocol.protocol*

```
1 typestate JavaIteratorProtocol {
2   HasNext = {
3     boolean hasNext(): <true: Next, false: end>
4   }
5   Next = {
6     String next(): HasNext
7   }
8 }
```

Notice in the protocol that the state change upon calling *hasNext* depends on the boolean value that is returned (line 3). This example tests if both tools support decisions on boolean values, not just enumeration values.

Listing 4.57: *Ok.java*

```
1  import java.util.*;
2
3  public class Ok {
4    public static void main(String args[]) {
5      JavaIterator it = new JavaIterator(Arrays.asList(args).iterator());
6
7      while (it.hasNext()) {
8        System.out.println(it.next());
9      }
10   }
11 }
```

The first code example shows the correct use of the iterator. For illustrative purposes, this iterator implementation is a wrapper around a standard iterator from the Java library.

Listing 4.58: *NotOk.java*

```
1  import java.util.*;
2
3  public class NotOk {
4    public static void main(String args[]) {
5      JavaIterator it = new JavaIterator(Arrays.asList(args).iterator());
6
7      while (!it.hasNext()) {
8        System.out.println(it.next());
9      }
10   }
11 }
```

The second code example shows the incorrect use of the iterator. Notice how in line 7, the loop condition is the negation of the return value of the *hasNext* call.

Listing 4.59: Mungo's output

```
1  JavaIteratorProtocol.protocol: 3-5: Semantic Error
2    Method boolean hasNext() should return an enumeration type.
3
4  JavaIteratorProtocol.protocol:3,25: error: unexpected token "true"
5
6  JavaIteratorProtocol.protocol:3,37: error: unexpected token "false"
```

Mungo reports that the *hasNext* method should return an enumeration value and reports syntax errors when it reads the *true* and *false* tokens in line 3 of the protocol. This illustrates that *Mungo does not support decisions on boolean values*.

Listing 4.60: Our tool's output

```
1  NotOk.java:5: error: Object did not complete its protocol. Type: JavaIteratorProtocol{
       ↪ Next}
2      JavaIterator it = new JavaIterator(Arrays.asList(args).iterator());
3                 ^
4  NotOk.java:8: error: Cannot call next on ended protocol
5        System.out.println(it.next());
6                           ^
7  2 errors
```

Our tool does support boolean values, which can be checked in the conditions of *if* statements or loops. It reports no errors in the *Ok.java* file and detects the issues resulting from the wrong code in line 7 of the *NotOk.java* file. Since the return value of *hasNext* was negated, the loop will exit when there are items to be read, which means the protocol will not complete (first error), and the loop will be entered when the protocol has reached the *end* state, where the *next* call (line 8) is not allowed (second error).

This feature is useful because it allows one to use, for example, an iterator in a more natural way, without having to use an enumeration value and testing it in a *switch* statement, which is more verbose.

4.4.3 Nullness checking

Null pointer errors are very common in Java so, it is important that tools avoid those issues by analyzing the code statically while allowing for common programming patterns. The following examples will compare how the two tools handle *null* values.

Listing 4.61: *FileProtocol.protocol*

```
1  typestate FileProtocol {
2    Init = {
3      FileStatus open(): <OK: Read, ERROR: end>
4    }
5    Read = {
6      String read(): Close
7    }
8    Close = {
9      void close(): end
10   }
11 }
```

This example makes use of the same file protocol presented in section 4.4.1.

Listing 4.62: *Ok.java*

```
1  public class Ok {
2    public static void main(String args[]) {
3      @Nullable File f = args.length == 0 ? null : new File();
4
5      if (f != null) {
6        use(f);
7      }
8    }
9
10   public static void use(@Requires("Init") File f) {
11     switch (f.open()) {
12       case OK:
13         System.out.println(f.read());
14         f.close();
15         break;
16       case ERROR:
17         break;
18     }
19   }
20 }
```

In the first code example, a variable *f* is declared (line 3), which might contain a *null* value or point to a file. The variable is declared with a *Nullable* annotation, provided by our tool, which is used to declare nullable variables. If the variable does not contain a *null* value (line 5), the file is passed to the *use* method, which expects a file in the *Init* state (notice the use of the *Requires* annotation in line 10). Note that Mungo will read the code as if the annotations *Nullable* and *Requires* were not there.

Listing 4.63: *NotOk.java*

```
1  public class NotOk {
2    public static void main1(String args[]) {
3      @Nullable File f = new File();
4
5      switch (f.open()) {
6        case OK:
7          System.out.println(f.read());
8          f = null;
9          f.close();
10         break;
11       case ERROR:
12         break;
13     }
14   }
15
16   public static void main2(String args[]) {
17     use(null);
18   }
19
20   public static void use(@Requires("Init") File f) {
21     // ...
22   }
23 }
```

The second code example presents two attempts to use a file. In the *main1* method (line 2), a file is initialized (line 3), attempted to be opened (line 5), and if the open operation succeeded, a read is performed (line 7). After that, the variable *f*, which pointed to the file, is assigned to *null* (line 8), and then the file is attempted to be closed (line 9). In the *main2* method (line 16), the *use* method is called with a *null* value (line 17). The *use* method is the same as in the previous code example, expecting a non-null file in the *Init* state.

Listing 4.64: Mungo's output

```
1  NotOk.java: 8-13: Semantic Error
2    Object reference is used uninitialised.
3
4  NotOk.java: 0-0: Semantic Error
5    Object created at NotOk.java: 3. Typestate mismatch. Found: end. Expected: void close().
       ↪
6
7  Ok.java: 3-25: Semantic Error
8    Object reference is used uninitialised.
```

Mungo detects the *null* assignment in line 8 of the *NotOk.java* file, reporting an *used uninitialised* error. It also reports the incorrect *close* method call in line 9, although without providing a line number and without indicating that the issue is directly related

with an attempt to call *close* on a *null* value.

Unfortunately, it also reports a false positive in the *Ok.java* file because it does not refine the type of the variable upon comparing it with *null* in a condition expression (line 5). Moreover, Mungo produces a false negative in the *NotOk.java* file by not detecting that a *null* value is being used as a parameter for the *use* method (line 17).

Listing 4.65: Our tool's output

```
1  NotOk.java:8: error: Cannot override because object has not ended its protocol. Type:
       ↪ FileProtocol{Close}
2          f = null;
3          ^
4  NotOk.java:9: error: Cannot call close on null
5          f.close();
6                ^
7  NotOk.java:17: error: incompatible types in argument
8     use(null);
9         ^
10   found : Null null
11   required: FileProtocol{Init} File
12 3 errors
```

Our tool detects all the *null* related issues in the *NotOk.java* file. It detects the attempt to call the *close* method on a *null* value (line 9) and reports a type incompatibility when calling the *use* method with *null* (line 17), where a non-null value was expected. Our implementation also reports that, by overriding the *f* variable (line 8), we might lose a reference to another file which is left with a non-completed protocol. In the *Ok.java* no error is reported, since the tool is able to refine the type of the *f* variable upon comparing it with *null* (line 5), avoiding a false positive.

4.4.4 Linearity checking

Linear use of objects is important. If linearity is not enforced, it is hard to know if there was some other piece of code that changed the state of an object, breaking the assumptions of the static analysis, and making it difficult to properly check protocol compliance and completion. In the following examples, we will compare how both tools handle situations where there is more than one reference to an object.

Listing 4.66: *FileProtocol.protocol*

```
1  typestate FileProtocol {
2    Read = {
3      String read(): Read,
4      void close(): end
5    }
6  }
```

To focus on the linearity enforcement of the two versions, the examples will work

with a simplified version of the file protocol: there is no *open* method, multiple reads are allowed and, at any point, the file may be closed by calling the *close* method.

Listing 4.67: *Ok.java*

```
1  public class Ok {
2    public static void main1() {
3      File f = new File();
4      use(f);
5    }
6    public static void main2() {
7      File f = new File();
8      File f2 = f;
9      use(f2);
10   }
11   public static void use(File f) {
12     System.out.println(f.read());
13     f.close();
14   }
15 }
```

In the first code example, there are two uses of files. In the *main1* method (line 2), a file is declared (line 3) and then passed as a parameter to the *use* method (line 4), which will read and close the file (lines 12-13). In the *main2* method (line 6), a file is declared (line 7), being referenced by the *f* variable, and then also referenced by the *f2* variable (line 8), which is then passed as a parameter to the *use* method (line 9). These two cases respect linearity and, even though a second reference to the file is present in *main2*, only the second one is used.

Listing 4.68: *NotOk.java* (Part 1)

```
1  import java.util.function.Supplier;
2
3  public class NotOk {
4    public static void main1() {
5      File f = new File();
6      use(f);
7      f.read();
8    }
9
10   public static void main2() {
11     File f = new File();
12     use(f);
13     use(f);
14   }
```

Listing 4.69: *NotOk.java* (Part 2)

```
16   public static void main3() {
17     File f = new File();
18     File f2 = f;
19     use(f2);
20     f.read();
21   }
22
23   public static void main4() {
24     File f = new File();
25     File f2 = f;
26     use(f2);
27     use(f);
28   }
29
30   public static void main5() {
31     File f = new File();
32     Supplier<String> fn = () -> {
33       return f.read();
34     };
35     f.close();
36     fn.get();
37   }
38
39   public static void use(File f) {
40     System.out.println(f.read());
41     f.close();
42   }
43 }
```

In the second code example, there are five cases where linearity is broken with the consequence of breaking the file protocol, allowing reads when the file has already been closed. In the *main1* method (line 4), a file is created, used in the *use* method (which will read it and close it), and then there is an attempt to read it again. In the *main2* method (line 10), a file is used twice. In the *main3* method (line 16), a file is created and referenced by two variables. The second variable is used to pass the file to the *use* method and then there is an attempt to read the file again (line 20). In the *main4* method (line 23), two variables reference the same file and are both used to pass the file to the *use* method. And finally, in the *main5* method (line 30), a file is created and referenced from within a lambda function. Before the lambda is called, the file is closed, forbidding the read operation (line 33) that would happen when the lambda is called (line 36).

Listing 4.70: Mungo's output

```
NotOk.java: 6-9: Semantic Error
  Object reference is used uninitialised.

NotOk.java: 12-9: Semantic Error
  Object reference is used uninitialised.

NotOk.java: 18-15: Semantic Error
  Object reference is used uninitialised.

NotOk.java: 25-15: Semantic Error
  Object reference is used uninitialised.
```

Mungo reports multiple *Object reference is used uninitialised* errors, which seem cryptic, since the variables are clearly initialized, making it hard to understand what Mungo is actually trying to enforce. Mungo is also unable to detect that an object is referenced from inside a lambda, which is then called in the wrong order, breaking the protocol of the file.

Listing 4.71: Our tool's output

```
NotOk.java:7: error: Cannot call read on moved value
    f.read();
        ^
NotOk.java:13: error: incompatible types in argument
    use(f);
       ^
  found : Moved File
  required: FileProtocol{Read} File
NotOk.java:20: error: Cannot call read on moved value
    f.read();
        ^
NotOk.java:27: error: incompatible types in argument
    use(f);
       ^
  found : Moved File
  required: FileProtocol{Read} File
NotOk.java:33: error: f was moved to a different closure
     return f.read();
            ^
5 errors
```

Our tool is able to detect all the issues in the *NotOk.java* file. Every time a reference is assigned to a new variable or passed as a parameter to a method, the variable that had a hold of that reference will be marked with the *Moved* type, which will disallow uses of it. That can be observed by the errors reported. In line 7, it is reported that a *read* call is not allowed on a "moved value". In line 13, a file with the *Moved* type is not compatible with a file in the *Read* state. In lines 20 and 27 the errors reported match the ones reported in

lines 7 and 13 respectively. Finally, in line 27, it is detected that the *f* variable is being referenced from inside a different closure, which is completely disallowed in this version of the tool.

Additionally, it is important to handle the case where an object is passed into a method which source code is not available to be analyzed.

Listing 4.72: *NotOk2.java*

```
1  import java.util.*;
2
3  public class NotOk2 {
4    public static void main1() {
5      List<File> list = new LinkedList<>();
6      list.add(new File());
7      File f1 = list.get(0);
8      File f2 = list.get(0);
9      f1.read();
10     f1.close();
11     f2.read();
12     f2.close();
13   }
14 }
```

In this example, there is a linked list capable of storing files. A single file is stored (line 6) and then retrieved twice (lines 7-8), which creates two references to the same file. The code does not follow the file protocol since the file is read, closed, and then read again.

Listing 4.73: Our tool's output

```
1  NotOk2.java:6: error: Passing an object with protocol to a method that cannot be analyzed
2     list.add(new File());
3             ^
4  NotOk2.java:7: error: incompatible types in assignment
5     File f1 = list.get(0);
6                     ^
7    found : FileProtocol{Read} | Ended | Moved File
8    required: FileProtocol{Read} File
9  NotOk2.java:8: error: incompatible types in assignment
10    File f2 = list.get(0);
11                    ^
12   found : FileProtocol{Read} | Ended | Moved File
13   required: FileProtocol{Read} File
14 NotOk2.java:9: error: Cannot call read on ended protocol, on moved value
15    f1.read();
16          ^
17 NotOk2.java:11: error: Cannot call read on ended protocol, on moved value
18    f2.read();
19          ^
20 5 errors
```

Our tool currently reports various errors. The first error (line 6) indicates that an object with protocol is being delegated to a method that cannot be analyzed. The two following errors (lines 7 and 8) indicate that the object returned by the *get* method might be aliased (i.e. includes the *Moved* type) which is not compatible with the type of the variable assigned to. The two last errors (lines 9 and 11) report that methods cannot be called on these potentially aliased objects. The reason we consider these to be aliased is because the *get* method cannot be analyzed.

Unfortunately, Mungo crashes with this last example for an unknown reason to us.

4.4.5 Force protocol completion

It is important that the protocol of objects reaches completion, to ensure necessary method calls are not forgotten, thus ensuring correctness, and ensuring that used resources are freed. The following examples will compare the two tools in their enforcement of protocol completion. To focus on the protocol completion aspect of typestate checking, these examples already present linear use of objects.

Listing 4.74: *FileProtocol.protocol*

```
1 typestate FileProtocol {
2   Read = {
3     String read(): Read,
4     void close(): end
5   }
6 }
```

This example makes use of the same file protocol presented in section 4.4.4.

Listing 4.75: *Ok.java*

```
1  public class Ok {
2    public static void main1() {
3      File f = new File();
4      System.out.println(f.read());
5      f.close();
6    }
7
8    public static void main2() {
9      File f = new File();
10     use(f);
11   }
12
13   public static void use(File f) {
14     System.out.println(f.read());
15     f.close();
16   }
17 }
```

In the first code example there are two uses of files. In the *main1* method (line 2), a

file is declared, then read, and then closed. In the *main2* method (line 8), a file is declared and then passed to the *use* method (line 10), which will then read and close the file (lines 14-15). Both cases show the correct completion of the file's protocol.

Listing 4.76: *NotOk.java*

```
1  public class NotOk {
2    public static void main1() {
3      File f = new File();
4    }
5
6    public static void main2() {
7      File f = new File();
8      use(f);
9    }
10
11   public static void use(File f) {
12
13   }
14 }
```

In the second code example there are two cases where files are created but not used to completion. In the *main1* method (line 2), a file is created but not used. In the *main2* method (line 6), a file is declared and passed as a parameter to the *use* method, which does nothing (line 11).

Listing 4.77: Mungo's output

```
1  NotOk.java: 3-14: Semantic Error
2    Object created at NotOk.java: 3. Typestate mismatch. Found: end. Expected: String read
         ↪ (), void close().
3
4  NotOk.java: 7-14: Semantic Error
5    Object created at NotOk.java: 7. Typestate mismatch. Found: end. Expected: String read
         ↪ (), void close().
```

Mungo is able to detect both issues where the files in lines 3 and 7 are not closed, being left in their initial state, thus the error that it found the *end* state while it expected *read* or *close* calls.

Listing 4.78: Our tool's output

```
1  NotOk.java:3: error: Object did not complete its protocol. Type: FileProtocol{Read}
2    File f = new File();
3        ^
4  NotOk.java:11: error: Object did not complete its protocol. Type: FileProtocol{Read}
5    public static void use(File f) {
6                        ^
7  2 errors
```

Our tool also detects the same issues, although the second one is reported in the declaration of the parameter of the *use* method (line 11).

Although both tools detect the errors in the previous code examples, there are some corner cases that should be handled as well, for example, when an object with protocol is stored inside another object without protocol or which source code is not available to be analyzed.

Listing 4.79: *NotOk2.java*

```
1  import java.util.*;
2
3  public class NotOk2 {
4    public static void main1() {
5      List<File> list = new LinkedList<>();
6      list.add(new File());
7    }
8
9    public static class FileWrapper {
10     public File file = new File();
11   }
12
13   public static void main2() {
14     FileWrapper file = new FileWrapper();
15   }
16 }
```

In this example, the *main1* method (line 4) includes the creation of a linked list and then the addition of a single file (lines 5-6). The program then terminates without the file having reached the end of its protocol. In the *main2* method (line 13), an object without protocol is created which holds a reference to a file (line 10). That file will not be used to completion.

Listing 4.80: Our tool's output

```
1  NotOk2.java:6: error: Passing an object with protocol to a method that cannot be analyzed
2      list.add(new File());
3               ^
4  NotOk2.java:10: error: Object did not complete its protocol. Type: FileProtocol{Read} |
       ↪ Ended | Moved
5      public File file = new File();
6                 ^
7  2 errors
```

Our tool detects the first issue by reporting that an object with protocol was passed into a method that cannot be analyzed and it also detects the second issue by reporting that the protocol of the file stored inside the wrapper object was not completed.

Unfortunately, Mungo crashes with this last example for an unknown reason to us.

4.4.6 Class analysis

When an object with protocol is stored inside another object, it is also important to check that its protocol is followed and that it reaches completion. Our tool is able to ensure that the use of objects with protocol inside another object, comply with their respective protocols, thanks to the *class analysis* explained in section 4.3.3.

Listing 4.81: *FileProtocol.protocol*

```
1 typestate FileProtocol {
2   Read = {
3     String read(): Read,
4     void close(): end
5   }
6 }
```

Listing 4.82: *FileWrapperProtocol.protocol*

```
1 typestate FileWrapperProtocol {
2   Init = {
3     void init(File): Read
4   }
5   Read = {
6     String read(): Read,
7     void close(): end
8   }
9 }
```

This example makes use of the same file protocol presented in section 4.4.4 and makes use of a *FileWrapper*. This wrapper holds a reference to a file. Its protocol matches the file protocol except that it needs to be initialized by the use of the *init* method, which receives a file and then allows the file to be operated (line 3 of the *FileWrapper* protocol).

Listing 4.83: *OkFileWrapper.java*

```
1  @Typestate("FileWrapperProtocol")
2  class OkFileWrapper {
3    private @Nullable File file = null;
4    public void init(File file) {
5      this.file = file;
6    }
7    public String read() {
8      return file.read();
9    }
10   public void close() {
11     file.close();
12   }
13 }
```

The first code example presents a correct implementation of the *FileWrapper*. Initially,

the *file* field is initialized with a *null* value (line 3). Notice the use of the *Nullable* annotation. The field is updated with a non-null value in the *init* method (line 5), which must be called first according to the protocol. Then, the *read* (line 7) and *close* (line 10) methods are available and call the respective *read* and *close* methods of the file which is stored inside the wrapper.

Listing 4.84: *NotOkFileWrapper1.java*

```
1  @Typestate("FileWrapperProtocol")
2  class NotOkFileWrapper1 {
3    private @Nullable File file = null;
4    public void init(File file) {
5
6    }
7    public String read() {
8      return file.read();
9    }
10   public void close() {
11     file.close();
12   }
13 }
```

The second code example presents an incorrect implementation of the *FileWrapper*. It matches the previous example except that there is no initialization of the *file* field with a non-null value in the *init* method (line 5).

Listing 4.85: *NotOkFileWrapper2.java*

```
1  @Typestate("FileWrapperProtocol")
2  class NotOkFileWrapper2 {
3    private @Nullable File file = null;
4    public void init(File file) {
5      this.file = file;
6    }
7    public String read() {
8      return file.read();
9    }
10   public void close() {
11
12   }
13 }
```

The third code example also presents an incorrect implementation. It matches the first implementation except that the *close* method (line 11) does not call the *close* method of the file stored inside the wrapper.

Listing 4.86: *NotOkFileWrapper3.java*

```
1  @Typestate("FileWrapperProtocol")
2  class NotOkFileWrapper3 {
3    private @Nullable File file = null;
4    public void init(File file) {
5      this.file = file;
6    }
7    public String read() {
8      file.close();
9      return "";
10   }
11   public void close() {
12     file.read();
13   }
14 }
```

The fourth code example also presents an incorrect implementation of the *FileWrapper*. Even though the file is properly initialized, notice how when the *read* method of the wrapper (line 7) is called, the *close* method of the file is called (line 8), and when the *close* method of the wrapper (line 11) is called, the *read* method of the file is called (line 12).

Listing 4.87: Mungo's output

```
1  NotOkFileWrapper1.java: 3-31: Semantic Error
2    Object reference is used uninitialised.
```

Mungo is only able to report an error related with a *null* value in the *NotOkFileWrapper1* file. All of the other issues are not detected.

Listing 4.88: Our tool's output (Part 1)

```
1  NotOkFileWrapper1.java:4: error: Object did not complete its protocol. Type: FileProtocol
       ↪ {Read}
2    public void init(File file) {
3                       ^
4  NotOkFileWrapper1.java:8: error: Cannot call read on null
5      return file.read();
6                  ^
7  NotOkFileWrapper1.java:11: error: Cannot call close on null
8      file.close();
9              ^
10 NotOkFileWrapper2.java:3: error: Object did not complete its protocol. Type: FileProtocol
       ↪ {Read}
11   private @Nullable File file = null;
12                           ^
```

Listing 4.89: Our tool's output (Part 2)

```
13 NotOkFileWrapper3.java:3: error: Object did not complete its protocol. Type: FileProtocol
       ↪ {Read}
14   private @Nullable File file = null;
15                           ^
16 NotOkFileWrapper3.java:8: error: Cannot call close on ended protocol
17     file.close();
18             ^
19 NotOkFileWrapper3.java:12: error: Cannot call read on ended protocol
20     file.read();
21             ^
22 7 errors
```

Our tool enforces protocol compliance of objects inside other objects and ensures that if the outer object reaches the end of its protocol, all of the inner objects are also used to completion. For the examples given, our tool detects the following errors:

- in the *NotOkFileWrapper1* file, it is detected that the file passed as parameter is not used (line 4), leaving the *file* field with a *null* value, which also results in the *read* and *close* being called on *null* (lines 8 and 11).

