Sérgio António Inácio da Silva

Licenciado em Engenharia Informática

Type-based Protocol Conformance and Aliasing Control in Concurrent Java Programs

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Co-orientadores : João Ricardo Viegas da Costa Seco,
Prof. Auxiliar,
Universidade Nova de Lisboa
Hugo Torres Vieira,
Prof. Auxiliar,
Universidade de Lisboa

Júri:

Presidente:	Prof. Rodrigo Seromenho Miragaia Rodrigues
Arguente:	Prof. Salvador Luís Bettencourt Pinto de Abreu
Vogal:	Prof. João Ricardo Viegas da Costa Seco

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2012

Type-based Protocol Conformance and Aliasing Control in Concurrent Java Programs

To my grandfathers, António and José

Acknowledgements

Firstly, I would like to express my gratitude to both of my advisors: João Seco for his guidance, support and perseverance throughout the last three years, giving me the opportunity to learn so much; to Hugo Vieira for his guidance, readiness to discuss new ideas, support and razor-sharp comments and observations throughout the development of this dissertation.

To my parents Flávio and Clara, whom have sacrificed so much, for their unwavering support, making it possible for me to progress in my academical studies. To my big brother and sister, João and Patrícia, for their constant support. I cannot thank you enough.

To Sara, for the unforgettable moments together, ever-present friendship and love, constant support and encouragement and for putting up with my ramblings and shenanigans. Thank you.

To my colleagues at FCT, whom I have met throughout these 5 years, thank you for your support and making it a good place to work in. A special thanks to Tiago, Paulo, Ricardo and Bruno, for their companionship, brainstorming sessions and allowing me to learn about their own theses.

To my friends, thank you for the good times. A special thanks to Pires for always being there, his good spirits, constant concern with the progress of this work and awesome movie sessions!

To my colleagues at Safira, Bruno, Andreia, Pedro, Gonçalo, Lukasz, Jan, Pietro and Luís thank you for your support and encouragement.

This work was partially supported by the PTDC/EIA-CCO/104583/2008 research scholarship.

Abstract

In an object-oriented setting, objects are modeled by their state and operations. The programmer should be aware of how each operation implicitly changes the state of an object. This is due to the fact that in certain states some operations might not be available, e.g., reading from a file when it is closed. Additional care must be taken if we consider aliasing, since many references to the same object might be held and manipulated. This hinders the ability to identify the source of a modification to an object, thus making it harder to track down its state. These difficulties increase in a concurrent setting, due to the unpredictability of the behavior of concurrent programs.

Concurrent programs are complex and very hard to reason about and debug. Some of the errors that arise in concurrent programs are due to simultaneous accesses to shared memory by different threads, resulting in unpredictable outcomes due to the possible execution interleavings. This kind of errors are generally known as race conditions.

Software verification and specification are important in software design and implementation as they provide early error detection, and can check conformity to a given specification, ensuring some intended correctness properties. To this end, our work builds on the work of Spatial-Behavioral types formalism providing object ownership support. Our approach consists in the integration of a behavioral type system, developed for a core fragment of the Java programming language, in the standard Java development process.

Keywords: Java, Verification, Type Systems, Concurrency, Spatial-Behavioral Types

Resumo

Em programas *object-oriented*, os objectos são modelados pelo seu estado e operações. O programador deve estar ciente de como cada operação muda implicitamente o estado do objecto, devido ao facto que, em certos estados algumas operações não estejam disponíveis, e.g., ler de um ficheiro quando este está fechado. Cuidado adicional deve ser adoptado quando na presença de *aliasing*, uma vez que podem existir várias referências para o mesmo objecto, podendo estes ser manipulados de diferentes formas. Devido a fenómenos de *aliasing*, determinar a origem de uma modificação num objecto torna-se difícil, dificultando também qual o conhecimento do seu estado actual. Num cenário concorrente, identificar o estado actual de um objecto torna-se mais difícil dada a possibilidade de ordens de execução.

A programação concorrente é complexa tornando o seu raciocínio e depuração bastante difíceis, sendo também bastante propensa a erros devido à sua natureza não determinista. Alguns dos erros que surgem em programas concorrentes devem-se a acessos não controlados a memória partilhada por parte de *threads*, tendo efeitos imprevisíveis devido à possibilidade de ordens de execução. Este tipo de erros são geralmente conhecidos como *race conditions*.

Verificação e especificação de software são técnicas importantes no desenho e implementação de software, uma vez que permitem verificar estaticamente a ocorrência de erros, bem como a conformidade a uma dada especificação, assegurando a presença de algumas propriedades de correcção. Para este fim, o nosso trabalho inspira-se no formalismo de tipos espaciais-comportamentais, fornecendo suporte para *ownership* e definição de protocolos em objectos. A nossa abordagem consiste no desenvolvimento de um sistema de tipos baseado em tipos comportamentais para um fragmento da linguagem de programação Java, integrando-o com o processo de desenvolvimento Java.

Palavras-chave: Java, Verificação, Sistemas de Tipos, Concorrência, Tipos Espaciais-Comportamentais

Contents

List of Figures

Listings

1

Introduction

In this chapter we start by motivating our work. We give an overview of the context in which our work is based on, giving a brief description of other techniques that aim to tackle some of the challenges that we have identified. Afterwards, we provide a description of our proposed solution to these challenges, with some practical examples that highlight the main features. We address the main contributions of our work and conclude with the outline of this document.

1.1 Motivation

With today's proliferation of multi-core systems, developing concurrent programs that fully use the available resources in a system has been a growing concern. As hardware evolves, the software we write must keep up with this progress and allow for better use of the available resources.

Concurrent programs are complex and very hard to reason about and debug. Some of the errors that arise in concurrent programs are due to simultaneous accesses to shared memory by different threads, resulting in unpredictable outcomes due to the possible execution interleavings. These kinds of errors are generally known as race conditions.

In order to discipline access to shared memory and control such interferences, several control mechanisms are at the programmers' disposal. For example, monitors [Hoa74], locks, and compare and swap primitives. Their use can be have positive and negative impacts, in the sense that, although they can effectively discipline synchronization between threads, if they are not used correctly, their scope might be unnecessarily broad and the we lose ability to efficiently use the resources at our disposal, or even result in deadlocks. Hence, given the number of possible execution interleavings, even an expert

programmer can fail to detect interferences and deadlocks.

Software verification and specification are important in software design and implementation as they provide early error detection, and can check conformity to a given specification, ensuring some intended correctness properties. Among the most commonly used approaches are type systems. For example, there are some approaches that relate to concurrency control that include tools that employ Hoare logic [Hoa69] and its extension for monitor operations such as ESC/Java [FLL+02], separation logic [Rey02] and behavioral type systems [Cai08, Mil08, Par11, CS13], that will be the basis of this work.

1.2 Context

In an object-oriented setting, objects are modeled by their state and operations. The programmer should be aware of how each operation implicitly changes the state of an object. This is due to the fact that in certain states some operations might not be available, e.g., reading from a file when it is closed. Additional care must be taken if we consider aliasing, since many references to the same object might be held and manipulated. This hinders the ability to identify the source of a modification to an object, thus making it harder to track down its state. These difficulties increase in a concurrent setting, due to the unpredictability of the behavior of concurrent programs. Many of the errors that appear in such programs are due to inconsistent states caused by unrestricted concurrent access to shared memory. Building robust and correct concurrent applications requires advanced knowledge of concurrency control mechanisms and, even then, over-conservative solutions might be used, e.g., synchronization on an entire method instead of applying a finer grained synchronization technique that protects critical regions of the code.

Mainstream object-oriented languages, like Java, can statically detect some type and initialization errors. However, they do not detect errors related to misuse of objects with respect to a specification. Correct usage of an object is usually enforced through some internal logic, that constrains the the object's allowed behavior at runtime. The only information related to these errors is present, if at all, in runtime errors or in informal documentation. In fact, if the implicit protocol of a class is not followed, one possible outcome can be some kind of unexpected behavior at runtime, e.g., reading from a file that has not been opened.

Some verification techniques have been proposed, that aim to check the correct usage of resources with respect to a protocol. Session Types, introduced in [Hon93, HVK98], aim to verify binary communication protocols made via message passing through dedicated channels. In this approach, types describe the sequence of messages, or protocols, that are exchanged through channels. For example, a channel typed as ?[int] represents a channel along which we expect to receive a message containing an integer value, while ![int] describes a channel where we output an integer value. Now, if two parties use

the same channel to communicate according to ?[int] and ![int], then there will be a synchronization between both parties, since they are performing dual actions. Any binary communication that follows this synchronization scheme is then guaranteed to evolve, thus respecting the protocol. Although this approach aims to verify correctness in a message passing concurrency paradigm, it has been applied to other contexts such as object-oriented programming [GVR+10, CV10b] and operating systems [FAH+06].

In [SY86], the programming language concept known as typestate is introduced. It aims to describe which operations are available under certain conditions, as well as how they change the state. *Plaid* is a typestate and object-oriented, concurrent programming language proposed by Aldrich et al. [ASSS09]. Its syntax resembles that of Java, but in *Plaid* typestates are first-class abstractions, and objects are modeled by their possible states (actually represented as classes). Some variable modifiers were included to control various forms of aliasing, in order to statically detect incorrect usage of aliases. However, determining the current state of an object is performed dynamically.

Concurrent Separation Logic (CSL) is a logic proposed by O'Hearn in [O'H04], as an extension of Separation Logic [ORY01, Rey02] for concurrent programs. CSL aims to verify the correctness of concurrent programs that directly manipulate shared memory. Simply put, two concurrent programs can be safely run if they operate in separate parts of the heap, i.e., they do not compete for the same resources.

This notion of separation relates to the one present in the work done in [Cai08], where the notion of spatial-behavioral types is introduced. In this work, a type system is proposed for a concurrent object-oriented setting, that aims at statically verifying correct usage of resources accounting for shared behavior and aliasing, by allowing specification of owned behaviors that are guaranteed to be independent of any other. Behavioral types are associated to objects and specify how an object must be used. A notion of behavioral separation is embedded in types, where $U \mid V$ denotes a behavior in which U is independent of V, i.e., U and V do not compete for the same resources, and therefore can be exercised concurrently.

The idea of behavioral independence was taken further in the work of [CS13]. In this work, a behavioral type system is presented, where types bear resemblance in form and meaning to the ones described in [Cai08], which purpose is to verify the correctness of programs in a λ-calculus extended with mutable heap variables, records and concurrency primitives, such as threads and synchronization mechanisms. Type assertions state behavioral global constraints on runtime values instead of properties about the program state, enforcing type safety with regard to interference, in a concurrent setting where aliasing can occur, by imposing usage control disciplines.

In [Par11] an implementation of a fragment this type system was developed, consisting in a type inference algorithm for a prototype language. However, its concurrency model was based on the structured par operator and it did not account for object type variables.

Our work then builds on the work of [Par11, CS13, Mil08], aiming at a fragment of

the spatial-behavioral type system of [Cai08] including support for object type variables, ownership types and finer-grained protocol specification at the method level, while targeting a mainstream programming language, namely Java.

1.3 Proposed Solution

The main goal of this work is to define a technique to statically check the correctness of the behavior of a concurrent, object-oriented program that operates on shared memory, with respect to a prescribed usage protocol. Our development is based on the work of [Cai08] and extending [Par11] to support object ownership as well the object-based concurrency model of the Java programming language, instead of the binary | operator, where $e_1 \mid e_2$ represents the concurrent evaluation of expressions e_1 and e_2.

Usage protocols are defined in Java interfaces, through the use of the Java annotation system. These protocols, whose syntax is based on spatial-behavioral types [Cai08], can be placed at some key locations in an interface definition. At the interface declaration level, a behavior annotation will define how the objects belonging to any class implementing it should be used. Additionally, these annotations can also be specified in method declarations, where a behaviorally annotated argument provides a kind of contract about the behavior that must be exercised on the argument by any implementation of the method. On the other hand, an annotation at the return type level restricts the behavior that can be exercised upon what is returned by the method.

Our type system will be defined on top of a fragment of the Java programming language, with a focus on concurrency. We define a set of typing rules, one for each construct of the language, in which we analyze how resources are used in each one. Concurrent behavior is defined according to the Java concurrency model, where we require all threads to be started and joined after declaration. This constraint is related to how we extract concurrent behavior, i.e., what is the concurrent behavior that is exercised by each thread on an object. If the thread's lifetime exceeds that of the method in which it is declared, i.e., if it is not joined, we are unable to precisely determine when the behavior will end. To this end, we analyze the thread's body to determine what use it makes of each resource. With this information, we examine what is executed concurrently with the thread by analyzing the behavior that occurs within its lifetime, i.e., the behavior that is exercised between starting and joining the thread. Consider the following example:

```
public void pushAndPop(int elem) {
  Thread t1 = new Thread(new Runnable(){
    public void run(){
      s1.push(elem);
      s2.push(elem);
    }
  });

  t1.start();
```

```
    s1.pop();
    t1.join();
    s2.pop();
  }
```

By analyzing the body of thread `t1`, we see that it uses fields `s1` and `s2`, of the class where the method is defined, calling push on both with argument `elem`. Now, between starting and joining the thread, a call to `pop` on `y` is performed. So, from this information we are able to determine that two concurrent operations made on field `s1`. Since in this case we are concurrently pushing and popping an element from `s1`, we can conclude that it represents a race. However, after the thread finishes execution, a `pop` method call is performed on `s2`. Since this happens *after* the thread finishes we are also able to conclude that the behavior exercised on `s2` is sequential, therefore ruling out any possibility of concurrent interference. Generalizing, we check the correctness of each declared resource with respect to a previously defined protocol. This conformance is checked by means of a subtyping relation, based on behavioral simulation, that captures how a usage that is given to an object is safe, even if does not exactly match the specification that is provided.

Checking correct usage of objects in a Java program is fully integrated with the compilation process of `javac`, hence we are able to provide the results of our typechecker by using the error issuing facilities provided by the compiler. This means we can precisely pinpoint the method in which the type error occurred, and provide feedback to the programmer much like in regular Java development.

In order to understand the general idea of this work, we will now present a toy example. Consider the following interface

```
@Protocol("open; (read & write); close")
public interface File {
  public void open();
  public String read();
  public void write(String content);
  public void close();
}

public void main(String[] args){
  File f = new ...;

  f.open();
  f.write("content");
  f.close();
}
```

that represents a File. As usual, before using a file for reading or writing, we must open it, and after using it, it must be closed. This kind of protocol can be defined by the `@Protocol` annotation that is done at the interface declaration, and its argument is a string containing the protocol definition based on method names. In the `main` method,

we can see that the usage given to `f` follows the specified protocol. Now, suppose we want to write to the file, while opening it the same time:

```
public static void main(String... args) throws InterruptedException {
  final File f = new ...;
  Thread t = new Thread(new Runnable(){
    public void run(){
      f.open();
    }
  });
  t.start();
  f.write("content");
  t.join();
  f.close();
}
```

By looking at the code that exists in-between starting and joining thread `t`, we can see that the `write` method is being called concurrently with `open`. Now, according to what was specified in `File` interface, this usage is not safe since the given protocol imposes sequentiality, i.e., opening a file must necessarily be performed *before* using it in any way, and the concurrent behavior exercised on `f` does not ensure that this order is respected.

However, we can still have the `write` call be made by the thread, but we need to make sure it occurs after the `open` method call and so we can write a corrected version of the previous program that guarantees this temporal constraint:

```
public static void main(String... args) throws InterruptedException {
  final File f = new ...;
  Thread t = new Thread(new Runnable(){
    public void run(){
      f.open();
    }
  });
  t.start();
  t.join();
  f.write("content");
  f.close();
}
```

Now, suppose we want to model an object that is responsible for copying the content from one file to another, i.e., reading the content from a file and writing it to the other.

```
@Protocol("copy*")
public interface FileCopy {
  public void copy(@Usage("read")File f1, @Usage("write")File f2);
}

public static void main(String[] args) {
```

```
    File f1 = new ...;
    File f2 = new ...;
    FileCopy fc = new ...;

    f1.open();
    f2.open();
    fc.copy(f1, f2);
    f1.close();
    f2.close();
  }
```

The `FileCopy` interface contains a `copy` method, taking two `File` objects as parameters. This method is responsible for reading the content from the first file `f1` and copying it to the second file `f2`. In order to ensure that any implementation of `FileCopy` obeys these constraints we use the annotations `@Usage("read")` and @Usage("write") annotations respectively on each argument.

Considering the `main` method in the example above, we create two files, `f1` and `f2` as well as `FileCopy` object, `fc`. Initially, we need to open both files to start copying and this is accomplished by calling the open `open` function. Copying the content from `f1` to `f2` is made by calling function `copy` on `fc`, passing both files as arguments. Lastly, we close both files by calling `close` on both. From our analysis on this program, we can conclude that `fc` is used as `copy`, `f1` is used according to `open;read;close`, while `f2` is used according to `open;write;close`. Although the behaviors exercised upon the files do not exactly match the ones that are specified in the `File` interface, they are still considered safe as we will see.

Now, suppose we try to copy the content of a file to itself:

```
public static void main(String... args){
    File f1 = new ...;
    FileCopy fc = new ...;

    f1.open();
    fc.copy(f1, f1);
    f1.close();
  }
```

In our programming model, we impose that the behavior that is exercised on each argument must not interfere with one another, i.e., they must be independent. Therefore, the usage given to `f1` by calling `copy` is `read | write`, since it is both the source and destination files, indicating that `read` and `write` must be independent. However, the protocol specification in the `File` imposes a choice between these two methods, i.e., after opening the file, the protocol is `read & write` meaning that only one of these two methods may be called. Thus, this program is deemed unsafe by our type system.

As discussed previously, we extend the work of [Par11] by including support for object type variables. This requires careful treatment regarding possible aliasing as well

as ensuring that the usage of the fields of a class follows what is specified for each type. In our approach, object ownership consists in every object reference being exclusively held, i.e., there is only a single reference being used for any object. Our approach to control aliasing is then based on behavioral types that embody this notion of exclusive ownership of an object. Consider the scenario in which file `f2` is an alias to `f1`:

```
public static void main(String[] args) {
  File f1 = new ...;
  File f2 = f1;

  f1.open();
  f2.read();
  f1.close();
}
```

In this case, the example would not be considered type safe since assigning `f1` to `f2` causes loss of ownership of the former, i.e., the only reference that can be used is `f2`.

Ensuring correct class field usage follows an assign-then-use behavior: after initializing, each field must be used as was prescribed in its type. Consider an adaptation of `File` interface to allow concurrent writing, as follows:

```
@Protocol("open; (reset | write); close")
public interface File {
  public void open();
  public void reset();
  public void write(String content);
  public void close();
}
public class FileImpl implements File {
  private String contents;

  public FileImpl(String cont){ contents = cont; }

  public void open(){}
  public void reset(){ contents = ""; }
  public void write(String content){ contents = content; }
  public void close(){}
}
```

In this implementation, allowing `write` to be called concurrently with `reset` may result in concurrent assignments on the `contents` field, representing a possible race condition, thus violating the assign-then-use behavior described previously. Therefore, the implementation `FileImpl` is not considered type safe, as it is not able to support concurrent calls to `write` and `reset` as was specified in the `File` interface.

Our solution then consists in the implementation of a behavioral type system, including the concurrent behavior extraction algorithms and subtyping algorithm based

on a labeled transition system. Our type system will be implemented in the Scala multi-paradigm programming language and due to the interoperability between Scala and Java, we are able integrate our type system into Java development by using the Annotation Processor and Compiler Tree API, developing a custom annotation processor that converts Java source to our core representation, upon which we check the correctness properties addressed by our type system. The process of checking a program is done after the parsing of source files and the feedback from the typechecker is given to the user by using `javac` standard messaging facilities.

1.4 Contributions

In this section, we present a summary of what was accomplished with this work, highlighting our main contributions. Our main goal was to design a behavioral type system targeting the Java programming language, building on the work done in [Cai08, Mil08, Par11, CS13], by including support for protocol specification at the method declaration level, specifying finer grained contracts about the behavior of parameters and returned objects, object ownership and an object-based concurrency model.

A behavioral type system with support for ownership types

In [Par11] an object-oriented language with concurrency primitives was developed, allowing to verify correct object usage against a protocol specification, however, not supporting any form of aliasing.

Our work extends the approach of [Par11], while building on some key notions presented in [Cai08, CS13], providing support for ownership types, so as to account for the construction dynamic data-structures. We provide a way to annotate method declarations that state contracts between the interface and possible implementations, as well as support for the Java concurrency model.

Application of the behavioral type system to a mainstream language

Our prototype implementation of the type system targets a mainstream language, namely Java, and may help to disseminate the main ideas around behavioral separation types. Our application to Java is made possible by the use of standard Java tools, guaranteeing full integration with the development of Java programs. The developed type system is invoked upon compiling the source code, hence it can be used as a tool without extensive configuration.

Furthermore, the specification of usage protocols in interfaces and methods contributes for providing meaningful information to programmers about how to safely use the behaviorally annotated interfaces. The expressiveness of usage protocols is such that it will allow to distinguish synchronization or independence between methods.

1.5 Outline

In this chapter, we introduced the initial motivation for our work, including an initial description of previous work upon which we base our development. We then provided an initial example of the proposed solution, to give a general feel of how it is used and how does it relate to Java development, followed by the main contributions of our work.

In Chapter 2, we will introduce base techniques used in this dissertation. We begin by introducing Spatial-Behavioral types, the theoretical basis of our work, followed by other background approaches such as Separation Logic, that aims to reason about programs that manipulate memory, and Session Types, a type-based approach that aims to verify correctness properties of concurrent programs that communicate via message passing.

Afterwards, in Chapter 3 we give a brief overview of other approaches that also aim to verify correctness properties in concurrent and object-oriented programs, focusing on ownership control and protocol conformity.

In Chapter 4 we will define our behavioral type system, some auxiliary algorithms to the system, followed by an explanation of each of the typing rules.

Afterwards, in Chapter 5 we present the technical details of our solution, explaining how the verification of a Java program is done, from development to the verification itself. We use a rather simple example to highlight the main features of our solution. We then describe a more complex example consisting of the annotation and verification of Java's `List` interface. We will discuss a possible usage protocol for this interface, and provide two implementations using different programming idioms. This example will serve to explain sensitive implementation details regarding some familiar constructions that are not possible under our ownership model.

