Cláudio José Albino de Vasconcelos

Master of Science

Behavioral type inference for concurrent object-oriented languages

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: António Ravara, Professor Auxiliar, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2016

Behavioral type inference for concurrent object-oriented languages

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2]. [1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

Abstract

The widespread use of service-oriented and cloud computing is creating a need for a communication-centered programming approach and for distributed concurrent software systems. Protocols play a central role in the design and development of such systems but mainstream programming languages still give poor support to ensure protocol compatibility. Testing alone is not sufficient to ensure protocol compatibility, so there is a pressing need for tools to assist in the development of these kind of systems.

Behavioral types is an possible solution to this problem and there is already a great amount of research on how to integrate these types in programming languages, namely object-oriented ones. The drawback some of the existing approaches to the use of behavioral types on programming languages consist on requiring the developers explicitly annotate the code with the behavioral types, what constitutes an extra burden to the programmer. Inference-based approaches would not only help programmers by alleviating them from such annotations but would also allow to deal with legacy code.

Our goal is to develop a tool that is capable of analyzing source code from programs written in a subset of Java and, if well typed, produce a new version annotated with its respective behavioral types, ensuring object interoperability. Concretely, the contribution is two-fold: The definition of an idealized, yet expressive, Java-like language with behavioral types (building on previous work); the definition of a type-inference system for the language without behavioral annotations but with pre-conditions and invariants.

Keywords: Behavioral types, object-oriented programming, type systems, concurrency, distributed systems, typestates, inference, session types

Resumo

O uso generalizado de computação centrada em serviços e *cloud computing* tem vindo a criar uma necessidade de uma abordagem baseada em programação centrada na comunicação e em sistemas distribuídos de software concorrente. Protocolos de comunicação desempenham um papel central na conceção e desenvolvimento de tais sistemas mas as linguagens de programação tradicionais continuam a não dar o suporte necessário para garantir a compatibilidade entre protocolo. Testar em si só não é suficiente para garantir a compatibilidade entre protocolos, pelo que existe uma necessidade urgente de ferramentas para auxiliar no desenvolvimento deste tipo de sistemas .

Tipos comportamentais são uma possível solução para este problema e já existe uma grande quantidade de trabalho de investigação feito sobre como integrar estes tipos em linguagens de programação, nomeadamente orientadas a objetos. A desvantagem de algumas das abordagens existentes para o uso de tipos comportamentais em linguagens de programação consiste na exigência feita aos programadores em anotar explicitamente o código com os tipos comportamentais, o que constitui uma carga extra para o programador. Abordagens baseadas em inferência não só ajudam os programadores aliviando-os de tais anotações, mas também permitiria lidar com *legacy code*.

O nosso objetivo é desenvolver uma ferramenta capaz de analisar código fonte de programas escritos num subconjunto do Java e, se bem tipificado, de produzir uma nova versão anotada com seus respetivos tipos comportamentais, garantindo interoperabilidade entre objetos. Concretamente, a contribuição é dupla : A definição de uma versão idealizada, mas expressiva, de uma linguagem semelhante ao Java com tipos comportamentais (a partir de trabalho anterior); a definição de um sistema de inferência de tipos para a linguagem sem anotações comportamentais, mas com pré-condições e invariantes.

Palavras-chave: Tipos comportamentais, Programação orientada a objetos, sistemas de tipos, concorrência, sistemas distribuídos, *typestates*, inferência, tipos de sessão