- in the *NotOkFileWrapper2* file, since the *close* method of the file is not called when the *close* method of the wrapper is called, the file is left in the *Read* state, without completing its protocol, thus the fourth error.

- in the *NotOkFileWrapper3*, since the file's *close* method was getting called in the *read* method of the wrapper, subsequent reads and a close will result in reads and closings being performed in a file which has already completed its protocol, thus the errors in lines 8 and 12. Additionally, since it is possible to call the *close* method of the wrapper immediately after initialization, which actually performs a read on the file, the file might be left in the *Read* state, thus the last error reported.

Note that, if an object is not associated with a protocol specification, a trivial protocol is attributed to it. That trivial protocol has only one state. In that state, the object may be "dropped", all its methods are available to be called, and any method call leaves the object in the same state.

Listing 4.90: *NotOkFileWrapper4.java*

```
1  class NotOkFileWrapper4 {
2    private @Nullable File file = null;
3    public void init(File file) {
4      this.file = file;
5    }
6    public String read() {
7      return file.read();
8    }
9    public void close() {
10
11   }
12 }
```

This fifth example presents an implementation of a *FileWrapper* where the *close* method does not actually close the file stored inside (line 10). Additionally, there is no *Typestate* annotation which would associate this implementation with a protocol specification. This means that the wrapper allows methods to be called in any order.

Listing 4.91: Our tool's output

```
1  NotOkFileWrapper4.java:2: error: Object did not complete its protocol. Type: FileProtocol
       ↪ {Read} | Null
2    private @Nullable File file = null;
3                           ^
4  NotOkFileWrapper4.java:4: error: Cannot override because object has not ended its
       ↪ protocol. Type: FileProtocol{Read} | Null
5      this.file = file;
6         ^
7  NotOkFileWrapper4.java:7: error: Cannot call read on null
8      return file.read();
9                  ^
10 3 errors
```

Since methods of this wrapper object may be called in any order, multiple errors are reported. The first error is reported because the protocol of the file might not be completed. This may happen in a scenario where the wrapper object is "dropped" immediately after the *init* method is called. The second error is reported because one could call the *init* method multiple times, which would overwrite the *file* field and make us lose a reference to the previous file, which protocol was not completed. The third error is reported because the *read* method could be called before the *init* method, which means that there could be an attempt to call a method on a *null* value.

Unfortunately, Mungo crashes with this last example for an unknown reason to us.

4.4.7 State refinement via annotations

This example presents an important feature implemented in our tool. It allows one to use annotations to specify the states in which an object passed in a parameter should be in, using the *Requires* annotation, the states in which an object passed in a parameter is left in, using the *Ensures* annotation, and in which states an object that is returned is in, using the *State* annotation.

Listing 4.92: *FileProtocol.protocol*

```
typestate FileProtocol {
  Init = {
    FileStatus open(): <OK: Read, ERROR: end>
  }
  Read = {
    String read(): Close
  }
  Close = {
    void close(): end
  }
}
```

This example makes use of the same file protocol presented in section 4.4.1.

Listing 4.93: *Ok.java*

```
1  public class Ok {
2    public static void main() {
3      File f = createFile();
4
5      switch (f.open()) {
6        case OK:
7          read(f);
8          f.close();
9          break;
10       case ERROR:
11         break;
12     }
13   }
14
15   public static @State("Init") File createFile() {
16     return new File();
17   }
18
19   public static void read(@Requires("Read") @Ensures("Close") File f) {
20     f.read();
21   }
22 }
```

In the *main* method of the *Ok.java* file, a file is first created with the help of the *createFile* method (line 3). Notice how before the return type of the *createFile* method

there is a *State* annotation indicating that the returned file is in the *Init* state (line 15). After that, there is an attempt to open the file (line 5). If that succeeds, the file is read, by passing it to the static *read* method (line 7), and then closed (line 8). Notice how the static *read* method uses the *Requires* and *Ensures* annotations to indicate that it expects a file in the *Read* state and that the file is left in the *Close* state.

Listing 4.94: *NotOk.java*

```
1  public class NotOk {
2    public static void main() {
3      File f = createFile();
4
5      switch (f.open()) {
6        case OK:
7          f.read();
8          read(f);
9          f.close();
10         break;
11       case ERROR:
12         break;
13     }
14   }
15
16   public static @State("Init") File createFile() {
17     return new File();
18   }
19
20   public static void read(@Requires("Read") @Ensures("Close") File f) {
21     f.read();
22   }
23 }
```

In the *main* method of the *NotOk.java* file, a file is first created with the help of the *createFile* method (line 3). After that, there is an attempt to open the file (line 5). If that succeeds, the file is read by directly calling the *read* method of the file (line 7), then the file is passed into the static *read* method (line 8), and then there is an attempt to close the file (line 9). Notice how the static *read* method expects a file in the *Read* state but is receiving it in the *Close* state.

Listing 4.95: Mungo's output

```
1  NotOk.java: 8-14: Semantic Error
2    Object reference is used uninitialised.
3
4  Ok.java: 7-14: Semantic Error
5    Object reference is used uninitialised.
```

Mungo reports two errors. One in the *Ok.java* file and one in the *NotOk.java* file. The message of these errors does not make it clear what the issue is. Also note that we did not expect any error to be reported in the *Ok.java* file.

Listing 4.96: Our tool's output

```
NotOk.java:8: error: incompatible types in argument
      read(f);
           ^
  found : FileProtocol{Close} File
  required: FileProtocol{Read} File
1 error
```

Our tool successfully verifies the use of the file in the *Ok.java* file and also detects the error in line 8 of the *NotOk.java* file, reporting that the static *read* method expected a file in the *Read* state but received one in the *Close* state.

4.4.8 Droppable transition

In the following example we show that our tool supports "droppable" transitions, which may be used in any state where an object may safely stop being used, without the need to reach the *end* state.

Listing 4.97: *MyComparatorProtocol.protocol*

```
1 typestate MyComparatorProtocol {
2   Start = {
3     int compare(int, int): Start,
4     drop: end
5   }
6 }
```

This example makes use of a comparator object. The protocol of this comparator has only one declared state. On that state, the *compare* method may be called. After the method call, the object remains in the same state. Additionally, the object may be safely "dropped" in that state, as indicated by the *drop: end* transition (line 4).

Listing 4.98: *MyComparator.java*

```
1 @Typestate("MyComparatorProtocol")
2 public class MyComparator {
3   public int compare(int a, int b) {
4     return a < b ? -1 : a > b ? 1 : 0;
5   }
6 }
```

The *compare* method of the comparator object receives two integer values and returns -1 if the first is smaller than the second, returns 1 if the first is greater than the second, and returns 0 if both are equal.

Listing 4.99: *Ok.java*

```
1 public class Ok {
2  public static void main() {
3   MyComparator comparator = new MyComparator();
4     System.out.println(comparator.compare(10, 5));
5  }
6 }
```

In the *main* method of the *Ok.java* file, a comparator object is first initialized. Following that, the *compare* method is called with two integers values, and the result is printed to the standard output channel. Then, the object is no longer used.

Mungo reports a syntax error in the protocol specification since it does not support the *drop: end* transition. Our tool does not report any errors since the use of the comparator object follows the specification.

4.4.9 Protocols for classes of libraries

In the following example we show that our tool supports that protocol specifications be associated with classes using a configuration file and that it supports *stub files*, which allow one to write annotations for a library when the code is not available to be edited.

Listing 4.100: Configuration file

```
1 java.util.Iterator=JavaIterator.protocol
```

To use this feature, a configuration file needs to be provided to map the full qualified name of a Java class with a protocol specification.

Listing 4.101: Stub file

```
1 package java.util;
2
3 public interface List<E> {
4   @State("HasNext") Iterator<E> iterator();
5 }
```

Additionally, a *stub file* is provided to indicate that the *iterator* method of a list returns an iterator in the *HasNext* state.

Listing 4.102: *NotOk.java*

```
1 import java.util.*;
2
3 public class NotOk {
4  public static void main(String[] args) {
5     Iterator<String> it = Arrays.asList(args).iterator();
6     System.out.println(it.next());
7   }
8 }
```

In this example, the array of arguments provided to the *main* method is taken to produce a list of strings. Then the *iterator* method on that list is called, and the *next* method is called without calling the *hasNext* method first.

Listing 4.103: Mungo's output

```
None
```

Listing 4.104: Our tool's output

```
Main.java:6: error: Cannot call next on state HasNext (got: HasNext)
  System.out.println(it.next());
                         ^
2 errors
```

Mungo does not report any errors because the iterator has no protocol associated with it. Our tool reports that the *next* method is being called when the iterator is in the *HasNext*, which means that the incorrect use of the iterator is properly detected.

4.4.10 Improved flow analysis

When type checking, it is important to understand the flow of execution so that type information is propagated in a correct way, first to avoid false negatives, and second, to reduce false positives. In our tool, the Checker Framework is responsible for building the control flow graph, so we did not need to implement that functionality. Since Checker has been used by other projects and battle tested, we trust its implementation. This section presents two examples.

Listing 4.105: *JavaIteratorProtocol.protocol*

```
typestate JavaIteratorProtocol {
  HasNext = {
    Boolean hasNext(): <True: Next, False: end>
  }
  Next = {
    String next(): HasNext
  }
}
```

The first example makes use of an iterator which, for illustrative purposes, is a wrapper around the standard Java iterator. The iterator has two states, the *HasNext* state, where one must check if there are items to be obtained, and the *Next* state, where one can extract the next item. The *hasNext* method (line 3), available in the *HasNext* state, returns an enumeration value, instead of a boolean value, so that the protocol is accepted by Mungo.

Listing 4.106: *NotOk.java*

```
1  import java.util.*;
2
3  public class NotOk {
4   public static void main(String[] args) {
5    JavaIterator it = new JavaIterator(Arrays.asList(args).iterator());
6
7      loop: do {
8        switch(it.hasNext()) {
9          case True:
10           System.out.println(it.next());
11           continue loop;
12         case False:
13           break loop;
14       }
15     } while(false);
16  }
17 }
```

The code for the first example shows the use of an iterator. There is a loop that keeps running while there are items in the iterator. In the loop body, the *hasNext* method is called (line 8). If it returns *True*, the next item is extracted, via the *next* method (line 10), and the loop continues. If it returns *False*, the loop stops.

Notice that the loop condition is not *true* but *false* (line 15). This means that at most one item will be obtained from the iterator. This is the case because the *continue* statement jumps to the condition, which evaluates to *false*, not the loop body. If there is more than one item in the iterator, the iterator will not be used to completion. Mungo does not detect that, while our tool does, as one can see by the following outputs.

Listing 4.107: Mungo's output - If condition is *false*

```
1  None
```

Listing 4.108: Our tool's output - If condition is *false*

```
1  NotOk.java:5: error: Object did not complete its protocol. Type: JavaIteratorProtocol{
       ↪ HasNext} | Ended
2    JavaIterator it = new JavaIterator(Arrays.asList(args).iterator());
3                ^
4  1 error
```

Mungo seems to wrongly assume that a *continue* statement jumps to the beginning of a loop, when in actuality, a *continue* statement jumps to the condition expression. If the condition were to be always *true*, there would be no practical difference, but if the condition were to be *false*, like in the example, bugs could go unnoticed.

Our tool, besides avoiding the previous false negative, also avoids a false positive in the case a literal boolean value *true* is used in the condition expression of a loop. In that

instance, it recognizes that when that condition is evaluated, the loop body will always be entered. This is useful because there are code generation tools, like StMungo [54], which produce loops with *true* conditions. This can be illustrated by changing the previous example to use *true* instead of *false* in the loop condition (line 15). Our tool reports no error in this scenario, which is correct. This particular case required special code, since it is not supported by Checker outside of the box, although there are plans for that[7].

Listing 4.109: Our tool's output - If condition is *true*

```
None
```

In the following example, which makes use of the same file protocol presented in section 4.4.1, we test how both tools handle the possible flows of execution around a *switch* statement.

Listing 4.110: *NotOk.java*

```
1  public class NotOk {
2    public static void main(String args[]) {
3      File f = new File();
4
5      switch (f.open()) {
6        case OK:
7          switch (f.read()) {
8            case "CLOSE":
9              f.close();
10             break;
11         }
12         break;
13       case ERROR:
14         break;
15     }
16   }
17 }
```

In this example, a file is created (line 3), and then opened (line 5). If the *open* operation failed, by returning *ERROR*, nothing more is done. If the *open* operation succeeded, by returning *OK*, the file is read (line 7). After the *read* operation, the file should be closed, but that only happens if the *read* method returns a string with the content *"CLOSE"*.

Listing 4.111: Mungo's output

```
None
```

Mungo does not detect that it is possible that no case in the *switch* statement matches, assuming that the code for the *"CLOSE"* case will always run.

[7] https://github.com/typetools/checker-framework/issues/3249

Listing 4.112: Our tool's output

```
1 NotOk.java:3: error: Object did not complete its protocol. Type: FileProtocol{Close} |
      ↪ Ended
2     File f = new File();
3         ^
4 1 error
```

Our tool understands that it is possible that the file is not closed so, it reports that the file either is the *end* state, or was left in the *Close* state, after the file was read.

4.4.11 Decisions based on equality checks in conditions

If a method returns an enumeration value, and if one is only interested in knowing if that matches or not another value, it might be too verbose to check it in a *switch* statement. The following example presents an enumeration value being compared with the use of a == operator.

Listing 4.113: *JavaIteratorProtocol.protocol*

```
1 typestate JavaIteratorProtocol {
2   HasNext = {
3     Boolean hasNext(): <True: Next, False: end>
4   }
5   Next = {
6     String next(): HasNext
7   }
8 }
```

This example makes use of the same iterator protocol presented in section 4.4.10.

Listing 4.114: *Ok.java*

```
1  import java.util.*;
2
3  public class Main {
4   public static void main(String[] args) {
5    JavaIterator it = new JavaIterator(Arrays.asList(args).iterator());
6
7      while(it.hasNext() == Boolean.True){
8        System.out.println(it.next());
9      }
10  }
11 }
```

In this example, an iterator is created and items are retrieved until there are no more items in it. Notice that the *hasNext* method returns an enumeration value, which is compared against *Boolean.True* with a == operator (line 7).

Listing 4.115: Mungo's output

```
Main.java: 0-0: Semantic Error
  Object created at Main.java: 6. Typestate mismatch. Found: String next(), end, Boolean
      ↪ hasNext(). Expected: <True, False>.
```

Mungo currently forces enumeration values to be checked in a *switch* statement. Because of this, Mungo reports an error indicating that it expected <*True*, *False*> (a *decision state*), but instead found the *end* state, a *next* call and a *hasNext* call.

Listing 4.116: Our tool's output

```
None
```

Our tool reports no errors in this example since it supports decisions to be based on equality checks done in conditions, properly recognizing what will be the new state of the object after the method call, according to the value returned.

4.5 Conclusion

In this chapter, we presented a tool for type-checking Java programs where objects are associated with typestates. This tool was inspired by Mungo [82, 54]. We also discussed the features this tool supports and the improvements over the current version of Mungo. In the following chapters, we will discuss how aliasing may be allowed, instead of forcing all objects to be used in a linear way.

Chapter 5

Theoretical work on Access Permissions

If multiple references to the same object exist, type information can get outdated if the object changes state via another reference. In the first version of the tool, we enforced that objects were used in a linear way to ensure that method calls on objects followed the protocol. While enforcing linear use of objects made the task of checking protocol compliance easier, it restricted what a programmer could do. For example, it is very common to have shared data, something that is not possible if linear use is enforced. In the following sections, we discuss different approaches that allow programs that share resources to be verified.

5.1 Owicki-Gries method and Rely-Guarantee

The **Owicki-Gries method** is an axiomatic method for proving properties of parallel programs [69]. The system Owicki and Gries proposed includes the Hoare logic [39] rules for sequential programs and a rule for parallel composition. This rule allows one to compose two programs into a concurrent program but requires that their proofs do not "interfere" [84, 55].

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \{P_2\}\ c_2\ \{Q_2\} \qquad \text{the two proofs are non-interfering}}{\{P_1 \wedge P_2\}\ c_1 \| c_2\ \{Q_1 \wedge Q_2\}}$$

Figure 5.1: Owicki-Gries method

Rely-Guarantee reasoning [51] is a compositional verification method for shared memory concurrency based on the Owicki-Gries method. In this system, specifications have four components: the pre-condition, which describes the initial state of the program; the post-condition, which relates the initial state to the final state; the rely condition,

which describes the interference the program can tolerate; and the guarantee condition, which describes the interference it imposes on other concurrent programs [51, 84].

5.2 Separation Logic

Separation Logic [73, 67, 45] is based on the logic of bunched implications [66], and includes the notion of resources. This program logic extends Hoare logic with new connectives and the separation conjunction. It has been mainly used to reason about sequential programs that manipulate pointer data structures. In this approach, each program state is divided into a heap and a store part, allowing explicit local reasoning about the memory. With the separation formula, one can ensure that two programs accessing the same location do not interfere to verify program behavior [74, 84].

Concurrent Separation Logic [11, 68] is a new realization of Separation Logic to allow reasoning about parallel programs that share resources. With this logic, if two threads operate on disjoint parts of the memory, they can be verified in a safe and isolated way. Additionally, this logic introduces the notion of resource invariants, where each rule includes a *precise* separation logic assertion that always holds during the execution of the program except when a thread is inside an atomic block [74, 84].

$$\frac{\{P\}\ C\ \{Q\} \qquad \{P'\}\ C'\ \{Q'\}}{\{P * P'\}\ C \| C'\ \{Q * Q'\}}$$

Figure 5.2: Disjoint concurrency rule

5.3 Access permissions

Access permissions are abstract capabilities that characterize the way a shared resource can be accessed by multiple references [74].

This notion is built on Linear Logic [34], which treats permissions as linear resources, and Separation Logic [73, 67], which reasons about program behavior against specifications. Classic Separation Logic does not support concurrent reads of a memory location by multiple references or threads. To add support for concurrency, Boyland (2003) and Bornat et al. (2005) combined Separation Logic with access permissions [74, 10, 8].

Access permissions are used to ensure that only a reference can write on a particular location at any given time, and to ensure that if a location is read by a thread, all other threads only have read permission for that location. This avoids interference in concurrent programs [74].

Plural [7] and Plaid [64] are examples of tools for Java where access permissions are combined with typestate abstractions to verify protocol compliance in sequential and

concurrent programs. Since then, many approaches were used to verify program behavior in concurrent programs with shared-memory [74].

5.3.1 Fractional permissions

Fractional permissions [10] are concrete mathematical values between 0 and 1 inclusive. These represent the permission for a shared resource. The absence of permission is represented by the value 0. Full permission is represented by the value 1. Shared read-only access is represented by a value strictly between 0 and 1. Fractional permissions can be split into a number of fractions and distributed among multiple references. For example, a permission s can be split into s_1 and s_2 such that $s = s_1 + s_2$, allowing two references to have access to the same resource. Permissions that were split may also be joined again. [74].

5.3.2 Counting permissions

Counting permissions [8] are similar to fractional permissions. While fractional permissions are values between 0 and 1 inclusive, counting permissions are values between 0 and a maximum constant value. The value 0 represents the absence of permission and the maximum constant value represents full permission. Read-only access is represented by a value between 0 and the maximum constant value [74].

5.3.3 Symbolic permissions

Symbolic permissions [7] are a simplified extension of fractional permissions. Instead of using concrete fractional values to represent and split permissions, symbolic permissions are represented with five types of permissions: *unique*, *full*, *share*, *pure*, and *immutable*. Like other types of permissions, these may be split and joined [74].

Unique provides exclusive access to a reference to read and modify the referenced object. *Full* grants a reference read and write access while also granting read-only access to other references. *Share* allows references to read and modify the object. *Pure* gives read-only access to a reference while other references may read or modify the object. *Immutable* gives read-only access to all references [74].

5.4 Other approaches

Other proposed approaches that provide aliasing control include **view-based typestates** [62], **adoption and focus** [29], and **capability calculus** [19]. *View-based typestates* are a generalization of access permissions that allows the creation of *views*, which are projections of an object where the fields and methods have specific permissions [62]. The *adoption and focus* strategy relaxes linearity by allowing one to alias objects via adoption, and get

a temporary linear view on an object via focus [29]. *Capability calculus* is a compiler intermediate language that supports *region-based memory management* [81, 80] and includes a safe type system, which controls the permissibility of operations, such as memory access and deallocation, and tracks aliasing information [19].

5.5 Motivating example: Cell example

To show the usefulness of access permissions, consider the following example, which makes use of a cell object and an item object. The protocol and implementation of the cell are provided in listings 5.1 and 5.2, respectively. The protocol and implementation of the item are provided in listings 5.3 and 5.4, respectively.

Listing 5.1: Cell protocol

```
typestate Cell {
  NoItem = {
    void setItem(Item): OneItem,
    drop: end
  }
  OneItem = {
    Item getItem(): OneItem,
    void setItem(Item): OneItem,
    drop: end
  }
}
```

Listing 5.2: Cell class

```
public class Cell {
  private Item item = null;
  public Item getItem() {
    return this.item;
  }
  public void setItem(Item item) {
    this.item = item;
  }
}
```

The cell can hold at most one item, which means the cell's protocol has only two states: one in which there is no item stored, and other in which there is one item stored. There are two methods that operate in this cell: *getItem* and *setItem*. The *getItem* can only be called if the cell has one item and it returns the currently stored item. The *setItem* method is used to store an item on the cell and can be called in both states, either to store a item for the first time or to overwrite a previously stored item.