Finally, in Chapter 6 we give some concluding remarks about the work that was done in this dissertation, discussing some possibilities of future work.

2

Background

In this chapter, we present some base techniques used in this thesis. We start by describing Spatial-Behavioral Types (SBT) [Cai08], from which our work builds on, which models systems in a distributed object core calculus based on the π-calculus [MPW92]. Next we overview the fundamental concepts of Separation Logic (SL) [Rey02], an extension of Hoare logic to reason about heap allocated data-structures. We then describe an extension of SL to concurrency — Concurrent Separation Logic [O'H04]. (Concurrent) Separation Logic closely relates to SBT by attempting to statically control reference aliasing, which is essential to any form of static verification of concurrency control mechanisms. We finish by referring to Session types [Hon93, HVK98], also related to SBT, as another technique to discipline concurrent resource usage, via linear constraints. Concurrent Separation Logic and Session Types are an important resource to this work, since controlling concurrent behavior is a challenging task to which the aforementioned techniques have made significant and related contributions.

2.1 Spatial-Behavioral Types

In this section we present Spatial-Behavioral Types [Cai08], a typing discipline for a general distributed, concurrent and object-oriented model, where objects represent both entities as well as resources and types express *how* they must be used. Any usage given to an object that does not conform to this specification is erroneous and is liable to make the system get into an illegal state.

Each object is an aggregate of operations, state and tasks. Object names are passed around in communication with other objects. Tasks contain code to be run, and can be executed concurrently with other tasks as long as they are *independent*, i.e., if they don't

U, V	$::=$		(Types)
		`stop`	(Stop)
	$\mid$	$U \mid V$	(Spatial Composition)
	$\mid$	$U; V$	(Sequential Composition)
	$\mid$	$U \wedge V$	(Conjunction)
	$\mid$	U°	(Owned)
	$\mid$	$\mathtt{l}(U)V$	(Method)

Figure 2.1: Syntax of Spatial-Behavioral Types

compete for the same resources. Objects can also be composed into networks, where this composition is denoted as $O \mid O'$.

The proposed type system captures essential constraints on resource usage that arise in concurrent systems. The syntax of types is presented in Figure 2.1 and we now describe the meaning of each type in terms of expected behavior, relating it to certain properties of objects. An object satisfies `stop` if it is idle, i.e., if no operations are running in it. $U \mid V$ means that the object satisfies both U and V independently with respect to resource separation, i.e., behaviors U and V are independent and therefore can be exercised concurrently. An object satisfies $U; V$ if it satisfies U and V in sequence, i.e., if V is only satisfied after U. Note that while the previous type represents independent behavior, sequential composition can indicate just the opposite: there is some kind of temporal dependency when competing for resources, e.g., pushing an element in a stack before popping it. $U \wedge V$ is satisfied if an object simultaneously satisfies both U and V. An object satisfies U° if it satisfies U, but this view of U is exclusively owned. This means that the behavior U does not depend on any other on the system, even if there are aliases to the object. This means that it can, for instance, be stored in the local state. Finally, an object satisfies $\mathtt{l}(U)V$ if it offers a method `l` that, when an argument of type U is passed, returns something of type V. Note that the last two described types can be combined to specify transfer of ownership when passing something as an argument to a method. For instance, consider an object o that offers a method $\mathtt{put}(U^\circ)V$, requiring its argument to be owned. Thus, on the client side, calling $o.\mathtt{put}(x)$ causes loss of ownership of x. However, it could be regained if the type V is U°.

A subtyping relation for this system is also defined in [Cai08], capturing the natural substitution principle for behavioral types. For instance, the rule

$$(A; B) \mid (C; D) <: (A \mid C); (B \mid D)$$

(which relates to the exchange law of Concurrent Kleene Algebras [HMSW09]) allows us to derive a rule that expresses interleaving, e.g., $(A \mid B) <: (A; B)$. Subtyping provides flexibility to the type system, thus allowing to typecheck more programs, since it defines how a different behavior than expected can still be used safely.

We now describe the work of [CS13] that builds on the Spatial-Behavioral types to

provide rich forms of aliasing and sharing in a concurrent object-oriented setting, followed by Featherweight Concurrent Java [Par11] that aims to check concurrent object-oriented programs by developing a typing algorithm based on Spatial-Behavioral Types approach.

2.1.1 The Type Discipline of Behavioral Separation

In [CS13], the notion of behavioral separation is introduced as a type-based approach to discipline interference caused by aliasing and concurrency, building on the notion of behavioral types that we have presented in Section 2.1. In this work a behavioral type system is presented, where types bear resemblance in form and meaning to the ones described in [Cai08], which purpose is to verify the correctness of programs in a λ-calculus extended with mutable heap variables, records and concurrency primitives, such as threads and synchronization mechanisms. In this work, program assertions state behavioral constraints on runtime values instead of properties about the program state, enforcing type safety with regard to interference, in a concurrent setting where aliasing can occur, by imposing usage control disciplines.

As in [Cai08], types are a kind of protocol that describe the usage behavior of values. For example, the type

$$File \triangleq (open : 0 \mapsto 0); (read : 0 \mapsto \mathsf{str}); (close : 0 \mapsto 0)$$

describes the behavior of a file that allows reading once after opening, requiring it to be closed afterwards; thus, any value that is typed as $File$ must be used exactly as how we have specified. Notice that the functional type constructor is slightly different from what was presented previously: $U \mapsto V$ describes a function that is behaviorally independent of its argument U and returns a value of type V being passed the proper argument of type U.

Until now, types only describe single runtime values. In order to express richer constraints in program expressions, where multiple different values might be used, the notion of type assertion is introduced. These assertions not only state usage constraints on the free identifiers of an expression, but also specify fine-grained behavioral dependencies between values. For example, the type assertion

$$x : U \,;\, y : T$$

states that x must be used as specified by U and only after can y be used as T. Another interesting example is the type:

$$(f : (U \mapsto V) \,;\, x : U) \mid y : U.$$

In this case, both x and y have types that are compatible with the argument type of f. However, we cannot use x as an argument to the function, since it can only be used after

f has been used. Thus, the expression $f(x)$ would not be considered well typed. On the other hand, the type assertion tells us that y is behaviorally independent of f, fulfilling the non-interference constraint between functional type of f and its possible argument, and so $f(y)$ is well typed.

The proposed type system also allows for rich forms of aliasing and sharing by introducing the shared $!U$ and isolated $\circ U$ types. $!U$ types a resource that can be infinitely used in a separated manner, e.g. $U \mid U$, while $\circ U$ states that the value may be safely used according to U and that this usage is not subject to any other global constraints (which relates to the previously presented owned type U° [Cai08]).

Types for heap variables include primitives that represent usage, writing and reading capabilities. A freshly allocated heap variable has type var, that denotes any allowed usage on the variable after allocation. Some constraints are imposed on this usage, as any variable must first be assigned a value that allows some behavior and only then be used. As we will see later on, some of our work is based on these notions.

This work presents simple but powerful notions of behavioral separation and, in novel ways, tackles the problem of dealing with object aliasing and sharing in a concurrent setting, effectively supporting recurrent programming idioms in object-oriented languages.

2.1.2 Featherweight Concurrent Java

Featherweight Concurrent Java (FWCJ) is a concurrent programming language developed as an extension to *Featherweight Java* in [Par11], offering simple primitives for parallel execution. A type system for FWCJ is developed in [Par11] following closely the spatial-behavioral types approach, aiming to verify correct object usage in a concurrent setting.

Consider the example shown in Figure 2.2 (taken from [Par11]). Classes are declared in a Java-like manner. The difference is that class headers directly declares the only constructor together with the class fields. In the example, the only constructor is `Cell(elem:Int)` and `elem` is the only class field. Therefore, calling the constructor with an integer value as an argument causes `elem` to be initialized with that same value.

FWCJ interfaces extend Java interfaces with the type that specifies the behavior of objects. A class implementing an interface must conform to the specified behavior. The behavioral specification is done by using construct **`usage`** `B;` where `B` is a spatial-behavioral type which syntax resembles the one defined in [Cai08], but adopting a simpler notation for methods (just its name).

In Figure 2.3 we present a complete example written in FWCJ. The `Cell` interface declares the `get` and `set` methods, offering the behavior specified by `(set|get)*`, where `*` represents zero or more repetitions of the behavior, and `CellImpl` is simply an implementation of the `Cell` interface. The main program is defined with a **`main`** block after the class and interface declarations: a new `CellImpl` object `c` is created, initializing its `elem` field with 5. The final instruction executes in parallel the set (with argument 6) and get

```
class Cell(elem:int) {
    void set(n:int) {
        elem = n
    }
    int get(){
        elem
    }
}
```

Figure 2.2: Class declaration in FWCJ

```
interface Cell {
    void set(n:int);
    int get();

    usage (set|get)*;
}
class CellImpl(elem:int) implements Cell {
    void set(n:int) {
        elem = n
    }
    int get(){
        elem
    }
}
main{
    let c = new CellImpl(5) in
        (c.set(6) | c.get())
}
```

Figure 2.3: Complete example in FWCJ

methods.

The typing algorithm presented in [Par11] aims at statically verifying the correct usage of objects, i.e., that objects conform to the behavior specified in the interface. A subtyping relation is also defined by means of a labeled transition system: a type τ is subtype of another type τ', if every transition in τ' is simulated by a transition in τ. The typing then consists in analyzing the usage of every object in the program, via a type inference algorithm, and check if the type declared in the associated interface is a subtype of the one that is used.

In this work, we aim to extend the approach of [Par11] with support for an object-oriented concurrency model, ownership types and object type variables as well as fine-grain protocol specification at the method level. The rest of the chapter focuses on closely related techniques, namely Separation Logic and Session Types, that address similar challenges in verifying correctness of concurrent programs.

2.2 Separation Logic

Separation Logic (SL) is an extension of Hoare Logic [Hoa69], proposed by Reynolds et al. in [ORY01, Rey02]. SL aims to provide a way to reason about imperative programs that directly manipulate memory, aiming to improve Hoare Logic, in particular

$$\frac{\{\mathsf{true}\}\ x := 3\ \{x = 3\}}{\{\mathsf{true} \wedge y = 2\}\ x := 3\ \{x = 3 \wedge y = 2\}}$$

Figure 2.4: Derivation that fails in the presence of aliasing

$$\begin{array}{rcll} P, Q & ::= & \ldots & \text{(Program)} \\ & & x := \mathsf{cons}(\overline{E}) & \text{(Allocation)} \\ & | & \mathsf{dispose}\ E & \text{(Deallocation)} \\ & | & [E] := E & \text{(Mutation)} \\ & | & x := [E] & \text{(Lookup)} \end{array}$$

Figure 2.5: Imperative programming language

with respect to aliasing as can be illustrated by the following example. Consider the statement $x := 3$, for which we can obtain the derivation described in Figure 2.4: the triple $\{\mathsf{true}\}\ x := 3\ \{x = 3\}$ states that from any state (satisfies true) the statement $x := 3$ leads to a state where $x = 3$ is true. Then, starting from a state where $y = 2$ holds then the assignment $x := 3$ leads to a state where $x = 3$ and $y = 2$ hold. However, the derivation fails if x and y are aliases, i.e., if they point to the same memory location.

In Separation logic, usual assertions in Hoare triples are extended with separation connectives and predicates as well as some useful abbreviations:

- **emp** denotes the empty heap;
- the points-to predicate $E \mapsto E'$, that denotes a singleton heap that contains one cell, at address E with content E';
- the *separating* conjunction $A * B$ states that formulas A and B hold for separate parts of the heap;
- the *separating* implication $A \mathrel{-\!\!*} B$ states that if the current heap is extended with a separate part in which A holds, then B holds in such extended heap;
- $E \mapsto -$ is shorthand for $\exists z.(E \mapsto z)$;
- $E \mapsto E_0, \ldots, E_n$ is shorthand for $(E \mapsto E_0) * \cdots * ((E + n) \mapsto E_n)$.

Hoare's simple **while** language is also extended with statements to manipulate pointers through statements that explicitly mutate and read memory, illustrated in Figure 2.5. Construct $x := \mathsf{cons}(E_1, \ldots, E_n)$ allocates contiguous memory cells initialized with the expressions E_i, and stores in x the pointer to the first cell. Statement $\mathsf{dispose}\ E$ deallocates the memory cell with address E. Two new assignment statements are introduced: $[E] := E'$ updates the contents of memory cell at address E with content E'; and $x := [E]$ assigns to x the value at memory cell addressed at E.

Every inference rule defined in Hoare Logic holds in Separation Logic, except for the *constancy* rule

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}}$$

$$\text{(MUT)}\ \frac{}{\{E \mapsto -\}\ [E] := E'\ \{E \mapsto E'\}} \qquad \text{(DEALL)}\ \frac{}{\{E \mapsto -\}\ \mathsf{dispose}\ E\ \{\mathsf{emp}\}}$$

$$\text{(ALLOC)}\ \frac{}{\{x = x' \wedge \mathsf{emp}\}\ x := \mathsf{cons}(E_0, \ldots, E_n)\ \{x \mapsto E_0\{^{x'}/_{x}\}, \ldots, E_n\{^{x'}/_{x}\}\}}$$

$$\text{(LKUP)}\ \frac{}{\{x = v \wedge E \mapsto v'\}\ x := [E]\ \{x = v' \wedge (E\{^{v}/_{x}\} \mapsto v')\}}$$

Figure 2.6: Inference rules for memory manipulation statements

which allows us to extend a local specification to one containing any assertions that do not mention names of the program (P). This rule is, nevertheless, unsound in Separation Logic in the presence of aliasing, as the following example illustrates. If x points to some unspecified value after assigning value 3 we have that x points to 3, hence

$$\frac{}{\{x \mapsto -\}\ [x] := 3\ \{x \mapsto 3\}}$$

Then, using the *constancy* rule, we derive

$$\frac{\{x \mapsto -\}\ [x] := 3\ \{x \mapsto 3\}}{\{x \mapsto - \wedge y \mapsto 2\}\ [x] := 3\ \{x \mapsto 3 \wedge y \mapsto 2\}}$$

since y is not being mentioned by the command $[x] := 3$. This is clearly unsound when x and y are aliases. Therefore, the same kind of local reasoning present in Hoare Logic is not possible in Separation Logic in the presence of aliasing. The ability to reason locally about programs is regained through the use of the *separating* conjunction

$$\text{(FRAME)}\ \frac{\{A\}\ P\ \{B\}}{\{A * R\}\ P\ \{B * R\}}$$

which allows to extend the local specification with *independent* resources. Returning to the previous example, we may now prove the intended local reasoning as follows

$$\frac{\{x \mapsto -\}\ [x] := 3\ \{x \mapsto 3\}}{\{x \mapsto - * y \mapsto 2\}\ [x] := 3\ \{x \mapsto 3 * y \mapsto 2\}}$$

by applying the frame rule. The conclusion is valid because a situation in which x and y are aliases is not possible since $x \mapsto - * y \mapsto 2$ expresses that x and y are independent.

We now informally describe the rules for the commands introduced in Figure 2.5, illustrated in Figure 2.6. The inference rule for memory mutating assignment (MUT) says that if the memory cell at address E contains some unspecified value then, after the assignment, it contains value E'. The inference rule for memory cell deallocation (DEALL) states that starting from a singleton heap containing a memory cell at address E then, after deallocating it, we are left with an empty heap. The rule for memory allocation (ALLOC) says that, starting in a state in which the heap is empty, executing the command

leads to a heap composed of the newly allocated memory cells initialized with the values $E_k\{^{x'}/_x\}$, where $E_k\{^{x'}/_x\}$ is the result of substituting every occurence of x in E_k by a fresh x'. Finally, the rule for the memory lookup operation (LKUP) states that starting in a state in which x has value v and the memory cell addressed at E contains a value v', then executing the command changes the value of variable x to v'.

Separation logic is useful to reason about shared mutable data structures in the presence of aliasing. The *separating* conjunction is a simple but powerful addition to the assertion language as it explicitly describes resource independence. This notion is central to guarantee race freedom in a concurrent setting, as discussed in Subsection 2.3. Some extensions to Separation logic have been proposed, namely Concurrent Separation Logic [O'H04] presented next. It has also inspired the creation of some verification tools, for instance, jStar [DP08] is a static analysis tool based on Separation logic for object-oriented programs written in Java.

2.3 Concurrent Separation Logic

Concurrent Separation Logic (CSL) is a logic proposed by O'Hearn in [O'H04], as an extension of Separation Logic for concurrent programs. The separating conjunction is one of the most important primitives in Separation Logic, as it allows to reason about independent parts of the heap. This kind of reasoning fits perfectly in a concurrent setting, in which race freedom is ensured by construction if processes are confined to disjoint fragments of the heap.

Suppose we add the statement $P_1 \parallel P_2$ to the language presented in Section 2.2, in which both P_1 and P_2 are executed concurrently. The rule for parallel composition is the following

$$\frac{\{A_1\}\ P_1\ \{B_1\} \qquad \{A_2\}\ P_2\ \{B_2\}}{\{A_1 * A_2\}\ P_1 \parallel P_2\ \{B_1 * B_2\}}$$

and it states that if the two processes act in two separate parts of the heap, their parallel composition is interference safe, since their effect on the heap is also isolated. To illustrate this concept, consider the following derivation

$$\frac{\{x \mapsto -\}\ [x] := 3\ \{x \mapsto 3\} \qquad \{y \mapsto -\}\ [y] := 4\ \{y \mapsto 4\}}{\{x \mapsto - * y \mapsto -\}\ [x] := 3 \parallel [y] := 4\ \{x \mapsto 3 * y \mapsto 4\}}$$

Race freedom between both processes is ensured since, $x \mapsto - * y \mapsto -$ states that x and y are not aliases. On the other hand, a derivation for a racy program such as

$$\frac{}{\{x \mapsto - * x \mapsto -\}\ [x] := 3 \parallel [x] := 4\ \{x \mapsto 3 * x \mapsto 4\}}$$

is impossible since x cannot point to disjoint locations of the heap at the same time. Thus, in the CSL formalism, racy programs do not have valid proof derivations.

$$
\begin{array}{lll}
P, Q & ::= & \text{(Processes)} \\
 & \mathbf{0} & \text{(Inaction)} \\
 & \mid \quad a(x).P & \text{(Input)} \\
 & \mid \quad \overline{a}(x).P & \text{(Output)} \\
 & \mid \quad (\nu x)P & \text{(Name Restriction)} \\
 & \mid \quad P \mid Q & \text{(Parallel Composition)} \\
 & \mid \quad P + Q & \text{(Alternative Choice)}
\end{array}
$$

Figure 2.7: π-calculus syntax

Concurrent Separation Logic introduces a central notion of separation when reasoning about concurrent programs that manipulate shared memory, where two concurrent programs are independent if they act on separate parts of the heap. This view of resource independence relates to the one in [Cai08], where two behaviors are independent if they do not compete for the same resources.

2.4 Session Types

A session is a communication protocol between two or more parties, established via a shared public channel and carried out in a private dedicated channel. Session interactions follow a protocol describing the type and the order of the messages exchanged on the private channel. These protocols allow to specify alternative behavior, branching according to the labels of the messages that are exchanged. Session Types, introduced in [Hon93, HVK98], describe the sequence of messages to be exchanged and their branching based on labels, in a structured way. We first briefly present the π-calculus in order to illustrate the Session Types discipline.

2.4.1 π-calculus

The π-calculus was introduced in [MPW92] as an extension of CCS [Mil80]. It is a minimalistic language designed to specify and study concurrent systems, focusing on communication. The specification of systems in the π-calculus is very high-level, abstracting away details that pertain to the real implementation.

We present the syntax for the π-calculus in Figure 2.7. It provides the basic constructs to model concurrent interactions. Concurrency is modeled by the construct $P \mid Q$, which denotes a process comprising two active sub-processes P and Q. Communication is modeled by the input ($a(x).P$) and output actions ($\overline{a}(x).P$), where names can be passed around. In a process of the form

$$a(x).P \mid \overline{a}(v).Q$$

a synchronization occurs on channel a, leading to process $P\{^{v}/_{x}\} \mid Q$. Name restriction, denoted $(\nu x)P$, creates a fresh name x whose scope is P. However, since names can be

communicated, the scope of a name may dynamically change. For example, in process

$$p(x).\overline{x}(ack) \mid (\nu s)(\overline{p}(s).s(msg))$$

name s, private to a part of the process, is sent along channel p. Hence, its scope is enlarged to the whole process. So, the synchronization on channel p leads to process

$$(\nu s)(\overline{s}(ack) \mid s(msg))$$

The thread on the left only learns about channel s after synchronization.

The π-calculus is useful to express and verify properties of distributed concurrent systems at a high level of abstraction. Type systems are among the most popular verification techniques for the π-calculus. For instance, [Mil91] addresses the problem of arity mismatch in communication, i.e., where the number of arguments in an action does not match the one in its co-action. In [PS96] a type system is proposed that assigns types according to the input and output usage of channels. More specialized type systems have been proposed, e.g., in [CRR02] types that describe the behavior of processes are themselves abstract process models, others define a generic framework for π-calculus type systems [IK04], verify deadlock freedom [Kob06] or analyze the usage of resources [Cai08].

2.4.2 Session Types

A session typing of a π-calculus process involves precisely characterizing the usage of each channel x_i via a session type α_i. A typing environment, i.e., a collection of channel type associations is denoted by Δ. Then, the typing judgment is of the form $\Delta \vdash P$ which states that process P is well typed under Δ.