Contents

List of Figures

Listings

Chapter 1

Introduction

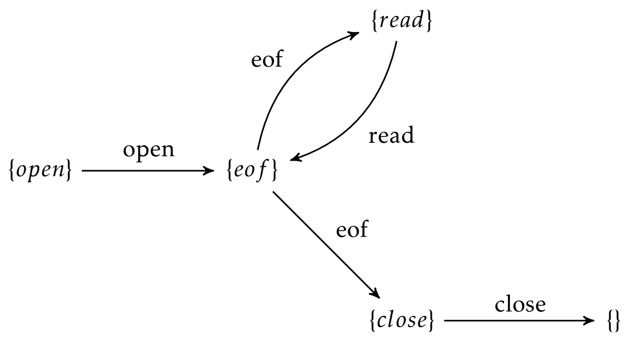
Modern software systems are inherently concurrent and distributed, and there is a necessity of guaranteeing that these systems are available, secure and reliable. Concurrent programming alone is already very challenging; developing multi-threaded applications is not only quite elaborate (or even difficult), but also error-prone due to how hard it is to reason about their behaviour. Debugging and testing is not enough to ensure these applications are safe because, as Dijkstra stated, "Program testing can at best show the presence of errors but never their absence" [12], and ensuring liveness properties requires demanding approaches since not only they are not decidable they also require algorithms with high complexity [25]. In a distributed context, these applications evolve into large-scale systems composed by many components that need to communicate between them while following protocols to ensure the system behaves correctly since these type of systems do not have shared memory. Because of this, communication protocols play a central role in the design and development of these type of systems and so they have created a necessity for a communication-based programming approach.

It is necessary to develop techniques that helps to create safe and well-behaved systems, such as language-based tools to provide correctness guarantees, or extensions of programming languages with constructs to specify the intended behaviour and to ensure that it is followed. It is important that these techniques allow programmers to test their systems statically while while they are developing them instead of testing only at the end, which can bring a lot of unnecessary complexity [13].

While large-scale software systems rely on communication protocols, none of the mainstream programming languages support them nor offer any type of static support to stateful behaviour. There are many (static) approaches to guarantee code correctness but they only focus on on safety properties, which is not enough as the absence of run-time errors does not imply that the application is well-behaved. Also, liveness properties are hard, sometimes impossible, to verify statically. Consider, for instance, an API to read files. Listing 1.1 shows the *File* class presented as an example in [4] but equipped with assertions. The assertions in the code suggest an usage protocol such as the one in Figure 1.1: One first opens the file; before reading one should test for (non-)emptiness, and once

Listing 1.1: Code for the *File* class

```
class File {

    int linesInFile; int linesRead;
    boolean open; boolean closed; boolean lineInBuffer;

    //@ invariant linesRead >= 0 && linesRead ≤ linesInFile;

    //@ requires !open;
    //@ ensures open && linesRead == 0 && linesInFile == 5 && !closed && !lineInBuffer;;
    void open(string filename) {...}

    //@ requires open && linesRead < linesInFile && !closed && lineInBuffer;
    //@ ensures open && linesRead + 1 ≤ linesInFile && !closed && !lineInBuffer;
    string read() {...}

    //@ requires open && linesRead ≤ linesInFile && !closed && !lineInBuffer;
    //@ ensures open && (linesRead == linesInFile −> !lineInBuffer) && !closed;
    boolean eof() {...}

    //@ requires open && linesRead == linesInFile && !closed;
    //@ ensures open && linesRead == linesInFile && closed;
    void close() {...}
}
```



$$\begin{tikzcd}
 & & \{read\} \arrow[dl, bend left, "read"] & \\
\{open\} \arrow[r, "open"] & \{eof\} \arrow[ur, bend left, "eof"] \arrow[dr, "eof"] & & \\
 & & \{close\} \arrow[r, "close"] & \{\}
\end{tikzcd}$$

Figure 1.1: File API usage protocol

the work is done, one should close the file. Guaranteeing (statically) that the code never goes wrong does not ensure its "usefulness" because while it can ensure that the client code does not execute a sequence of instructions that can cause the program to fail, such as reading a file before opening it, it does not guarantee that it fully follows the usage protocol required by the API by allowing situations such as the client code not closing the file at the end. An important (liveness) property is that client code using the API should always follow its protocol.

Behavioural types [24] are a valid approach to the problem, as behavioural-typed languages allows programmers to specify usage protocols as types that can be statically verified by the type system to ensure the usage protocol is well implemented. At the moment none of the mainstream languages provide official support to behavioural types but there is an extensive line of research working towards its mainstream use [1].