Listing 5.3: Item protocol

```
typestate Item {
  S0 = {
    void changeState(): S1,
    drop: end
  }
  S1 = {
    void changeState(): S1,
    drop: end
  }
}
```

Listing 5.4: Item class

```
public class Item {
  private int state = 0;
  public int getState() {
    return this.state;
  }
  public void changeState() {
    this.state = 1;
  }
}
```

The item has two states, *S0* and *S1*, and one method, *changeState*. Calling the method in the *S0* state, changes the state of the item to *S1*. Calling the method in *S1*, keeps the item in the same state.

Listing 5.5: Main 1

```
1  Cell c = new Cell();
2  c.setItem(new Item());
3
4  Item item = c.getItem();
5
6  item.changeState();
7
8  Item item2 = c.getItem();
9
10 item2.changeState();
```

Consider the first scenario (listing 5.5) where a cell is created (line 1), an item is stored (line 2), then borrowed via the *getItem* method (line 4), and used by the caller (line 6). With linearity enforced, the item would be "moved" from the cell to the caller of the *getItem* method, and the cell would lose the ownership of the item. We would like to be able to temporarily borrow the item, use it, and then "return" it to the cell, without having to create an explicit method for that, allowing it to be borrowed again later.

Listing 5.6: Main 2

```
1  Cell c = new Cell();
2  c.setItem(new Item());
3
4  Item item = c.getItem();
5
6  Thread t = new Thread(() -> {
7    item.changeState();
8  });
9
10 t.start();
11 t.join();
12
13 Item item2 = c.getItem();
14 item2.changeState();
```

The second scenario (listing 5.6) is similar to the previous one except that the item held by the cell is used in a separate thread (line 7). In this instance, we would like that the item be made available again to the cell after the thread finishes its execution (line 11).

Listing 5.7: Main 3

```
1  Cell c = new Cell();
2  c.setItem(new Item());
3
4  Item item = c.getItem();
5
6  Thread t = new Thread(() -> {
7    item.changeState();
8  });
9
10 t.start();
11
12 Item item2 = c.getItem();
13 // Error should be reported here
14 item2.changeState();
15
16 t.join();
```

In the third scenario (listing 5.7), we expect the type-checker to report an error because we are trying to borrow the item a second time from the cell and use it without waiting for the thread to finish (line 14). If we do not call the *join* method on the thread first, there is no guarantee that no data-race will occur.

Although these scenarios are simple in nature, they show a common pattern that exists in imperative programming languages: sharing of objects. The sharing may occur in sequential and concurrent contexts, in single-threaded and multi-threaded contexts. Forcing the linear use of objects makes it easier to statically keep track of the state of the

objects, but prevents common uses cases, as we seen.

Although borrowing is a concept already existing in languages like Rust [72], this concept alone is not enough for a typestate-oriented language. Since the types evolve, we need to ensure that borrowed objects are still used in a way that respects their protocol and does not break the assumptions the owners made.

Access permissions enable the sharing of objects in a controlled way while still allowing for the use of objects to be checked statically. In the next chapter, we will explore tools and languages that either support access permissions directly or that allow specifications to indicate which memory locations are required and modified.

Chapter 6

Practical work on Access Permissions

In this chapter, we will explore tools and languages that either support access permissions directly or that allow specifications to indicate which memory locations are required and modified. The languages and tools analyzed are Spec#, Chalice, Dafny, VeriFast, and Plaid. To study these, we will be using the motivating example presented in the previous chapter and test how these verify it in the three scenarios. Table 6.1 summarizes the results obtained from analyzing each language and tool.

	Spec#	Chalice	Dafny	VeriFast	Plaid
Specifications	Assertions + Ownership discipline	Assertions	Assertions	Assertions	Permission + State annotations
Resource management	Modifies clause	Access permissions	Modifies clause	Separation Logic	Access permissions
Typestate-oriented	×	×	×	×	✓
Concurrency support	×	✓	×	✓	✓
Actively maintained	×	×	✓	✓	×

Table 6.1: Comparison of languages and tools

6.1 Spec#

The Spec# language is a superset of C#, an object-oriented language targeted for the *.NET Platform*. Spec# adds support for distinguishing non-null object references from

possibly-null object references, method pre- and postconditions, a discipline for managing exceptions, and support for constraining the data fields of objects. Spec# includes an automatic program verifier, called Boogie [5], which checks specifications statically [4, 59].

One of the key features of Spec# is the introduction of concepts like *aggregate objects*, *peers*, and *owners*, to enforce an ownership discipline, which controls the use of objects referenced by other objects in its fields, allowing invariants between them to be verified. Unfortunately, Spec# does not check for concurrency errors such as data races and deadlocks [59]. In the following listings, we present an implementation of the motivating example in Spec#.

Listing 6.1: Spec#: Item class

```
1  class Item {
2    int state;
3
4    public Item()
5      ensures state == 0;
6    { state = 0; }
7
8    public void ChangeState()
9      modifies state; ensures state == 1;
10   { state = 1; }
11 }
```

The *Item* class has a *state* field that may hold an integer value: 0 or 1. The initial *state* value is 0 and may be changed to 1 by calling the *ChangeState* method. Notice how in the constructor (line 5) and in the *ChangeState* method (line 9), there are *ensures* clauses that specify what is the value of the *state* field after they are ran. Additionally, the *ChangeState* method includes a *modifies* clause (line 9) that indicates that the method may change the value of the *state* field.

Listing 6.2: Spec#: Cell class

```
1  using Microsoft.Contracts;
2
3  class Cell {
4    [Peer] public Item? item = null;
5
6    public Item GetItem()
7      requires item != null; ensures item == result;
8    { return item; }
9
10   public void SetItem([Captured] Item i)
11     modifies item; ensures i == item;
12   { item = i; }
13 }
```

The *Cell* class has an *item* field which may hold a reference to an item or may be *null*.

The field must be *public* otherwise Spec# will not allow the exposure of the item via the *GetItem* method. After initialization, the *item* field is *null*. There is a *GetItem* method (line 6), which can only be called if *item* is not *null*, and which returns the current stored item. There is also a *SetItem* method (line 10) which stores an item in the cell, by modifying the *item* field, like specified in the *modifies* clause. Notice the use of two annotations, *Peer* (line 4) and *Captured* (line 10).

The *Peer* annotation is used when the field references an object that can also be accessed by clients of the enclosing class. For example, in a typical collection-iterator pattern, the iterator has a field that references the collection. This field would be marked as *Peer* since clients of the iterator may also access the collection directly [59]. In this example, this annotation will allow clients of the cell to use the item directly.

The *Captured* annotation indicates that the *SetItem* method has the right to take ownership of the object referenced by the parameter, and that a caller should not expect to be able to directly use the object after the call [59] (expect later, by borrowing the item via the *GetItem* method).

Listing 6.3: Spec#: Main

```
1  class Main {
2    void Run() {
3      Cell c = new Cell();
4      c.SetItem(new Item());
5      Item item = c.GetItem();
6      item.ChangeState();
7      Item item2 = c.GetItem();
8      item2.ChangeState();
9    }
10 }
```

In the *Run* method of the *Main* class, a cell is created (line 3), an item is created and stored in it (line 4), and then borrowed (line 5). After borrowing the item, the *ChangeState* method is called on the item (line 6). Following that, the item may be borrowed again (line 7). The code is accepted by Spec#.

Unfortunately, since Spec# does not check for concurrency errors [59], there will be no guarantee of the absence of data races in the scenario where the state of the item is changed in a separate thread.

6.2 Chalice

Chalice is an imperative language and program verifier for reasoning about concurrent programs [60, 58]. Like Spec#, Chalice is also built on Boogie [5].

It supports thread creation, locking and channels. To allow for code verification, programmers indicate the assumptions about their code via annotations, which the verifier then analyzes to check that they are never violated. These annotations can indicate that

a parameter must not be null, a memory location can safely be accessed by a thread, or that a memory location is protected by a certain lock [60].

The key feature of the Chalice language is that it centers around access permissions. To require permission to write on a field *f* of an object *o*, one must use *acc(o.f)* in the *requires* clause of a method. To require only read access, one can use *rd(o.f)* [60]. Chalice does not use *modifies* clauses to know what locations can be modified by a method call instead, it deduces those locations by looking at the permissions that the method requires. If a method does not demand access to a memory location, then that method cannot modify that location [60]. Note that the annotations *acc* and *rd* can only refer to fields. If one requires full permission to all the fields of an object, one can use *acc(o.*)*. In the following listings, we present an implementation of the motivating example in Chalice.

Listing 6.4: Chalice: Item class

```
1  class Item {
2    var state: int;
3
4    method ChangeState()
5      requires acc(state)
6      ensures acc(state) && state == 1
7    { state := 1; }
8  }
```

The *Item* class has a *state* field that may hold an integer value: 0 or 1. The initial *state* value is specified when initializing the object. That value may be changed to 1 by calling the *ChangeState* method. Notice how in the *ChangeState* method (line 4), there is a *requires* clause (line 5) that requests full permission to write in the *state* field, and there is a *ensures* clause (line 6) that specifies that the permission is returned to the caller and that the value of the *state* field after the method call is 1.

Listing 6.5: Chalice: Cell class

```
1  class Cell {
2    var item: Item;
3
4    method GetItem() returns (res: Item)
5      requires rd(item) && item != null
6      ensures rd(item) && res == item && res != null
7    { res := item; }
8
9    method SetItem(i: Item)
10     requires acc(item) && i != null
11     ensures acc(item) && i == item && item != null
12   { item := i; }
13 }
```

The *Cell* class has an *item* field which may hold a reference to an item or may be *null*. There is a *GetItem* method (line 4), which can only be called if *item* is not *null*, and which

returns the current stored item. Notice how it requires read access to the *item* field, and then returns that permission to the caller (lines 5 and 6). There is also a *SetItem* method (line 9) which stores an item in the cell, by modifying the *item* field. To do that, the method requires full permission to the *item* field, and then returns that permission to the caller (lines 10 and 11).

Listing 6.6: Chalice: Main 1

```
1  class Main {
2    method Main() {
3      var c := new Cell { item := null };
4      var i := new Item { state := 0 };
5      call c.SetItem(i);
6      call item := c.GetItem();
7      call item.ChangeState();
8      call item2 := c.GetItem();
9      call item2.ChangeState();
10   }
11 }
```

In the first scenario (listing 6.6), in the *Main* method, a cell is initialized with its *item* field having a *null* value (line 3), an item is initialized with its *state* field having a 0 integer value (line 4), and the item is stored in the cell (line 5). After that, the item is borrowed from the cell (line 6), its state is changed (line 7), and then the item is borrowed again (line 8). The code is accepted by Chalice.

Listing 6.7: Chalice: Main 2

```
1  class Main {
2    method Main() {
3      var c := new Cell { item := null };
4      var i := new Item { state := 0 };
5      call c.SetItem(i);
6      call item := c.GetItem();
7      fork tk := item.ChangeState();
8      join tk;
9      call item2 := c.GetItem();
10     call item2.ChangeState();
11   }
12 }
```

The second scenario (listing 6.7) is similar to the previous one, except the state of the item is changed in a separate thread (line 7), and before trying to use the item again, we wait for the thread to finish (line 8). The code is also accepted by Chalice, as expected.

Listing 6.8: Chalice: Main 3

```
1  class Main {
2    method Main() {
3      var c := new Cell { item := null };
4      var i := new Item { state := 0 };
5      call c.SetItem(i);
6      call item := c.GetItem();
7      fork tk := item.ChangeState();
8      call item2 := c.GetItem();
9      // The precondition at 5.14 might not hold. Insufficient fraction at 5.14 for Item.
           ↪ state.
10     call item2.ChangeState();
11     join tk;
12   }
13 }
```

In the third scenario (listing 6.8), there is an attempt to use the item before ensuring that the thread has finished (line 10). In this instance, Chalice reports an error, which is correct. The *ChangeState* method requires full permission to the *state* field of the item, but that permission was transferred to the thread, and can only be obtained again after the *join* statement, which waits for the thread to finish.

6.3 Dafny

Dafny is an imperative and sequential language, with support for generic classes, dynamic allocation, inductive datatypes, and specification constructs, including pre- and postconditions, frame specifications (read and write sets), and termination metrics [56, 20].

Its static verifier, powered by Boogie [5] and Z3 [22], is then used to verify the functional correctness of programs, being able to prove that there are no runtime errors, such as index out of bounds, null dereferences and division by zero, and that the code terminates, except in specially designated loops [65]. Although Dafny does not seem to include concurrency support, it has been used to model concurrency [57], to verify concurrent programs [61], and as a subject of future extension with concurrency support [26]. In the following listings, we present an implementation of the motivating example in Dafny.

Listing 6.9: Dafny: Item class

```
1  class Item {
2    var state: int;
3
4    constructor()
5      ensures state == 0
6    { state := 0; }
7
8    method changeState()
9      modifies `state
10     ensures state == 1
11   { state := 1; }
12 }
```

The *Item* class has a *state* field that may hold an integer value: 0 or 1. The initial *state* value is 0 and may be changed to 1 by calling the *changeState* method. Like in Spec#, in the constructor (line 5) and in the *changeState* method (line 10), there are *ensures* clauses that specify what is the value of the *state* field after they are ran. Also like in Spec#, the *changeState* method includes a *modifies* clause (line 9) that indicates that the method may change the value of the *state* field. The only difference is syntactical: instead of writing *modifies state*, one writes *modifies 'state*, with a backtick, to indicate that the *state* field may be changed. In Dafny, *modifies state* would indicate that the object pointed by *state* may be changed, not the field itself, which in this case does not make sense, since *state* holds an integer, not an object reference.

Listing 6.10: Dafny: Cell class

```
1  class Cell {
2    var item: Item?;
3
4    constructor()
5      ensures item == null
6    { item := null; }
7
8    method getItem() returns (res: Item)
9      requires item != null
10     ensures res == item
11   { return item; }
12
13   method setItem(i: Item)
14     modifies `item
15     ensures item == i
16   { item := i; }
17 }
```

The *Cell* class implementation is very similar to the Spec# one. It has an *item* field which may hold a reference to an item or may be *null*. After initialization, the *item* field is *null*. There is a *getItem* method (line 8), which can only be called if *item* is not *null*,

and which returns the current stored item. There is also a *setItem* method (line 13) which stores an item in the cell, by modifying the *item* field, like specified in the *modifies* clause.

Listing 6.11: Dafny: Main

```
1 method Main() {
2   var c := new Cell();
3   var i := new Item();
4   c.setItem(i);
5   var item := c.getItem();
6   item.changeState();
7   var item2 := c.getItem();
8   item2.changeState();
9 }
```

In the *Main* method, a cell is created (line 2), an item is created (line 3), and stored in the cell (line 4). After that, the item is borrowed (line 5), and its state is changed by the *changeState* method (line 6). Following that, the item may be borrowed again (line 7). The code is accepted by Dafny.

Unfortunately, since Dafny does not have built-in support for threads, we cannot test the scenarios where the state of the item is changed in a separate thread.

6.4 VeriFast

VeriFast is a verification tool, based on separation logic [67], for single-threaded and multithreaded C and Java programs [46]. The main features of VeriFast include: checking that a method satisfies its corresponding method contract, which can specify the structure of the heap and provide an upper bound on the set of modifiable memory locations via permissions; support for predicates, inductive data types, functions and lemmas [76].

In the following listings, we present an implementation of the motivating example in Java with VeriFast specifications.

Listing 6.12: VeriFast: Item class

```
1  //@ predicate Item(Item i; int state) = i.state |-> state;
2
3  public class Item {
4    private int state = 0;
5
6    public void changeState()
7      //@ requires Item(this, _);
8      //@ ensures Item(this, 1);
9    { state = 1; }
10 }
```

The *Item* class has a *state* field that may hold an integer value: 0 or 1. The initial *state* value is 0 and may be changed to 1 by calling the *changeState* method. Notice the use of the *Item* predicate to reason about the current state of the item.

Listing 6.13: VeriFast: Cell class

```
1  //@ predicate Cell(Cell c; Item item) = c.item |-> item;
2
3  public class Cell {
4    private Item item;
5
6    public Cell()
7      //@ requires true;
8      //@ ensures Cell(this, null);
9    { item = null; }
10
11   public Item getItem()
12     //@ requires Cell(this, ?i) &*& i != null;
13     //@ ensures Cell(this, i) &*& result == i;
14   { return item; }
15
16   public void setItem(Item i)
17     //@ requires Cell(this, _) &*& i != null;
18     //@ ensures Cell(this, i);
19   { item = i; }
20 }
```

The *Cell* class has an *item* field which may hold a reference to an item or may be *null*. The initial *item* field value is *null* (line 9). There is a *getItem* method (line 11), which can only be called if *item* is not *null*, and which returns the current stored item. There is also a *setItem* method (line 16) which stores an item in the cell, by modifying the *item* field. Notice the use of the *Cell* predicate to reason about the state of the cell, before and after each method call.

Listing 6.14: VeriFast: Main 1

```
1  public class Main {
2    public static void main(String[] args)
3      //@ requires true;
4      //@ ensures true;
5    {
6      Cell c = new Cell();
7      c.setItem(new Item());
8
9      Item item = c.getItem();
10     item.changeState();
11
12     Item item2 = c.getItem();
13     item2.changeState();
14   }
15 }
```

In the first scenario (listing 6.14), in the *Main* class, a cell is created and an item is stored inside of it (lines 6 and 7). After that, the item is borrowed from the cell (line 9),

its state is changed (line 10), and then the item is borrowed again (line 12). The code is accepted by VeriFast.

Listing 6.15: VeriFast: Runnable class

```
1  class StateChanger implements Runnable {
2
3    private Item item;
4
5    //@ predicate pre() = this.item |-> ?i &*& i != null &*& Item(i, _);
6    //@ predicate post() = this.item |-> ?i &*& i != null &*& Item(i, _);
7
8    public StateChanger(Item i)
9      //@ requires i != null &*& Item(i, _);
10     //@ ensures pre();
11   {
12     this.item = i;
13   }
14
15   public void run()
16     //@ requires pre();
17     //@ ensures post();
18   {
19     item.changeState();
20   }
21 }
```

For the following scenarios, which make use of threads, there was the need for the declaration of a *StateChanger* class, which implements the *Runnable* interface, because VeriFast does not parse lambda expressions.

Listing 6.16: VeriFast: Main 2

```
1  public class Main {
2    public static void main(String[] args) {
3      Cell c = new Cell();
4      c.setItem(new Item());
5
6      Item item = c.getItem();
7
8      Thread t = new Thread(new StateChanger(item));
9      t.start();
10     t.join();
11
12     Item item2 = c.getItem();
13     item2.changeState();
14   }
15 }
```

The second scenario (listing 6.16) is similar to the previous one, except the state of the item is changed in a separate thread, and before trying to use the item again, we wait for

the thread to finish. Unfortunately, the code is not accepted by VeriFast, reporting that the *changeState* call (line 13) cannot be performed. The reason seems to be that it has no knowledge that the *heap chunk* for the item, available in the post-condition, corresponds to the same item required in the pre-condition. Because the *Runnable* interface requires the specification of two predicates, *pre* and *post*, which are independent from one and another, there seems to be no way of informing the verifier that the item is still the same.

Listing 6.17: VeriFast: Main 3

```
1  public class Main {
2    public static void main(String[] args) {
3      Cell c = new Cell();
4      c.setItem(new Item());
5
6      Item item = c.getItem();
7
8      Thread t = new Thread(new StateChanger(item));
9      t.start();
10
11     Item item2 = c.getItem();
12     item2.changeState();
13
14     t.join();
15   }
16 }
```

The third scenario (listing 6.17) presents a data-race example, where there is an attempt to call *changeState* on the item without waiting for the thread that borrowed it to finish first. In this instance, VeriFast reports that there is no *heap chunk* available to allow for the *changeState* call outside the thread.

6.5 Plaid

Plaid [77, 37] is a typestate-oriented programming language designed for concurrency. It includes the concept of access permissions [64, 7], which are associated with each type to express the aliasing and the mutability of the corresponding object's typestate [74]. Furthermore, access permissions information is used to automatically parallelize code [38].

Unfortunately, since Plaid does not seem to be maintained any longer, we could not reproduce the motivating example. If we could, the example would look something like what is presented in the following listing (according to what we could gather from [77] and [2]).

Listing 6.18: Plaid example

```
state Item {
  val int state = 0;
  method void changeState() {
    this.state = 1;
  }
}

state Cell {
  val shared Item item = null;
}

state Cell0 case of Cell {
  method void setItem(unique Item i)
    [unique Cell0 >> unique Cell1]
  {
    this.item = i;
    this <- Cell1;
  }
}

state Cell1 case of Cell {
  method shared Item getItem() {
    return this.item;
  }
}

method void main() {
  val unique cell = new Cell;
  val unique i = new Item;
  cell.setItem(i);

  val item1 = cell.getItem();
  item1.changeState();

  val item2 = cell.getItem();
  item2.changeState();
}
```

6.6 Summary

There are many approaches that can be used to specify program behavior and reason about protocols. Languages like Spec#, Chalice, Dafny, and VeriFast, are very rich, allowing one to express with precision what is the expected behavior of a program. Nonetheless, they have the drawback of potentially being more difficult to use due to their complexity and due to the undecidability of specifications in the general case. Spec# targets the *.NET Platform* but forces programs to follow an ownership discipline, which may impose

some overhead. Chalice is the most interesting among the four languages because it has built-in support for threads and includes the concept of access permissions, from which we expect to get some inspiration from. Dafny does not support threads but is actively maintained. VeriFast is available for C and Java, but the specifications are usually very verbose. Plaid is distinguishable from the other languages since it does not require complex specifications and has built-in support for typestate-oriented features and access permissions. Unfortunately, it does not seem to be maintained.

In conclusion, the current technology is, in general, able to verify the motivating example. Unfortunately, it usually requires many annotations and expertise on the programmer's part.