In the following, we give an overview of Session Types adopting the presentation of [HOP11], for the sake of simplicity. The most basic types are $![\alpha].\beta$, $?[\alpha].\beta$ and $\mathtt{end}$, where α and β are also session types. The first type describes a channel that first outputs something of type α and then is used as described in β, while the second first inputs something of type α and then proceeds as described in β. Finally, a channel typed as $\mathtt{end}$ has no behavior and we abbreviate the type $\alpha.\mathtt{end}$ simply as α. There is also a notion of a dual type of a type α, denoted by $\overline{\alpha}$ that is defined as

$$\overline{?[\alpha].\beta} = ![\alpha].\overline{\beta} \qquad \overline{![\alpha].\beta} = ?[\alpha].\overline{\beta} \qquad \overline{\mathtt{end}} = \mathtt{end}$$

For example, consider the π-calculus process:

$$\underbrace{pub(x).\overline{x}(\text{"}hi\text{"})}_{P} \mid \underbrace{\overline{pub}(ch).ch(msg)}_{Q}$$

The main idea is that we type each process individually. Then, the parallel composition of the two processes is well typed if the channels that occur in both processes have dual

$$
\begin{array}{rcll}
P, Q & ::= & \ldots & \text{(Processes)} \\
 & & a \triangleleft l.P & \text{(Selection)} \\
 & | & a \triangleright \{l_i : P_i\}_{i \in I} & \text{(Branching)} \\
 & & & \\
\alpha, \beta & ::= & \ldots & \text{(Session Types)} \\
 & & \oplus\{l_i : \alpha_i\}_{i \in I} & \text{(Select)} \\
 & | & \&\{l_i : \alpha_i\}_{i \in I} & \text{(Branch)}
\end{array}
$$

$$
\overline{\oplus\{l_i : \alpha_i\}_{i \in I}} = \&\{l_i : \overline{\alpha_i}\}_{i \in I} \qquad \overline{\&\{l_i : \alpha_i\}_{i \in I}} = \oplus\{l_i : \overline{\alpha_i}\}_{i \in I}
$$

Figure 2.8: Selection and Branching constructs and (dual) types

types of each other. This is a central notion of the session types formalism. Since the two types are dual, both processes will perform symmetrical actions and therefore will always evolve through synchronization on that channel.

Process P is expecting to receive, through pub, a channel that is then used to output a string. So, the usage given to parameter x is ![string] and therefore the one given to pub is the input of such channel, typed ?[![string]]. On the other hand, Q sends ch through the pub channel and then receives a message in ch. So, ch is used according to ?[string]. The argument type of pub is determined by the process that receives in it, shown above. Thus, in Q, we type pub as the output of the argument type, hence ![![string]]. Therefore, Q specifies two different usages to ch. The parallel composition of P and Q is, therefore, well typed since the types for each common channel are dual of each other: pub has types ?[![string]] and ![![string]]; ch has types ![string] and ?[string].

There are other types that allow for richer interactions (cf., [Vas09]). For instance, a usual situation in this kind of communication is that of a clients' choice between several services offered. This notion is referred to as *branching* and its dual action is *selection*. Suppose that we extend the syntax of π-calculus processes with labeled branching and selection constructs, with their respective session types, as shown in Figure 2.8. A process of the form $a \triangleleft l.P$ selects option l offered by a process prefixed at a. On the other hand, a process of the form $a \triangleright \{l_i : P_i\}$ offers the options with labels l_i, behaving as P_i when chosen. As an example, suppose we have a server that can add two numbers and return the result or return a number's symmetric value, encoded as the process

$$\overline{calc} \triangleright \{add : calc(x).calc(y).\overline{calc}(x + y), sym : calc(x).\overline{calc}(-x)\}$$

Then, we type $calc$ as $\&\{add$:?[int].?[int].![int], sym :?[int].![int]$\}$. Conversely, the respective client that only uses the add service, can be encoded as

$$calc \triangleleft add.\overline{calc}(3).\overline{calc}(3).calc(res)$$

where $calc$ is typed as $\oplus\{add$:![int].![int].?[int]$\}$. However, the two described types are not dual to each other. Nevertheless, it should not be mandatory that the client process is

ready to choose each of the services offered by the server process – in fact, choosing just one of them is a realistic scenario. Subtyping for session types, as proposed in [GH99], allows for these kinds of programs to be considered valid. Therefore, considering subtyping, we can conclude that the parallel composition of both client and server processes is well typed.

Session types were originally introduced for describing binary interactions, however extensions to consider multiparty interaction have been proposed [HYC08, CV10a]. Session types have been used outside of the π-calculus setting, such as functional programming [VGR06, GV10], object-oriented programming [GVR+10, CV10b], and in operating systems design [FAH+06].

2.5 On Session Types and Concurrent Separation Logic

Session Types (ST) is a typing formalism for statically verifying correctness properties of programs that communicate via message passing. On the other hand, Concurrent Separation Logic (CSL) is a formal system for reasoning about concurrent programs that communicate via shared memory. In [HOP11], it is argued that although both of these formalisms achieve independent reasoning of used resources, be they message channels or the heap, no formal link has been made between them. This is due to the languages that are used to model interactions: ST type message channels in π-calculus processes, while CSL reasons about imperative languages that directly mutate shared memory. In the aforementioned paper, a relation between ST and CSL is explored.

We now overview simplifications of both formalisms that are proposed in [HOP11] to simplify comparison, Baby Session Types (BST) and Basic Concurrent Separation Logic (BCSL).

BST is a stripped-down version of ST, only containing types for input, output and non-usage (as described in Subsection 2.4). The sequent of the BST typing system has the form

$$P \triangleright \Delta$$

that states that P is well typed under context Δ, where Δ is a collection of channel type associations. A subset of π-calculus is used to model processes, containing the inaction, output, input and composition constructs. A rule worthy of discussion is that of parallel composition:

$$\frac{P_1 \triangleright \Delta_1 \qquad P_2 \triangleright \Delta_2}{P_1 \parallel P_2 \triangleright \Delta_1 \circ \Delta_2}$$

where $\circ$ denotes multiset union. For a process in which there occurs synchronization on some channel, its type in Δ_1 should be dual of the one in Δ_2. The context that results from the union $\Delta_1 \circ \Delta_2$ then denotes a multiset in which two different, but dual types can occur for said channel. Based on this definition, a context Δ is *consistent* if for any channel name there are at most two occurrences – and when there are two, they must be dual of each other.

$$\text{(SND)}\ \frac{}{\{k :![\alpha].\beta * j : \alpha\}\ k!j\ \{k : \beta\}} \qquad \text{(RCV)}\ \frac{\{A * k : \beta\}\ P\ \{B\}}{\{A * k :?[\alpha].\beta\}\ k?j.P\ \{B\}}$$

Figure 2.9: Inference rules for BST instantiation

BCSL is inspired on Abstract Separation Logic [COY07] in which the propositions and primitive commands are abstracted away. In this simplification, primitive commands are left unspecified. However, some restrictions are imposed. For any two commands c_1 and c_2, the parallel and sequential composition, $c_1 \parallel c_2$ and $c_1; c_2$ respectively, must be defined. Thus, in comparison with CSL, the inference rules for BCSL are fewer. Local reasoning is retained, as well as the rule for parallel composition showing resource independence. On the other hand, in a particular instance of BCSL, additional rules may be defined for each primitive command.

An instantiation of BCSL with BST (BCSL/ST) is then defined, where pre and post-conditions are typing judgements in BST. The set of commands Com is extended with primitives for channel sending $k!j$ and receiving $k?j.C$. Notice that the former construct does not have the explicit continuation, as in traditional π-calculus processes. This is the case because $k?j.C$ binds j in P, whereas the continuation of $k!j$ does not and can be expressed by the sequential composition ;. Specific inference rules for both constructs are presented in Figure 2.9. A translation from BST to BCSL programs is then defined, denoted by $\langle\langle P \rangle\rangle$, and the soundness and completeness theorem of this transformation states that $P \triangleright \Delta$ is provable in BST if and only if $\{\Delta\}\ \langle\langle P \rangle\rangle\ \{\mathsf{emp}\}$ is provable in BCSL/ST.

The relation between the two formalisms provides some initial insights on how can we reason about message passing programs in a local and resource independent way. In conclusion, the relation that was explored in [HOP11] paves the way for further research in building the bridge between these two apparently disjoint formalisms.

2.6 Summary

In this section we presented some background techniques of this work. Spatial-Behavioral Types provide a typing discipline in which one can statically verify the correct usage of object against a defined protocol with a focus on object ownership. Separation Logic defines a proof system for imperative programs that operate on shared memory, focusing on resource independence. This notion is essential in a concurrent setting, in which race freedom is ensured as long as independent resources are used by the various processes. Finally, we gave an overview of Session Types, a formalism to statically determine correctness properties in programs that communicate through message passing in a structured way.

3

Related work

This thesis will ultimately focus on statically determining, using a behavioral type system, if concurrent object-oriented programs respect the the protocols that types prescribe, with a focus on ownership and protocol conformity. In this chapter we overview other approaches that also aim at verifying correctness properties in concurrent and object-oriented programs, focusing on ownership control and protocol conformity. We first describe ownership types [CPN98, CD02] that address the problem of object aliasing resorting to a notion of ownership, similarly to our type system. Finally, we overview the typestate concept that aims at statically verifying legal method calls on objects, considering its state, a notion of protocol conformity which relates to our approach. As an example, we overview the fundamental concepts of the typestate-oriented programming language *Plaid* [ASSS09], that considers a high-level view of object state.

3.1 Ownership Control

Aliasing is one of the most common features in mainstream object-oriented languages. In fact, without aliasing, implementing most data structures and design patterns would be hard. However, if changes are made to an object, other objects that refer to it may not be aware of these modifications. Thus, reasoning about object-oriented programs becomes a bigger challenge. Statically verifying correction and object encapsulation is hard because of the inability to determine what objects are aliases of others.

Based on the notion of ownership types introduced by in [CPN98], Drossopoulou and Clarke developed a type system [CD02] in order to tackle the problem of aliasing control. In this system, every object has an owner that can be another object or a special constant **`world`** that denotes the entire system. Notice that if an object A owns an object B, it

doesn't necessarily mean that A holds a reference to B.

In object-oriented programs, the owner relationship forms a graph with **`world`** as its root. Consider the example depicted in Figure 3.2, where W denotes **`world`**, and the remaining nodes denote distinct objects. An edge from a node N to M means that N owns M. So we can generalize ownership by stating that an object A owns B if all paths from **`world`** to B go through A. Therefore, no object outside of owner A can have a reference to B. So, for instance, we can actually conclude that A also owns E.

A Java extension was proposed in [CD02] to allow owner declarations. These are made in a way that resembles the declaration of generic types in regular Java classes. Consider the example of a simple linked list presented in Listing 3.1. As with generic types, we declare two different types of owners for the linked list: the first is the owner of the list, while the second is the owner of the data that is stored in the list. Notice that the owner of each link is the list itself.

Another important concept is that of an *effect*. In this proposed extension, each method is accompanied by an *effect clause*, that represents the boundaries - in the ownership graph - of which objects are accessed by it. This information is very useful, for instance, when determining if two methods can be called concurrently, without the need for any synchronization. There are two kinds of effect clauses: **`writes`**$(w_1, \ldots, w_n)$ and **`reads`**$(r_1, \ldots, r_m)$, where both w_k and r_k are objects. Additionally, finer-grain specificity is possible when describing where a method reads or writes in the ownership graph, through the use of the **`under`**$(o[.n])$ clause, where o is an object and n is an optional integer that represents the depth of the ownership tree that will be manipulated. For instance, **`under`**(o) states that a method will operate on o and the objects it owns, while **`under`**$(o.1)$ states that it will operate solely on objects owned by o. A method with a **`writes`**$(w_1, \ldots, w_n)$ clause can write in an object x if is a descendant of w_i in the ownership graph, for some i. Conversely, a method with a **`reads`**$(r_1, \ldots, r_m)$ clause can read an object x if it is a descendant of w_i or r_j, for some i or j. In conclusion, two methods write in disjoint parts of the ownership graph, then there can be no interference. In the proposed system, the objects that can be used in these clauses are **`this`** or any object field.

Ownership types also allow us to determine when one object is an alias to another. Suppose we have two lists, declared as `List<this,world>` and `List<this,this>` in class `Main`. We could conclude that they could never be aliases of each other, since the types that own the data of both lists are different.

In this work, Java-like classes are annotated in a similar way as generic types, with zero or more names that represent the owners of the objects instantiated from that class. With these annotations we are able to conclude if two objects can be aliases, given their owners. However, this translates to some annotation burden for the programmer. In our approach, we apply an linear model of ownership, in which every object can have just a single reference to it being held.

```
1   class List<owner, data> {
2       Link<this,data> head;
3
4       void addData(Data<data> d) writes under(this){
5           head = new Link<this,data>(d, head);
6       }
7   }
8
9   class Main<> {
10      List<this,world> list;
11
12      Main() writes this {
13          list = new List<this,world>;
14      }
15
16      void populate() writes under(this.1){
17          list.add(new Data<world>);
18          list.add(new Data<world>);
19      }
20
21      static void main() writes under(world) {
22          Main<> main = new Main<>;
23          main.populate();
24      }
25  }
```

Figure 3.1: An example demonstrating ownership types

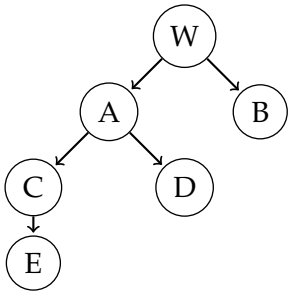

Figure 3.2: Ownership graph

3.2 Typestate

Object-oriented programming is a paradigm that has gained popularity over the years as it allows modeling real-world entities, but also to create reusable and modular libraries of abstract entities. However, these libraries are complex aggregates of objects and state. In mainstream languages, there is no way to enforce the correct order of method calls - or at least, how to let the user know it is not being respected. Any information about correct usage of a library is (if it exists at all) present in the documentation.

Typestate is a programming language concept that describes which operations are available under certain contexts, as well as how these operations change them. While traditional types determine what operations are permitted and how they are invoked, typestates determine a subset of these operations made available under a given context [SY86].

In the object-oriented paradigm, with the use of typestates we can effectively specify

```
state File{
    public final String filename;
}

state ClosedFile extends File{
    public void open() [ClosedFile >> OpenFile]{}
}

state OpenFile extends File{
    private FilePtr filePtr;

    public void read(...) {}
    public void write(...) {}
    public void close() [OpenFile >> ClosedFile]{}
}
```

Figure 3.3: File example in Plaid

```
if(file instate OpenFile){
    file.write();
    ...
}
```

Figure 3.4: Checking state in Plaid

which methods are available at each state and statically check if a certain method call in the program is allowed. A well-known example is that of files. Reading and writing from a file is only allowed after having opened it. After that, we may close the file and in this state, we can only (re-)open it.

Plaid is a typestate and object-oriented, concurrent programming language proposed by Aldrich et al. [ASSS09]. Its syntax resembles that of Java, but in *Plaid* typestates are first-class abstractions. Instead of being modelled by classes, objects are modelled by their changing states. Despite this difference, inheritance mechanisms for states are equal to that of Java's classes. Each object state has its own set of fields and operations that may cause the state to change. However, a state change in an object can only occur to any other subtype of its superstate.

An example containing the specification of a File API in *Plaid* is presented in Listing 3.3. The first state, `File`, is a base state that contains the fields that any `File` object contains, in any state. `ClosedFile` extends the `File` state and offers the `open` operation. This operation causes a state transformation to a new `OpenFile` state, through the transition annotation `[ClosedFile >> OpenFile]`. This new state also extends the `File` state, declaring a new field for the actual file pointer. In this state, we can now read from and write to the file. Notice that both `read()` and `write()` operations do not have a transition annotation and therefore do not change the state.

Dynamically testing if an object is in a given state is achieved throught the use of the **`instate`** operator, that resembles the Java counterpart **`instanceof`**, presented in Listing 3.4.

There is still the case where objects are passed as arguments to functions. For example, consider the *Plaid* code in Listing 3.5. The only way for the `read()` call to be valid is if

```
int readFromFile(ClosedFile f){
    openFile(f);
    int x = f.read();
    f.close();
    return x;
}
```

Figure 3.5: Invoking another function with an object as argument

```
void f(A x){
    ...
    g();
}
```

Figure 3.6: Aliasing in Plaid

the `openFile` method actually opens the file, i.e., leaves it in the `OpenFile` state. This is achieved by declaring the method as **`void`** `openFile(ClosedFile >> OpenFile f){}`, meaning that there is the guarantee that starting in the `ClosedFile` state, `f` will be left in the `OpenFile` state after the function returns.

Aliasing and Permissions

The same aliasing problems, described in the Section 3.1, are present in *Plaid* as tracking the state of aliased objects is undecidable [BBA08]. To illustrate this problem, consider the example in Listing 3.6. It could be the case that `x` is stored in a global variable and, afterwards, `g()` does something to change the state of that global variable, and tracking the state of `x` becomes harder.

To cope with aliasing, both concerns of aliasing and access control are combined. In *Plaid*, they consist in variable modifiers, that can occur in state fields or function arguments much like **`public`** and **`final`**, and they indicate in what ways an object can (or cannot) be aliased. We now present each of the modifiers and their meaning.

`pure` references provide read-only access to the object, stating nothing about the quantity and quality of other references;

`unique` means that there are no aliases to that object and therefore, the owner of the object has exclusive access to it;

`full` guarantees not only unrestricted state modification to the object's user, but also that any other reference to it must be read-only;

`immutable` states that unlimited and immutable aliases to this object can occur, but its state is never modified;

`shared` is the least restrictive permission. It gives modifying permissions, but states nothing about other types of aliasing to the object;

`none` is a reference that does not allow any kind of access, be it modifying or not. This is useful for modeling loss of ownership.

These modifiers give a clear semantics on how object references are shared and used, also in a concurrent setting. In *Plaid*, a dataflow graph can be constructed using these permissions, statically determining data dependencies.

Plaid is a programming language that implements the notion of typestate first presented in [SY86], handling state changes as a higher order operation on objects. To this end, methods are annotated with the state changes that will be made on the parameters when the same method is invoked. This relates to our approach of annotating method declarations in Java interfaces, specifying the behavior that is exercised on the arguments. However, in *Plaid*, since state is a first class abstraction as checking the current state of an object is performed through the use of a dynamic operator. In our approach, there is no notion of state, since the conformance of usage protocols is done statically.

3.3 Summary

In this chapter, we gave an overview of related approaches to static verification of concurrent object-oriented programs. In [CD02], ownership types provide a way to statically check correctness and resource independence in the presence of aliasing. Disjointness in owner types of objects allows to statically determine aliasing properties, while the disjointness in methods effects determines that there is no interference between them. Finally, typestate [SY86] extended the traditional types with a notion of context (or state), which determines what operations are available. A typestate-oriented programming language is proposed in [ASSS09], in which changes in object state are first-class abstractions. It allows constraint definitions on what forms of aliasing are permitted, as well as what state changes occur when a method uses an object parameter. State changes dynamically and, as such, the programmer must use primitives to check what state an object is in.

4

Core Language and Type System

Our work targets the verification of Java programs, namely to to verify correctness properties of programs with respect to usage protocol definitions, taking into account aliasing and the Java concurrency model, where the definition of usage protocols is possible in Java through the use of the annotation API.

To this end, we have identified a core fragment of Java focusing on some basic constructs, such as conditional, loops, method calls and assignments, and constructs that represent thread creation, starting and joining.

With the purpose of checking and ensuring correct object usage, we have defined a behavioral type system upon the aforementioned core fragment of Java, inspired by the work presented in [Cai08, Par11, CS13]. In our development, types describe the usage that can be exercised upon objects, and essentially these types are compared to the protocols that are declared in interfaces and methods. The comparison between types is supported by a subtyping relation, so that it is still possible to accept a different protocol than expected, therefore widening the number of programs addressed by our type system.

In this chapter, we will introduce our programming model, providing an overview of the syntax and semantics. Next, we present the developed type system, starting by presenting the intuitive meaning of each type. We discuss how we can infer concurrent behavior from a type, presenting an algorithm that aims to determine, given a behavioral type, what is the concurrent behavior that is exercised by different threads. Afterwards, we present each of the typing rules, providing a detailed description of its meaning. We conclude with some remarks regarding our solution.

$$
\begin{array}{lcll}
P & ::= & \overline{D}\ \mathbf{main}\{s\} & \text{(Program)} \\
D & ::= & & \text{(Definition)} \\
 & & \mathbf{interface}\ I\ \mathbf{usage}\ \mu\ \{\overline{M_d}\} & \text{(Interface)} \\
 & \mid & \mathbf{class}\ C(\overline{x:T})\ \mathbf{implements}\ I\ \{\overline{M}\} & \text{(Class)} \\
 & & & \\
\mu & ::= & & \text{(Protocol)} \\
 & & \mu\,;\mu & \text{(Sequence)} \\
 & \mid & \mu \mid \mu & \text{(Parallel)} \\
 & \mid & \mu \,\&\, \mu & \text{(Choice)} \\
 & \mid & \mu^{\circ} & \text{(Owned)} \\
 & \mid & \mu^{*} & \text{(Repetition)} \\
 & \mid & \mu! & \text{(Replication)} \\
 & \mid & m & \text{(Method)} \\
 & & & \\
T & ::= & \mathbf{int} \mid \mathbf{string} \mid \mathbf{boolean} \mid \mathbf{void} \mid Id \mid Id.\mu & \text{(Basic Types)} \\
 & & & \\
M_d & ::= & T\ m(\overline{x:T}) & \text{(Method Signature)} \\
 & & & \\
M & ::= & & \text{(Method)} \\
 & & T\ m(\overline{x:T})\ \{s\ [\mathbf{return}\ e]\} & \text{(Standard Method)} \\
 & \mid & \mathbf{sync}\ T\ m(\overline{x:T})\ \{s\ [\mathbf{return}\ e]\} & \text{(Sync Method)}
\end{array}
$$

Figure 4.1: Core Syntax (Definitions)

4.1 Syntax

In this language, a program is a set of interface and class definitions, accompanied by a `main` entry point. The top-level syntax for definitions is given in Figure 4.1.

Interface definitions are extended with the specification of a protocol that describes how objects implementing this interface should be used. The specification of a usage protocol is done according to the protocol language presented in Figure 4.1, which includes the operators presented in [Cai08, Par11]: method name m, sequential usage $\mu\,;\,\mu$, behavioral independence $\mu \mid \mu$, choice $\mu \,\&\, \mu$, behavioral ownership μ°, sequential repeated usage μ^{*} and replicated usage $\mu!$.

Classes are defined in a Java-like manner and must implement an interface. Similarly to [Par11], a class declaration also defines its fields and the only constructor that initializes all of them and method definitions also follow that of Java.