Assertions are officially supported by Java since 2006 [1] and not only there is a lot of written code with assertions, many programmers already have experience in using assertions in code and so they may find easier to specify the behaviour of the code by stating, for each method, the required and ensured state of fields and parameters through assertions. Also, while specifying behaviour through behavioural types may be more intuitive and easy the check, type systems are not mature enough to allow programmers to fully rely on behavioural types to guarantee code correctness. Consider the usage protocol in Figure 1.1: Type systems can guarantee that the reading operation is executed after the opening operation but it does not guarantee that the opening operation does, in fact, open the file, allowing for the read operation to operate on non initialized fields. With assertions we can specify that after the opening operation the necessary fields must be initialized. As of now, the properties guaranteed by the type systems and assertions are complementary.

The aim of our work is to bridge the world of programming in Java with assertions with the world of behavioural typed programming, in particular in Java. We developed an algorithm that converts Java with assertions in a form of behavioural types (henceforth called *usage*, a textual representation of a finite automata). Usages represent all the safe sequences of method calls and are (enhanced forms of) class types, checkable at compile-time. Our work will focus on a small programming language similar to Java that uses behavioural types in the form of usages, which we refined its type-checking system to provide enough expressiveness to the language to support the development of more complex and realistic applications.

To summarize, our approach is the following: given a program written in a subset of Java, fully annotated with assertions that we take correct and ensure correctness, returns its usage. The goal is to provide developers with abstractions extracted from the code to represent its behaviour. Furthermore, these abstractions can be attached to the code and statically verified.

1.1 Contributions

The contributions of this thesis are:

- An revision of the Mool language, a small object-oriented programming language that integrates behavioural types in the form of usages. This revision is composed by (1) an analysis to the formalization of the language where we present a set of aspects where the language is not well defined or it is too restricting and (2) a proposal for a new version of its formalization, completed with revised syntax, operational semantics and type system, that solves those aspects;

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html

- A definition of an algorithm that takes a program written in a variation of our revised version of the Mool language and returns it annotated with usages and a prototype tool that implements the algorithm.

1.2 Thesis outline

The thesis is structured as follow:

- Chapter 2 presents a state of the art study on behavioural types, more specifically in the form of session types, and type inference.
- Chapter 3 presents our analysis to the Mool language and a proposal for an updated version of the language;
- Chapter 4 presents the behavioural type inference algorithm we defined for the Mool language;
- Chapter 5 summarises the accomplishments of this thesis and and presents our ideas for future work.

Chapter 2

Related work

This chapter presents the state-of-the-art on static code analysis and behavioural types. Sections 2.1, 2.2 and 2.3 presents, in a general way, the state of the art on techniques for static analysis of code. Section 2.4 presents a study on behavioural types focusing on session types and their integration in programming languages, specially in the object-oriented paradigm. Section 2.5 presents a study on for session type inference, focusing on the techniques which we will use on our work.

2.1 Deductive proof systems

Deductive proof systems allows to statically guarantee code correctness through the analysis of code annotated with assertions that describe the specification of the system it implements, namely its safety properties, which are then proved valid through the use of theorem provers [16]. The specification of the system through assertions requires an extra effort from the programmer in analysing and extracting the properties of the system and then translating them into assertions. Why3 is an example of a platform that allows to perform deductive proofs on code. It offers the WhyML language, a ML dialect, than can be used to write code and program annotations and uses a set of external theorem provers to verify the code. There are a number of projects consisting on platforms for deductive proofs of programs written in mainstream languages using WhyML as an intermediate language [17].

2.2 Model checking

First proposed by Clarke and Emerson [7] and by Sifakis and Queille [31], model checking is a verification technique that consists on analysing a model automatically extracted from code and validate it by checking if it complies with the specification of the system. The models are finite-state automata that represent, in an abstract way, the behaviour of the system and are exhaustively explored while checking if certain properties are satisfied.

While this approach is automatic, it can take a long time to give an answer and it is not decidable when dealing with data-intensive systems [2, p. 7 - 18].