Chapter 7

Typestate-oriented tool: version 2

In this chapter, we are going to present a language of assertions that focuses on allowing a program that uses typestates to be type-checked even in the presence of aliasing. The chapter starts by presenting the language of assertions (section 7.1). Following that, we present the inference algorithm, which removes the need for assertions to be explicitly written (section 7.2). Then, we compare our language of assertions and tool with other languages and tools (section 7.3). In the last section, we test our tool with some examples (section 7.4).

7.1 Language of assertions

In the following sections, we start by presenting the components of our language and their meaning (section 7.1.1). Then we discuss the guarantees that the assertions provide and the well-formedness properties (sections 7.1.2 and 7.1.3). Following that, the concepts of packing and unpacking are expanded upon as well as transferring of permissions (sections 7.1.4 and 7.1.5). Immediately after, it is explained how protocol completion is ensured (section 7.1.6). Next, it is explained how to check if an assertion is implied by another (section 7.1.7), how to compute the upper bound between two assertions (section 7.1.8), and how nullable values and union types are handled (section 7.1.9).

7.1.1 Introduction

Each assertion in the language is composed of *access*, *equality*, *typeof*, *packed* and *unpacked* predicates.

Access predicates specify the fractional permissions [10] for *access locations*. A fractional permission is represented with a fractional number between zero and one. If the fraction is equal to zero, it means there is no access to that location, so no reads and no

writes are allowed. If the fraction is equal to one, there is full access to that location, so reads and writes are both allowed. If the fraction is a value between zero and one, only reads are allowed.

An *access location* may represent a local variable or a field of an object. Conceptually, we can think of a local variable as a field of the closure of a function call. Because of that, we are not going to distinguish between the stack and the heap, and we will be thinking only in terms of memory locations. These *access locations* may also refer to the object pointed by a variable or field. Even though an object is the composition of its fields, we will need to be able to reason about the access permission to an object itself, separately from its fields. The need for it will be explained later.

Equality, *typeof*, *packed* and *unpacked* predicates, talk about *locations*. A *location* represents a local variable or a field of an object.

Equalities assert that two *locations* point to the same object (they hold the same memory address number). Tracking equalities is useful so that we can transfer permissions between two locations, for example, to restore full access to an object that was aliased between two locations.

Typeof predicates indicate the current type of an object pointed by a variable or field.

Packed and *unpacked* predicates specify that an object is packed or unpacked, respectively, via the associated *location*. When an object is packed, the concrete types of its fields are hidden behind the abstract typestate view. When an object is unpacked, the concrete types of its fields are exposed.

For illustrative purposes, the syntax used to represent the assertions is formalized by the following grammar.

Listing 7.1: Assertions' grammar

```
1  Assertion := Term | Term "∧" Assertion
2  Term := Access | Equality | TypeOf | Packed | Unpacked
3
4  Access := "access" "(" AccessLocation "," f ")"
5  Equality := "eq" "(" Location "," Location ")"
6  TypeOf := "typeof" "(" Location "," t ")"
7  Packed := "packed" "(" Location ")"
8  Unpacked := "unpacked" "(" Location ")"
9
10 Location := id | id "." Location
11 AccessLocation := id | id "." "0" | id "." AccessLocation
```

In the grammar, f is a meta-variable ranging over all rational numbers between zero and one; id is a meta-variable ranging over values of the set of all the valid Java identifiers; t is a meta-variable ranging over values of the set of all the types in the type system.

Assertions are represented as conjunctions of the predicates mentioned. *Access* predicates are represented as *access(x,f)*, where x is an *access location* and f a fractional value.

An *access location* is represented as a sequence of Java identifiers separated by dots, potentially terminated with .0 if the *access location* refers to the object itself. *Equalities* are represented as *eq(x,y)*, where *x* and *y* are *locations*. A *location* is also represented as a sequence of Java identifiers separated by dots, except it never terminates with .0. *Typeof* predicates are represented as *typeof(x,T)*, where *x* is a *location* and *T* a textual representation of a type from our type system. *Packed* and *unpacked* predicates are represented as *packed(x)* and *unpacked(x)* respectively.

In the following sections, code may also be presented to show how the ideas are implemented in practice. In the implementation, assertions are represented by the structure presented in the following listing.

Listing 7.2: Assertions' type structure

```
type assertion = {
  accesses: access list;
  equalities: equality list;
  typeOfs: typeOf list;
  packs: packed list;
  unpacks: unpacked list;
} and

access = Access of accessLoc * fraction and
equality = Equality of loc * loc and
typeOf = TypeOf of loc * ttype and
packed = Packed of loc and
unpacked = Unpacked of loc and

fraction = int * int and
loc = string list and
accessLoc = string list * bool
```

Access predicates are represented as a pair with an *access location* and a fractional value. *Equalities* are pairs of two *locations*. The *typeof* predicate is a pair with a *location* and a type from our type system. *Packed* and *unpacked* predicates are composed of a *location*. A *fraction* is represented as a pair of integers. A *location* is represented with a non-empty list of valid Java identifiers. An *access location* is represented with a tuple with a non-empty list of valid Java identifiers and a boolean value. If the boolean value is *false*, the *access location* refers to a memory location, like a variable or a field. If the boolean value is *true*, the *access location* refers to the object pointed by the variable or field.

An example of an assertion in textual form and the corresponding data structure that represents it, in the implementation, are presented in the following listings.

Listing 7.3: Assertion example produced by the grammar in listing 7.1

```
access(x,1) ∧ access(y,1) ∧
access(x.0,0) ∧ access(y.0,1) ∧
eq(x,y) ∧ typeof(y,State "HasNext") ∧ packed(y)
```

Listing 7.4: Assertion example according to the type structure in listing 7.2

```
{
  accesses = [
    Access((["x"], false), (1, 1));
    Access((["y"], false), (1, 1));
    Access((["x"], true), (0, 1));
    Access((["y"], true), (1, 1))
  ];
  equalities = [Equality(["x"], ["y"])];
  typeOfs = [TypeOf(["y"], State "HasNext")];
  packs = [Packed ["y"]];
  unpacks = [];
}
```

In this example, there is full access to read or write into x and y, no access to the object via x, full access to an object via y, y is packed and in state *HasNext*, and x and y are aliases. An example of Java code where this assertion would hold true, is presented in the following listing.

Listing 7.5: Example of Java code where an assertion holds

```
1 Iterator x = collection.iterator();
2 Iterator y = x;
3 // access(x,1) ∧ access(y,1) ∧
4 // access(x.0,0) ∧ access(y.0,1) ∧
5 // eq(x,y) ∧ typeof(y,State "HasNext") ∧ packed(y)
```

Details regarding the language of assertions, and how it works in practice, will be explained in the following sections.

7.1.2 Assertions' guarantees

This language of assertions guarantees that:

1. The usage of objects will follow the corresponding protocol, even in the presence of aliasing;

2. No data-races occur at the level of variables and fields;

3. Method calls that change the state of an object do not interfere with each other.

The following listings present examples of code that contains bad behavior that would be detected by our type system.

Listing 7.6: Wrong method order example

```
1 Iterator it = collection.iterator();
2 it.next(); // Error: Cannot call "next" in HasNext state
```

In the first example, the method *next* is being called in a newly created iterator without ensuring that there are items to be read. The *hasNext* should have been called first.

Listing 7.7: Data-race example

```
1  Iterator it = collection.iterator();
2
3  new Thread(() -> {
4    while(it.hasNext())
5      it.next();
6  }).start();
7
8  new Thread(() -> {
9    while(it.hasNext())
10     it.next();
11 }).start();
```

In the second example, the same iterator object is being manipulated by two different threads, which are both trying to read all the items of the iterator. This is an example of bad behavior since, for example, by the time one of the threads attempts to call *next*, the other thread might have read all the items already.

7.1.3 Assertions' well-formedness

An assertion, to be well-formed, must respect the following properties. Assume that the fractional values *f1* and *f2* are greater than zero ($f1 > 0 \land f2 > 0$).

1. An *access* predicate that refers to an object pointed by a variable or field can only hold if read access to that variable or field is available. For example, *access(x.0,f1)* holds if *access(x,f2)* holds, and *access(x.y.0,f1)* holds if *access(x.y,f2)* holds;

2. An *access* predicate that refers to a field of an object can only hold if read access to the variable or field that holds that object is available. For example, *access(x.y,f1)* holds if *access(x,f2)* holds;

3. An equality predicate, such as *eq(x,y)*, can only hold if *access(x,f1)* $\land$ *access(y,f2)* holds. This means that we need at least read access to the variables/fields x and y, ensuring they do not get overwritten;

4. A *typeof* predicate with *Unknown* always holds;

5. A *typeof* predicate with a type that is subtype of *Null* | *Primitive*, such as *typeof(x,Null)*, can only hold if *access(x,f1)* holds. This means that we need at least read access to the variable/field x, ensuring it does not get overwritten;

6. A *typeof* predicate with a type that is not a subtype of *Null* | *Primitive* (and it is not *Unknown*), such as *typeof(x,State "HasNext")*, can only hold if *access(x,f1)* $\land$ *access(x.0,f2)* holds. This means that we need at least read access to the variable/field

x, ensuring it does not get overwritten, and at least read access to the object pointed by x, ensuring its state does not change;

7. Predicates such as *packed(x)* or *unpacked(x)* can only hold if *access(x,f1)* $\wedge$ *access(x.0,f2)* holds;

8. An object cannot be packed and unpacked at the same time via the same variable or field;

9. If *packed(x)* is true, no predicates over the object's fields via x should exist in the assertion;

10. The sum of all fractional permissions that allow access to the same location, must be a value less or equal than one. For example, if x and y are the only variables that point to the same object, and *access(x.0,k1)* $\wedge$ *access(y.0,k2)* holds, then the sum of *k1* and *k2* must be a number less or equal to one.

A function that checks if an assertion is well-formed, may be defined by the following ML code. The code for property 10 is omitted for brevity.

Listing 7.8: Checking assertions' well-formedness (Part 1)

```
let isValid (a:assertion) : bool =
  (* Properties 1 and 2 *)
  for_all (fun (Access((x, dotZero), (f1, f2))) ->
    f1 = 0 ||
    if dotZero
      then hasAccess a x
      else hasAccess a (parent x)
  ) a.accesses &&
  (* Property 3 *)
  for_all (fun (Equality(x, y)) ->
    hasAccess a x &&
    hasAccess a y
  ) a.equalities &&
  (* Properties 4, 5 and 6 *)
  for_all (fun (TypeOf(x, t)) ->
    isUnknown t || (
    hasAccess a x &&
    (isNullOrPrimitive t || hasAccessDotZero a x))
  ) a.typeOfs &&
  (* Property 7 and 8 *)
  for_all (fun (Packed x) ->
    hasAccess a x &&
    hasAccessDotZero a x &&
    not (isUnpacked a x)
  ) a.packs &&
```

Listing 7.9: Checking assertions' well-formedness (Part 2)

```
25  (* Property 7 and 8 *)
26  for_all (fun (Unpacked x) ->
27    hasAccess a x &&
28    hasAccessDotZero a x &&
29    not (isPacked a x)
30  ) a.unpacks &&
31  (* Property 9 *)
32  for_all (fun (Packed x) ->
33    not (hasPredicatesOverFields a x)
34  ) a.packs
```

Each *for_all* call checks if the predicates follow the properties. The *parent* function, given a *location* with more than one identifier, returns a new *location* without the last identifier, and given a *location* with only one identifier, the function just returns it. The *hasAccess* function checks if the assertion contains an *access* predicate such as *access(x,f)*, where $f > 0$. The *hasAccessDotZero* function checks if the assertion contains an *access* predicate such as *access(x.0,f)*, where $f > 0$. The *isPacked* and *isUnpacked* functions check if in the assertion the locations are packed or unpacked, respectively. The *hasPredicatesOverFields* function checks if the assertion contains predicates over *locations* prefixed with *x*, like *access(x.y,f)* or *typeof(x.z,T)*. The *isNullOrPrimitive* function returns *true* if the given type is a subtype of *Null* | *Primitive*.

Note that in the implementation, if an *access* predicate for a certain *access location* is absent, it is assumed that the associated fractional permission is zero, if a *typeof* predicate for a certain *location* is absent, it is assumed that the associated type is *Unknown*, and if a *packed* predicate for a certain *location* is absent, it is assumed that the associated object is unpacked.

7.1.4 Packing and Unpacking

When an object is packed, the concrete types of its fields are hidden behind the abstract typestate view. When an object is unpacked, the concrete types of its fields are exposed. In our system, methods can only be called on packed objects, to ensure that the invariants hold before the execution of methods. But since accessing objects stored in fields of other objects is also a use case, it is important to be able to perform packing and unpacking on objects, coercing between their abstract typestate views and their concrete views.

The concepts of packing and unpacking applied to typestates were probably introduced in [25], and have been used in other approaches, for example in [7] and [62]. An earlier appearance of these concepts, although not applied to typestates, is [87]. Originally, unpacking was not allowed for aliased objects [25]. Later Bierhoff and Aldrich [7] refined the notion of unpacking to instead pack and unpack specific permissions, which allows unpacking of shared objects. Our approach also allows for aliased objects to be unpacked and packed in a consistent way.

To allow packing or unpacking of an aliased object, we need to "remember" the permission to the object we unpacked or packed. This is the reason why access predicates such as *access(x.0,f)*, are needed. Although 0 is not a valid field name, we use this representation to refer to the access permission for an object itself, without referring to its fields.

To illustrate how packing and unpacking work in practice, imagine an object with field y, which stores a reference to another object for which it has read access. Suppose for example that for any state of the object, the following invariant holds.

$$access(y, 1) \wedge access(y.0, 1/4) \wedge packed(y)$$

Now assume that there is a reference for that object in variable x, such that the following assertion holds. Notice that there is only read permission available for that object via x, which means that this object may be aliased.

$$access(x.0, 1/2) \wedge packed(x)$$

To unpack the object pointed by x, one just needs to multiply the access fractions in the invariant (1 and 1/4) by the object's access fraction (1/2), combine the modified invariant with the previous assertion, and replace *packed(x)* with *unpacked(x)*. This process results in the following assertion.

$$access(x.0, 1/2) \wedge unpacked(x) \wedge$$
$$access(x.y, 1/2) \wedge access(x.y.0, 1/8) \wedge packed(x.y)$$

Packing is only allowed if all the truths about the fields of the object are consistent with the invariant. In the previous assertion, x could be packed again, since $1 \times 1/2 = 1/2$, $1/4 \times 1/2 = 1/8$ and the object referenced by the field y is packed, like required by the invariant. Given that packing is allowed, one just needs to remove the predicates that refer to fields of the object and replace *unpacked(x)* with *packed(x)*.

But if instead of the previous assertion, we had the following one, we would not be able to pack x again since $1/4 \times 1/2 \neq 0$.

$$access(x.0, 1/2) \wedge unpacked(x) \wedge$$
$$access(x.y, 1/2) \wedge access(x.y.0, 0)$$

The following listings present the *pack* and *unpack* operations and their helper functions implemented in ML.

Listing 7.10: *revealedInv* function

```
let revealedInv (t:ttype) (f:fraction) (x:loc) (inv:ttype->assertion) : assertion =
  let assertion = inv t in
  {
    accesses = map (fun
      (Access((y, zero), k)) -> Access((x @ y, zero), multFrac k f)
    ) assertion.accesses;
    equalities = map (fun
      (Equality(a, b)) -> Equality(x @ a, x @ b)
    ) assertion.equalities;
    typeOfs = map (fun
      (TypeOf(y, t)) -> TypeOf(x @ y, t)
    ) assertion.typeOfs;
    packs = map (fun
      (Packed y) -> Packed(x @ y)
    ) assertion.packs;
    unpacks = map (fun
      (Unpacked y) -> Unpacked(x @ y)
    ) assertion.unpacks;
  }
```

The *revealedInv* function, given a type, a fraction, a location and an *inv* function, gets the invariant corresponding to the given type, and returns a new assertion that corresponds to the invariant but modified to be exposed: locations referring to fields are prefixed with the location to the object and *access* predicates have their fractional values multiplied by the fraction initially passed as parameter. The *inv* function, given a type, returns the corresponding invariant in the form of an assertion. This invariant indicates what is true about the fields of the object when in a given typestate and must only include predicates over locations accessible via those fields. Invariants may be provided by the programmer or inferred by executing an algorithm similar to the *class analysis* (section 4.3.3).

Listing 7.11: *unpack* function

```
let unpack (a:assertion) (x:loc) (inv:ttype->assertion) : assertion =
  let Access(_, f) = getAccess a (x, true) in
  let TypeOf(_, t) = getTypeOf a x in
  let revealed = revealedInv t f x inv in {
    accesses = a.accesses @ revealed.accesses;
    equalities = a.equalities @ revealed.equalities;
    typeOfs = a.typeOfs @ revealed.typeOfs;
    packs = (remove (Packed x) a.packs) @ revealed.packs;
    unpacks = a.unpacks @ ((Unpacked x) :: revealed.unpacks);
  }
```

The *unpack* function, given an assertion, a location and an *inv* function, takes the current fractional permission to the object in the given location (using the *getAccess* function), takes the current type of the object (using the *getTypeOf* function), computes the

revealed invariant (with the *revealedInv* function), and returns a new assertion that combines the revealed invariant with the given assertion, with the additional care of replacing the *packed* predicate for an *unpacked* predicate for the location we are unpacking.

Listing 7.12: *pack* function

```
let pack (a:assertion) (x:loc) (inv:ttype->assertion) : assertion =
  let Access(_, f) = getAccess a (x, true) in
  let TypeOf(_, t) = getTypeOf a x in
  let revealed = revealedInv t f x inv in {
    accesses = remove_list revealed.accesses a.accesses;
    equalities = remove_list revealed.equalities a.equalities;
    typeOfs = remove_list revealed.typeOfs a.typeOfs;
    packs = (Packed x) :: remove_list revealed.packs a.packs;
    unpacks = remove_list revealed.unpacks (remove (Unpacked x) a.unpacks);
  }
```

The *pack* function, given an assertion, a location and an *inv* function, takes the current fractional permission to the object in the given location, takes the current type of the object, computes the revealed invariant, and returns a new assertion without the predicates that are common between the revealed invariant and the given assertion, with the additional change that the object becomes packed instead of unpacked.

7.1.5 Permission transfer

Since assertions track equality between memory locations, it is possible to transfer permissions. Making use of aliasing information is an idea also suggested in [10]. For example, imagine a context in which variables x and y are alias to the same object, which means that *eq(x,y)* is true, and that there is 1/2 access to the object via x — *access(x.0,1/2)* — and also 1/2 access to the object via y — *access(y.0,1/2)*. If one wants to perform a mutable operation on the object, it needs full permission to it. Since x and y are known to be alias, the access permission of one can be transferred to the other, for example, from x to y, resulting in assertion *access(x.0,0)* $\wedge$ *access(y.0,1)*, which allows one to perform a mutable operation on the object via the variable y.

If two variables are alias to the same object, it is also possible to assert equalities between their fields, which also allows permissions to be transferred between their fields and objects referenced by those fields.

Additionally, by knowing that two variables point to the same object, it is possible to intersect the types of both, to get a more refined knowledge of the type of the object. For example, given assertion *eq(x,y)* $\wedge$ *typeof(x,State "HasNext"| State "Next")* $\wedge$ *typeof(y,State "Next")*, we can be sure that x is in the *Next* state.

Transferring of permissions can be done as needed. Like in the initial example, a permission was transferred to allow for a mutable operation. Additionally, we can transfer permissions before computing the least upper bound of two assertions to avoid losing "resources" and information (explained later).

7.1.6 Protocol completion

To guarantee that protocols reach the final state in a system with access permissions, we need to ensure that no access permissions (or "resources") are lost unless the object is already in the final state or in a state in which it can be "dropped". The reason for this is that if we were able to arbitrarily "forget" about an alias, this would mean that we could also "forget" about other aliases, and potentially lose all references to an object for which the protocol did not complete. To this end, we analyze each pair of consecutives assertions (i.e. pair where the first implies the second) in the code to check if there was some access permission in the first that was completely lost in the second. If that was the case, and if the object was not in the final state nor in a state in which it could be "dropped", we report an error. We also check the end of methods to ensure objects have their protocol completed, unless we know that the object will be "returned" to the caller, via the post-condition.

7.1.7 Implication

It is important to mention that the pre-condition strengthening rule (figure 7.1) and the post-condition weakening rule (figure 7.2), which apply to Hoare logic [39, 36], also apply to our language of assertions. What needs to be specified is how to know if an assertion implies or is implied by another assertion.

$$\frac{\vdash P \implies P' \qquad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$

Figure 7.1: Pre-condition strengthening

$$\frac{\vdash \{P\}\ C\ \{Q'\} \qquad \vdash Q' \implies Q}{\vdash \{P\}\ C\ \{Q\}}$$

Figure 7.2: Post-condition weakening

As presented before, our language of assertion includes *access*, *equality*, *typeof*, *packed* and *unpacked* predicates.

Packing and unpacking serve the purpose of coercing between the abstract view and the concrete view of objects, to hide or expose the facts about the fields of the objects. Because of that, any assertion where an object is packed, implies an assertion resulting from unpacking the object, and any assertion where an object is unpacked and its fields are consistent with the object's invariant (to allow for packing), implies an assertion resulting from packing the object. In the following paragraph, assume that the objects are all unpacked in the assertions.

An assertion a implies an assertion b if when a holds, b also holds. In other words, a implies b if b is a weaker version of a. To check if a implies b, we need to check for each predicate in b, if there is a predicate in a that implies it. An *access* predicate is implied by another *access* predicate, for the same *access location*, if the first offers an equal or greater access permission than the second. An *equality* predicate that holds in b must also hold in a. A *typeof* predicate is implied by another *typeof* predicate, for the same *location*, if the type in the first predicate is a subtype of the type in the second predicate. The following are some examples of implications that are true or false.

$$access(x,1) \implies access(x,0)$$

The implication is true since full permission is greater than no permission.

$$access(x,0) \not\Longrightarrow access(x,1)$$

The implication is false since we cannot ensure that full permission is available from having no permission.

$$access(x,1) \wedge typeof(x,Null) \implies access(x,1) \wedge typeof(x,Unknown)$$

The implication is true since *Null* is a subtype of *Unknown*.

$$access(x,1) \wedge typeof(x,Object) \not\Longrightarrow access(x,1) \wedge typeof(x,Null)$$

The implication is false since *Object* is not a subtype of *Null*.

$$access(x,0.5) \wedge access(y,0.5) \wedge eq(x,y) \implies access(x,0.5) \wedge access(y,0.5)$$

The implication is true since an assertion without information about an equality is implied by one where that equality holds.

$$access(x,0.5) \wedge access(y,0.5) \not\Longrightarrow access(x,0.5) \wedge access(y,0.5) \wedge eq(x,y)$$

This implication is not true because we cannot assert that an equality is true without having knowledge of that.