In this language, we define statements and expressions, the former of which do not denote any value. The body of a method is a block of statements, followed by a return statement (if the return type of the method is not **void**). The syntax of statements and expressions is defined in Figure 4.2. We include some constructions such as the method call $x.m(\overline{e})$, assignment $x = e$, the conditional **if** e **then** s **else** and while loop **while** e **do** s. We have added two constructs that represent the two main operations on Java thread objects: $\mathbf{start}(t)$ starts the execution of thread t, and $\mathbf{join}(t)$ suspends the execution of

s	$::=$		(Statement)
		$x.m(\overline{e})$	(Method Invocation)
	$\mid$	$x = e$	(Assignment)
	$\mid$	$\textbf{if } e \textbf{ then } s \textbf{ else } s$	(If)
	$\mid$	$\textbf{while } e \textbf{ do } s$	(While)
	$\mid$	$\textbf{start}(t)$	(Thread Start)
	$\mid$	$\textbf{join}(t)$	(Thread Join)
	$\mid$	$\textbf{thread } t = \textbf{thread}(s)$	(Thread Decl)
	$\mid$	$T\ x$	(Local Variable Decl)
	$\mid$	$\overline{s}$	(Statement Block)
e	$::=$		(Expression)
		x	(Identifier)
	$\mid$	v	(Value)
	$\mid$	$x.m(\overline{e})$	(Method Invocation)
	$\mid$	$e\ op\ e$	(Binary Operator)
	$\mid$	$\textbf{new } C(\overline{e})$	(Object Creation)
v	$::=$	$n \mid \textbf{true} \mid \textbf{false} \mid \textbf{null} \mid \text{STRING}$	(Value)

Figure 4.2: Core Syntax (Statements and Expressions)

the current thread until t finishes execution. Finally, we include the local variable declaration $T\ x$, where T indicates the type associated with name x. The syntax for our core language borrows from Java syntax in some constructs, while some of the constructs of the latter are mapped to our own through the use of source code analysis. For instance, we gather thread typed variables that are declared and intercept calls to `start` and `join` and map them to our own $\textbf{start}(t)$ and $\textbf{join}(t)$ constructs. Also, a declaration of the form `int x = 1;` is mapped to int $x;\ x = 1$ in our language.

Expressions always denote some value and as expected, together with basic values, we include the identifier, method call, binary operation and object instantiation expressions. Notice that the method call is included in both expressions and statements. As in Java, methods returning any type can be called and not have their results stored somewhere, case in which we will not perform any usage upon the result. There are cases however, where this is not permitted since the result of a method might require some mandatory usage.

The available types include the usual basic types, **int**, **boolean**, **string** and **void** as well as type identifiers Id that refer to the type of an interface or class. We extend this set of types, with a variant of the type identifier, $Id.\mu$. This type allows us to specify a portion of the usage that is associated with type Id. This is useful when we want to impose some constraints on the usage of an object, e.g., when it is passed as an argument to a method. As an example consider, the following method declaration that reads from a file that is passed as a parameter:

```
public void readHelper(File.read! f);
```

```
interface I usage m {
    void m(X.m1;m2 arg)
}

class C implements I {
    void m(X arg) {
        arg.m2();
        arg.m1();
    }
}
```

Figure 4.3: Contracts in Interfaces

We extend the `File` type with the `read!` annotation that constrains the usage of `f`, stating that is to be used as a read-only file. We can then be sure that, in the context of a method call, any file that is passed as an argument to this method will only be used as `read!`.

Considering this extension to the type identifier, the binding between a class and the interface it implements is more than just implementing the methods that are declared in it. There are contracts that need to be respected in the implementation regarding the declaration of the restricted capability of what is returned as well as the behavior exercised upon the parameters. Consider the program presented in Figure 4.3. The program should not be valid, as the implementation of method `m` does not fulfill the stipulated contract in interface `I` that `arg` must be used according to `m1;m2`.

There is a constraint regarding the programs that are addressed by our type analysis, related to thread usage throughout the program. Simply put, threads are a kind of local linear resource, which we *must* use after having being created. Thus, we separate thread declaration from regular variable declaration with the construct $\mathbf{thread}\, t = \mathbf{thread}(s)$, where we require that thread t is initialized upon declaration. Additionally, we require that every thread is started and joined exactly once in the scope following its declaration. The reason for this constraint is that we need to know the thread's body as well as the entire lifetime of the thread to be able to precisely track down concurrent behavior for each resource. So, if a thread lives longer than the scope of the method, we are unable to know what behavior is still being exercised after a method returns. Programs that do not conform to this restriction are discarded by our type system, as we will see later on.

4.2 Type System

In this section the developed type system will be presented and discussed, starting by the type language. We introduce and define a subtyping relation on types, that defines how a behavior that is different from the expected one can safely be used. We also define some auxiliary functions on types, such as the concurrent behavior extraction function, and proceed by giving a detailed description of the purpose of each rule.

τ, σ	$::=$		(SBT)
		stop	(Stop)
	$\mid$	$\mathsf{start}(t)$	(Thread Start)
	$\mid$	$\mathsf{join}(t)$	(Thread Join)
	$\mid$	$\mathsf{set}(\tau)$	(Reference write)
	$\mid$	$\tau \,;\, \tau$	(Sequential)
	$\mid$	$\tau \mid \tau$	(Parallel)
	$\mid$	$\tau \,\&\, \tau$	(Choice)
	$\mid$	τ^*	(Repetition)
	$\mid$	$\tau!$	(Replicated)
	$\mid$	τ°	(Owned)
	$\mid$	$m : \overline{\tau} \to \sigma$	(Method)
	$\mid$	$\overline{\tau} \Rightarrow \sigma$	(Class)
	$\mid$	γ	(Simple type)
γ	$::=$	$\mathsf{int} \mid \mathsf{boolean} \mid \mathsf{string}$	(Basic types)

Figure 4.4: Core Syntax (Spatial-Behavioral Types)

The intent of this behavioral type system, where types describe the way an object is used, is to check and ensure that every object in a program is used in a way such that is compatible with its associated protocol. To this end, the system is composed of several typing rules that check correct usage for each construction - from expressions to classes - of the presented programming language model, serving as building blocks to ensure the correctness of an entire program. From each construction, we extract the usage of each named resource, comparing it to its declared usage.

4.2.1 Behavioral Types

The syntax for types is presented in Figure 4.4. As previously mentioned a behavioral type τ describes the usage pattern of an object. Most of the types bear a resemblance in form and meaning to the ones that were presented in Section 2.1, with the addition of new types to account for the behavior that is exercised on Java threads, and, in turn, the behavior they themselves exercise in other objects. We will now proceed to precisely describe the meaning of each type.

The stop type indicates that the object is never used. The method type $m : \overline{\tau} \Rightarrow \sigma$ describes an object that is used by calling a method m that takes as parameters objects that are used described by each τ_i, respectively, and returns an object that is used according to the behavior described by σ.

The start and join types are two additional types that were created in order to extract concurrent behavior exercised by threads. The $\mathsf{start}(t)$ type describes an object that will be used by thread t. This type serves as a marker that indicates where the possibly concurrent behavior of t will begin, to help determine which behavior is activated concurrently with the one exercised by t. Analogously to the start type, the $\mathsf{join}(t)$ type

indicates where the point in which thread t stops using the object. In conjunction with the previously presented type, we are able to infer the behavior that is exercised concurrently with whatever behavior exists in-between occurrences of the start and join types (for the same thread).

The $\tau_1 \,;\, \tau_2$ type describes an object that is used according to τ_1, and only after that as prescribed by τ_2. While the previous type enforces a kind of temporal dependence on the usages τ_1 and τ_2, the parallel type is $\tau_1 \mid \tau_2$ is agnostic of any dependence that exists between τ_1 and τ_2. It then describes the behavior of an object that is used according to τ_1 and τ_2, in any order. This means that the behaviors that the τ_1 and τ_2 types capture do not interfere with each other and so both may be activated in parallel. The choice type $\tau_1 \,\&\, \tau_2$ describes an object that is either used as stipulated by τ_1 or τ_2, and this choice is made by the client using the object.

The τ° type types an object whose usage described by τ is completely owned. This means that there is no other reference to the object being held anywhere else in the program. We use this type in order to impose a kind of linearity constraint on aliasing. By default, basic types, method call results and object instantiations are owned. Basic types are owned since they are read-only values and are stateless, call results since in order to return something we must completely own it, and object instantiation due to the fact that every freshly created object is unique, hence, independent from any other.

The $\mathbf{set}(\tau)$ type indicates that we perform an assignment on the object, with an object that should be used as τ. This type is used to keep track of the many possible assignments that exist on a resource and allows us to check for possible races in concurrent accesses to the object.

The $\overline{\tau} \Rightarrow \sigma$ type is the type associated to each class indicating that it has a set of fields, each one used as described by each behavioral type τ_i, respectively. The instantiation of this class then returns an object that should be used as prescribed by the type σ.

Having presented the intuitive meaning of types, we proceed to the describe the reasoning behind the concurrent behavior analysis from an object's behavioral type, i.e., from the information we have about the delimitation of the behavior of a thread, how we obtain a type that states what behavior is activated in parallel with whatever use the thread makes of the object.

Analysis of Concurrent Behavior

Types related to threads, start and join, only specify the beginning and end of the behavior that is exercised by a single thread. So, intuitively, the behavior that occurs between, or concurrently with these limits is actually activated in parallel with the behavior that the thread exercises on the object. To this end, we have defined a function that given a behavioral type τ, a thread name t and the usage that t makes of an object o, τ_o, extracts the behavior that occurs between $\mathsf{start}(t)$ and $\mathsf{join}(t)$ in τ and maps it to a type τ' such that whatever behavior exists between the lifetime of the thread is specified concurrently

with τ_o. For example, suppose we type an object o with (we use the short-hand notation m to express the behavioral type $m : \emptyset \to \mathbf{stop}$):

$$(m_1 \,;\, \mathbf{start}(t) \,;\, m_2) \,;\, (m_3 \,;\, \mathbf{join}(t) \,;\, m_4).$$

We can see that m_2 occurs after the thread t is started, and that m_3 occurs before it is joined. So, this behavior should run concurrently with whatever behavior t exerts on o, and let us call that behavior τ_o. The behavioral type of o should then be mapped to

$$m_1 \,;\, ((m_2 \,;\, m_3) \mid \tau_o) \,;\, m_4$$

and note that m_1 and m_4 occur in instants that do not interfere with t so, the temporal dependencies between these behaviors and the lifetime of the thread are maintained. There are, however, cases in which we do not determine the most precise type as we will explain later on, and approximate in a sound way the intended mapping.

In order to present the definition of the function that realizes the behavior extraction, we first present some auxiliary functions. We define some functions that allow us to determine the set of thread names that are started or joined in a behavioral type.

Definition 4.2.1 (Thread names)**.** *We inductively define on types the functions* $\mathsf{st}(\tau)$ *and* $\mathsf{jn}(\tau)$*, that extract the set of thread names that are started or joined in* τ*, respectively, as follows. Additionally, we define the set of all thread names occurring in* τ*,* $\mathsf{th}(\tau)$*, as* $\mathsf{st}(\tau) \cup \mathsf{jn}(\tau)$*.*

$$\begin{aligned}
\mathsf{st}(\mathsf{start}(x)) &= \{x\} \\
\mathsf{st}(\tau_1 ; \tau_2) &= \mathsf{st}(\tau_1) \cup \mathsf{st}(\tau_2) \\
\mathsf{st}(\tau_1 \mid \tau_2) &= \mathsf{st}(\tau_1) \cup \mathsf{st}(\tau_2) \\
\mathsf{st}(\tau_1 \,\&\, \tau_2) &= \mathsf{st}(\tau_1) \cup \mathsf{st}(\tau_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{jn}(\mathsf{join}(x)) &= \{x\} \\
\mathsf{jn}(\tau_1 ; \tau_2) &= \mathsf{jn}(\tau_1) \cup \mathsf{jn}(\tau_2) \\
\mathsf{jn}(\tau_1 \mid \tau_2) &= \mathsf{jn}(\tau_1) \cup \mathsf{jn}(\tau_2) \\
\mathsf{jn}(\tau_1 \,\&\, \tau_2) &= \mathsf{jn}(\tau_1) \cup \mathsf{jn}(\tau_2)
\end{aligned}$$

We now move on to the definition of the algorithms used to analyze concurrent behavior in behavioral types. Firstly, we define algorithm $\mathcal{C}$, that, given a thread name t and a type τ, extracts the behavior that occurs before, concurrently and after an occurence of t, either $\mathsf{start}(t)$ or $\mathsf{join}(t)$.

Definition 4.2.2 (Algorithm $\mathcal{C}$)**.** *Given a behavioral type* τ *and a thread name* t*, we define algorithm* $\mathcal{C}^t(\tau)$ *on types by case analysis, as described in Figure 4.5.*

$$\mathcal{C}^t(\mathsf{start}(t)) = (\mathsf{stop}, \mathsf{stop}, \mathsf{stop})$$

$$\mathcal{C}^t(\mathsf{join}(t)) = (\mathsf{stop}, \mathsf{stop}, \mathsf{stop})$$

$$\mathcal{C}^t(\tau_1;\tau_2) = \begin{cases} (\tau', \pi, \tau''; \tau_2), & \text{if } t \in \mathsf{th}(\tau_1) \wedge \mathcal{C}^t(\tau_1) = (\tau', \pi, \tau'') \\ (\tau_1; \tau', \pi, \tau''), & \text{if } t \in \mathsf{th}(\tau_2) \wedge \mathcal{C}^t(\tau_2) = (\tau', \pi, \tau'') \end{cases}$$

$$\mathcal{C}^t(\tau_1 \mid \tau_2) = \begin{cases} (\tau', \pi \mid \tau_2, \tau''), & \text{if } t \in \mathsf{th}(\tau_1) \wedge \mathcal{C}^t(\tau_1) = (\tau', \pi, \tau'') \\ (\tau', \pi \mid \tau_1, \tau''), & \text{if } t \in \mathsf{th}(\tau_2) \wedge \mathcal{C}^t(\tau_2) = (\tau', \pi, \tau'') \end{cases}$$

$$\mathcal{C}^t(\tau_1 \,\&\, \tau_2) = \Big\{ (\gamma_1 \,\&\, \gamma_2, \pi_1 \,\&\, \pi_2, \sigma_1 \,\&\, \sigma_2), \quad \text{if } t \in \mathsf{th}(\tau_1) \cap \mathsf{th}(\tau_2) \wedge \mathcal{C}^t(\tau_i) = (\gamma_i, \pi_i, \sigma_i)$$

Figure 4.5: Algorithm $\mathcal{C}$

$$\mathcal{B}^t_\tau(\sigma) = \sigma, \text{ if } t \notin \mathsf{th}(\sigma)$$

$$\mathcal{B}^t_\tau(\tau_1;\tau_2) = \begin{cases} \mathcal{B}^t_\tau(\tau_1); \tau_2, & \text{if } t \in \mathsf{st}(\tau_1) \cap \mathsf{jn}(\tau_1) \\ \tau_1; \mathcal{B}^t_\tau(\tau_2), & \text{if } t \in \mathsf{st}(\tau_2) \cap \mathsf{jn}(\tau_2) \\ (\gamma_1; ((\sigma_1; \gamma_2) \mid \tau); \sigma_2) \mid \pi_1 \mid \pi_2, & \text{if } t \in \mathsf{st}(\tau_1) \cap \mathsf{jn}(\tau_2) \wedge \mathcal{C}^t(\tau_i) = (\gamma_i, \pi_i, \sigma_i) \end{cases}$$

$$\mathcal{B}^t_\tau(\tau_1 \mid \tau_2) = \begin{cases} \mathcal{B}^t_\tau(\tau_1) \mid \tau_2, & \text{if } t \in \mathsf{st}(\tau_1) \cap \mathsf{jn}(\tau_1) \\ \tau_1 \mid \mathcal{B}^t_\tau(\tau_2), & \text{if } t \in \mathsf{st}(\tau_2) \cap \mathsf{jn}(\tau_2) \end{cases}$$

$$\mathcal{B}^t_\tau(\tau_1 \,\&\, \tau_2) = \begin{cases} \mathcal{B}^t_\tau(\tau_1) \,\&\, \mathcal{B}^t_\tau(\tau_2), & \text{if } t \in \mathsf{st}(\tau_1) \cap \mathsf{jn}(\tau_1) \wedge t \in \mathsf{st}(\tau_2) \cap \mathsf{jn}(\tau_2) \\ \mathcal{B}^t_\tau(\tau_1) \,\&\, \tau_2, & \text{if } t \in \mathsf{st}(\tau_1) \cap \mathsf{jn}(\tau_1) \\ \tau_1 \,\&\, \mathcal{B}^t_\tau(\tau_2), & \text{if } t \in \mathsf{st}(\tau_2) \cap \mathsf{jn}(\tau_2) \end{cases}$$

Figure 4.6: Algorithm $\mathcal{B}$

We can now define the main behavior analysis algorithm.

Definition 4.2.3 (Algorithm $\mathcal{B}$)**.** *Given a behavioral type σ of an object, a thread name t and a usage τ that t exercises on the object, we define algorithm $\mathcal{B}^t_\tau(\sigma)$ on types by case analysis, as illustrated in Figure 4.6.*

The first case states that if t does not occur in the thread names of σ, then the result is exactly σ. For most of the other cases of the algorithm consist in recursive calls for the inner types for each of the binary operators. The case of parallel composition $\tau_1 \mid \tau_2$ consists in recursive calls for each of the types. Since a thread cannot be started and joined concurrently, we can safely assume that the same thread cannot occur in both τ_1 and τ_2. We apply the same reasoning for the choice type $\tau_1 \,\&\, \tau_2$, with the exception that it is possible that a thread can be started and joined in both branches of the type.

The interesting case appears in the sequence type $\tau_1 \,;\, \tau_2$. If a thread is started and

joined in τ_1, then τ_2 will remain unchanged since this behavior will surely be exercised after the thread is joined and therefore will not possibly interfere with the thread's behavior - and we apply the analogous reasoning for τ_1. The last case consists of when a thread is started in τ_1 and joined in τ_2. The next step is to determine what behavior occurs inbetween (and in parallel with) the starting and joining of a thread, and produce a type in which this behavior is $|$-composed with the behavior of the thread on an object. We need a way to determine what happens before, in parallel and after the occurrences of t in both τ_1 and τ_2, by using algorithm $\mathcal{C}$.

With this information we can now reconstruct the type. So, applying algorithm $\mathcal{C}$ to τ_1 and τ_2 yields two triples $(\sigma_1, \pi_1, \gamma_1)$ and $(\sigma_2, \pi_2, \gamma_2)$. γ_1 and σ_2 are the behavior that occurs inbetween starting and joining a thread. So, intuitively, if the behavior of the thread upon the object is τ_o, then $(\gamma_1 \,;\, \sigma_2) \mid \tau_o$ is the expected type containing precisely what behavior is exercised concurrently within the thread's lifetime. Now, σ_1 and γ_2 respectively occur before starting and after joining the thread, and thus we are able to construct the type $\sigma_1 \,;\, ((\gamma_1 \,;\, \sigma_2) \mid \tau_o) \,;\, \gamma_2$, respecting the temporal dependencies between these types and the thread's execution scope. Lastly, π_1 and π_2 are the behaviors that occur in parallel with the starting and joining of the thread. The limitation of this approach consists in how to precisely compose these two types with the previously constructed one and is related to the inability of the type language to express richer temporal dependencies. Consider two types $\tau = m_1 \,;\, (\mathsf{start}(t) \mid m_2)$ and $\sigma = (m_1 \,;\, \mathsf{start}(t)) \mid m_2$. Although they do not have the same meaning, the result of $\mathcal{C}^t(\tau)$ is exactly the same as that of $\mathcal{C}^t(\sigma)$:

$$\frac{\dfrac{\mathcal{C}^t(\mathsf{start}(t)) = (\mathsf{stop}, \mathsf{stop}, \mathsf{stop})}{\mathcal{C}^t(\mathsf{start}(t) \mid m_2) = (\mathsf{stop}, m_2, \mathsf{stop})}}{\mathcal{C}^t(\tau) = (m_1, m_2, \mathsf{stop})} \qquad \frac{\dfrac{\mathcal{C}^t(\mathsf{start}(t)) = (\mathsf{stop}, \mathsf{stop}, \mathsf{stop})}{\mathcal{C}^t(m_1 \,;\, \mathsf{start}(t)) = (m_1, \mathsf{stop}, \mathsf{stop})}}{\mathcal{C}^t(\sigma) = (m_1, m_2, \mathsf{stop})}$$

and so, the temporal dependency between m_1 and m_2 in τ is lost. Consequently both types π_1 and π_2 may or may not be after σ_1 and so the conservative and sound way to correctly place the extracted concurrent behaviors is $(\sigma_1 \,;\, ((\gamma_1 \,;\, \sigma_2) \mid \tau_o) \,;\, \gamma_2) \mid \pi_1 \mid \pi_2$.

4.2.2 Subtyping

We define the subtyping relation between behavioral types, denoted $\tau <: \sigma$, that follows the substitution principle. That is, whenever something of type τ is expected, we can safely provide something of type σ. Suppose we define a method with an argument such that we require that the behavior $m_1 \mid m_2$ is exercised upon it. Although this means m_1 and m_2 can be run concurrently, we don't actually need to have two threads running in parallel each calling m_i, for instance. Providing a serial behavior $m_1 \,;\, m_2$ should also be valid, since the prescribed parallel composition accepts any execution order - which relates to the expansion law [Mil89].

Based on the work done in [Par11], our subtyping relation is defined with the help

of a labeled transition system. A type τ is a subtype of another type σ, if and only if τ simulates σ, i.e., if σ exhibits a transition to σ' by some label then τ must exhibit a matching transition to τ', and τ' is a subtype of σ'. However, when the behavior of the supertype reaches the stop type, the subtype must also allow no behavior to be exercised.