The following listing presents a ML implementation of a function that checks if one assertion implies another assertion.

Listing 7.13: *implies* function

```
let implies (a:assertion) (b:assertion) (inv:ttype->assertion) : bool =
  let (a, b) = unpackNecessary a b inv in
  for_all (fun (Access(l, fB)) ->
    let Access(_, fA) = getAccess a l in gteFrac fA fB
  ) b.accesses &&
  for_all (fun (Equality(x,y)) ->
    areEqual a x y
  ) b.equalities &&
  for_all (fun (TypeOf(l, tB)) ->
    let TypeOf(_, tA) = getTypeOf a l in isSubtype tA tB
  ) b.typeOfs
```

The function *implies* starts by ensuring that all the locations that are packed or unpacked in a are also packed or unpacked (respectively) in b (using the *unpackNecessary* function). Following that, the function checks that for each access permission available in b, there is a greater or equal access permission in a (using the *getAccess* and *gteFrac* functions), it checks that for each equality that holds in b, the same equality holds in a (using the *areEqual* function), and that for each *typeof* predicate in b, there is one in a associated with a type that is subtype of the type in the predicate in b (using the *getTypeOf* and *isSubtype* functions).

The implementation uses some helper functions. The *unpackNecessary* function, given two assertions, returns two new assertions where the locations that are packed/unpacked in one are also packed/unpacked in the other (respectively). The *getAccess* function attempts to find an *access* predicate in the given assertion for the given location. If none is found, it is assumed that there is no permission for that *access location*. The *gteFrac* returns *true* if the first fractional permission is greater or equal than the second. The *areEqual* function checks if in the given assertion, the two given locations are known to refer to the same object. The *getTypeOf* function finds an *typeof* predicate in the given assertion for the given location. If none is found, it is assumed the type *Unknown* for that location. The *isSubtype* function returns *true* if the first type is a subtype of the second.

Note that the above implementation is simplified. It does not consider equality transitivity, it does not consider the fact that if two *locations* refer to the same object all its fields are also the same, and it does not perform permission transfer to cover other possible implications.

7.1.8 Assertions' upper bound

It is important to define how to compute the least upper bound of two assertions. For example, after executing an *if-else* statement, we need to know what the assertion will be by combining the assertions of the *if* and *else* branches. When dealing with types, one just needs to compute the union of the types from both branches. When dealing with access permissions, the result is more difficult to define.

Imagine, for example, an *if* branch where an object is passed to a method call, leaving the current context with no permission to access it, and an *else* branch which keeps full access to the object in the current context. In this example, we need to combine two different access predicates: *access(x.0,0)* and *access(x.0,1)*. One solution could be to support disjunctions in assertions, like so: *access(x.0,0)* $\vee$ *access(x.0,1)*. The problem is that the checking would become more complex to perform without much gain, since there would always be the chance that we lost access to the object, which meant we would need to report an error anyway, if the object were to be used. A different solution is to choose the "worst" scenario, in other words, choose the smaller fraction value. This makes sense because in actuality, a non-aliased type is a subtype of an aliased-type, so when merging the assertions, the "upper bound" can be chosen. Unfortunately, this means that "resources" can be lost, which would compromise the assurance that protocols reach completion. To avoid that, we report an error if the fractional values are not the same and the object is neither in the final state or in a state in which it can be "dropped".

Assertions are also composed of equality predicates. When computing the least upper bound between two assertions, one where an equality is true, and another where the equality is false or unknown, we do not preserve the knowledge about the equality.

Finally, it is important to define the case where in one assertion, an object is packed, and in another, the object is unpacked. One solution could be to unpack the object (in the assertion it is packed) and compute the upper bound like previously explained. Another solution could be to try to pack the object (in the assertion it is unpacked), and report an error if that was not possible. In this instance, we believe we can preserve precision without much overhead, so we chose the first solution.

The following listings present some scenarios where the least upper bound between two assertions is computed. Assume that there is always full permission to read or to write into the local variables.

Listing 7.14: Assertions' upper bound: example 1

```
1  Cell cell = new Cell();
2  // access(cell.0,1) ∧ packed(cell) ∧ typeof(cell,State "NoItem")
3
4  if (condition) {
5    take(cell);
6    // access(cell.0,0)
7  } else {
8    addItem(cell);
9    // access(cell.0,1) ∧ packed(cell) ∧ typeof(cell,State "OneItem")
10 }
11
12 // access(cell.0,0)
```

The first scenario (listing 7.14) uses a cell which stores an item. If a certain condition is true, the cell is "moved" and the current context loses permission to access it. If the condition is false, the cell is temporarily borrowed. In this example, we combine two

access permissions, one with full permission, and one with no permission to the cell, resulting in one with no permission. Since with no permission to an object, we cannot ensure that it is packed or that it is in a given typestate, such information is not preserved. An error is also reported since a "resource" might have been lost.

Listing 7.15: Assertions' upper bound: example 2

```
1  Cell cell1 = new Cell();
2  Cell cell2 = null;
3  // access(cell1.0,1) ∧ packed(cell1) ∧ typeof(cell1,State "NoItem") ∧
4  // typeof(cell2,Null)
5
6  if (condition) {
7    cell2 = cell1;
8    // access(cell1.0,0) ∧ eq(cell1,cell2) ∧
9    // access(cell2.0,1) ∧ packed(cell2) ∧ typeof(cell2,State "NoItem")
10 } else {
11   cell2 = new Cell();
12   // access(cell1.0,1) ∧ packed(cell1) ∧ typeof(cell1,State "NoItem") ∧
13   // access(cell2.0,1) ∧ packed(cell2) ∧ typeof(cell2,State "NoItem")
14 }
15
16 // access(cell1.0,0) ∧
17 // access(cell2.0,1) ∧ packed(cell2) ∧ typeof(cell2,State "NoItem")
```

In the second example (listing 7.15), one cell is created, and stored in variable *cell1*, and variable *cell2* is assigned to *null*. If the condition is true, *cell2* becomes an alias for the cell also referenced from the local variable *cell1*. If the condition is false, *cell2* will reference a new cell object. Notice how the permission for the object pointed by *cell2* is the same in both branches, and that in both branches, the object is packed and is in state *NoItem*. Because of that, such information is preserved. But since in the *if* branch, there is no access to the object referenced by *cell1*, the upper bound of the two assertions will affirm that there is no access to the object. The information about the equality between *cell1* and *cell2*, which was true in the *if* branch, is not included in the resulting assertion, since access to the object pointed by *cell1* was lost and because that equality was unknown in the *else* branch. Notice how if the condition were to be false, the object referenced by *cell1* would just be lost. Because of that, an error needs to be reported, since protocol completion might have been compromised.

Listing 7.16: Assertions' upper bound: example 3

```
1  Cell cell = new Cell();
2  c.setItem(new Item());
3
4  Item item = null;
5  // access(cell.0,1) ∧ packed(cell) ∧ typeof(cell,State "OneItem") ∧
6  // typeof(item,Null)
7
8  if (condition) {
9    item = cell.getItem();
10   // access(cell.0,1) ∧ access(cell.item.0,0) ∧
11   // access(item.0,1) ∧ eq(cell.item,item) ∧
12   // typeof(cell,State "OneItem") ∧ unpacked(cell) ∧
13   // typeof(item,NoProtocol) ∧ packed(item)
14 } else {
15   // Unpacking to compute the upper bound...
16
17   // access(cell.0,1) ∧ access(cell.item.0,1) ∧
18   // typeof(cell,State "OneItem") ∧ unpacked(cell) ∧
19   // packed(cell.item) ∧
20   // typeof(item,Null)
21 }
22
23 // access(cell.0,1) ∧ access(cell.item.0,0) ∧
24 // typeof(cell,State "OneItem") ∧ unpacked(cell) ∧
25 // typeof(item,NoProtocol | Null) ∧ packed(item)
```

In the third scenario (listing 7.16), a cell is created and an item is stored on it, and a local variable *item* is declared with the value *null*. If the condition is true, the item is borrowed and assigned to the local variable *item*. This results in the cell becoming unpacked. If the condition is false, nothing happens, and the cell remains packed with one item, and the local variable remains with the value *null*. In this scenario, the cell is unpacked in one branch, and packed in a different branch. To compute the least upper bound, the cell object is unpacked in the assertion from the *else* branch, and then the upper bound is computed as explained before. The resulting assertion gives full access to the cell object, no access to the item via the *item* field, asserts that the cell remains in the *OneItem* state, and that it is packed. The local variable *item* may refer to the item or it may have value *null*. If it refers to the item, we may also say that it is packed.

The following listing presents the ML implementation of a function that computes the least upper bound between two assertions.

Listing 7.17: *assertionsUpperBound* function

```
let assertionsUpperBound (a:assertion) (b:assertion) (inv:ttype->assertion) : assertion =
  (* Returns a new assertion without the predicates that may no longer hold *)
  let pruneInvalid (a:assertion) : assertion = ... in
  (* Perform the necessary unpacks *)
  let (a, b) = unpackNecessary a b inv in
  pruneInvalid {
    accesses = map (fun (Access(l, f)) ->
      accessUpperBound (Access(l, f)) (getAccess b l)
    ) a.accesses;
    equalities = filter_map (fun (Equality(x,y)) ->
      if areEqual b x y then Some(Equality(x,y)) else None
    ) a.equalities;
    typeOfs = map (fun (TypeOf(l, t)) ->
      typeOfUpperBound (TypeOf(l, t)) (getTypeOf b l)
    ) a.typeOfs;
    packs = a.packs @ b.packs;
    unpacks = a.unpacks @ b.unpacks;
  }
```

The implementation starts by finding the locations that are packed in one assertion and not in the other, and then proceeds to unpack those. Given two assertions, where locations that are packed (or unpacked) in one, are also packed (or unpacked) in the other, the computing of the least upper bound continues: the upper bound of the access predicates is computed, by choosing the smaller fractional values; if an equality holds in one assertion and not in the other, it is not included in the final result; the upper bound of the *typeof* predicates is computed, by producing an union type; and the lists of *pack* and *unpack* predicates are concatenated. Finally, predicates that may no longer hold, because some *access* predicate now has fractional value zero, are removed in the *pruneInvalid* function.

Note that the above implementation is simplified. It does not consider equality transitivity, it does not consider the fact that if two *locations* refer to the same object all its fields are also the same, and it does not perform permission transfer to cover other possible implications.

7.1.9 Nullable values and union types

In this section, we address how access permissions, *packed* and *unpacked* predicates, function in the situation a location stores the *null* value. Additionally, we explain how an invariant is computed given an union type.

If a location stores the *null* value, we may assume that we have full access permission to it. This is correct since we can think of *null* as an immutable object with no fields, which can be arbitrarily shared. With that reasoning, we can also say that *null* is packed or even unpacked, since it has no fields. This device allows us to avoid losing information

when computing the least upper bound between two assertions, one in which a value is *null* and other in which it is not *null* (like in the third example of the previous section).

When speaking about fields of a potentially *null* value, we can also assert that we have full permission to access them and that they have the *Bottom* type. This works because accessing a field of a *null* value would result in a null pointer error anyway, which the type-checker detects and reports in compile-time. By assuming that we have full access to the field, we also avoid reporting two errors when an access is made.

To compute an invariant given an union type, we establish that the assertion the *inv* function returns, is the result of computing the least upper bound of the invariants of each typestate in the union. If one of the types in the union type is *Null*, we default for an assertion that gives full access to any field, to avoid losing information when computing the upper bound, using the previously explained reasoning.

To illustrate how nullable values and union types are handled in practice, suppose there is a cell object which may or may not include an item stored inside of it and that is packed.

$$access(cell.0, 1) \land packed(cell) \land typeof(cell, State"OneItem"|State"NoItem")$$

To unpack the object, we need to obtain the invariants for both states, *OneItem* and *NoItem*. For this example, assume that the cell object has a *item* field where the item is stored, and that the invariants for both states are the following, respectively.

$$access(item, 1) \land access(item.0, 1) \land packed(item) \land typeof(item, NoProtocol)$$

$$access(item, 1) \land typeof(item, Null)$$

Before performing the actual unpack operation, we need to compute the least upper bound between these two assertions. Notice that in the *NoItem* state, the *item* is *null*. To avoid losing information, we arbitrarily include some *access* and *packed* predicates, resulting in the following assertion for the *NoItem* state.

$$access(item, 1) \land access(item.0, 1) \land packed(item) \land typeof(item, Null)$$

Computing the least upper bound between the invariants of both states results in what follows.

$$access(item, 1) \land access(item.0, 1) \land packed(item) \land typeof(item, NoProtocol|Null)$$

With the previous assertion for the type *OneItem* | *NoItem* computed, we can unpack the cell object, resulting in the following assertion.

$access(cell.0, 1) \land unpacked(cell) \land typeof(cell, State"OneItem"|State"NoItem") \land$
$access(cell.item, 1) \land access(cell.item.0, 1) \land$
$packed(cell.item) \land typeof(cell.item, NoProtocol|Null)$

7.2 Inference algorithm

Until now, we have presented the language of assertions as if the programmer would write them explicitly in the code. Since that could become a burden to the programmer, in this section, we are going to present an algorithm to automatically infer the assertions from the code. This algorithm is inspired by the work done in [30]. In that work, a constraint system over symbolic permissions, which expresses the requirements at each program point, is inferred and then solved using linear programming [21]. A similar approach was also previously presented in [90].

Just like in Hoare logic [39], each statement in the code has one assertion that precedes it, supplying the pre-conditions to be met for the code to run correctly, and one assertion that follows it, with the facts resulting from executing that statement.

The inference algorithm analyzes each statement in the code and produces constraints. These constraints limit the possible assertions that occur before or after each statement. For example, for the statement *iterator.next()*, the assertion that precedes it must allow the *iterator* variable to be read, it must indicate that the object is in a state that allows for the *next* call, and the permission to modify that state must be provided. The assertion that follows it must provide the effects of calling the *next* method.

After inferring the constraints, these are provided to the Z3 Solver [22] to ensure that they are satisfiable.

The implementation was done in Kotlin and relies on the Checker Framework, like the first version of the tool, and on the Z3 Solver. Although we use Z3, the algorithm produces input for any SMT Solver[1].

7.2.1 Implementation

The implementation of the inference algorithm is composed of four steps: a skeleton for the assertions for each method of each class is constructed, assertions are instantiated and associated with each expression or statement in the code, constraints are inferred on those assertions, and then the Z3 Solver [22] is used to find a satisfiable solution.

Like discussed in previous sections, assertions are composed of *access*, *typeof*, *eq*, *packed* and *unpacked* predicates. *Access* predicates specify the permissions for *access locations* (i.e. variables, fields or objects pointed by variables or fields). *Typeof* predicates indicate the

[1] http://smtlib.cs.uiowa.edu/

current type of *locations* (i.e. variable or fields). *Eq* predicates assert that two *locations* are aliases. Finally, *packed* and *unpacked* predicates indicate if the information about the fields of an object is hidden behind an abstract view or exposed.

To build an initial structure for the assertions, each method of each class is analyzed and all the variables and associated fields are gathered. For now, we are working with all the objects unpacked and we are ignoring subtyping. To know the fields associated with variables, we check the Java type of each variable (which is statically known), find the class declaration, and gather all the declared fields. We also check the Java type of each field and gather all the fields of those, recursively. After gathering all these, we build a list of possible equalities that would need to be tracked, by combining two by two the variables or fields that have compatible Java types, making sure that all equalities that could be true by transitivity are also included in the list.

With an assertion skeleton constructed for each method, concrete assertions are instantiated before and after each expression or statement in each method, with *symbolic permissions* associated with each *access location* and *symbolic types* associated with each *location*. Each assertion is also associated with *symbolic equalities* (with boolean type) where each represents if a given equality (from the list of possible equalities) is true in that assertion. These symbols are the values for which Z3 will try to find satisfiable values. Additionally, assertions for the pre- and post-conditions are associated with each method. During this instantiation, consecutive assertions are connected to track which assertions imply which.

Following that, each expression and statement is analyzed independently and constraints are inferred on the assertion that precedes each and the assertion that follows each.

Finally, the solving phase is performed in two sub-phases. First, the constraints over the *symbolic permissions* and *symbolic equalities* are given to Z3 to find a satisfiable answer. Using the model produced by Z3, the constraints over the *symbolic types* are simplified (replacing the *symbolic permissions* and *symbolic equalities* with concrete fractional and boolean values, respectively), and given to Z3 to find a satisfiable answer. Splitting the solving phase in two sub-phases works because the access permissions and equalities inferred do not depend on the types, but the inferred types do depend on these, for example, to assert that an object has a given type, there needs to be enough permission. Additionally, performance is improved.

The code is only considered correct if, and only if, the two solving phases report that there are satisfiable interpretations for the constraints. If the first phase does not provide a satisfiable model, the second phase does not occur.

7.2.2 Constraints

In this section are presented the constraints inferred when statically analyzing each piece of code. Previously, the assertions were presented as if they were manually provided

in the code, and the access permissions were presented with concrete fractional values. Now that we want to infer the assertions, and for the purposes of explaining the inferred constraints, we will extend the language of assertions with variables (e.g. f_1), arithmetic (e.g. $f_1 + f_2$), number equalities and inequalities (e.g. $f_1 = f_2$ or $f_3 > 0$), implication, predicates and functions, which will be explained in the following sections. The variables are the values for which the Z3 Solver will try to find satisfiable answers.

7.2.2.1 Method declarations

Each method is associated with a pre-condition and a post-condition. The requirements needed for the method call (expressed in the pre-condition) and the results produced by the method call (expressed in the post-condition), are inferred by analyzing the body of the method.

To allow the tracking of the information about the objects passed in the parameters of the object, there are ghost variables that hold references to those objects. Thus, these ghost variables hold the same references that the parameters hold. This artifact is needed because in Java, one can assign to the parameters of the method, which would mean that we could lose track of the access permissions and types of the objects passed.

For example, a method which accepts two parameters (a and b), would be analyzed as if the parameters were constant and assigned to the corresponding local variables at the beginning of the method's code.

Listing 7.18: Parameters with their counterparts

```
void method(final A param0, final B param1) {
  A a = param0;
  B b = param1;
  // ...
}
```

Additionally, all the possible equalities for the given method are constrained to be *false*. Without this enforcement, Z3 could arbitrarily assign *true* to these, meaning that all the possible equalities would hold, which would be incorrect, because Z3 only attempts to find a satisfiable model that makes the constraints hold.

With respect to all the other variables and fields referenced in the method, the pre-condition provides no permissions to those. These are introduced when variables are declared, when other methods are called, etc... For a method with no arguments, the pre-condition corresponds to the empty assertion. The empty assertion is an assertion in which no locations have permissions. It is similar to the "empty heap" assertion in separation logic [73] and also corresponds to the *true* assertion.

To keep track of the permissions and the type of a value returned by a method, we also make use of a ghost variable that represents such value. So, a *return* statement, is seen as an assignment to that ghost variable.

Listing 7.19: Return values

```
Object method() {
  return new Object();
}
// Seen as:
Object method() {
  returnValue = new Object();
}
```

7.2.2.2 Variable declarations

When a variable is declared, full access to that variable is introduced and its type is *Unknown*. All the associated fields get no permissions and the *Unknown* type. All the other variables and fields remain with their access permissions and types.

Listing 7.20: Constraints inferred for variable declarations

```
Object x;
// access(x,1)
```

7.2.2.3 Variable reads

To read from a variable, there needs to be read access to that variable. Reads have no side-effects so, any fact that was true before this expression, remains true after this expression.

Listing 7.21: Constraints inferred for variable reads

```
// access(x,f1) ∧ f1 > 0
x;
```

7.2.2.4 Field reads

To read from a field, there needs to be read access to the variable or field that holds the object, its type should be a subtype of *Object* (ensuring it is not *null*), and there needs to be read access to the field itself. Reads have no side-effects so, any fact that was true before this expression, remains true after this expression.

Listing 7.22: Constraints inferred for field reads

```
// access(x,f1) ∧ access(x.y,f2) ∧ f1 > 0 ∧ f2 > 0
// typeof(x,t1) ∧ isSubtype(t1,Object)
x.y;
```

7.2.2.5 Assignments

Assignments are very common in imperative programming languages and they are the instructions that introduce aliasing so, it is important to handle these correctly. To assign

to a variable, there needs to be write access to that variable. To assign to a field, there needs to be read access to the variable or field that holds the object, its type should be a subtype of *Object* (ensuring it is not *null*), and there needs to be write access to the field itself.

Since assignments overwrite the value previously stored, it is important to not lose the "residual" access permission and type of the previous object pointed by the variable (or field). To keep track of this "residual" information, we use a ghost local variable that represents the previous object. With this artifact, no facts about that object are lost. To ensure protocol completion of all objects, the state of these ghost variables is checked at the end of methods and at the end of loops. Since a loop may contain assignments in its body and execute an arbitrary number of times, there is no enough amount of ghost variables that can be created to represent all possible values, so the end of loops must be checked as well.

The use of ghost variables is a technique already used by verifiers to facilitate the verification process. For example, they may be used to refer to a value of a variable at some program point different from the point where an assertion is declared [40]. In this instance, ghosts variables are used to keep track of the previous values of variables before they get assigned to.

In practice, when analyzing the assignment, we imagine the existence of an assignment to a ghost variable before the actual assignment is performed. In other words, if we are analyzing an assignment such as $x = y$, we need to consider the assignment *oldX* = x before. To ensure the total amount of permissions is preserved, each assignment transfers all the access permissions and type information from the assigned expression to the assignee, leaving the assigned expression with no permission and the *Unknown* type. After the main assignment, there is an equality that holds between the assigned expression and the assignee. With that, permissions can be transferred again if necessary later.

Listing 7.23: Constraints inferred for an assignment

```
// access(x,1) ∧ access(y,f1) ∧ f1 > 0
// access(x.0,f2) ∧ access(y.0,f3)
// typeof(x,t1) ∧ typeof(y,t2)
oldX = x;
// access(oldX, 1) ∧ access(x,1) ∧ access(y,f1)
// access(oldX.0,f2) ∧ access(x.0,0) ∧ access(y.0,f3)
// typeof(oldX,t1) ∧ typeof(x,Unknown) ∧ typeof(y,t2)
x = y;
// access(oldX, 1) ∧ access(x,1) ∧ access(y,f1)
// access(oldX.0,f2) ∧ access(x.0,f3) ∧ access(y.0,0)
// typeof(oldX,t1) ∧ typeof(x,t2) ∧ typeof(y,Unknown)
// eq(x,y)
```

Since assignments have side-effects, it is important to infer the correct constraints to represent the changes that are performed and to preserve information about locations

that were not modified. The permissions for the assignee and the assigned expression remain the same, the permission and type of the object pointed by the ghost variable in the post-condition correspond to the permission and type of the object pointed by the assignee in the pre-condition, the permission and type of the assignee in the post-condition correspond to the permission and type of the assigned expression, and the assigned expression is left with no permission and the *Unknown* type.

To handle equalities, it is not enough to include *eq(x,y)* in the post-condition. To facilitate the reasoning, we employ the assignment rule of Hoare logic [39].

$$\overline{\{P[E/x]\}\ x := E\ \{P\}}$$

Figure 7.3: Assignment rule

By applying the assignment rule of Hoare logic in the two assignments presented, we get the following intermediate assertions.

Listing 7.24: Applying the assignment rule of Hoare logic

```
{P[y/x][x/oldX]}
oldX = x;
{P[y/x]}
x = y;
{P}
```

This means that each equality holds in the post-condition if and only if it holds in the pre-condition by replacing the assignee with the assigned expression and replacing the ghost variable with the assignee. By looking at the previous example, we know that *eq(x,y)* holds in the post-condition since *eq(y,y)* trivially holds in the pre-condition.

Listing 7.25: Applying the assignment rule of Hoare logic (2)

```
{eq(x,y)[y/x][x/oldX]}
oldX = x;
{eq(x,y)[y/x]}
x = y;
{eq(x,y)}
```

Listing 7.26: Applying the assignment rule of Hoare logic (3)

```
{eq(y,y)}
oldX = x;
{eq(y,y)}
x = y;
{eq(x,y)}
```

7.2.2.6 Object instantiations

When a new object is initialized, there is full access to it and to its fields, and the object is in its initial state, according to the corresponding protocol. In the following listing, the rules for assignment and method calls also apply but are ignored for brevity.

Listing 7.27: Constraints inferred for new objects

```
x = new SomeClass();
// access(x.0,1) ∧ typeof(x,initialState(SomeClass))
```

7.2.2.7 Method calls

In general, to call a method, an assertion that implies the pre-condition of the method must hold. After the method call, the post-condition, and any assertion implied by it, hold. The constraints upon the pre- and post-conditions are found when analyzing the code of the method.

Non-static method calls, i.e. method calls on objects, are reasoned with as if the method had one additional argument representing the *this* value, which means that, for example, *iterator.next()* would be interpreted as *next(iterator)*. Furthermore, when calling methods on objects, there are some additional constraints. To call the method, the object must be in a state that allows for that method call (i.e. $available(t_1$,method)), and after the method call, the object is in the state corresponding to the transition via that method call (i.e. $transition(t_1$,method)). If the method changes the state of the object, then full permission to the object is also required.

Listing 7.28: Constraints inferred for mutable methods

```
// access(x,f1) ∧ f1 > 0 ∧ access(x.0,1)
// typeof(x,t1) ∧ available(t1,method)
x.method();
// access(x,f1) ∧ access(x.0,1)
// typeof(x,transition(t1,method))
```

If according to the protocol, the object remains in the same state after the method call, then full access permission is not required (i.e. access(x.0,f_2) $\wedge$ $f_2 > 0$ instead of access(x.0,1)).

Listing 7.29: Constraints inferred for readable methods

```
// access(x,f1) ∧ f1 > 0 ∧ access(x.0,f2) ∧ f2 > 0
// typeof(x,t1) ∧ available(t1,method)
x.method();
// access(x,f1) ∧ access(x.0,1)
// typeof(x,t1)
```

For the method to be called, the permissions of the arguments need to be equal or greater than the permissions of the corresponding parameters in the pre-condition, and

the types of the arguments need to be subtypes of the types of the corresponding parameters in the pre-condition.

Listing 7.30: Method's requirements (1)

```
void fillCells(Cell param1, Cell param2) {
  // access(param1.0,1)
  // access(param2.0,1)
  // typeof(param1,State "NoItem")
  // typeof(param2,State "NoItem")
  param1.setItem(new Item());
  param2.setItem(new Item());
}

void main() {
  Cell c1 = new Cell();
  Cell c2 = new Cell();
  fillCells(c1, c2);
}
```

In this example, to call the method *fillCells*, the permission for the object in *param1* needs to be equal to one, the permission for the object in *param2* needs to be equal to one as well, and both objects need to be in the *NoItem* state. This means that the permission for the object in *c1* needs to be equal to one (because *c1* corresponds to *param1*), the permission for the object in *c2* needs to be equal to one (because *c2* corresponds to *param2*), and both objects need to be in the *NoItem* state.

Although each argument expression corresponds in general to a single parameter, such is not always the case. For example, if a method that requests two objects is called with the same object in both arguments, then the permission for that object needs to be equal or greater than the sum of the requested permissions of both parameters in the method's pre-condition. If this was not the case, we could potentially allow for the duplication of permissions, which would be incorrect.

Listing 7.31: Method's requirements (2)

```
void main() {
  Cell c = new Cell();
  fillCells(c, c);
}
```

In this second example, the object in *c* would correspond to both *param1* and *param2* in the *fillCells* method, which means that the permission for that object would need to be at least two. This is impossible since permissions are values between zero and one and an error would be reported as intended.

After the method is called, the permissions and types of the arguments must take into account the effects the method produced, which include if locations were modified. A location is considered modified if the method requested full permission to it or to any location that reaches the first. For example, if full permission is requested for *cell.item*,

then *cell.item.state* may be modified indirectly by replacing the object in *cell.item* for another one.

For each argument (and corresponding fields), if it was modified, then the corresponding permissions and types are only those ensured by the method's post-condition; otherwise, the corresponding permissions are the subtraction between the permissions before the method call and the permissions the method call took, and the corresponding types remain the same.

Listing 7.32: Method's effects (1)

```
void fillCells(Cell param1, Cell param2) {
  // access(param1.item,1)
  // access(param2.item,1)
  // typeof(param1.item,Null)
  // typeof(param2.item,Null)
  param1.item = new Item();
  param2.item = new Item();
  // access(param1.item,1)
  // access(param2.item,1)
  // typeof(param1.item,State "S0")
  // typeof(param2.item,State "S0")
}

void main() {
  Cell c1 = new Cell();
  Cell c2 = new Cell();
  fillCells(c1, c2);
}
```

To discuss how the effects of methods are handled, consider this third example where the code of *setItem* was inlined. The method requests full permission to the *item* fields in *param1* and *param2*, requests that they hold the *null* value, ensures that full permissions remain, and that the fields hold an item in the *S0* state. Since *c1* corresponds to *param1* and *c2* corresponds to *param2*, and since the method requested full permission to modify their fields, the permissions and types for *c1.item* and *c2.item* are those ensured for *param1.item* and *param2.item* (respectively) in the method's post-condition.

The effects of methods also have an impact on which equalities hold after a method call. For each pair of locations that may refer to the same object, we build constraints such that: if the locations were not modified, then the equality is true if and only if it was true before the method call; if both locations were modified, then the equality is true if and only if that is asserted in the post-condition; if only one location was modified, then the equality is true if and only if that is asserted in the post-condition or that can be proven via transitivity (i.e. if now exists a location that is equal to the modified location and the non-modified location).

Listing 7.33: Method's effects (2)

```
class Cell {
  void setItem(Item param1) {
    // access(this.item,1)
    this.item = param1;
    // access(this.item,1)
    // eq(this.item,param1)
  }
}

void main() {
  Cell cell = new Cell();
  Item item = new Item();
  cell.setItem(item);
  // eq(cell.item,item)
}
```

In this example, a cell and an item are instantiated, and then the *setItem* method is called to store the item in the cell. This method requires full permission to write into the *item* field and ensures that the field holds the item passed in the parameter (i.e. *eq(this.item,param1)*). Since *cell.item* corresponds to *this.item*, which is modified, and *item* corresponds to *param1*, the equality between *cell.item* and *item* in the *main* context holds because the equality between *this.item* and *param1* holds in the post-condition of the *setItem* method.

Listing 7.34: Method's effects (3)

```
void main() {
  Cell cell = new Cell();
  Item item = new Item();
  Item item2 = item;
  // eq(item,item2)
  cell.setItem(item);
  // eq(item,item2)
  // eq(cell.item,item)
  // eq(cell.item,item2)
}
```

Suppose now that there is an alias for the item object in variable *item2* (i.e. *eq(item,item2)*). The equality between *item* and *item2* holds after the method because it holds before the method and none of those variables was modified. The equality between *cell.item* and *item* holds after the method call as previously explained. Finally, the equality between *cell.item* and *item2* after the method call is proven by transitivity, because there is a variable (i.e. *item*) which is now equal to both *cell.item* and *item2*.

7.2.2.8 Threads

With our tool, threads are associated automatically with a protocol with three states: *not_started*, *started* and *end*. When a thread object is instantiated, it is in the *not_started* state. In this state, only the *start* method is available which, when called, changes the state of the thread to *started*. Only in the *started* state, the *join* method is available, which waits for the thread to finish, and changes the state of the thread to the final state, *end*.

Currently, threads are only supported if instantiated with a lambda expression and if used in the local context where they were created.

The pre- and post-conditions of the *start* and *join* methods are inferred from the pre- and post-conditions of the associated lambda expression. The pre-condition of *start* is the pre-condition of the lambda expression. The post-condition of *start* is such that the permissions requested by the thread are not returned to the caller. The pre-condition of *join* requires nothing (except that the thread is in the *started* state and full permission is available to it). The post-condition of *join* corresponds to the post-condition of the lambda expression, to allow for the permissions to be returned to the caller.

7.2.2.9 Control flow statements

In Java, statements are generally executed from top to bottom, in the order that they appear. However, control flow statements are used to break up the flow of execution. These include decision-making statements (*if-else*, *switch*), looping statements (*for*, *while*, *do-while*), and branching statements (*break*, *continue*, *return*) [18].

To ease the reasoning about the flow of execution of programs, each method declaration is analyzed not by visiting each node in the abstract syntax tree, but by visiting each node in a *control flow graph* [3], which is built by the Checker Framework.

As previously mentioned, each expression or statement in the code has an assertion that precedes it and an assertion that follows it. For expressions that evaluate to a boolean value, there are two assertions that follow it, one for when the expression is *true*, and other for when the expression is *false*. In terms of the control flow graph, this means that each node in the graph is associated with an assertion that precedes it and an assertion that follows it (or two assertions that follow it). Additionally, the assertions that follow each node, imply the assertions that precede the successor nodes (i.e. nodes that can be reached by following the out-edges).

For example, when considering an *if-else* statement, we know that the assertion that follows it is the result of computing the upper bound of the assertions resulting from the *if* and the *else* branches (represented as a_1 and a_2, respectively, in the following listing).

Listing 7.35: The assertion that follows an *if-else* statement

```
if (condition) {
  ...
  // a1
} else {
  ...
  // a2
}
// assertionsUpperBound(a1,a2)
```

Another way to look at an *if-else* statement is to consider that the assertion resulting from the *if* branch and the assertion resulting from the *else* branch need both to imply the assertion that immediately follows the *if-else* statement (represented as a_3, in the following listing).

Listing 7.36: The assertion that follows an *if-else* statement (2)

```
if (condition) {
  ...
  // a1
} else {
  ...
  // a2
}
// (a1 ⟹ a3) ∧ (a2 ⟹ a3)
// a3
```

The analysis of loop statements especially benefits from looking at the code via a control flow graph. In Java there are three types of loop statements: *while*, *do-while*, and *for* [48]. In a *while* loop, the condition is evaluated, and if true, the loop body is executed and control returns to the condition, otherwise, the loop exits. In a *do-while* loop, the loop body is executed at least once, and iterations continue until the condition is false. In the *for* loop, the declarations are executed (if any), and while the condition is true, the loop body and then the update expression are executed.

Loops may contain *break* and *continue* statements. When a *break* statement is executed, control is transferred to the enclosing labeled statement (or the innermost enclosing *while*, *do-while*, or *for* statement, if no label is given) and immediately completes [48]. When a *continue* statement is executed, control is transferred to the enclosing labeled statement (or the innermost enclosing *while*, *do-while*, or *for* statement, if no label is given) and immediately ends the current iteration and begins a new one (starting with the evaluation of the condition) [48].

An example of a control flow graph of a *while* statement, without branching statements, can be seen in figure 7.4.

Let *pre(Cond)* be the pre-condition for *Condition*, and *post(CondTrue)* and *post(CondFalse)* be the post-conditions for when *Condition* evaluates to *true* or *false* (respectively), and let

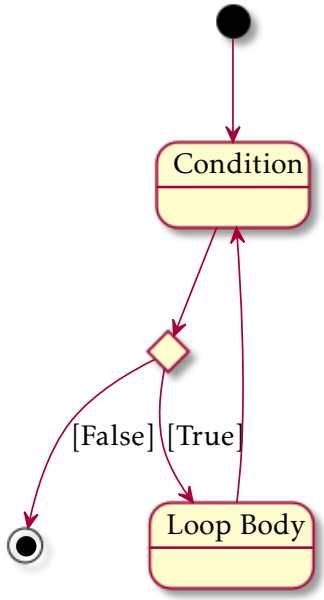

Figure 7.4: While's control flow graph

pre(LoopBody) and *post(LoopBody)* be the pre- and post-conditions (respectively) for the *Loop Body*.

For the *while* loop, the inference algorithm will produce constraints such that *pre(Cond)* is implied by the assertion that holds before executing the loop statement, *post(CondTrue)* implies *pre(LoopBody)*, *post(LoopBody)* implies *pre(Cond)*, and *post(CondFalse)* implies the assertion that holds when the loop exits.

If we look at the *while rule* in Hoare logic [39], we may observe the similarities.

$$\frac{\{B \wedge P\}\ \mathrm{S}\ \{P\}}{\{P\}\ \text{while B do S done}\ \{\neg B \wedge P\}}$$

Figure 7.5: While rule

The assertion $B \wedge P$ corresponds to *post(CondTrue)* and *pre(LoopBody)*, P corresponds to *pre(Cond)* and *post(LoopBody)*, and $\neg B \wedge P$ corresponds to *post(CondFalse)*.

What we are left to define are the specific constraints that together build the implications between assertions.

7.2.2.10 Implications

As previously explained, one of the initial phases of the inference algorithm is responsible for connecting consecutive assertions with implications. Because of the different flows of execution that are possible in programs, an assertion might be implied by more than one assertion. For example, as we seen, the assertion that immediately follows an *if-else* statement, is implied by the assertions that result from the *if* and *else* branches.

In this section, we will explain which concrete constraints are produced so that the implication between assertions is enforced while allowing for the transferring of permissions.

For the purposes of explanation, we will be using an *if-else* statement as example. Let x and y be two variables which may point to an object with a z field. Let $tail$ be the assertion that follows the statement and let $head_{if}$ and $head_{else}$ be the assertions that result from the *if* and *else* branches, respectively, which imply $tail$ (i.e. $head_{if} \implies tail$ and $head_{else} \implies tail$).

$$
\begin{aligned}
head_{if} \iff & \\
& access(x, f_1) \land access(x.0, f_2) \land access(x.z, f_3) \land \\
& access(y, f_4) \land access(y.0, f_5) \land access(y.z, f_6) \land \\
& typeof(x, t_1) \land typeof(y, t_2)
\end{aligned}
$$

$$
\begin{aligned}
head_{else} \iff & \\
& access(x, f_7) \land access(x.0, f_8) \land access(x.z, f_9) \land \\
& access(y, f_{10}) \land access(y.0, f_{11}) \land access(y.z, f_{12}) \land \\
& typeof(x, t_3) \land typeof(y, t_4)
\end{aligned}
$$

$$
\begin{aligned}
tail \iff & \\
& access(x, f_{13}) \land access(x.0, f_{14}) \land access(x.z, f_{15}) \land \\
& access(y, f_{16}) \land access(y.0, f_{17}) \land access(y.z, f_{18}) \land \\
& typeof(x, t_5) \land typeof(y, t_6)
\end{aligned}
$$

For each one of the variables in $tail$, the access permission of each is the minimum between the access permissions of the same variable in $head_{if}$ and $head_{else}$. For example, if the permission for variable x in $head_{if}$ is f_1, and the permission in $head_{else}$ is f_7, then the permission for x in $tail$ (i.e. f_{13}) is $min(f_1, f_7)$.

$f_{13} = min(f_1, f_7)$
$f_{16} = min(f_4, f_{10})$

For each one of the fields in $tail$, the access permission of each needs to consider the possible existence of aliases between the objects. For example, if the permission for field $x.z$ in $head_{if}$ is f_3, and the permission in $head_{else}$ is f_9, and no aliases for x are known,

then the permission for $x.z$ in $tail$ (i.e. f_{15}) is $min(f_3, f_9)$. If, for example, y is an alias of x (which means that $y.z$ and $x.z$ are the same location), then the following holds:

$$min(f_3, f_9) + min(f_6, f_{12}) = f_{15} + f_{18}$$

Which is equivalent to:

$$(min(f_3, f_9) - f_{15}) + (min(f_6, f_{12}) - f_{18}) = 0$$

With this constraint, f_{15} and f_{18} are not fixed, thus allowing the transferring of permissions. We also note that each term of the sum is the subtraction between the minimum of the permissions in $head_{if}$ and $head_{else}$, and the permission in $tail$, which together amount to zero, meaning that there is conservation of permissions between aliases. By looking at this pattern, we realize that we can use the set of possible equalities (computed in the initial phases of the algorithm) and build all the necessary formulas.

Assuming there can be no other aliases for x and y, we would produce the following constraint when constraining the access permission for $x.z$:

$$(min(f_3, f_9) - f_{15}) + (\text{if } eq(x, y) \text{ then } min(f_6, f_{12}) - f_{18} \text{ else } 0) = 0$$

And we would produce the following constraint when constraining the access permission for $y.z$:

$$(min(f_6, f_{12}) - f_{18}) + (\text{if } eq(y, x) \text{ then } min(f_3, f_9) - f_{15} \text{ else } 0) = 0$$

In practice, if $eq(x, y)$ were true in both $head_{if}$ and $head_{else}$, then the constraints would correspond to:

$$(min(f_3, f_9) - f_{15}) + (min(f_6, f_{12}) - f_{18}) = 0$$

$$(min(f_6, f_{12}) - f_{18}) + (min(f_3, f_9) - f_{15}) = 0$$

If $eq(x, y)$ were false in $head_{if}$ or in $head_{else}$, then the constraints would correspond to:

$$min(f_3, f_9) = f_{15}$$

$$min(f_6, f_{12}) = f_{18}$$

For each one of the variables and fields in $tail$, the access permission of each to the object pointed by each, follows the same reasoning as explained before. The difference is that instead of talking about the permission to a field, we would be talking about

the permission for the object pointed by a variable or field. For example, given that the permission for the object pointed by x is f_2, f_8 and f_{14} in $head_{if}$, $head_{else}$ and $tail$ (respectively), and the permission for the object pointed by y is f_5, f_{11} and f_{17} in $head_{if}$, $head_{else}$ and $tail$ (respectively), then the following constraints would be produced:

$(min(f_2, f_8) - f_{14}) + (\text{if } eq(x,y) \text{ then } min(f_5, f_{11}) - f_{17} \text{ else } 0) = 0$

$(min(f_5, f_{11}) - f_{17}) + (\text{if } eq(y,x) \text{ then } min(f_2, f_8) - f_{14} \text{ else } 0) = 0$

For each one of the variables and fields in $tail$, the type of each is, in general, the union of the types in $head_{if}$ and $head_{else}$. If aliases are known in $head_{if}$ and $head_{else}$, then the type is refined by intersecting with the types of the aliases. For example, given two variables x and y, and given that the type of x is t_1, t_3 and t_5 in $head_{if}$, $head_{else}$ and $tail$ (respectively), and the type of y is t_2, t_4 and t_6 in $head_{if}$, $head_{else}$ and $tail$ (respectively), then the following constraints would be produced:

$intersection(union(t_1, t_3), \text{ if } eq(x,y) \text{ then } union(t_2, t_4) \text{ else } Unknown) = t_5$

$intersection(union(t_2, t_4), \text{ if } eq(y,x) \text{ then } union(t_1, t_3) \text{ else } Unknown) = t_6$

In practice, if $eq(x,y)$ were true in both $head_{if}$ and $head_{else}$, then the constraints would correspond to:

$intersection(union(t_1, t_3),\ union(t_2, t_4)) = t_5$

$intersection(union(t_2, t_4),\ union(t_1, t_3)) = t_6$

If $eq(x,y)$ were false in $head_{if}$ or in $head_{else}$, then the constraints would correspond to the following. Note that $intersection(t, Unknown)$ is always t.

$union(t_1, t_3) = t_5$

$union(t_2, t_4) = t_6$

For each pair of possible equalities between variables and fields, an equality holds in $tail$ if it holds both in $head_{if}$ and $head_{else}$.