In order to grasp the general idea of the subtyping algorithm, we will now move on to an example. Consider two types $\tau = m_1 \,\&\, m_2$ and $\sigma = m_1$. τ describes the behavior of an object that is used by calling either of m_1 and m_2, while σ describes one that is used by calling m_1. Intuitively, it should be valid if we use an object typed as τ if we use it by calling m_1, since it is one of the branches. Looking at τ, we can see that it has two distinct transitions, one for each branch:

$$m_1 \,\&\, m_2 \xrightarrow{m_1} \mathsf{stop}$$
$$m_1 \,\&\, m_2 \xrightarrow{m_2} \mathsf{stop}.$$

On the other hand, σ only has one transition $m_1 \xrightarrow{m_1} \mathsf{stop}$. So we can see that every transition on σ has a corresponding one in τ. We now need to check if the types after transition are still in the subtyping relation, i.e., if $\mathsf{stop} <: \mathsf{stop}$ - and this is true since both sides of the relation are the same. Therefore, we can conclude that $m_1 \,\&\, m_2 <: m_1$. However, this is not true for the symmetric case, since $m_1 \,\&\, m_2$ offers more transition possibilities than m_1. We present the set of rules that realize the labeled transition system.

Labeled Transition System

The labeled transition system is defined on types in Figure 4.8. The possible labels for type transitions are also behavioral types and they can be a method call, an assignment, an owned type, or a basic type usage.

Definition 4.2.4 (Transition on types)**.** *τ transitions by a type σ to a type τ', if we can derive $\tau \xrightarrow{\sigma} \tau'$ from the rules in Figure 4.8.*

Notice that the presented labeled transition system is non-deterministic, since we can construct a choice type of the form $m_1 \,;\, m_2 \,\&\, m_1 \,;\, m_3$, for instance, and we can see that there are two transitions with matching labels, but to completely different types. Thus we will have to take into account all transition possibilites when defining the subtyping relation.

The first three rules, LTS-METHOD, LTS-SET and LTS-BASIC reflect the transition in which the types match the label, and so all of the types transition to the stop type.

There are two rules defined for the $;$ operator. The first one, LTS-SEQ, a transition is made in the left type of the sequence, τ. So the resulting type is the sequential composition between the *residual* type after the transition and the type that occurs after τ. If, however the τ transitions to the stop type, then we can skip ahead to the continuation, and this case is represented in the rule LTS-SEQSTOP. So, for instance, $m_1 \,;\, m_2$ transitions by m_1 to m_2 since m_1 transitions to stop (according to LTS-METHOD).

For parallel composition there are four rules, including symmetric cases. In particular, LTS-PAR$_L$ reflects the case in which the left type transitions to some other type, and so the type evolves to a parallel type in which the left type is the residual after the inner transition, and the right type remains the same. The LTS-PARSTOP$_L$ rule, similarly to LTS-SEQSTOP, represents the case in which the one of the inner types transitions to the stop type. Thus in this case, there is a transition to a type that represents the remaining behavior in the opposite inner type.

The rules for the choice type $\&$ are LTS-CHOICE$_L$ and LTS-CHOICE$_R$. In LTS-CHOICE$_L$, the type transitions to whatever type remains after performing a transition in the left branch of the choice. LTS-CHOICE$_R$ is symmetric to the previous rule, where the transition is made on the right branch. Notice that since the choice is disjoint, choosing one of the branches causes the other one to be forgotten.

For repetition types, $\tau!$ and τ^*, we have rules LTS-REPL and LTS-STAR. The replicated type provides behavior to an infinite number of threads. So, the possibility for a thread to initiate the behavior the type offers must always be available. This notion is realized in rule LTS-REPL, in which if τ transitions to some type τ', then $\tau!$ transitions to $\tau' \mid \tau!$, so that the possibility for a thread to consume behavior τ is always possible. With sequential repetition, τ^*, there must be the possibility to perform τ any number of times, sequentially. So, if τ transitions to some type τ', then τ^* transitions to $\tau' ; \tau^*$, where the possibility to exercise τ is regained after it the residual τ' is exercised.

The rules for the owned type τ° are twofold. The LTS-OWNEDSTOP represents a transition whose label is the entire owned type. This transition will represent a situation in which there is loss of ownership. The LTS-OWNED rule states that if the inner type τ transitions by some label to τ', then the owned type transitions to the owned version of the residual type. The reasoning behind this rule is that if we use an object that is owned, its usage does not cause loss of ownership. So, the remaining behavior is still owned, which is consistent with the fact that the usage of something that is owned does not need to be used in its entirety, since there is no other reference to the object.

The last rule, LTS-CANSTOP is a special case of the sequence type, in which the left inner type does not have a requirement to be used - it can be a sequential or replicated repetition, for instance. This notion of stoppable behavior is defined in inductive predicate `canStop`, presented in Figure 4.7.

The CS-STAR and CS-REPL rules represent the possibility of the sequential and replicated repetitions be stopped, since they can be used zero or more times. The CS-OWNED rule reflects the possibility that an object typed as τ° can be used according to τ or not at all, and therefore the owned type can be stopped. The remaining rules extend the notion of stoppable behavior to the binary type operators. However, the choice type $\&$ only requires one of the branches to be able to stop, since we can just choose that one.

$$\text{CS-STAR}\ \ \texttt{canStop}(\tau^*) \qquad \text{CS-REPL}\ \ \texttt{canStop}(\tau!) \qquad \text{CS-OWNED}\ \ \texttt{canStop}(\tau^\circ)$$

$$\text{CS-CHOICE}\ \frac{\texttt{canStop}(\tau) \vee \texttt{canStop}(\tau')}{\texttt{canStop}(\tau \,\&\, \tau')} \qquad \text{CS-PAR}\ \frac{\texttt{canStop}(\tau) \quad \texttt{canStop}(\tau')}{\texttt{canStop}(\tau \mid \tau')}$$

$$\text{CS-SEQ}\ \frac{\texttt{canStop}(\tau) \quad \texttt{canStop}(\tau')}{\texttt{canStop}(\tau\,;\,\tau')}$$

Figure 4.7: canStop predicate

Subtyping Relation

Having defined the rules for the labeled transition system, we can now define the subtyping relation.

Definition 4.2.5 (Subtyping Relation)**.** *A type τ is a subtype of a type σ, read $\tau <: \sigma$, if $\tau = \sigma$ or if we can derive $\tau <: \sigma$ from the rules in Figure 4.9.*

The general subtyping rule SUB-GEN, is built upon the labeled transition system. It states that if every transition that a type σ is able to do is covered by a compatible one in τ, and if the residual types after both transitions are in the subtype relation, then $\tau <: \sigma$. This rule covers every binary and unary operators on types, and is accompanied by a set of simpler rules that define the subtyping relation for basic types. Notice that it isn't necessarily the case that σ will eventually reach the stop type; it can, for instance, be a sequential repetition type. We then assume that if $\tau <: \sigma$ are both repeatable behaviors then, if after checking every behavior, we return to $\tau <: \sigma$ we assume that σ would continue to be indefinitely simulated by τ and therefore we assume that they are in the subtype relation.

SUB-METHOD states that a method type m_1 is a subtype of a method type m_2 if they have the same name, the return type of m_1 is a subtype of m_2's and if each of the arguments in m_2 is a subtype of the corresponding one in m_1. These subtyping constraints follow the usual notion of covariance in the return types and contravariance in the method types, present in object oriented type systems.

SUB-CLASS is a similar rule, in which we see a class type as a method that given a set of fields, returns a fresh object with those fields initialized: field types must be related in a contravariant way, and the object types in a covariant way.

The rule for the $^\circ$ type, SUB-OWN, states that two owned types are subtypes of each other if their contents are also subtypes of each other.

The rule for the set type, SUB-SET, states that two assignments on a variable are subtype of each other if their contents are contravariant. The general idea behind this rule is that we do not allow something with less behavior than what is declared to be assigned

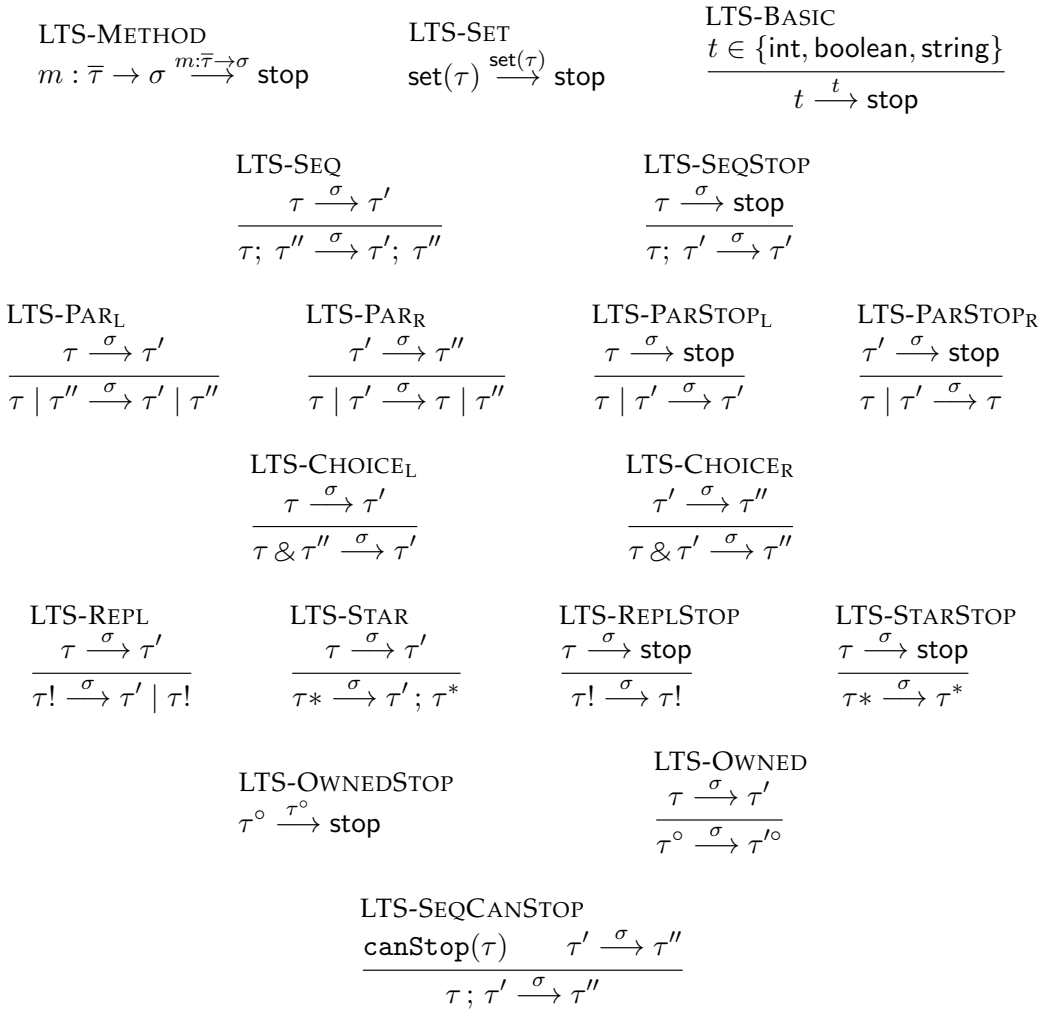

Figure 4.8: Labeled Transition System

to a variable, which relates to the same notion in Java, i.e., there should be an error when assigning something of type `List` to a variable of type `ArrayList`.

Since the labeled transition system is not defined on stop, we need a way to know if a behavior is able to stop. Thus the SUB-STOP rule realizes the notion that if, for any behavioral type τ, $\mathtt{canStop}(\tau)$ holds, then that means τ is able to stop and therefore is a subtype of stop.

We now present an example illustrating the application of the subtyping rules. Consider two types $\tau = (m_1 \,\&\, m_2)^*$ and $\sigma = (m_1\,;\, m_2)\,;\, m_1$.

By SUB-GEN, we need to check if every transition in σ is simulated by one in τ and if both residual types are still in subtyping relation.

$$\textsc{Sub-Method}\ \frac{\tau_r <: \sigma_r \qquad \sigma_i <: \tau_i \qquad i \in \{1,\ldots,n\}}{m : \overline{\tau} \to \tau_r <: m : \overline{\sigma} \to \sigma_r} \qquad \textsc{Sub-Class}\ \frac{\tau_r <: \sigma_r \qquad \sigma_i <: \tau_i \qquad i \in \{1,\ldots,n\}}{\overline{\tau} \Rightarrow \tau_r <: \overline{\sigma} \Rightarrow \sigma_r}$$

$$\textsc{Sub-Own}\ \frac{\tau <: \sigma}{\tau^\circ <: \sigma^\circ} \qquad \textsc{Sub-Set}\ \frac{\tau' <: \tau}{\mathsf{set}(\tau) <: \mathsf{set}(\tau')} \qquad \textsc{Sub-Stop}\ \frac{\mathtt{canStop}(\tau)}{\tau <: \mathsf{stop}}$$

$$\textsc{Sub-Gen}\ \frac{\forall \sigma', \gamma, \exists \tau', \delta \qquad \sigma \xrightarrow{\gamma} \sigma' \qquad \tau \xrightarrow{\delta} \tau' \qquad \delta <: \gamma \qquad \tau' <: \sigma'}{\tau <: \sigma}$$

Figure 4.9: Subtyping Relation

So by applying the LTS-SEQ rule,

$$\textsc{LTS-Seq}\ \frac{\textsc{LTS-SeqStop}\ \dfrac{\textsc{LTS-Method}\ \dfrac{}{m_1 \xrightarrow{m_1} \mathsf{stop}}}{m_1 \,;\, m_2 \xrightarrow{m_1} m_2}}{\sigma \xrightarrow{m_1} m_2 \,;\, m_1}$$

and by LTS-STARSTOP,

$$\textsc{LTS-StarStop}\ \frac{\textsc{LTS-Choice}_\textsc{L}\ \dfrac{\textsc{LTS-Method}\ \dfrac{}{m_1 \xrightarrow{m_1} \mathsf{stop}}}{m_1 \,\&\, m_2 \xrightarrow{m_1} \mathsf{stop}}}{\tau \xrightarrow{m_1} \tau}$$

and now we have to check if $\tau <: m_1 \,;\, m_2$. Again, we must apply the SUB-GEN rule. By LTS-SEQSTOP, we have that

$$\textsc{LTS-SeqStop}\ \frac{\textsc{LTS-Method}\ \dfrac{}{m_1 \xrightarrow{m_1} \mathsf{stop}}}{m_1 \,;\, m_2 \xrightarrow{m_1} m_2}$$

and τ transitions to τ again, by reapplying the same rules as in the previous case. Thus, we must now check that $\tau <: m_2$, by rule SUB-GEN yet again. By applying LTS-METHOD, we know that $m_2 \xrightarrow{m_2} \mathsf{stop}$, and

$$\text{LTS-METHOD}\ \frac{}{m_2 \xrightarrow{m_2} \mathsf{stop}} \qquad \text{LTS-CHOICE}_\text{R}\ \frac{}{m_1 \,\&\, m_2 \xrightarrow{m_2} \mathsf{stop}} \qquad \text{LTS-STARSTOP}\ \frac{}{\tau \xrightarrow{m_2} \tau}$$

So, we finally check if $\tau <: \mathsf{stop}$, which it is because $\mathtt{canStop}(\tau)$ holds (recall that τ is of the form τ'^*), and therefore we can apply rule SUB-STOP. Thus, $\tau <: \sigma$ and we can conclude that we can use an object according to σ where a usage τ is expected.

Our subtyping relation defines how a different behavior from what was specified can still be safely used, which is useful to confer some flexibility to our type system, e.g., if a protocol definition states that we can exercise some behavior τ^*, exercising just once or none at all should still be considered safe.

4.2.3 Typing Rules

Before moving on to the rules of the type system, we will introduce some key concepts depicting the general idea behind our development, as well as some auxiliary functions and operators. As described before, our main goal is to ensure that the behavior of every object is correct with respect to the protocols prescribed by the types. This behavior is described by a protocol using the operators that were presented in Figure 4.1. The idea is then, for each construct of the presented core language, to capture the usage that is exercised upon every used object (in the scope of that construct). After gathering the behavior covered by each object's scope, we then check it against what was prescribed in its interface. Thus, our typing sequent needs to account for what resources are used in a program, and how they are being used.

The typing sequent for expressions, methods and classes follows the template

$$\Delta \vdash X : \tau,$$

that states that X has type τ under environment Δ. The typing sequent for statements and programs follows the template

$$\Delta \vdash s$$

which, similarly to the previous sequent, states that s is well-typed under Δ but doesn't have a type annotation since neither constructions produce a value.

We use Δ to denote an environment that contains associations between objects names and behavioral types, $x : \tau$ describing the behavior τ exercised upon variable x, and we use $\mathsf{Dom}(\Delta)$ to denote the set of all names occurring in Δ. The syntax for environments is given by the grammar:

$$\Delta \quad ::= \quad \cdot \mid x : \tau, \Delta \quad \text{(Type Environment)}$$

We use the notation $\Delta(x)$ to denote the associated type with x in Δ. We assume every name in the environment is unique, and so each name identifies a single variable.

The type syntax admits terms that represent undesired behavior, e.g., a type $\mathsf{int} \mid \mathsf{set}(\mathsf{int})$ indicates that a resource of type int is being read and written upon at the same time. We impose some well-formedness rules on types so that we can reject types that are not conformant. To this end we define the predicate `hasWrite` on types,

$$\texttt{hasWrite}(\mathsf{set}(\tau)) \qquad \frac{\texttt{hasWrite}(\tau_1) \vee \texttt{hasWrite}(\tau_2)}{\texttt{hasWrite}(\tau_1 \mathbin{op} \tau_2)}$$

that indicates if an assignment, that is, an occurrence of the set type exists within a given type. Since the purpose is to find a single occurrence of an assignment, we recursively check if any of the inner types of a complex binary type has an occurence. With this predicate, we can now say that a behavioral type $\tau_1 \mid \tau_2$ is well-formed if $\neg\texttt{hasWrite}(\tau_1) \wedge \neg\texttt{hasWrite}(\tau_2)$ holds, meaning that no concurrent behavior should involve assignments. We can apply this sort of reasoning to the replication type $\tau!$. Since this type offers infinite independent copies of the behavior τ, no assignments can be made in it either, and thus $\tau!$ is well-formed if $\neg\texttt{hasWrite}(\tau)$ holds.

Another well-formedness constraint we impose on types is related to variable usage, based on the work done in [CS13]. Intuitively, we can only use a variable as long as it is assigned with a value that allows some behavior to be exercised. Additionally, after we fully use the prescribed protocol for the type of the variable we can only reuse its' name if we assign it a value that again allows some behavior. Generalizing, every variable must follow an assign-then-use policy that can be specified by our type language. So, if a variable has type T, its usage throughout the program must follow (with respect to subtyping) the type $(\mathsf{set}(T)\,;\,T)^*$, that represents the iterated assign-then-use behavior.

In order to ease the notation of the type rules, we extend previously defined type operators and behavior extraction function to typing environments. As we will see in the typing rules we will need to compose behavioral types in different environments, though for the same variable name. Algorithm $\mathcal{B}$ is also extended to typing environments in order to extract concurrent behavior for a set of resources that may be used by a thread. We extend the type operators to typing environments. Starting with the $;$ and $\mid$ operators,

$$(\Delta_1 \mathbin{op} \Delta_2)(x) = \begin{cases} \Delta_1(x) \mathbin{op} \Delta_2(x), & \text{if } x \in \mathsf{Dom}(D_1) \cap \mathsf{Dom}(D_2) \\ \Delta_i(x), & \text{if } x \in \mathsf{Dom}(D_i) \end{cases}$$

the aggregation of both environments for this type is an environment in which each x belonging to both environments is associated with a sequential composition between the types occurring in each one, in order. Otherwise, it is the type associated to the identifier, contained in one of the environments. However, the environment aggregation for the

choice type has a difference

$$(\Delta_1 \,\&\, \Delta_2)(x) = \begin{cases} \Delta_1(x) \,\&\, \Delta_2(x), & \text{if } x \in \mathsf{Dom}(D_1) \cap \mathsf{Dom}(D_2) \\ \Delta_1(x) \,\&\, \mathsf{stop}, & \text{if } x \in \mathsf{Dom}(D_1) \\ \mathsf{stop} \,\&\, \Delta_2(x), & \text{if } x \in \mathsf{Dom}(D_2) \end{cases}$$

that resides in the case when x only belongs to one of the environments. The composed type when $x \in \mathsf{Dom}(\Delta_1)$ is $\Delta_1(x) \,\&\, \mathsf{stop}$ since there is a branch in which we choose not to use x - and the same applies in the analogous case when $x \in \mathsf{Dom}(\Delta_2)$. For the unary type operators, the rules are straightforward and follow the equality $\Delta^{op}(x) = (\Delta(x))^{op}$.

Algorithm $\mathcal{B}$ is also extended to environments, being parametrized with a thread name t and, instead of an usage of an object by t, contains an environment representing the the usages of every object contained in the definition of t

$$\mathcal{B}^t_{\Delta_t}(\Delta) = \bigcup_{x \in \mathsf{Dom}(\Delta)} x : \mathcal{B}^t_{\Delta_t(x)}(\Delta(x))$$

Having defined the auxiliary functions and operators, we can now describe the inference rules for the type system, presented in figures 4.10, 4.11, 4.12, 4.14 and 4.15. Our approach for presenting the type system will be describing the typing rules for each of the expressions and statements, moving on to methods, classes and concluding with programs.

We begin by describing the typing rules for expressions, presented in Figure 4.10. In these rules we assume that every basic type is shared since they are immutable and therefore can be shared infinitely. Also we assume that every basic type, method call return type and instantiation return type are implicitly owned, and so we ommit the $^\circ$ annotation.

Null The rule for the **null** expression

$$\cdot \vdash \mathbf{null} : \mathsf{stop} \quad (\textsc{Null})$$

states that the null object does not offer any behavior, thus it is typed as stop. No resources are used in this expression and therefore it is well-typed under the empty environment $\cdot$.