Additionally, to make sure that the assertions remain well-formed after some permission transfer has happened, and following the properties presented in section 7.1.3, we constraint the *symbolic permissions* and the *symbolic types* in the $tail$ assertion so that:

- An access permission greater than zero to the object pointed by a variable or field

is only available if the permission to the variable or field is also greater than zero (property 1);

$$f_{13} = 0 \implies f_{14} = 0$$
$$f_{16} = 0 \implies f_{17} = 0$$

- An access permission greater than zero to a field is only available if the permission to the variable or field that holds the corresponding object is also greater than zero (property 2);

$$f_{13} = 0 \implies f_{15} = 0$$
$$f_{16} = 0 \implies f_{18} = 0$$

- An equality only holds if there are permissions greater than zero for the variables or fields the predicate refers to (property 3);

- A type may be *Null* or *Primitive* only if there is a permission greater than zero for the corresponding variable and field. Otherwise, the type is *Unknown* (properties 4 and 5);

- A type may be an object type only if there is a permission greater than zero for the variable and field that holds the reference to that object and if there is a permission greater than zero to the object itself. Otherwise, the type is *Unknown* (properties 4 and 6).

$$xIsObject = \neg isSubtype(union(t_1, t_3), Null \mid Primitive)$$
$$yIsObject = \neg isSubtype(union(t_2, t_4), Null \mid Primitive)$$

$$f_{13} = 0 \vee (f_{14} = 0 \wedge xIsObject) \implies t_5 = Unknown$$
$$f_{16} = 0 \vee (f_{17} = 0 \wedge yIsObject) \implies t_6 = Unknown$$

Properties 7, 8 and 9, which speak about the *packed* and *unpacked* predicates, always hold because in the current implementation, objects are always unpacked. Property 10, which ensures that the sum of the access permissions that refer to the same memory location do not exceed one, is enforced by the constraints presented before, which preserve the total amount of permissions. In conclusion, as long as the assertions $head_{if}$ and $head_{else}$ are well-formed, $tail$ will always be well-formed.

7.2.3 Implementation details

In this section, we will discuss some details regarding the implementation of the inference algorithm that we not explained before for simplicity.

7.2.3.1 Expressions

Expressions in Java may evaluate to object references for which we need to track the access permissions and types. In the implementation, expressions are like variables or fields, which means that they appear in the assertions like any other variable or field.

But since, for example, *typeof("new Object()",Object)* was not considered as part of the language of assertions, we can instead suppose the existence of ghost variables that are unique for each expression and that track all the permissions and types of each expression.

Listing 7.37: Tracking permissions and types of expressions

```
expr1 = new Object()
// access(expr1,1) ∧ access(expr1.0,1) ∧
// typeof(expr1,Object)
```

7.2.3.2 Analysis of expressions

In the first version of the tool, the result of analyzing each expression was composed by a *then store* and an *else store*, which include the type information that is true when the expression is *true* or when it is *false*, respectively. In this version of the tool, a similar thing is employed. The difference is that instead of two *stores*, we have two assertions. If the expression does not evaluate to a boolean value, then these two assertions are actually the same.

7.2.3.3 Z3 definitions

For Z3 [22] to solve the constraints, it requires some definitions.

Each *symbolic fraction* is associated with a constant declaration of a real value, constrained to be between zero and one. Since real values are represented in Z3 as fractional values, precision is not lost.

Listing 7.38: Symbolic fraction in Z3

```
1 (declare-const f20 Real)
2 (assert (<= 0 f20))
3 (assert (<= f20 1))
```

Each *symbolic equality* is associated with a constant declaration of a boolean value which is *true* when an equality between two locations holds in a given assertion. This implies that for each assertion and for each pair of possible equalities, there is an unique boolean value that expresses if that equality holds.

Listing 7.39: Symbolic equality in Z3

```
1 (declare-const eq4 Bool)
```

During the implementation process, we attempted to represent types in various ways. First we attempted to define constants for each singleton type, use a function with two

arguments to represent the union between two types, and then define some properties over that function. Then we tried to represent union types with a list of singleton types. And finally, we attempted to represent union types with a set of singleton types. Unfortunately, for each one of these experiments, Z3 was too slow in proving properties over the subtyping relationships between types. Because of this, it was likely that the inference algorithm would perform poorly in trying to find satisfiable types that would fulfill the subtyping constraints.

For these reasons, each *symbolic type* is associated with a constant declaration of a value that belongs to an enumeration. This enumeration includes values representing each one of the singleton types and values representing some union types. To avoid the need to compute all the possible unions types, the set of union types that we consider are those that are mentioned in the constraints produced. This seems to be a good compromise between performance and precision. With this approach, all the types considered all discretely defined, allowing us to define functions over types for all the possible combinations using the Z3 API. The functions over types used in the Z3 context are: *isSubtype*, *union* and *intersection*.

Listing 7.40: Symbolic types in Z3

```
(declare-datatypes () ((Type
  Unknown Bottom Object NoProtocol Null Primitive Ended State1 State0 Union_State1_State2
)))
(assert (distinct
  Unknown Bottom Object NoProtocol Null Primitive Ended State1 State0 Union_State1_State2
))
(declare-const t15 Type)
```

To define the *isSubtype* function, we combine the types in pairs and for each pair we call the *isSubtype* function written in Kotlin, and place the result (true or false) directly in the function definition.

Listing 7.41: *isSubtype* function in Z3

```
(define-fun isSubtype ((a Type) (b Type)) Bool
  (and
    (=> (= a Unknown) (and
      (=> (= b Unknown) true)
      (=> (= b Bottom) false)
      (=> (= b Object) false) ...
    ))
    (=> (= a State1) (and
      (=> (= b State0) false)
      (=> (= b Union_State1_State2) true) ...
    ))
    ...
  )
)
```

To define the *union* function, we combine the types in pairs and for each pair we call the *union* function written in Kotlin. For each result, we check if it belongs to the list of types considered. If it does, the result is placed directly in the function definition. Otherwise, the closest supertype that does belong in the list is placed in the function definition. This is important because we are not considering all the possible types, just an approximation.

Listing 7.42: *union* function in Z3

```
(define-fun union ((a Type) (b Type)) Type
  (ite (= a Unknown)
    (ite (= b Unknown)
      Unknown
      (ite (= b Object)
        Unknown ...
      )
    )
    (ite (= a State1)
      (ite (= b State0)
        Union_State1_State2 ...
      ) ...
    ) ...
  )
)
```

To define the *intersection* function, we combine the types in pairs and for each pair we call the *intersection* function written in Kotlin. For each result, we check if it belongs to the list of types considered. If it does, the result is placed directly in the function definition. Otherwise, the closest supertype that does belong in the list is placed in the function definition.

Listing 7.43: *intersection* function in Z3

```
(define-fun intersection ((a Type) (b Type)) Type
  (ite (= a Unknown)
    (ite (= b Unknown)
      Unknown
      (ite (= b Object)
        Object ...
      )
    )
    (ite (= a State1)
      (ite (= b State0)
        Bottom ...
      ) ...
    ) ...
  )
)
```

Previously, to explain which constraints are produced when analyzing method calls on

objects, we used the *available* and *transition* functions. The *available* function indicates if a method is available in a given typestate. The *transition* function, given an initial typestate and a method, returns the typestate to which the object transits via that method call. The *available* function does not need to be defined in Z3. When analyzing the method call, we pre-compute the typestates in which the object must be in, and constraint the current type to be subtype of the union of those typestates. To handle transitions, specialized *transition* functions are defined in Z3 for each method, which given the current typestate, returns the new typestate resulting from that method call. Note that these specialized functions only need to consider as input the typestates in which the method is available to be called.

Listing 7.44: Specialized *transition* function for the *setItem* method in Z3

```
(define-fun transition_setItem ((a Type)) Type
  (ite (= a State_NoItem)
    State_OneItem
    (ite (= a State_OneItem)
      State_OneItem
      (ite (= a Union_NoItem_OneItem)
        State_OneItem
        Unknown
      )
    )
  )
)
```

7.2.4 Limitations of the implementation

The current implementation of the inference algorithm includes some limitations. Currently, all objects are considered to be unpacked. This causes problems if for example we want to work with recursive data structures, like a linked-list, where each node has a reference for the following node via a field (e.g. *next*). The problem is that it is impossible to gather all the possible "chains" (e.g. *node.next.next*) to build the structure of the assertions, since a linked-list may have an arbitrary number of connected nodes. The ideal solution would be to infer when an object may be packed and when it must be unpacked, but we leave that as subject of future work.

Another limitation is that the analysis of threads only works if the thread is started and waited upon in the context in which it was created. If a thread object is passed to another method, the information about which permissions it retains (upon the *start* call) and which permissions it gives back (upon the *join* call) is lost.

Like mentioned, subtyping is not yet considered. When analyzing a method call, the pre-conditions and post-conditions of the method are obtained by looking at the pre- and post-conditions associated with the method declaration (instantiated in the initial phases of the algorithm). This ignores dynamic method dispatch in Java, where the actual

method to be called is resolved at runtime, rather than compile-time.

Finally, we need to investigate how to better report errors to the programmer. When Z3 reports that it did not find any satisfiable solution for the constraints, it returns a set of constraints that together produce a contradiction. We would like to take this information and indicate to the programmer the code location where the error is. Assuming that this set of constraints is simplified to highlight the concrete contradiction, and by tracking in which code locations certain constraints were inferred, we can give the programmer an approximation of where the error lies.

7.2.5 Protocol inference

Although the programmer is free from explicitly writing all the assertions, it still needs to write the protocol for classes. It would be possible to try to infer the protocol itself, by finding which methods' post-conditions imply the pre-conditions of other methods, and building the protocol that allows for those sequences of method calls, following a strategy like [85].

The problem with inferring the protocol specification is that we might hit, what some would call, the Assertion Inference Paradox [31]. In general, a program is correct if the implementation satisfies the specification. But if we also infer the specification, we might be reasoning in a circle. Nonetheless, inferring the specification is useful to compare it for consistency with the intended behavior. Additionally, the inference process might produce an inconsistent specification, which can reveal a flaw in the implementation.

Another concern is that inferring the protocols would not be useful given the current language of assertions. This language of assertions deals with access permissions, aliasing and the types of objects, and it is not expressive enough to, for example, assert some properties over integers values. Because of that, it is likely that the inferred protocols would allow for many undesired sequences of method calls, which the programmer would need to remove anyway.

Considering the points presented, we still believe that inferring the protocols is useful, providing a base specification from which the programmer can work, but given the current expressiveness of the language, we leave this topic as subject of future work.

7.3 Comparison with other languages

The following tables compare the *access* annotations from different languages as they relate with how one can require access to some part of the memory. As one can observe, our language is not completely new and it is inspired by already existent ideas, even if the notation is slightly different.

The annotations in table 7.1 are used when one requires full permission to a memory location, so that it can be read and modified.

Our language	Spec#	Chalice	Dafny	VeriFast
access(x,1)	modifies x	acc(x)	modifies `x	x ↦ _
access(x.y,1)	modifies x.y	acc(x.y)	modifies x`y	x.y ↦ _

Table 7.1: Full access annotations

Our language (0 < f < 1)	Chalice	VeriFast (0 < f < 1)
access(x,f)	rd(x)	[f]x ↦ _
access(x.y,f)	rd(x.y)	[f]x.y ↦ _

Table 7.2: Read access annotations

The annotations in table 7.2 are used when one only needs read permission to a memory location.

Additionally, the concept of packing and unpacking, as well as unpacking of potentially aliased objects, can be replicated in VeriFast. One just needs to consider the use of an *Invariant* predicate which represents the object being packed.

Listing 7.45: Summing two number objects: VeriFast

```
1  //@ predicate Invariant(Number num; int value) = num.value |-> value;
2
3  int sum(Number a, Number b)
4    //@requires [1/2]Invariant(a, ?i1) &*& [1/2]Invariant(b, ?i2);
5    //@ensures [1/2]Invariant(a, i1) &*& [1/2]Invariant(b, i2);
6  {
7    //@assert [?f1]a.value |-> _ &*& [?f2]b.value |-> _;
8    //@assert f1 == 1/2 &*& f2 == 1/2;
9    return a.value + b.value;
10 }
```

Consider for example two *Number* objects which store an integer value in their *value* fields, and a *sum* method which returns the sum of those two integer values. The method requires that the invariants of the objects be meet with fraction 1/2, allowing the same object to be passed in both parameters. This implies that we have 1/2 permission to the heap location corresponding to the field, which is verified by the assertions in lines 7 and 8. Similar annotations in our language are shown in listing 7.46.

Listing 7.46: Summing two number objects: Language of assertions

```
1  int sum(Number a, Number b)
2    //requires: access(a.0, 1/2) ∧ packed(a) ∧ access(b.0, 1/2) ∧ packed(b)
3    //ensures: access(a.0, 1/2) ∧ packed(a) ∧ access(b.0, 1/2) ∧ packed(b)
4  {
5    // access(a.0, 1/2) ∧ unpacked(a) ∧ access(a.value, 1/2)
6    // access(b.0, 1/2) ∧ unpacked(b) ∧ access(b.value, 1/2)
7    return a.value + b.value;
8  }
```

Even though it is possible to replicate ideas of our language of assertions in other languages and tools, our tool has a particular focus on providing typestate-oriented features without much annotation burden, thanks to the inference algorithm. Additionally, our tool ensures that the protocols of objects are completed. To our knowledge, this is not supported by mainstream tools, even those which allow for very expressive specifications.

7.4 Working examples

In this section are presented six examples: four with well-behaved code (i.e. methods are called in correct order, there are no data-races at the level of variables or fields, and there is no interference between concurrent calls of methods that change the state of the object), and two with code that is not well-behaved, because it allows for interference.

These examples make use of a *cell* object and an *item* object, like the motivating example in section 5.5. The protocol of the *item* was slightly modified by the addition of the *getState* method that returns the current value of the *state* field, while leaving the object in the same typestate.

The inferred assertions for well-behaved code are presented as comments. Note that when an *access* predicate is not mentioned, it is assumed that there is no permission to that *access location*, and when a *typeof* predicate is not mentioned, it is assumed that the asserted type is *Unknown*. Additionally, if an equality is not mentioned, it means that it does not hold in that program point. When the code is not well-behaved, no assertions are inferred.

All six examples start with the initialization of a cell object, the initialization of an item object, which is stored inside the cell via the *setItem* method, and the retrieval of the stored item via the *getItem*, which is then assigned to the local variable *item*. For the well-behaved examples, the inferred assertions for the first three code statements are presented as comments in listings 7.47, 7.48, and 7.49.

Listing 7.47: Examples introduction (Part 1)

```
1 Cell cell = new Cell();
2 // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "NoItem")
3 // access(cell.item,1)
```

After the cell object is initialized and assigned to the *cell* variable (line 1), there is full access to the variable, full access to the object pointed by the variable (i.e. the cell), the cell is in the *NoItem* state, and there is full permission to the *item* field of the cell. At initialization, the field has the *null* value, but since the field will be overwritten in the next statement, Z3 just attributed the *Unknown* type instead of *Null*. This is correct since *Unknown* is the supertype of all types.

Listing 7.48: Examples introduction (Part 2)

```
5  cell.setItem(new Item());
6  // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "OneItem")
7  // access(cell.item,1) ∧ access(cell.item.0,1) ∧ typeof(cell.item,State "S0")
8  // access(cell.item.state,1) ∧ typeof(cell.item.state,Primitive)
```

After the item is stored on the cell (line 5), there is still full permission to the *cell* variable and the cell object, but now the cell is in the *OneItem* state (line 7). There is full permission to the *item* field, and full permission to the item object, which is the *S0* state (line 8). Additionally, there is full permission to the *state* field of the item, which holds a primitive value (line 8).

Listing 7.49: Examples introduction (Part 3)

```
10 Item item = cell.getItem();
11 // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "OneItem")
12 // access(cell.item,1)
13 // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S0")
14 // access(item.state,1) ∧ typeof(item.state,Primitive)
15 // eq(item,cell.item)
```

After the item is retrieved from the cell (line 10), there is still full permission to the *cell* variable and the cell object, the cell remains in the *OneItem* state, and there is full permission to the *item* field (lines 11 and 12). However, the permission to the item object is now available via the *item* variable instead of the *item* field on the cell (lines 13 and 14), and an equality between both is now asserted (line 15). Notice how the total amount of permissions is preserved, without duplication, and the fact that *item* and *cell.item* are aliases is known.

Listing 7.50: Example 1 (Part 1)

```
1  item.changeState();
2  // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S1")
3  // access(item.state,1) ∧ typeof(item.state,Primitive)
4  // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "OneItem")
5  // access(cell.item,1)
6  // eq(item,cell.item)
7
8  Item item2 = cell.getItem();
9  // access(item,1)
10 // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "OneItem")
11 // access(cell.item,1)
12 // access(item2,1) ∧ access(item2.0,1) ∧ typeof(item2,State "S1")
13 // access(item2.state,1) ∧ typeof(item2.state,Primitive)
14 // eq(item,item2) ∧ eq(item,cell.item) ∧ eq(cell.item,item2)
```

Listing 7.51: Example 1 (Part 2)

```
16 item2.changeState();
17 // access(item,1)
18 // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "OneItem")
19 // access(cell.item,1)
20 // access(item2,1) ∧ access(item2.0,1) ∧ typeof(item2,State "S1")
21 // access(item2.state,1) ∧ typeof(item2.state,Primitive)
22 // eq(item,item2) ∧ eq(item,cell.item) ∧ eq(cell.item,item2)
```

In the first example (listings 7.50 and 7.51), with the item object now available in the *item* local variable, the state of the item is changed from *S0* to *S1* (line 1). All the permissions remains the same and the cell remains in the same state. Then the item is retrieved again from the cell, being now referenced from the *item2* variable as well (line 8), which means that there is aliasing between *cell.item*, *item* and *item2*. Since in the following statement there is an attempt to change the state of the item again (line 16), but now via the *item2* variable, the permissions concerning the item are transferred to *item2* to allow for that operation. Since the item was already in the *S1* state, it remains in the *S1* state according to the protocol.

Listing 7.52: Example 2 (Part 1)

```
1  Thread t = new Thread(() -> {
2    // access(item,3/4) ∧ access(item.0,1) ∧ typeof(item,State "S0")
3    // access(item.state,1) ∧ typeof(item.state,Primitive)
4    item.changeState();
5    // access(item,3/4) ∧ access(item.0,1) ∧ typeof(item,State "S1")
6    // access(item.state,1) ∧ typeof(item.state,Primitive)
7  });
8
9  // access(t,1) ∧ access(t.0,1) ∧ typeof(t,State "NotStarted")
10 // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "OneItem")
11 // access(cell.item,1)
12 // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S0")
13 // access(item.state,1) ∧ typeof(item.state,Primitive)
14 // eq(cell.item,item)
```

In the second example (listings 7.52 and 7.53), with the item object available in the *item* local variable, the state of the item will be changed from *S0* to *S1* in a thread (line 4). All the permissions and types remain the same after the thread object was created (with the exception of the thread being introduced in the *NotStarted* state), since threads only start upon the *start* call.

Listing 7.53: Example 2 (Part 2)

```
16 t.start();
17
18 // access(t,1) ∧ access(t.0,1) ∧ typeof(t,State "Started")
19 // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "OneItem")
20 // access(cell.item,1)
21 // access(item,1/4)
22 // eq(cell.item,item)
23
24 t.join();
25
26 // access(t,1) ∧ access(t.0,1) ∧ typeof(t,Ended)
27 // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "OneItem")
28 // access(cell.item,1)
29 // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S1")
30 // access(item.state,1)
31 // eq(cell.item,item)
32
33 Item item2 = cell.getItem();
34
35 // access(t,1) ∧ access(t.0,1) ∧ typeof(t,Ended)
36 // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "OneItem")
37 // access(cell.item,1)
38 // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S1")
39 // access(item.state,1) ∧ typeof(item.state,Primitive)
40 // access(item2,1)
41 // eq(cell.item,item) ∧ eq(item2,item) ∧ eq(item2,cell.item)
42
43 item2.changeState();
44
45 // access(t,1) ∧ access(t.0,1) ∧ typeof(t,Ended)
46 // access(cell,1) ∧ access(cell.0,1) ∧ typeof(cell,State "OneItem")
47 // access(cell.item,1) ∧ access(item,1)
48 // access(item2,1) ∧ access(item2.0,1) ∧ typeof(item2,State "S1")
49 // access(item2.state,1)
50 // eq(cell.item,item) ∧ eq(item2,item) ∧ eq(item2,cell.item)
```

After the thread is started (line 16), 3/4 of permission is used in the thread to read the *item* variable, and only 1/4 is left in the current context. Additionally, the thread requests full access to the item object and its *state* field, so that its typestate and field can change. The thread is now in the *Started* state. After that, the *join* method is called (line 24), which waits for the thread to finish. After the thread finishes, full permission to the *item* variable, the item object and its field are restored. Following that, like in the previous example, the item is retrieved again from the cell and used (lines 33 and 43). The operation is allowed because full permission to the item is available.

Listing 7.54: Example 3

```
1  Thread t = new Thread(() -> {
2    item.changeState();
3  });
4
5  t.start();
6
7  Item item2 = cell.getItem();
8
9  item2.changeState();
10
11 t.join();
```

The third example (listing 7.54) is similar to the second example except there is an attempt to change the state of the item before waiting for the thread to finish (line 9). This code is not well-behaved because it presents interference: both the thread and the main thread attempt to change the state of the item concurrently. This is detected by the fact that the thread as requested full permission to the item but has not given it back yet. This creates a contradiction in the constraints, since the permission for the item in the main thread before line 9 is zero, while one is required.