Binary Operators The rules for binary operators all follow similar reasoning. For example, checking a binary arithmetic expression requires checking both sub-expressions

$$\frac{\Delta_1 \vdash e_1 : \mathsf{int} \qquad \Delta_2 \vdash e_2 : \mathsf{int} \qquad op \in \{+, -, /, *\}}{\Delta_1 \,;\, \Delta_2 \vdash e_1 \, op \, e_2 : \mathsf{int}} \quad (\textsc{Arithmetic Operator})$$

We require that both types are of type int and so, as usual, the compound expression is also of type int. Additionally, the resources that are used in e_1 and e_2 are

composed with the sequence operator in the conclusion, $\Delta_1 \,;\, \Delta_2$, and this is done for any of the binary expressions.

Identifier The rule for identifier is

$$x : \tau \vdash x : \tau \quad (\text{IDENTIFIER})$$

that states that x has type τ if the environment contains just the information that the identifier is used according to the type.

Method Call The method call expression $x.m(\overline{e})$ is typed according to the rule

$$\frac{\Delta_i \vdash e_i : \tau_i \qquad i \in \{1, \ldots, n\}}{x : (m : \overline{\tau} \to \sigma) \mid \Delta_1 \mid \ldots \mid \Delta_n \vdash x.m(\overline{e}) : \sigma} \quad (\text{METHOD CALL})$$

in which we check each of the arguments.

Behaviorally, we know that object x is used by calling method m, hence in the conclusion we have the environment containing an association between x and the behavioral type that realizes this usage, $x : (m : \overline{\tau} \to \sigma)$, in which $\overline{\tau}$ are the types of each argument and σ is the return type. Thus, since calling this method returns something of σ, the type of this expression is also σ.

Now, we impose independence between the arguments since we do not define a specific order of evaluation, and so the resulting environment consists of every Δ_i containing the resources of each e_i composed using the parallel $\mid$ operator. Additionaly, if the argument contains the same upon which we are performing the method call, then its usage should also be independent of the method call itself and therefore the resulting environment is $x : (m : \overline{\tau} \to \sigma)$ and each Δ_i composed with the parallel operator.

New Object Object creation is checked in a similar manner to the method call

$$\frac{\Delta_i \vdash e_i : \tau_i \qquad i \in \{1, \ldots, n\}}{C : (\overline{\tau} \Rightarrow \sigma) \mid \Delta_1 \mid \ldots \mid \Delta_n \vdash \mathbf{new}\ C(\overline{e}) : \sigma} \quad (\text{NEW OBJECT})$$

in which we check each of the arguments of the constructor that is being called.

Behaviorally, we know that class C is being used to instantiate an object that is used according to σ. The validity of the usage of σ according to the interface of C will be checked later on, as we will see. Thus, in the conclusion we include the environment that contains the association between C and the behavioral type that realizes the usage of C, $C : \overline{\tau} \Rightarrow \sigma)$, where $\overline{\tau}$ represents the types of each of the arguments and σ is the returned object type upon instantiation. Therefore, the type of this expression is also σ.

$$\frac{\Delta_i \vdash e_i : \tau_i \qquad i \in \{1, \ldots, n\}}{x : (m : \overline{\tau} \to \sigma) \mid \Delta_1 \mid \ldots \mid \Delta_n \vdash x.m(\overline{e}) : \sigma} \quad \text{(METHOD CALL)}$$

$$\frac{\Delta_i \vdash e_i : \tau_i \qquad i \in \{1, \ldots, n\}}{C : (\overline{\tau} \Rightarrow \sigma) \mid \Delta_1 \mid \ldots \mid \Delta_n \vdash \textbf{new}\ C(\overline{e}) : \sigma} \quad \text{(NEW OBJECT)}$$

$$\frac{\Delta_1 \vdash e_1 : \mathsf{boolean} \qquad \Delta_2 \vdash e_2 : \mathsf{boolean} \qquad op \in \{\&\&, ||\}}{\Delta_1 \,;\, \Delta_2 \vdash e_1\, op\, e_2 : \mathsf{boolean}} \quad \text{(LOGICAL OPERATOR)}$$

$$\frac{\Delta_1 \vdash e_1 : \mathsf{int} \qquad \Delta_2 \vdash e_2 : \mathsf{int} \qquad op \in \{+, -, /, *\}}{\Delta_1 \,;\, \Delta_2 \vdash e_1\, op\, e_2 : \mathsf{int}} \quad \text{(ARITHMETIC OPERATOR)}$$

$$\frac{\Delta_1 \vdash e_1 : \mathsf{int} \qquad \Delta_2 \vdash e_2 : \mathsf{int} \qquad op \in \{\leq, <, \geq, >\}}{\Delta_1 \,;\, \Delta_2 \vdash e_1\, op\, e_2 : \mathsf{boolean}} \quad \text{(INEQUALITY OPERATOR)}$$

$$\frac{\Delta_1 \vdash e_1 : \tau \qquad \Delta_2 \vdash e_2 : \tau \qquad op \in \{==, \neq\}}{\Delta_1 \,;\, \Delta_2 \vdash e_1\, op\, e_2 : \mathsf{boolean}} \quad \text{(EQUALITY OPERATOR)}$$

$$x : \tau \vdash x : \tau \quad \text{(IDENTIFIER)} \qquad \cdot \vdash \textbf{null} : \mathsf{stop} \quad \text{(NULL)}$$

Figure 4.10: Typing rules for expressions

When instantiating an object, like in a method call, we do not impose an order of evaluation of what is being passed as an argument to the constructor, hence every Δ_i that concerns the resources used in each e_i is composed using the $\mid$ operator. Any objects of the same class that are created in any of the arguments should be independent of the one we are creating (which is necessarily true since every newly created object is unique), and so the resulting environment consists of the parallel composition between each Δ_i and the environment that contains the information about the usage of C.

We will now describe the typing rules for statements, presented in Figure 4.11.

Block In the rule for the statement block

$$\frac{\Delta_1 \vdash s \qquad \Delta_2 \vdash l}{\Delta_1 \,;\, \Delta_2 \vdash s; l} \quad \text{(STATEMENT BLOCK)}$$

we check the statement at the beginning of the block as well as the rest of the block. The resulting environment is the sequential composition between Δ_1, that contains the resources used in s, and Δ_2 containing the resources used in the rest of the block.

Assignment When checking an assignment we must first check the type of the expression on the right-hand side:

$$\frac{\Delta \vdash e : \tau}{\Delta \mid x : \mathbf{set}(\tau) \vdash x = e} \quad (\textsc{Assignment})$$

We are performing an assignment on x with a resource of type τ and so the environment $x : \mathsf{set}(\tau)$ captures this behavior. Now, if x is in Δ its usage should be independent of the assignment and so the resulting environment is the parallel composition of both Δ and $x : \mathsf{set}(\tau)$ environments. Notice that according to this rule and the well-formedness rules presented earlier, we can not perform an assignment of the form $x = x + 1$.

If The conditional statement requires checking the condition expression and both branches:

$$\frac{\Delta \vdash e : \mathsf{boolean} \qquad \Delta_1 \vdash s_1 \qquad \Delta_2 \vdash s_2}{\Delta \,;\, (\Delta_1 \,\&\, \Delta_2) \vdash \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2} \quad (\textsc{If})$$

The reasoning behind how the environments are composed in the conclusion is related to how the execution of such a statement is usually done: firstly, we evaluate the condition, using the objects contained in Δ. Only afterwards, based on the truth value of the condition, the appropriate branch is executed, and we combine the resources of both branches according to $\Delta_1 \,\&\, \Delta_2$. So, the resulting environment consists of sequential composition of the environments in this order $\Delta \,;\, (\Delta_1 \,\&\, \Delta_2)$.

While Similarly to the conditional statement previously presented, the rule for the while loop requires that its condition and body are checked:

$$\frac{\Delta_1 \vdash e : \mathsf{boolean} \qquad \Delta_2 \vdash s}{\Delta_1 \,;\, (\Delta_2 \,;\, \Delta_1)^* \vdash \mathbf{while}\ e\ \mathbf{do}\ s} \quad (\textsc{While})$$

We first evaluate the condition, Δ_1, then if it is true, repeatedly execute the body of the loop Δ_2, followed by a re-evaluation of the loop's condition. Of course, if the condition is initially false, the body of the loop will not run. Thus, we sequentially compose the environments Δ_1 for evaluating the condition and $(\Delta_2 \,;\, \Delta_1)^*$ for repeatedly executing (zero or more times) the body of the loop and re-checking the condition.

Method Call This case is identical to the method call in the expressions. The key difference is that the result of the call will not be used since it is not being stored anywhere.

$$\frac{\Delta_i \vdash e_i : \tau_i \qquad i \in \{1, \ldots, n\}}{x : (m : \overline{\tau} \to \mathsf{stop}) \mid \Delta_1 \mid \ldots \mid \Delta_n \vdash x.m(\overline{e})} \quad (\textsc{Method Call})$$

Therefore, the behavioral type associated to the object upon which we invoke the method has stop as the return type.

Variable Declaration This rule is a special case of the statement block rule in which the first statement is a variable declaration. The reason for having the continuation after the declaration is that we need to know the entire scope of the variable so as to account for its entire usage, and since the language is a fragment of Java, we do not include the familiar **let**-binding construct.

$$\frac{\Delta, x : \tau \vdash l \qquad (\mathsf{set}(T^{\circ}); T)^{*} <: \tau}{\Delta \vdash (T\ x);\ l} \quad (\textsc{Variable Decl})$$

Analyzing the rest of the statement block tells us the resources that are used after the variable declaration. We know that the usage given to x exists in the environment that results from checking the continuation, hence the "split" $\Delta, x : \tau$, that separates the usage of x from the rest of the environment.

Intuitively, any usage of x must first be preceded by an assignment of something that conforms to type T. After using the variable (with respect to subtyping), we are able to perform another assignment and use it again. This behavior is captured by the the type $(\mathsf{set}(T^{\circ}); T)^{*}$. The extracted usage τ of x must then conform to this protocol, hence the side condition $(\mathsf{set}(T^{\circ}); T)^{*} <: \tau$. The resulting environment is Δ since at this point we know that if the side conditions hold, the use of variable x is considered to be correct and can no longer be used since at this point, its entire scope has been analyzed.

Thread Start and Join The environments for the $\mathbf{start}(t)$ and $\mathbf{join}(t)$ have some particularities.

$$\Delta_0, t : (start : \emptyset \to \mathsf{stop}) \vdash \mathbf{start}(t) \quad (\textsc{Thread Start})$$

$$\Delta_0, t : (join : \emptyset \to \mathsf{stop}) \vdash \mathbf{join}(t) \quad (\textsc{Thread Join})$$

For example, starting a thread t behaviorally means that t is used according to $start : \emptyset \to \mathsf{stop}$. Additionally, since t can exercise concurrent behavior with other resources, starting a thread should also produce an environment containing $x_i : \mathsf{start}(t)$, for each x_i used by t, which we will denote Δ_0. This reasoning is analogous in the case of the **join** statement, in which Δ_0 contains associations of the form $x_i : \mathsf{join}(t)$, for each x_i used by t.

Thread Declaration Like the variable declaration, thread declaration is also a special

case of the statement block.

$$\frac{\begin{array}{cc}\Delta, t : \tau \vdash l & \Delta' \vdash s \\ \Delta'' = \mathcal{B}^t_{\Delta'}(\Delta) & (start : \emptyset \to \mathsf{stop}\,;\, join : \emptyset \to \mathsf{stop}) <: \tau\end{array}}{\Delta'' \vdash (\mathbf{thread}\ t = \mathbf{thread}(s));\ l} \quad \text{(THREAD DECL)}$$

Checking the continuation of the statement block tells us the resources that are used after the declaration of the thread. We know that the usage that is exercised on t is given in the environment that result from checking the statement block, hence the "split" $\Delta, t : \tau$, where τ denotes this usage of t. Since we impose that every thread is to be started and joined after declaration we require that the usage τ of t is a supertype of $(start : \emptyset \to \mathsf{stop}\,;\, join : \emptyset \to \mathsf{stop})$.

Now, since the t is started and joined in the continuation of the statement block, we know that it can contain resources that are used by t which usage contains the types $\mathsf{start}(t)$ and $\mathsf{join}(t)$. So, applying algorithm $\mathcal{B}$ to environment Δ, with thread t and resource environment Δ' that is obtained by checking the thread body, yields the environment Δ'' that contains the appropriate concurrent usage for the resources of t.

Since the entire scope of t has been analyzed and is considered to be correct, the resulting environment is Δ''.

There are two rules for methods, void and non-void, presented in Figure 4.12.

Methods In a non-void method, we check the body s and the return expression e.

$$\frac{\begin{array}{ccc}\Delta_1 \vdash s & \Delta_2 \vdash e : \tau & \Delta = \Delta_1; \Delta_2 \\ T_i <: \Delta(x_i) & \tau <: T_r & i \in \{1, \ldots, n\}\end{array}}{\Delta \setminus \overline{x} \vdash T_r\ m(\overline{x : T})\ \{s\ \mathbf{return}\ e\} : (m : \overline{T} \to T_r)} \quad \text{(METHOD)}$$

Since the return statement comes after the body, we sequentially compose the respective environments, $\Delta_1\,;\,\Delta_2$, yielding the environment Δ. Now, the usage that is given in the method body to each argument must be a supertype of what is declared in the formal parameter, i.e., $T_i <: \Delta(x_i)$ and that the return type τ of the return expression is a subtype of the declared return type T_r. The environment in the conclusion is Δ, excluding the parameters, which we consider correct with respect to their declaration, and the method is typed as $m : \overline{T} \to T_r$.

Regarding the void methods, the same constraints on parameters apply but not the one for the return type, since there is no return statement. The method is then typed as $m : \overline{T} \to \mathsf{stop}$.

Class Checking a class requires checking each of the declared methods. We know that the environment of each method only contains occurrences of the class fields and

$$\frac{\Delta'' = \mathcal{B}^t_{\Delta'}(\Delta) \qquad \begin{array}{c}\Delta, t : \tau \vdash l \qquad \Delta' \vdash s \\ (start : \emptyset \to \mathsf{stop}\,;\, join : \emptyset \to \mathsf{stop}) <: \tau\end{array}}{\Delta'' \vdash (\mathbf{thread}\ t = \mathbf{thread}(s));\, l} \quad \text{(THREAD DECL)}$$

$$\frac{\Delta, x : \tau \vdash l \qquad (\mathbf{set}(T^\circ); T)^* <: \tau}{\Delta \vdash (T\ x);\, l} \quad \text{(VARIABLE DECL)}$$

$$\frac{\Delta_1 \vdash s \qquad \Delta_2 \vdash l}{\Delta_1\,;\,\Delta_2 \vdash s; l} \quad \text{(STATEMENT BLOCK)} \qquad \frac{\Delta \vdash e : \tau}{\Delta \mid x : \mathbf{set}(\tau) \vdash x = e} \quad \text{(ASSIGNMENT)}$$

$$\frac{\Delta_i \vdash e_i : \tau_i \qquad i \in \{1, \ldots, n\}}{x : (m : \overline{\tau} \to \mathsf{stop}) \mid \Delta_1 \mid \ldots \mid \Delta_n \vdash x.m(\overline{e})} \quad \text{(METHOD CALL)}$$

$$\frac{\Delta \vdash e : \mathsf{boolean} \qquad \Delta_1 \vdash s_1 \qquad \Delta_2 \vdash s_2}{\Delta; (\Delta_1 \,\&\, \Delta_2) \vdash \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2} \quad \text{(IF)}$$

$$\frac{\Delta_1 \vdash e : \mathsf{boolean} \qquad \Delta_2 \vdash s}{\Delta_1\,;\,(\Delta_2\,;\,\Delta_1)^* \vdash \mathbf{while}\ e\ \mathbf{do}\ s} \quad \text{(WHILE)}$$

$$\Delta_0, t : (start : \emptyset \to \mathsf{stop}) \vdash \mathbf{start}(t) \quad \text{(THREAD START)}$$

$$\Delta_0, t : (join : \emptyset \to \mathsf{stop}) \vdash \mathbf{join}(t) \quad \text{(THREAD JOIN)}$$

Figure 4.11: Typing rules for statements

class names that were instantiated. So, to determine the usage of each field we need to aggregate the set of resulting environments according to the class' interface. This aggregation yields an environment that contains how each field is used by imposing the protocol that is declared in the interface.

From the types of each method, we need to compose each of them according to the underlying protocol as well, to determine what is the object type that is returned every time we instantiate the class in question.

$$\frac{\begin{array}{c}\Delta_i \vdash M_i : \tau_i \qquad \mathsf{fieldUsage}(I, \overline{(\tau, \Delta)}) = (\sigma, \Delta') \\ (\mathsf{set}(T_j^\circ)\,;\, T_j^\circ)^* <: \mathsf{set}(T_j^\circ)\,;\, \Delta'(x_j) \qquad i \in \{1, \ldots, n\} \qquad j \in \{1, \ldots, m\}\end{array}}{\Delta' \setminus \overline{x} \vdash \mathbf{class}\ C(\overline{x : T})\ \mathbf{implements}\ I\ \{\overline{M}\} : \overline{T} \Rightarrow \sigma} \quad \text{(CLASS)}$$

This aggregation is defined with the `fieldUsage` function, inductively defined on protocols presented in Figure 4.13. This function takes two parameters, the protocol that is defined in interface I and the set of pairs containing the type and environment resulting from typechecking each of the methods.

Having the object type σ and protocol-aggregated environment Δ', we can now check the correct usage of each field. Similarly to the local variable declaration,

$$\frac{\begin{array}{ccc}\Delta_1 \vdash s & \Delta_2 \vdash e : \tau & \Delta = \Delta_1; \Delta_2 \\ T_i <: \Delta(x_i) & \tau <: T_r & i \in \{1, \ldots, n\}\end{array}}{\Delta \setminus \overline{x} \vdash T_r\ m(\overline{x : T})\ \{s\ \mathbf{return}\ e\} : (m : \overline{T} \to T_r)} \quad (\textsc{Method})$$

$$\frac{\Delta \vdash s \qquad T_i <: \Delta(x_i) \qquad i \in \{1, \ldots, n\}}{\Delta \setminus \overline{x} \vdash \mathbf{void}\ m(\overline{x : T})\ \{s\} : (m : \overline{T} \to \mathsf{stop})} \quad (\textsc{Void Method})$$

Figure 4.12: Typing rules for methods

$$\begin{aligned}\texttt{fieldUsage}(m, \overline{(\tau, \Delta)}) &= (\tau_m, \Delta_m) \\ \texttt{fieldUsage}(\mu_1\ op\ \mu_2, \overline{(\tau, \Delta)}) &= (\tau_1\ op\ \tau_2, \Delta_1\ op\ \Delta_2) \\ \texttt{fieldUsage}(\mu^{op}) &= (\tau^{op}, \Delta^{op})\end{aligned}$$

Figure 4.13: `fieldUsage` function

any usage of the field must be preceded by an assignment. The difference is that the usage of the field by the class is completely owned, hence any usage must be a supertype of $(\mathsf{set}(T_j^\circ)\,;\, T_j^\circ)^*$. We extract the usage of each field from the environment, $\Delta'(x_j)$, but we can assume that the field is always initialized since there is a single constructor that initializes every field. Therefore, the usage to be checked is actually $\mathsf{set}(T_j^\circ)\,;\, \Delta'(x_j)$. In the conclusion, the environment that contains the resources used by the class is Δ' excluding every field, since we have checked their correctness. Lastly, the class is typed as $\overline{T} \Rightarrow \sigma$.

Finally, the rule for programs is presented in Figure 4.15.

Program Verifying a program consists in checking each class and the main entry point. Notice that the environment that results from checking a single class only has names that correspond to classes in the program. This is also the case for the main statement. So the objective now is to check if every instantiation of a class follows the specified protocol. We compose with the parallel operator the environments that contain the class names used in every other class, yielding environment Δ'. This is because initially, every created object for a given class is independent from the rest.

For each class C_i, we check if its class type τ_i is compatible with its occurrence in Δ', $\Delta'(C_i)$. We can think of a class as an object factory that offers the possibility to infinite threads to create new objects, and so the extracted usage for each class $\Delta'(C_i)$ must be a supertype of the replicated version of τ_i, hence $\tau_i! <: \Delta'(C_i)$.

$$\frac{\begin{array}{c}\Delta_i \vdash M_i : \tau_i \qquad \mathsf{fieldUsage}(I, \overline{(\tau, \Delta)}) = (\sigma, \Delta') \\ (\mathsf{set}(T_j^\circ)\,;\, T_j^\circ)^* <: \mathsf{set}(T_j^\circ)\,;\, \Delta'(x_j) \qquad i \in \{1, \ldots, n\} \qquad j \in \{1, \ldots, m\}\end{array}}{\Delta' \setminus \overline{x} \vdash \textbf{class}\ C(\overline{x : T})\ \textbf{implements}\ I\ \{\overline{M}\} : \overline{T} \Rightarrow \sigma} \quad (\textsc{Class})$$

Figure 4.14: Typing rule for class

$$\frac{\begin{array}{c}\Delta_i \vdash C_i : \tau_i \qquad \Delta \vdash s \qquad \Delta' = \Delta \mid \Delta_1 \mid \ldots \mid \Delta_n \\ \tau_i! <: \Delta'(C_i) \qquad i \in \{1, \ldots, n\}\end{array}}{\cdot \vdash \overline{I}\ \overline{C}\ \textbf{main}\{s\}} \quad (\textsc{Program})$$

Figure 4.15: Typing rule for program

4.3 Summary

In this chapter, we have presented a core programming model that aims to capture essential constructs from the Java programming language, also building upon the work done in [Par11]. We have included, to some extent, support for the object-based thread model, where we require the thread to be initialized upon declaration, as well as being started and joined within the scope of the declaration. This restriction was required in order to fully determine the scope of concurrent behavior on an object. Starting a thread inside a method but not joining it causes its behavior to exceed the lifetime of the method itself and we would lose the ability to pinpoint what is the behavior exercised on argument objects, since there might be some usage being done even after the method returned.

We have also included support for ownership types, although enforcing linearity constraints, contrasting with the possibility of shared behavior independence [Cai08] that allows for richer but controlled forms of aliasing.

Focusing on the presented core language, we have designed a behavioral type system whose fundamental purpose is to extract the usage of every object in a program and checking it against the protocols in its declared interfaces. A program is then considered correct if every object's - possibly concurrent - usage is valid with respect to a subtyping relation, built upon a labeled transition system defined on types. For each declared variable/field, we impose a write-then-use sequential behavior, that ensures that after an object is completely used, using it again is only allowed after reassigning a new value that is able to fulfill the declared capabilities.

5

Application to Java

One of the contributions of this dissertation is a prototype implementation of the previously presented type system and applying it to the Java programming language. Our initial goal was to make the implementation completely integrate with any Java program, by using standard Java APIs. By using the Pluggable Annotation Processor API released with Java 6.0, we are able to write an Annotation Processor that analyzes the Java source files, whose code is run in the annotation processing phase, after the parsing phase. The ability to define usage protocols and specify a portion of a protocol in method arguments and return types is accomplished by the use of standard Java annotations. Throughout this chapter we will present the main features of the implementation with a paper reviewing example, showing how we can typecheck a Java program. We proceed by discussing a possible implementation of a `java.util` interface, concluding with some remarks regarding technical details and limitations.

5.1 Architecture

In `javac`, the annotation processing phase is run after parsing each file, until there are no more files to process, as depicted in Figure 5.1. The Process Annotations phase calls upon every Processor whose name is passed as argument to `javac`. We have defined our own Annotation Processor whose function is shown in Figure 5.2. For each parsed source file, we obtain its read-only abstract syntax tree representation by using the Java Compiler Tree API, and convert it to our own core representation. After all files were processed, we then invoke the our typechecker to check the program.

Annotations can be persisted at various phases of the compilation process. In our case, we only require that the annotations exist during the annotation processing phase,

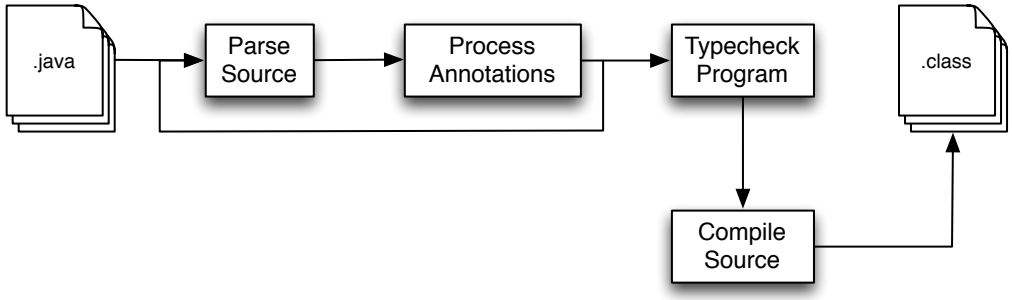

Figure 5.1: `javac`'s different phases

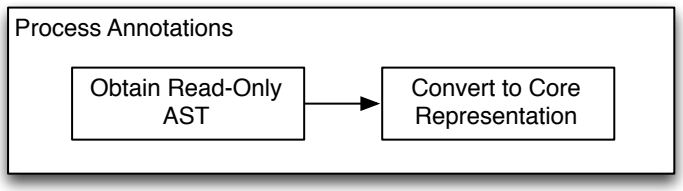

Figure 5.2: Annotation Processing

i.e., when the code is compiled there is no need to keep the metadata associated with each of the annotations. We were then able to provide an implementation of the type system that seamlessly integrates with the Java Compiler, that is invoked by a custom Annotation Processor after analyzing the Java code.

5.2 Typechecking

Our type system is invoked by means of a custom Annotation Processor. The Java source code is analyzed by using the Java Compiler Tree API, that provides us a read-only abstract syntax tree (AST) of the entire compilation unit. The read-only constraint was somewhat restrictive. Thus we have implemented the AST representation for the core language and using the Compiler Tree API's visitor classes we have developed a partial translation from the Java AST to the core language AST, having then defined the typechecker on this simplified AST. All of the constructions presented in Section 4.1 have their equivalents in Java, and their translation is straightforward. Additionally, we have included support for other constructions that are not presented in the core language syntax, such as `for` loops.

Both the typechecker and AST were implemented in the Scala multi-paradigm programming language that offers full interoperability with Java, as well as functional programming facilities that are well-known in the design of type systems, allowing for an elegant and compact implementation. This interoperability allows us to write Java code that translates the Java AST to our own, and call the main Scala typechecking function

from the annotation processor. This way, the typechecking of a program is made on the Scala side, maintaining a way to issue typechecking errors to `javac` by using standard Java classes.

5.3 Paper Reviewing Example

Consider an interface that represents a scientific paper that can be read and annotated with comments, presented in Figure 5.3. When defining the interface and its protocol, we are imposing some constraints on whatever the implementation of `Paper` might be. Since the `read` method must only read the textual information of the paper, it can't perform any changes to the internal state, hence the usage of this method can be replicated (`read!`). On the other hand, the `annotate` method will perform changes to the paper's content, and so it cannot be called concurrently with the `read` method. Additionally, we would like the paper to be read and reviewed several times, since there can be more than one reviewer. Thus, the protocol that respects these constraints is `(read! & annotate)*`.

Our annotation processor will then analyze the abstract syntax tree that corresponds to the `Paper` interface, producing its equivalent in Core representation:

$$
\begin{array}{l}
\textbf{interface}\ \mathit{Paper}\ \textbf{usage}\ (\mathit{read}! \,\&\, \mathit{annotate})^*\ \{ \\
\quad \textbf{string}\ \mathit{read}(); \\
\quad \textbf{void}\ \mathit{annotate}(\mathit{cmts} : \textbf{string}); \\
\}
\end{array}
$$

A possible implementation of the `Paper` interface could be the `PaperImpl` class presented in Figure 5.4. This simple class merely contains a `String` field representing the paper's content. The translation from this Java class to Core is as follows:

$$
\begin{array}{l}
\textbf{class}\ \mathit{PaperImpl}(\mathit{content} : \textbf{string})\ \textbf{implements}\ \mathit{Paper}\ \{ \\
\quad \textbf{string}\ \mathit{read}()\{ \\
\quad\quad \textbf{return}\ \mathit{content}; \\
\quad \} \\
\quad \textbf{void}\ \mathit{annotate}(\mathit{cmts} : \textbf{string})\{ \\
\quad\quad \textbf{string}\ \mathit{cont}; \\
\quad\quad \mathit{cont} = \mathit{content}; \\
\quad\quad \mathit{content} = \mathit{cont} + \mathit{cmts}; \\
\quad \} \\
\}
\end{array}
$$

Now that we have the core representation of the interface and class, we can now check its correctness. As an example, let us typecheck the *annotate* method. In order to simplify the derivation, we will begin by checking the each statement in the method's body individually and then compose the results.

$$\textsc{Assignment}\ \frac{\textsc{Identifier}\ \dfrac{}{content : \mathsf{string} \vdash content : \mathsf{string}}}{cont : \mathsf{set}(\mathsf{string}), content : \mathsf{string} \vdash cont = content} \tag{5.1}$$

$$\textsc{Assignment}\ \frac{\textsc{BinOp}\ \dfrac{\textsc{Identifier}\ \dfrac{}{cont : \mathsf{string} \vdash cont : \mathsf{string}} \qquad \textsc{Identifier}\ \dfrac{}{cmts : \mathsf{string} \vdash cmts : \mathsf{string}}}{cont : \mathsf{string}, cmts : \mathsf{string} \vdash cont + cmts : \mathsf{string}}}{content : \mathsf{set}(\mathsf{string}), cont : \mathsf{string}, cmts : \mathsf{string} \vdash content = cont + cmts} \tag{5.2}$$

Let Δ_1 and Δ_2 equal the resulting environments from the derivations in 5.1 and 5.2, respectively. Checking the statement block comprised of both statements then yields:

$$\textsc{Block}\ \frac{\dfrac{5.1}{\Delta_1 \vdash cont = content : \mathsf{string}} \qquad \dfrac{5.2}{\Delta_2 \vdash content = cont + cmts}}{\Delta_1 \,;\, \Delta_2 \vdash \begin{pmatrix} cont = content; \\ content = cont + cmts \end{pmatrix}} \tag{5.3}$$

Where $\Delta_1 \,;\, \Delta_2$ denotes the sequential composition between the two environments, that contains the associations: $content : (\mathsf{string} \,;\, \mathsf{set}(\mathsf{string}))$, $cont : (\mathsf{set}(\mathsf{string}) \,;\, \mathsf{string})$ and $cmts : \mathsf{string}$. We now must check the correctness of the $cont$ local variable.

$$\textsc{Variable Decl}\ \frac{\dfrac{5.3}{\Delta_1 \,;\, \Delta_2 \vdash \begin{pmatrix} cont = content; \\ content = cont + cmts \end{pmatrix}} \qquad \dfrac{(\Delta_1 \,;\, \Delta_2)(cont) = (\mathsf{set}(\mathsf{string}) \,;\, \mathsf{string})}{(\mathsf{set}(\mathsf{string}) \,;\, \mathsf{string})^* <: (\mathsf{set}(\mathsf{string}) \,;\, \mathsf{string})}}{\Delta_3 \vdash \begin{pmatrix} \textbf{string}\ cont; \\ cont = content; \\ content = cont + cmts \end{pmatrix}} \tag{5.4}$$

where Δ_3 does not contain $cont$. This resulting environment then contains the names $content$, which is a class field and $cmts$, the method's argument, associated to the use that is exercised upon them by the entire method body. Having checked the statement block that defines the method's body (that we abbreviate by s), we can now check the

```
1  @Protocol("(read! & annotate)*")
2  public interface Paper {
3    public String read();
4    public void annotate(String cmts);
5  }
```

Figure 5.3: Paper interface

correctness of the $annotate$ method itself, which consists in checking the each argument is correctly used.

$$\text{VOID METHOD}\ \frac{\Delta_3 \vdash s \qquad \mathsf{string} <: \mathsf{string} = \Delta_3(cmts)}{\Delta_4 \vdash \mathbf{void}\ annotate(cmts : \mathbf{string})\{s\} : annotate : (\mathsf{string} \to \mathsf{stop})} \tag{5.5}$$

Where Δ_4 only contains the usage of the $content$ field, $content : \mathsf{string}\,;\,\mathsf{set}(\mathsf{string})$.

We check the `read` method as follows:

$$\text{NON-VOID METHOD}\ \frac{content : \mathsf{string} \vdash content : \mathsf{string} \qquad \mathsf{string} <: \mathsf{string}}{content : \mathsf{string} \vdash \mathbf{string}\ read()\{\mathbf{return}\ content;\} : read : \emptyset \to \mathsf{string}} \tag{5.6}$$

So, the type and environment we get from checking `annotate` and `read` respectively are,

$$(annotate : \mathsf{string} \to \mathsf{stop}, content : \mathsf{string}\,;\,\mathsf{set}(\mathsf{string}))$$

and

$$(read : \emptyset \to \mathsf{string}, content : \mathsf{string}).$$

By applying function `fieldUsage` to the set

$$\{(read : \emptyset \to \mathsf{string}, annotate : \mathsf{string} \to \mathsf{stop}), (content : \mathsf{string}, content : \mathsf{string}\,;\,\mathsf{set}(\mathsf{string}))\}$$

we obtain the object type and environment:

$$((read : \emptyset \to \mathsf{string})! \,\&\, (annotate : \mathsf{string} \to \mathsf{stop}))^*$$

$$content : (\mathsf{string}! \,\&\, (\mathsf{string}\,;\,\mathsf{set}(\mathsf{string})))^*$$

And the usage of field $content$ conforms to the protocol that is prescribed for each field in our type system. So we can consider the `PaperImpl` correct, since it respects the annotation provided in its interface with respect to the implementation: the `read` method does not write on the object's state and therefore can be replicated, and the `annotate` method does not interfere with `read`, due to the synchronization imposed by the combination of `&` and `*`.

To illustrate how to constrain the usage that can be exercised on method parameters and return values, consider the `Reviewer` interface, described in Figure 5.5. A reviewer must first receive a paper to review and only then can he perform his review, and this is done for any paper he might need to review. After reviewing, he can give back the paper, properly annotated. This protocol is written as `(receivePaper ; review ; getAnnotatedPaper)*`. Intuitively, since the paper needs to be received before being reviewed, we know that in a possible implementation of the `Reviewer` interface we will need to store the paper in the object's state, i.e., the reviewer will have to own it. To this end, we annotate the argument of function `receivePaper` with the `@Usage` annotation, that takes up two arguments: `usage` describing what the usage will be given to the argument, and `owned` which is a flag that indicates if the provided behavior is owned.

```
1  public class PaperImpl implements Paper {
2    private String content;
3
4    public PaperImpl(String content) {
5      //Constructor must include every field
6    }
7
8    public String read() {
9      return content;
10   }
11   public void annotate(String cmts){
12     //Dummy implementation that simply appends the comments
13     String cont = content;
14     content = cont + cmts;
15   }
16 }
```

Figure 5.4: Paper Implementation

Notice that we have omitted the `usage` argument in the annotation - by default it is the declared interface protocol for the type of the argument.

Now, suppose the reviewer would like his annotations to be protected, that is, no more annotations are to be made on the paper. This is achieved by restricting the behavior made available on the return type of the `getAnnotatedPaper`, by using the `@Return` protocol, that has a single parameter containing the protocol. So, in this case, since the paper should be read-only, we only allow the protocol `read!` to be exercised upon the paper upon returning it.

Consider the implementation `ReviewerImpl` of the `Reviewer` interface, presented in Figure 5.6. It has a single field of type `Paper`, and receiving a paper stores it into the field. Reviewing it is simply annotating the paper, and returning the reviewed paper consists in returning the `paper` field.

Each method consists in a single instruction that uses the `paper` field in some way. In `receivePaper`, it is used as $\mathsf{set}(Paper^{\circ})$, where $Paper$ denotes an abbreviation of the protocol associated with the interface with the same name. The `review` method consists of calling `annotate` upon the field, and so it is used as $annotate : \mathsf{string} \rightarrow \mathsf{stop}$. In the `getAnnotatedPaper` method, the field is used as $((read : \emptyset \rightarrow \mathsf{string})!)^{\circ}$. So, composing these three environments according to the `Reviewer` protocol, yields the usage

$$(\mathsf{set}(Paper^{\circ})\,;\,(annotate : \mathsf{string} \rightarrow \mathsf{stop})\,;\,((read : \emptyset \rightarrow \mathsf{string})!)^{\circ})^{*}$$

and we can see that it obeys the field protocol $(\mathsf{set}(Paper^{\circ})\,;\,Paper^{\circ})^{*}$, when preceded by an initialization.

```
1  @Protocol("(receivePaper ; review ; getAnnotatedPaper)*")
2  public interface Reviewer {
3
4    public void receivePaper(@Usage(owned=true)Paper p);
5
6    public void review(String comments);
7
8    @Return("read!")
9    public Paper getAnnotatedPaper();
10 }
```

Figure 5.5: Reviewer Interface

```
1  public class ReviewerImpl implements Reviewer {
2    private Paper paper;
3
4    public ReviewerImpl(Paper p){
5      //Constructor must include every field
6    }
7
8    public void receivePaper(Paper p) {
9      paper = p;
10   }
11
12   public void review(String comments) {
13     paper.annotate(comments);
14   }
15
16   public Paper getAnnotatedPaper() {
17     return paper;
18   }
19 }
```

Figure 5.6: Reviewer Implementation

5.4 Implementing and Checking the `List` Interface

In this section we will present a full example in which we annotate an interface from the standard Java API, and provide an implementation. For this example we choose the `List` interface from the `java.util` package. We have chosen this example as it represents the interface for a data structure, and an implementation of such an interface will allow us to discuss some details regarding ownership of objects. The source code for the `List` interface is presented in Figure 5.7, containing only a few key methods. Our example implementation consists in a singly linked list. However, due to the lack of generic types in our formalization, we will describe an implementation of the `List` interface for `Cell` objects, that store a single integer value, allowing for reading and modification. The `Cell` interface is presented in Figure 5.8. Infinite readers are allowed but only one writer, and so we annotate it with the protocol `@Protocol("(get! & set)*")`.

Before getting into the implementation of the list, let us first discuss what the declared interface protocol should be. Determining the size of the list and checking if an element

```
1  public interface List {
2    public int size();
3
4    public boolean contains(Cell elem);
5
6    public void add(int pos, @Usage(owned=true)Cell elem);
7
8    public Cell remove(int pos);
9  }
```

Figure 5.7: Java's List Interface

```
1  @Protocol("(get! & set)*")
2  public interface Cell {
3    public int get();
4    public void set(int i);
5  }
```

Figure 5.8: Cell Interface

exists should be independent of each other and themselves. However, adding and removing elements will necessarily write upon memory and should not be allowed to run concurrently, either with any of the read-only methods and themselves. So, one possible protocol that realizes these constraints can be

```
@Protocol("((size & contains)! & add & remove)*")
```

We define the interface that represents a node of the list, presented in Figure 5.9. The protocol for the `Node` is worthy of discussion. Unlike the read-only operations in the list, with the exception of the `atEnd` method, the ones in this interface actually return the reference that the node holds. So, although these operations are read-only, there is a loss of ownership the moment we decide to retrieve the pointer to the next node or the content. Hence, each call to `getContent/getNext` must be succeeded by a call to `setContent/setNext`. All of this behavior should be exercised sequentially, and so we get the protocol:

```
@Protocol("((getContent;setContent) &
           (getNext;setNext) &
           atEnd! & setNext & setContent)*")
```

The constraint that the `setX` must be called after `getX` seems awkward at first glance, since this can potentially mean corrupting the structure of the list in the process, when used with the wrong arguments. In reality, our linear model of ownership makes it impossible to use list nodes this way. Consider part of the implementation of the `List` interface, only including the `add` function as listed in Figure 5.10.

This follows a traditional implementation of the insertion function in which, starting from the head of the list, iterate over the node structure with as many steps as the `pos`

```
1  public interface Node {
2    public Cell getContent();
3
4    public void setContent(@Usage(owned=true)Cell c);
5
6    public Node getNext();
7
8    public void setNext(@Usage(owned=true)Node n);
9  }
```

Figure 5.9: Node Interface

```
1  public class LinkedList implements List {
2
3    private Node head;
4    //constructor
5
6    public void add(int pos, Cell elem) {
7      int i = 0;
8
9      Node node = head;
10     while(i < pos){
11       Node aux = node.getNext();
12       //node.setNext(?);
13       node = aux;
14       ...
15     }
16     ...
17   }
18   ...
19 }
```

Figure 5.10: List Implementation (First try)

argument. According to our protocol for interface `Node`, we are required to call `setNext` immediately after obtaining the next node reference. However, this implementation requires us to store this reference in the local variable `aux` so that we can incrementally move forward on the list. So, now we need to return the reference to the node and at the same time store it in the local state, rendering this implementation impossible to accomplish, given the linearity we enforce on object ownership.

The solution to this problem is then to provide a kind of functional implementation in which every operation of the list, except for `size` is recursively defined in the node structure. The new version of the `Node` interface is presented in Figure 5.11. The newly added `removeNext` function "disconnects" the node from the next, returning the pointer to it. As we will see, this function does indeed return the the next node pointer, but before that it should set its value to something that represents the null pointer (since we are disconnecting both nodes). Similarly to the previous method, the `removeElement` method returns the element of a node, leaving a dummy value in its stead. The protocol for this interface should allow `atEnd` and `contains` to be activated concurrently, while imposing sequentiality between the rest of the methods, and so a possible protocol is

```
1  @Protocol("((atEnd & contains)! & add &
2                setNext & removeElement &
3                removeNext & remove)*")
4  public interface Node {
5    public boolean atEnd();
6
7    public void add(int pos, @Usage(owned=true)Cell elem);
8
9    public boolean contains(@Usage(usage="get*")Cell elem);
10
11   public Cell remove(int pos);
12
13   public void setNext(@Usage(owned=true)Node n);
14
15   public Cell removeElement();
16
17   public Node removeNext();
18 }
```

Figure 5.11: Node Interface (Corrected)

```
@Protocol("((atEnd & contains)! & add & remove & setNext & removeNext)*")
```

We provide implementations of the `Node` interface one that implements the full expected behavior, `ListNode` in Figure 5.12, and another one that serves as a dummy node implementation, `NullNode` in Figure 5.13, so that we know that we have reached the end of the node chain. The reason for this last class is that if we were to assign the **null** value to a variable (which at least happens when creating a singleton node, i.e., with no next pointer), then it would need to be declared as having the stop protocol, and consequently we would not be able to use said field in any way. For this reason we also include the `atEnd` method to determine if we are at a null node.

Having implemented the `Node` interface, we can define an implementation of the `List` interface. As we mentioned earlier, our implementation will be a linked list with only a list head pointer, as well as a size counter. We present the implementation in Figure 5.14. The `size` method consists in returning the size counter. The `contains` and method is simply calling upon the head's `contains` method with the same argument. Adding an element first requires us to check if we are inserting at the 0 position, case in which we just create a new node with the element, having the current head pointer as next, and setting the head to point to that new node. With removal, we must check if we are removing at 0 position, and if we are we remove the element contained in the `head` pointer, and have it point to the next, obtained by calling `removeNext`.

A limitation related to the linearity of our approach to aliasing is that we cannot have a `get` method. This is because returning the element from a node causes that node to lose ownership of it, and so in a behavioral sense, retrieving an element has the same effect as removing it. No implementation can maintain the list coherence in retrieval, except if we were to return copies of the elements that were being stored, and even then it does not come close to a real world implementation.

```
1  public class ListNode implements Node {
2      private Cell elem;
3      private Node next;
4
5      public ListNode(Cell elem, Node next) {
6        this.elem = elem; this.next = next;
7      }
8
9      public boolean atEnd() {
10       return false;
11     }
12
13     public void add(int pos, Cell elem) {
14       if(pos == 0){
15         Node newNode =
16           new ListNode(elem, next);
17         next = newNode;
18       }
19       else{
20         next.add(pos-1,elem);
21       }
22     }
23
24     public boolean contains(Cell c) {
25       boolean result = false;
26
27       if(elem.get() == c.get()){
28         result = true;
29       }
30       else{
31         result = next.contains(c);
32       }
33
34       return result;
35     }
```

```
36   public Cell remove(int pos) {
37     Cell result;
38
39     if(pos == 0){
40       result = next.removeElement();
41       Node aux = next.removeNext();
42       next = aux;
43     }
44     else{
45       result = next.remove(pos-1);
46     }
47
48     return result;
49   }
50
51   public void setNext(Node n) {
52     next = n;
53   }
54
55   public Node removeNext() {
56     Node n = next;
57     next = new NullNode();
58     return n;
59   }
60
61   public Cell removeElement() {
62     Cell el = elem;
63     //Dummy cell
64     elem = new CellImpl(-1);
65     return el;
66   }
67 }
```

Figure 5.12: ListNode Class

5.5 Summary

In this implementation, we have presented an annotation system that seamlessly with regular Java development. The main challenges throughout the development of our solution were related to the implementation of the subtyping relation and associated labeled transition system presented in Section 4.2. We extended the approach of [Par11] with support for recursive types in the core language, and deal with cases of the subtyping relation involving repetition types. In the case of the labeled transition system, determining valid sets of transition for both the subtype and supertype was a complex task. In the implementation, the representation of the labeled transition system and its transitions was something that was reformulated many times, but in the end the result was a cleanly defined transition system, that we believe is very close to its associated formal definition.