Listing 7.55: Example 4 (Part 1)

```
1  Thread t1 = new Thread(() -> {
2    // access(item,1/4) ∧ access(item.0,1) ∧ typeof(item,State "S0" | State "S1")
3    // access(item.state,1) ∧ typeof(item.state,Primitive)
4    item.changeState();
5    // access(item,1/4) ∧ access(item.0,1) ∧ typeof(item,State "S1")
6    // access(item.state,1) ∧ typeof(item.state,Primitive)
7  });
8
9  // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,State "NotStarted")
10 // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S0")
11 // access(item.state,1) ∧ typeof(item.state,Primitive)
12
13 Thread t2 = new Thread(() -> {
14   // access(item,1/4) ∧ access(item.0,1) ∧ typeof(item,State "S0" | State "S1")
15   // access(item.state,1) ∧ typeof(item.state,Primitive)
16   item.changeState();
17   // access(item,1/4) ∧ access(item.0,1) ∧ typeof(item,State "S1")
18   // access(item.state,1) ∧ typeof(item.state,Primitive)
19 });
20
21 // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,State "NotStarted")
22 // access(t2,1) ∧ access(t2.0,1) ∧ typeof(t2,State "NotStarted")
23 // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S0")
24 // access(item.state,1) ∧ typeof(item.state,Primitive)
25
```

In the fourth example (listings 7.55 and 7.56), two threads are instantiated (lines 1 and 13), and both of them will attempt to change the state of the item (lines 4 and 16). Additionally, both threads require 1/4 of permission to read from the *item* variable and require full permission to the item object to change its state.

Listing 7.56: Example 4 (Part 2)

```
26 t1.start();
27
28 // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,State "Started")
29 // access(t2,1) ∧ access(t2.0,1) ∧ typeof(t2,State "NotStarted")
30 // access(item,3/4)
31
32 t1.join();
33
34 // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,Ended)
35 // access(t2,1) ∧ access(t2.0,1) ∧ typeof(t2,State "NotStarted")
36 // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S1")
37 // access(item.state,1)
38
39 t2.start();
40
41 // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,Ended)
42 // access(t2,1) ∧ access(t2.0,1) ∧ typeof(t2,State "Started")
43 // access(item,3/4)
44
45 t2.join();
46
47 // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,Ended)
48 // access(t2,1) ∧ access(t2.0,1) ∧ typeof(t2,Ended)
49 // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S1")
50 // access(item.state,1)
```

The thread *t1* is started first (line 26). Notice how only 3/4 of permission is left for the *item* variable and no permission is available for the item object in the main thread. After that, the *join* method is called on *t1*, which restores full permission to the item (line 32). Following that, the thread *t2* can be started (line 39) and waited upon (line 45). The code is well-behaved because the two threads do not concur.

Listing 7.57: Example 5

```
1  Thread t1 = new Thread(() -> {
2    item.changeState();
3  });
4  Thread t2 = new Thread(() -> {
5    item.changeState();
6  });
7
8  t1.start();
9  t2.start();
10
11 t1.join();
12 t2.join();
```

The fifth example (listing 7.57) is similar to the previous one except that the two threads will concur. Notice how *t1* is started and then *t2* is also started without waiting for the thread *t1* to finish. Since both of them will attempt to change the state of the item concurrently, thus requiring both full permission to it, the inference algorithm does not find any satisfiable answer, as expected.

Listing 7.58: Example 6 (Part 1)

```
1  Thread t1 = new Thread(() -> {
2    // access(item,1/6) ∧ access(item.0,1/6) ∧ typeof(item,State "S0")
3    // access(item.state,1/6) ∧ typeof(item.state,Primitive)
4    item.getState();
5    // access(item,1/6) ∧ access(item.0,1/6) ∧ typeof(item,State "S0")
6    // access(item.state,1/6) ∧ typeof(item.state,Primitive)
7  });
8
9  Thread t2 = new Thread(() -> {
10   // access(item,1/6) ∧ access(item.0,1/6) ∧ typeof(item,State "S0")
11   // access(item.state,1/6) ∧ typeof(item.state,Primitive)
12   item.getState();
13   // access(item,1/6) ∧ access(item.0,1/6) ∧ typeof(item,State "S0")
14   // access(item.state,1/6) ∧ typeof(item.state,Primitive)
15 });
16
17 // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,State "NotStarted")
18 // access(t2,1) ∧ access(t2.0,1) ∧ typeof(t2,State "NotStarted")
19 // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S0")
20 // access(item.state,1/3) ∧ typeof(item.state,Primitive)
```

The sixth example (listings 7.58 and 7.59) is similar to the previous one except instead of both threads trying to change the state of the item, they only attempt to read from it (lines 4 and 12). Notice how both threads only require 1/6 of permission to read from the *item* variable and 1/6 of permission to the item object, and use the item while in the *S0* state without changing it.

Listing 7.59: Example 6 (Part 2)

```
22 t1.start();
23
24 // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,State "Started")
25 // access(t2,1) ∧ access(t2.0,1) ∧ typeof(t2,State "NotStarted")
26 // access(item,5/6) ∧ access(item.0,5/6) ∧ typeof(item,State "S0")
27 // access(item.state,1/6) ∧ typeof(item.state,Primitive)
28
29 t2.start();
30
31 // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,State "Started")
32 // access(t2,1) ∧ access(t2.0,1) ∧ typeof(t2,State "Started")
33 // access(item,2/3) ∧ access(item.0,2/3) ∧ typeof(cell.item,State "S0")
34
35 t1.join();
36
37 // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,Ended)
38 // access(t2,1) ∧ access(t2.0,1) ∧ typeof(t2,State "Started")
39 // access(item,5/6) ∧ access(item.0,5/6) ∧ typeof(item,State "S0")
40 // access(item.state,1/6) ∧ typeof(item.state,Primitive)
41
42 t2.join();
43
44 // access(t1,1) ∧ access(t1.0,1) ∧ typeof(t1,Ended)
45 // access(t2,1) ∧ access(t2.0,1) ∧ typeof(t2,Ended)
46 // access(item,1) ∧ access(item.0,1) ∧ typeof(item,State "S0")
47 // access(item.state,1/3) ∧ typeof(item.state,Primitive)
```

Although the two threads concur, since they only perform read operations on the item object, the code is considered safe. Since each thread requires 1/6 of permission to the item object, only 5/6 is left after the first *start* call (line 22), and only 4/6 (i.e. 2/3) of permission is left after the second *start* call (line 29). After both *join* calls (lines 35 and 42), $2 \times 1/6$ of permission is restored, which means that the main thread as restored full access to the item. Notice how the information about the state of the item was never lost because read access was still available in main thread while the other threads were running. This example shows that sharing of objects is allowed as long as there is no interference between method calls that change the state of the object or modify its fields in any way.

Chapter 8

Conclusions and Future work

8.1 Summary

In this thesis, we presented a tool that type-checks a Java program where objects are associated with typestates. The first version focused on providing basic features while enforcing the linear use of objects. The features included the support for the association of protocols with classes and verification of the use of instances of those classes, prevention of null pointer errors, verification of the completion of protocols, and "droppable" states.

The second version allows aliasing of objects in a controlled way using a language of assertions. This language guarantees that the usage of objects will follow the protocols, no data-races occur at the level of variables and fields, and that method calls that change the state of an object do not interfere. Additionally, the language allows unpacking of aliased objects and transferring of permissions between aliases. To relieve the programmer from manually writing the assertions, we implemented an inference algorithm which analyzes the code statically and constructs all the required assertions.

The features provided are useful for real applications. For example, one can naturally define a protocol for a Java readable stream to ensure that the data is only read when available and that upon reading all the data, the stream is closed, freeing any resources associated with it. Additionally, if the stream is connected to another stream from which it gets all the data, the usage of the underlying stream is also verified. With the language of assertions, one can also share objects. For example, one can create a producer object and a consumer object which have references to each other. The producer notifies the consumer that more data is available, and the consumer calls the methods of the producer to retrieve the new data. Examples are available at the project's repository[1].

[1] https://github.com/jdmota/java-typestate-checker/

8.2 Future work

The current version of the tool includes some limitations: the inference algorithm does not infer when an object should be packed or unpacked; dynamic method dispatch in Java, where the actual method to be called is resolved at runtime, rather than compile-time, is not considered; and subtyping, as well as generics, are not yet supported. In the future, we plan to start by fixing these limitations.

To support more use cases, we would also like to allow for concurrent write operations to be performed on shared objects. The solution could be to change the *access* predicate to inform if mutations can be performed in an object that is aliased. That additional information could also indicate if the aliases are available only in a single threaded environment or may be shared between threads. In a single threaded context, one would just need to ensure that the object is left in a state from a set of states that all aliases agree upon and, in this context, state refinements would be allowed. For example, calling the *hasNext* method on a iterator in the *HasNext* state, to refine the state to *Next*, if the method returns *true*, would be safe, since in a single threaded environment, there would be no operations that could happen in between that would change the state of the object. If the aliases are shared between threads, then additional care is needed. The object must be left in a state from a set of states that all aliases agree upon but also, method calls on the object need to be synchronized, and state refinements would not be performed, since there would be no guarantee that other operations happened in between in different threads.

Another aspect that is subject of future work is how certain data structures could be handled. For example, in doubly linked list, each internal node has a reference to the following node and a reference to the previous node, and there may be an arbitrary number of internal nodes in the structure. It would be useful to track the aliasing between these internal nodes. Because of that, and also to deal with other kinds of collections (e.g. arrays), there might be the need to include quantifiers in our language of assertions, and extend the inference algorithm to find loop invariants that speak about collections, with strategies like [31]. The support for collections is also important to verify the usage of objects stored inside of collections.

Finally, it would be useful to be able to associate access permissions with locks or monitors, which would become available when these were acquired.

Bibliography

[1] W. Ahrendt et al. "Deductive Software Verification–The KeY Book." In: *Lecture Notes in Computer Science* 10001 (2016).

[2] J. Aldrich et al. "Permission-based programming languages: Nier track." In: *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE. 2011, pp. 828–831.

[3] F. E. Allen. "Control flow analysis." In: *ACM Sigplan Notices* 5.7 (1970), pp. 1–19.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. "The Spec# programming system: An overview." In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer. 2004, pp. 49–69.

[5] M. Barnett et al. "Boogie: A modular reusable verifier for object-oriented programs." In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2005, pp. 364–387.

[6] N. E. Beckman, D. Kim, and J. Aldrich. "An empirical study of object protocols in the wild." In: *European Conference on Object-Oriented Programming*. Springer. 2011, pp. 2–26.

[7] K. Bierhoff and J. Aldrich. "Modular typestate checking of aliased objects." In: *ACM SIGPLAN Notices* 42.10 (2007), pp. 301–320.

[8] R. Bornat et al. "Permission accounting in separation logic." In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2005, pp. 259–270.

[9] J. Boyland. "Alias burying: Unique variables without destructive reads." In: *Software: Practice and Experience* 31.6 (2001), pp. 533–553.

[10] J. Boyland. "Checking interference with fractional permissions." In: *International Static Analysis Symposium*. Springer. 2003, pp. 55–72.

[11] S. Brookes. "A semantics for concurrent separation logic." In: *Theoretical Computer Science* 375.1-3 (2007), pp. 227–270.

[12] J. Campos and V. T. Vasconcelos. "Channels as objects in concurrent object-oriented programming." In: *arXiv preprint arXiv:1110.4157* (2011).

[13] J. Campos and V. T. Vasconcelos. *Mool: Mini Object-Oriented Language*. Accessed: 2020-11-19. URL: http://rss.di.fc.ul.pt/tools/mool/.

[14] L. Cardelli. "Type systems." In: *ACM Computing Surveys* 28.1 (1996), pp. 263–264.

[15] G. de Caso et al. "Program abstractions for behaviour validation." In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, pp. 381–390.

[16] *Checker Framework - 2019 - Google Summer of Code Archive*. Accessed: 2020-04-30. URL: https://summerofcode.withgoogle.com/archive/2019/organizations/4937505709752320/.

[17] E. M. Clarke Jr et al. *Model checking*. MIT press, 2018.

[18] *Control Flow Statements (The Java Tutorials)*. Accessed: 2020-11-06. URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html.

[19] K. Crary, D. Walker, and G. Morrisett. "Typed memory management in a calculus of capabilities." In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1999, pp. 262–275.

[20] *Dafny: A Language and Program Verifier for Functional Correctness*. Accessed: 2020-07-01. URL: https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/.

[21] G. B. Dantzig. *Linear programming and extensions*. Vol. 48. Princeton university press, 1998.

[22] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[23] M. Degen, P. Thiemann, and S. Wehr. "Tracking linear and affine resources with Java (X)." In: *European Conference on Object-Oriented Programming*. Springer. 2007, pp. 550–574.

[24] R. DeLine and M. Fähndrich. *The Fugue protocol checker: Is your software baroque*. Tech. rep. Technical Report MSR-TR-2004-07, Microsoft Research, 2004.

[25] R. DeLine and M. Fähndrich. "Typestates for objects." In: *European Conference on Object-Oriented Programming*. Springer. 2004, pp. 465–490.

[26] P. Denissen, K. Huizing, and R. Kuiper. "Extending Dafny to Concurrency." In: (2017).

[27] W. Dietl et al. "Building and using pluggable type-checkers." In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 681–690.

[28] T. Ekman and G. Hedin. "The jastadd extensible java compiler." In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. 2007, pp. 1–18.

[29] M. Fahndrich and R. DeLine. "Adoption and focus: Practical linear types for imperative programming." In: *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation*. 2002, pp. 13–24.

[30] P. Ferrara and P. Müller. "Automatic inference of access permissions." In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2012, pp. 202–218.

[31] C. A. Furia and B. Meyer. "Inferring loop invariants using postconditions." In: *Fields of logic and computation*. Springer, 2010, pp. 277–300.

[32] R. Garcia et al. "Foundations of typestate-oriented programming." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.4 (2014), p. 12.

[33] S. J. Gay et al. "Modular session types for distributed object-oriented programming." In: *ACM Sigplan Notices* 45.1 (2010), pp. 299–312.

[34] J.-Y. Girard. "Linear logic." In: *Theoretical computer science* 50.1 (1987), pp. 1–101.

[35] *Google core libraries for Java*. Accessed: 2020-04-30. URL: https://github.com/google/guava/.

[36] M. Gordon. "Background reading on hoare logic." In: *Lecture Notes, April* (2012).

[37] T. P. Group. *The Plaid programming language*. Accessed: 2020-11-19. URL: https://www.cs.cmu.edu/~aldrich/plaid/.

[38] T. P. Group. *The Plaid Programming Language - Introduction*. Accessed: 2020-11-19. URL: https://www.cs.cmu.edu/~aldrich/plaid/plaid-intro.pdf.

[39] C. A. R. Hoare. "An axiomatic basis for computer programming." In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[40] M. Hofmann and M. Pavlova. "Elimination of ghost variables in program logics." In: *International Symposium on Trustworthy Global Computing*. Springer. 2007, pp. 1–20.

[41] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language primitives and type discipline for structured communication-based programming." In: *European Symposium on Programming*. Springer. 1998, pp. 122–138.

[42] W. Huang et al. "ReIm & ReImInfer: Checking and inference of reference immutability and method purity." In: *ACM SIGPLAN Notices* 47.10 (2012), pp. 879–896.

[43] H. Hüttel et al. "Foundations of session types and behavioural contracts." In: *ACM Computing Surveys (CSUR)* 49.1 (2016), pp. 1–36.

[44] *Introducing Kotlin support in Spring Framework 5.0*. Accessed: 2020-04-29. URL: https://spring.io/blog/2017/01/04/introducing-kotlin-support-in-spring-framework-5-0.

[45] S. S. Ishtiaq and P. W. O'hearn. "BI as an assertion language for mutable data structures." In: *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2001, pp. 14–26.

[46] B. Jacobs et al. "VeriFast: A powerful, sound, predictable, fast verifier for C and Java." In: *NASA Formal Methods Symposium*. Springer. 2011, pp. 41–55.

[47] *JastAdd*. Accessed: 2020-04-28. URL: http://jastadd.org/web/.

[48] *Java Language Specification: Chapter 14. Blocks and Statements*. Accessed: 2020-09-29. URL: https://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html.

[49] D. Jemerov and S. Isakova. *Kotlin in action*. Manning Publications Company, 2017.

[50] JetBrains. *Kotlin Programming Language*. Accessed: 2020-04-28. URL: https://kotlinlang.org/.

[51] C. B. JONES. "SPECIFICATION AND DESIGN OF (PARALLEL) PROGRAMS Cliff B. JONES." In: ().

[52] *JSR 308 Explained: Java Type Annotations*. Accessed: 2020-04-30. URL: https://www.oracle.com/technical-resources/articles/java/ma14-architect-annotations.html.

[53] *KMS Compliance Checker*. Accessed: 2020-05-04. URL: https://github.com/awslabs/aws-kms-compliance-checker.

[54] D. Kouzapas et al. "Typechecking protocols with Mungo and StMungo." In: *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*. ACM. 2016, pp. 146–159.

[55] O. Lahav and V. Vafeiadis. "Owicki-Gries reasoning for weak memory models." In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2015, pp. 311–323.

[56] K. R. M. Leino. "Dafny: An automatic program verifier for functional correctness." In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2010, pp. 348–370.

[57] K. R. M. Leino. "Modeling Concurrency in Dafny." In: *School on Engineering Trustworthy Software Systems*. Springer. 2017, pp. 115–142.

[58] K. R. M. Leino and P. Müller. "A basis for verifying multi-threaded programs." In: *European Symposium on Programming*. Springer. 2009, pp. 378–393.

[59] K. R. M. Leino and P. Müller. "Using the Spec# language, methodology, and tools to write bug-free programs." In: *Advanced Lectures on Software Engineering*. Springer, 2007, pp. 91–139.

[60] K. R. M. Leino, P. Müller, and J. Smans. "Verification of concurrent programs with Chalice." In: *Foundations of Security Analysis and Design V*. Springer, 2009, pp. 195–222.

[61] J. Mediero Iturrioz. "Verification of Concurrent Programs in Dafny." In: (2017).

[62] F. Militão, J. Aldrich, and L. Caires. "Aliasing control with view-based typestate." In: *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*. 2010, pp. 1–7.

[63] R. Milner. "A theory of type polymorphism in programming." In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

[64] K. Naden et al. "A type system for borrowing permissions." In: *ACM SIGPLAN Notices* 47.1 (2012), pp. 557–570.

[65] T. Nipkow et al. "Getting started with Dafny: A guide." In: *Software Safety and Security: Tools for Analysis and Verification* 33 (2012), p. 152.

[66] P. W. O'Hearn and D. J. Pym. "The logic of bunched implications." In: *Bulletin of Symbolic Logic* (1999), pp. 215–244.

[67] P. O'Hearn, J. Reynolds, and H. Yang. "Local reasoning about programs that alter data structures." In: *International Workshop on Computer Science Logic*. Springer. 2001, pp. 1–19.

[68] P. W. O'hearn. "Resources, concurrency, and local reasoning." In: *Theoretical computer science* 375.1-3 (2007), pp. 271–307.

[69] S. Owicki and D. Gries. "Verifying properties of parallel programs: An axiomatic approach." In: *Communications of the ACM* 19.5 (1976), pp. 279–285.

[70] M. M. Papi et al. "Practical pluggable types for Java." In: *Proceedings of the 2008 international symposium on Software testing and analysis*. 2008, pp. 201–212.

[71] F. Pfenning. "Benjamin C. Pierce. Types and programming languages. The MIT Press, Cambridge, Massachusetts, 2002, xxi+ 623 pp." In: *Bulletin of Symbolic Logic* 10.2 (2004), pp. 213–214.

[72] *References and Borrowing - The Rust Programming Language*. Accessed: 2020-05-04. URL: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.

[73] J. C. Reynolds. "Separation logic: A logic for shared mutable data structures." In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74.

[74] A. Sadiq, Y.-F. Li, and S. Ling. "A survey on the use of access permission-based specifications for program verification." In: *Journal of Systems and Software* 159 (2020), p. 110450.

[75] J. Siek and W. Taha. "Gradual typing for objects." In: *European Conference on Object-Oriented Programming*. Springer. 2007, pp. 2–27.

[76] J. Smans et al. "Verifying java programs with VeriFast." In: *Aliasing in Object-oriented Programming, pages XXX–XXX. Springer* (2012).

[77] J. Sunshine et al. "First-class state change in plaid." In: *ACM SIGPLAN Notices* 46.10 (2011), pp. 713–732.

[78] K. Takeuchi, K. Honda, and M. Kubo. "An interaction-based language and its typing system." In: *International Conference on Parallel Architectures and Languages Europe*. Springer. 1994, pp. 398–413.

[79] *The Checker Framework*. Accessed: 2020-04-28. URL: https://checkerframework.org/.

[80] M. Tofte and J.-P. Talpin. "Implementation of the typed call-by-value λ-calculus using a stack of regions." In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1994, pp. 188–201.

[81] M. Tofte and J.-P. Talpin. "Region-based memory management." In: *Information and computation* 132.2 (1997), pp. 109–176.

[82] A. P. University of Glasgow. *[St]Mungo*. Accessed: 2020-11-19. URL: http://www.dcs.gla.ac.uk/research/mungo/index.html.

[83] *Using Kotlin for Android Development*. Accessed: 2020-04-29. URL: https://kotlinlang.org/docs/reference/android-overview.html.

[84] V. Vafeiadis. *Modular fine-grained concurrency verification*. Tech. rep. University of Cambridge, Computer Laboratory, 2008.

[85] C. Vasconcelos and A. Ravara. "From object-oriented code with assertions to behavioural types." In: *Proceedings of the Symposium on Applied Computing*. 2017, pp. 1492–1497.

[86] V. T. Vasconcelos et al. "Sessions, from types to programming languages." In: *Bulletin of the EATCS* 103 (2011), pp. 53–73.

[87] D. Walker and G. Morrisett. "Alias types for recursive data structures." In: *International Workshop on Types in Compilation*. Springer. 2000, pp. 177–206.

[88] *What is Ownership? - The Rust Programming Language*. Accessed: 2020-05-05. URL: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[89] J. M. Wing. "FAQ on π-Calculus." In: *Microsoft Internal Memo* (2002).

[90] H. Yasuoka and T. Terauchi. "Polymorphic fractional capabilities." In: *International Static Analysis Symposium*. Springer. 2009, pp. 36–51.

[91] E. Zoppi et al. "Contractor. net: inferring typestate properties to enrich code contracts." In: *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*. ACM. 2011, pp. 44–47.