Another main issue, although it was solved relatively early, was the joint compilation of Java and Scala sources into a single `jar` file, since Scala code was being used in Java and vice-versa, traditional ways to compile the source code seemed a daunting task. To solve this issue, we have used the Maven [mav12] build manager by Apache. This versatile tool provided the necessary facilities not only to have a joint Scala/Java project for the

```
1  public class NullNode implements Node {
2    public boolean atEnd() {
3      return true;
4    }
5
6    public void add(int pos, Cell elem) {}
7
8    public boolean contains(Cell c) {
9      return false;
10   }
11
12   public Cell remove(int pos) {
13     return new CellImpl(-1); //Dummy cell
14   }
15
16   public void setNext(Node n) {}
17
18   public Node removeNext() {
19     return new NullNode();
20   }
21   public Cell removeElement() {
22     return CellImpl(-1);
23   }
24 }
```

Figure 5.13: NullNode Class

development of the implementation, as well as to package and compress the type system into a distributable `jar` file.

```
1  public class LinkedList implements List {
2
3    Node head;
4    int nElems;
5
6    public LinkedList(Node head, int nElems) {
7      this.head = head;
8      this.nElems = nElems;
9    }
10
11   public int size() {
12     return nElems;
13   }
14
15   public boolean contains(Cell elem) {
16     return head.contains(elem);
17   }
18
19   public void add(int pos, Cell elem) {
20     if(pos >= 0 && pos <= nElems) {
21       if(pos == 0){
22         //Add first
23         Node newNode = new ListNode(elem, head);
24         head = newNode;
25       }
26       else{
27         //We insert after the head
28         head.add(pos-1, elem);
29       }
30       int nElemsAux = nElems;
31       nElems = nElemsAux + 1;
32     }
33   }
34
35   public Cell remove(int pos) {
36     Cell result;
37     if(pos >= 0 && pos < nElems) {
38       if(pos == 0){
39         result = head.removeElement();
40         Node aux = head.removeNext();
41         head = aux;
42       }
43       else{
44         result = head.remove(pos-1);
45       }
46       int nElemsAux = nElems;
47       nElems = nElemsAux - 1;
48     }
49     else{
50       //Dummy cell
51       result = new CellImpl(-1);
52     }
53     return result;
54   }
55 }
```

Figure 5.14: Linked List Class

6

Final Remarks

In this dissertation, our main goal was to design a behavioral type system targeting the Java programming language, building on the work done in [Par11], by including support for protocol specification at the method declaration level, specifying finer grained contracts about the behavior of parameters and returned objects, object ownership and an object-based concurrency model.

We have proposed an algorithm that is capable of analyzing concurrent behavior from a behavioral type, given the information about the execution scope of a thread. In order to check correct object usage in a program, we have presented a behavioral type system, applied to a core fragment of Java, that guarantees correct use of objects with respect to a usage protocol specification. We described presented a behavioral type system, aiming to check behavioral correctness of constructs like branching statements, assignments and method calls, up to class conformance to a behaviorally annotated interface.

The prototype implementation of our development consisted in a typechecker that is fully integrated in a Java development environment, allowing programs to be behaviorally annotated by using standard Java APIs and annotations.

6.1 Contributions

This work presents a behavioral type system for a fragment of the Java programming language, based on spatial-behavioral types proposed in [Cai08]. We have fully integrated an implementation of the type system with regular Java development by using standard Java annotations, as well as the Compiler Tree and Annotation Processor APIs.

With our type system, we are able to statically check if concurrent programs follow

disciplined usage of objects as prescribed by a protocol annotation on their interface definition. We have included a subtyping algorithm, based on a labeled transition system, that checks if a different usage than what is expected can still be safely used, which in turn increases the number of programs that our type system addresses and, in development, makes the typechecker less rigid to the programmer.

The validation of our approach was done by developing an example suite of Java programs that were checked by our type system, aiming not only to test each construct of the language, but also some complex programming idioms found in Java.

6.2 Future Work

To conclude, we enumerate the aspects of our work that can be improved in the future:

Extraction of Concurrent Behavior Algorithm $\mathcal{C}$ poses a limitation in which we lose temporal dependencies imposed by the ; operator between behavior that occurs before some arbitrary concurrent behavior. It would be interesting to study if this limitation is tied to the algebraic structure of types, or if there is an algorithm that is capable of producing the most specific type, and if so, provide a proof of this property.

Subtyping The general case of the subtyping relation is defined solely upon the presented labeled transition system. It would be interesting to prove that this case indeed covers the rest of the type operators for which a subtyping rule isn't directly defined.

Support a broader set of Java's core constructs There are some frequently used constructions that were not included in our programming model, such as calls to the implicit `this` object, promoting code reuse. However, this would imply a careful study of how internal calls change the state of the object and guaranteeing that they do not break the protocol specification provided in the interface.

Support for richer ownership types In our implementation, we follow a linear model of object ownership. This strictness limits the construction of programs, such as dynamic data structures that rely on richer forms of object ownership. Although in the `List` example we have provided an example implementation of this data structure, it did not follow a more traditional implementation that is usually found in object-oriented languages.

Provide a way to define protocols for Java library interfaces Much like the specification we provided for the `List` interface, it would be useful to provide a set of specification files for standard Java interfaces, for use when typechecking a program that uses them. This prototype implementation only checks classes that were defined by the user and so classes that belong to the Java API are not considered, e.g., the `Object` class.

6.3 Summary

In this chapter we have provided a sum up of the work that was developed in this dissertation, followed by the main contributions and key aspects that could be improved in future work. These improvements include some more technical details of the implementation to improve usability as well as some theoretical aspects that would ensure the soundness of our approach.

Bibliography

[ASSS09] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented Programming. In *OOPSLA*, pages 1015–1022, 2009.

[BBA08] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying Correct Usage of Atomic Blocks and Typestate. In Gail E. Harris, editor, *OOPSLA*, pages 227–244. ACM, 2008.

[Cai08] Luís Caires. Spatial-Behavioral Types for Concurrency and Resource Control in Distributed Systems. *Theor. Comput. Sci.*, 402:120–141, July 2008.

[CD02] David G. Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, pages 292–310, 2002.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local Action and Abstract Separation Logic. In *LICS*, pages 366–378, 2007.

[CPN98] David G. Clarke, John Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, pages 48–64, 1998.

[CRR02] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as Models: Model Checking Message-passing Programs. In *POPL*, pages 45–57, 2002.

[CS13] Luís Caires and João Seco. The Type Discipline of Behavioral Separation. In *POPL*, 2013.

[CV10a] Luís Caires and Hugo Torres Vieira. Conversation Types. *Theor. Comput. Sci.*, 411(51-52):4399–4440, 2010.

[CV10b] Joana Campos and Vasco T. Vasconcelos. Channels as Objects in Concurrent Object-Oriented Programming. In Kohei Honda and Alan Mycroft, editors, *Proceedings Third Workshop on Programming Language Approaches to Concurrency and Communication-centric Software*, volume 69 of *EPTCS*, pages 12–28, 2010.

[DP08] Dino Distefano and Matthew J. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA*, pages 213–226, 2008.

[FAH+06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys*, pages 177–190. ACM, 2006.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. *SIGPLAN Not.*, 37:234–245, 2002.

[GH99] Simon J. Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In S. Doaitse Swierstra, editor, *ESOP*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1999.

[GV10] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear Type Theory for Asynchronous Session Types. *J. Funct. Program.*, 20(1):19–50, 2010.

[GVR+10] Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312, 2010.

[HMSW09] C. A. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene Algebra. In *Proceedings of the 20th International Conference on Concurrency Theory*, CONCUR 2009, pages 399–414, 2009.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[Hoa74] C. A. R Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17:549–557, 1974.

[Hon93] Kohei Honda. Types for dynamic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[HOP11] Akbar Hussain, Peter W. O'Hearn, and Rasmus L. Petersen. On Separation, Session Types and Algebra, 2011. Unpublished.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284, 2008.

[IK04] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the Pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

[Kob06] Naoki Kobayashi. A New Type System for Deadlock-Free Processes. In *CONCUR*, pages 233–247, 2006.

[mav12] Apache Maven Project. `http://maven.apache.org/`, September 2012.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice-Hall, Upper Saddle River, NJ, USA. Prentice Hall, 1989.

[Mil91] Robin Milner. The Polyadic Pi-Calculus: a Tutorial. Technical report, Logic and Algebra of Specification, 1991.

[Mil08] Filipe Militão. Design and Implementation of a Behaviorally Typed Programming System for Web Services. Master's thesis, Universidade Nova de Lisboa, Portugal, 2008.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I + II. *Information and Computation*, 100(1):1–77, 1992.

[O'H04] Peter W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, pages 49–67, 2004.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL*, pages 1–19, 2001.

[Par11] Nuno Parreira. Implementação de uma Linguagem Concorrente com Tipos Comportamentais. Master's thesis, Universidade Nova de Lisboa, Portugal, 2011.

[PS96] Benjamin C. Pierce and Davide Sangiorgi. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[Rey02] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74, 2002.

[SY86] R. E. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, January 1986.

[Vas09] Vasco Thudichum Vasconcelos. Fundamentals of Session Types. In Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors, *SFM*, volume 5569 of *Lecture Notes in Computer Science*, pages 158–186. Springer, 2009.

[VGR06] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type-checking a Multithreaded Functional Language with Session Types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

More Examples

```
@Protocol("(set & get!)*")
public interface Cell {
  public int get();
  public void set(int i);
}

public class CellImpl implements Cell {
  private int data;

  public CellImpl(int data) {
    this.data = data;
  }

  public int get() {
    return data;
  }

  public void set(int i){
    data = i;
  }

  public static void main(String[] args){
    Cell c = new CellImpl();
    c.set(3);
    c.get();
  }
}
```

Listing A.1: Correct usage of Cell

```
@Protocol("set;get")
public interface Cell {
  public int get();
  public void set(int i);
}

public class CellImpl implements Cell {
  private int data;

  public CellImpl(int data) {
    this.data = data;
  }

  public int get() {
    return data;
  }

  public void set(int i){
    data = i;
  }

  public static void main(String[] args){
    Cell c = new CellImpl(); //Usage should be set;get
    c.get();
    c.set(3);
  }
}
```

Listing A.2: Misused Cell

```
@Protocol("(set & get!)*")
public interface Cell {
  public int get();
  public void set(int i);
}

public class CellImpl implements Cell {
  private int data;

  public CellImpl(int data) {
    this.data = data;
  }

  public int get() {
    return data;
  }

  public void set(int i){
    data = i;
  }

  public static void main(String[] args){
    final Cell c = new CellImpl();
    Thread t1 = new Thread(new Runnable(){
      public void run(){
        c.get();
      }
    });
    Thread t2 = new Thread(new Runnable(){
      public void run(){
        c.get();
      }
    });
    t1.start();
    t2.start();
    t2.join();
    t1.join();
    c.set(3);
  }
}
```

Listing A.3: Correct concurrent usage of Cell

```
@Protocol("(set & get!)*")
public interface Cell {
  public int get();
  public void set(int i);
}

public class CellImpl implements Cell {
  private int data;

  public CellImpl(int data) {
    this.data = data;
  }

  public int get() {
    return data;
  }

  public void set(int i){
    data = i;
  }

  public static void main(String[] args){
    final Cell c = new CellImpl();
    Thread t1 = new Thread(new Runnable(){
      public void run(){
        c.get();
      }
    });
    Thread t2 = new Thread(new Runnable(){
      public void run(){
        c.get();
      }
    });
    t1.start();
    t2.start();
    c.set(3);
    t2.join();
    t1.join();

  }
}
```

Listing A.4: Incorrect concurrent usage of Cell

```
@Protocol("set | get")
public interface Cell {
  public int get();
  public void set(int i);
}

public class CellImpl implements Cell {
  //There is a data race on data
  //since it used according to
  //set(int) | int
  private int data;

  public CellImpl(int data) {
    this.data = data;
  }

  public int get() {
    return data;
  }

  public void set(int i){
    data = i;
  }
}
```

Listing A.5: Incorrect implementation of Cell

```
@Protocol("inc")
public interface Counter {
  public void inc(@Usage(usage="get;set")Cell c);
}
public class CounterImpl implements Counter {
  public void inc(Cell c) {
    int x = c.get();
    c.set(x+1);
  }
}
@Protocol("get;set")
public interface Cell {
  public int get();
  public void set(int i);
}

public class CellImpl implements Cell {
  private int data;

  public CellImpl(int data) {
    this.data = data;
  }

  public int get() {
    return data;
  }

  public void set(int i){
    data = i;
  }

  public static void main(String[] args){
    Counter co = new CounterImpl();
    Cell c = new CellImpl();
    co.inc(c); // c: get;set
  }
}
```

Listing A.6: Counter

```
@Protocol("inc")
public interface Counter {
  public void inc(@Usage(owned=true)Cell c);
}
public class CounterImpl implements Counter {
  private Cell cell;
  public CounterImpl(Cell c){ cell = c; }
  public void inc(Cell c) {
    int x = c.get();
    c.set(x+1);
    cell = c;
  }
}
@Protocol("(set & get!)*")
public interface Cell {
  public int get();
  public void set(int i);
}

public class CellImpl implements Cell {
  private int data;

  public CellImpl(int data) {
    this.data = data;
  }

  public int get() {
    return data;
  }

  public void set(int i){
    data = i;
  }

  public static void main(String[] args){
    Counter co = new CounterImpl();
    Cell c = new CellImpl();
    co.inc(c);

    //Will fail since there is loss of
    //ownership of c by calling inc
    c.set(3);
  }
}
```

Listing A.7: Loss of ownership

```
@Protocol("inc;getReadOnly")
public interface Counter {
  public void inc(@Usage(owned=true)Cell c);

  @Return("get!")
  public Cell getReadOnly();
}
public class CounterImpl implements Counter {
  private Cell cell;

  public CounterImpl(Cell c) { cell = c; }
  public void inc(Cell c) {
    int x = c.get();
    c.set(x+1);
    cell = c;
  }
  public Cell getReadOnly(){
    return cell;
  }
}
@Protocol("(set & get!)*")
public interface Cell {
  public int get();
  public void set(int i);
}

public class CellImpl implements Cell {

  private int data;

  public CellImpl(int data) {
    this.data = data;
  }

  public int get() {
    return data;
  }

  public void set(int i){
    data = i;
  }

  public static void main(String[] args){
    Counter co = new CounterImpl();
    Cell c = new CellImpl();
    co.inc(c);

    @Usage(usage="get!")Cell c = co.getReadOnly();
    c.get();
  }
}
```

Listing A.8: Return type restriction

```
@Protocol("open; (read! & write)*; close")
public interface File {

  public void open();

  public String read();
  public void write(String content);

  public void close();

}
public class FileImpl implements File{
  private String data;

  public FileImpl(String d){
    data = d;
  }

  public void open() { }
  public String read() {
    return data;
  }
  public void write(String content) {
    String auxData = data;
    data = auxData + content;
  }
  public void close() { }

  public static void main(String args){
    final File f = new FileImpl("");

    Thread t1 = new Thread(new Runnable(){
      public void run(){
        f.read();
      }
    });
    Thread t2 = new Thread(new Runnable(){
      public void run(){
        f.read();
      }
    });
    f.open();
    t1.start();
    t2.start();
    t2.join();
    t1.join();
    f.close();
  }
}
```

Listing A.9: Correct File usage

```
@Protocol("open; (read! & write)*; close")
public interface File {

  public void open();

  public String read();
  public void write(String content);

  public void close();

}
public class FileImpl implements File{
  private String data;

  public FileImpl(String d){
    data = d;
  }

  public void open() { }
  public String read() {
    return data;
  }
  public void write(String content) {
    String auxData = data;
    data = auxData + content;
  }
  public void close() { }

  public static void main(String args){
    final File f = new FileImpl("");

    Thread t1 = new Thread(new Runnable(){
      public void run(){
        f.read();
      }
    });
    Thread t2 = new Thread(new Runnable(){
      public void run(){
        f.read();
      }
    });
    t1.start();
    t2.start();
    //Opening a file concurrently with read()
    f.open();
    t2.join();
    t1.join();
    f.close();
  }
}
```

Listing A.10: Incorrect concurrent File usage

```
@Protocol("((atEnd & contains)! & add & setNext & removeElement & removeNext & remove)*")
public interface Node {
  public boolean atEnd();

  public void add(int pos, @Usage(owned=true)Cell elem);

  public boolean contains(@Usage(usage="get*")Cell elem);

  public Cell remove(int pos);

  public void setNext(@Usage(owned=true)Node n);

  public Cell removeElement();

  public Node removeNext();
}
```

Listing A.11: Node Interface

```
  public class NullNode implements Node {

    public boolean atEnd(){
      return true;
    }

    public void add(int pos, Cell elem){}

    public boolean contains(Cell elem){
      return false;
    }

    public Cell remove(int pos){
      return new CellImpl(-1);
    }

    public void setNext(@Usage(owned=true)Node n){}

    public Node removeNext(){
      return new NullNode();
    }

    @Override
    public Cell removeElement() {
      return new CellImpl(-1);
    }
  }
```

Listing A.12: NullNode

```
public class ListNode implements Node {
  private Cell elem;
  private Node next;

  public ListNode(Cell elem, Node next) {
    this.elem = elem;
    this.next = next;
  }

  public boolean atEnd() {
    return false;
  }
  public void add(int pos, Cell elem) {
    if(pos == 0){
      Node newNode = new ListNode(elem, next);
      next = newNode;
    }
    else{
      next.add(pos-1,elem);
    }
  }
  public boolean contains(Cell c) {
    boolean result = false;
    if(elem.get() == c.get()){
      result = true;
    }
    else{
      result = next.contains(c);
    }
    return result;
  }
  public Cell remove(int pos) {
    Cell result;
    if(pos == 0){
      result = next.removeElement();
      Node aux = next.removeNext();
      next = aux;
    }
    else{
      result = next.remove(pos-1);
    }

    return result;
  }
  public void setNext(Node n) {
    next = n;
  }
  public Node removeNext() {
    Node n = next;
    next = new NullNode();
    return n;
  }
  public Cell removeElement() {
    Cell el = elem;
    //Dummy cell
    elem = new CellImpl(-1);
    return el;
  }
}
```

Listing A.13: ListNode

```
@Protocol("((size & contains)! & add & remove)*")
public interface List {
  public int size();
  public boolean contains(@Usage(usage="get*")Cell elem);
  public void add(int pos, @Usage(owned=true)Cell elem);
  public Cell remove(int pos);
}
public class LinkedList implements List {
  private Node head;
  private int nElems;

  public LinkedList(Node head, int nElems) {
    this.head = head;
    this.nElems = nElems;
  }
  public int size() {
    return nElems;
  }
  public boolean contains(Cell elem) {
    return head.contains(elem);
  }
  public void add(int pos, Cell elem) {
    if(pos >= 0 && pos <= nElems) {
      if(pos == 0){
        //Add first
        Node newNode = new ListNode(elem, head);
        head = newNode;
      }
      else{
        //We insert after the head
        head.add(pos-1, elem);
      }
      int nElemsAux = nElems;
      nElems = nElemsAux + 1;
    }
  }
  public Cell remove(int pos) {
    Cell result;
    if(pos >= 0 && pos < nElems) {
      if(pos == 0){
        result = head.removeElement();
        Node aux = head.removeNext();
        head = aux;
      }
      else{
        result = head.remove(pos-1);
      }
      int nElemsAux = nElems;
      nElems = nElemsAux - 1;
    }
    else{
      //Dummy cell
      result = new CellImpl(-1);
    }
    return result;
  }
}
```

Listing A.14: List Interface and Implementation

```
public static void main(String[] args){
  final List l = new LinkedList(new NullNode(), 0);
  int i = 0;
  while(i < 100){
    l.add(0,new CellImpl(i));
    int x = i;
    i = x + 1;
  }

  Thread t1 = new Thread(new Runnable(){
    public void run(){
      int j = 0;
      while(j < 100){
        l.contains(new CellImpl(j));
        int x = j;
        j = x + 1;
      }
    }
  });

  Thread t2 = new Thread(new Runnable(){
    public void run(){
      int j = 0;
      while(j < 100){
        l.remove(0);
        int x = j;
        j = x + 1;
      }
    }
  });

  //t1 does not interfere with t2
  t1.start();
  t1.join();
  t2.start();
  t2.join();
}
```

Listing A.15: Correct concurrent List usage

```
public static void main(String[] args){
  final List l = new LinkedList(new NullNode(), 0);
  int i = 0;
  while(i < 100){
    l.add(0,new CellImpl(i));
    int x = i;
    i = x + 1;
  }

  Thread t1 = new Thread(new Runnable(){
    public void run(){
      int j = 0;
      while(j < 100){
        l.contains(new CellImpl(j));
        int x = j;
        j = x + 1;
      }
    }
  });

  Thread t2 = new Thread(new Runnable(){
    public void run(){
      int j = 0;
      while(j < 100){
        l.remove(0);
        int x = j;
        j = x + 1;
      }
    }
  });

  //t1 interferes with t2 since
  //remove is being called concurrently with contains
  t1.start();
  t2.start();
  t2.join();
  t1.join();
}
```

Listing A.16: Incorrect concurrent List usage