2.3 Types and type systems

A type system is an essential component of any typed language because these programming languages rely on them to detect the absence of type-related errors, helping determine if the system is well behaved. But, for the type system to be effective, its formalization must be correct and compliant to the behaviour we expect from the system [5]. A well formalized type system can bring a lot of benefits to a typed language. Statically typed languages require the programmer to declare properly each type, which not only enforces disciplined programming but more importantly helps to provide useful information to help detect early some programming errors during typecheking, which can help the language being more secure, improve the code's readability, since these typing declarations can serve as documentation, and contribute to the execution efficiency of the code [30, p. 4 - 8].

2.3.1 Type-checking

Typechecking is the process of checking a program written in a typed language to ensure that the program is well typed, i.e., is safe and well behaved. This is done with an algorithm, called the typechecker, that goes through the program and verifies that the program is well typed. If the typechecker cannot guarantee that the program is well typed or detects that the program is ill behaved, the program is considered ill typed [5].

In statically typed languages, typecheckers are embedded into the compiler and the process of typechecking is done during compile time, allowing the detection of errors early [30, p. 3].

2.3.2 Type inference

Type inference consists of deducing the type of a term within a given type system based on type annotations. Like in typechecking, the process of type inference should be static, i.e., run at compile time [5].

2.4 Behavioural types and type systems

Born from the idea of describing the behaviour of the system through types, behavioural types are an approach to code correctness that consist on the the specification of interaction patterns of processes through expressive type languages which can then be used to reason about the behaviour of the code. Since behavioural types are essentially types, these can be used to verify statically if the code is well-behaved using type checking algorithms. In this context, if the code is proven to be well-typed, it is guaranteed to have a consistent communication pattern and so it will never cause a runtime error due to the interaction between processes [24].

2.4.1 Typestates

Typestates are an extension of the concept of type that makes it possible to define which operations are allowed in a particular context, helping to detect during compilation time, particularly during type-checking, unexpected sequences of operations (for example, reading a file after closing it) [32].

Garcia, Tanter, Wolff and Aldrich introduced the concept of typestate-oriented programming [18], which consists on the extension of object-oriented programming with the notion of typestate. In typestate-oriented programming each state is represented by a class, with each of these classes having their own representation and methods. In this context, the class of the object represents its typestate, which can dynamically change during runtime. Aldrich also developed the Plaid language [34], a typestate-programming language that uses these foundations as its core.

2.4.2 Session types

The concept of session was first introduced by Honda et al. [22, 23], and it consists on structuring the interaction between two or more clients that communicate through the use of bidirectional channels. Takeuchi, Honda and Kubo were the first to formalize sessions using type theory in [33], where they present a small programming language based on the Milner's π-calculus and its type system based on the concept of interaction between processes.

There has been a large amount of effort into researching the integration of session types into mainstream paradigms, namely functional and object-oriented paradigms [1]. Neunauer and Thiemann present in [28] an approach to session type use on Haskell by encoding session types into the language using its original type system, while preserving typing. Vasconcelos et al. present in [36] the idea of transferring the concept of session types from the context of the π-calculus to a multi-threaded functional language that, unlike [28], extends the typing system with judgments that statically describe dynamic changes in the types of channels and function types that describe the changes the function performs on the channels it interacts with. Vasconcelos et al. extended on previous work in [19], where they present the formal semantics of buffered communication, introducing linearity to channels.

The idea of integrating session types into object-oriented languages was first proposed by Dezani-Ciancaglini et al. in [9], which would then be built over with the formalization the Moose language [10], a simple multi-threaded object-oriented programming language extended with session-based communication primitives and types. In this work, session types describe the channel usages, namely the sequence of messages passed through the channel. Channels can be shared or linear: Once created, the channel starts in a shared state until a connection is established, which then it goes to a linear state, meaning that the channel is live and, dues to its linearity, a message can only be transmitted once. To achieve type preservation, a live channel can only be used by two threads and a thread can only have one reference to a single live channel. [10] guarantees that a well-typed program guarantees the correct and complete execution of the communication protocol between two threads.

Stoop [11], developed by Dezani-Ciancaglini et al., is an approach to session type integration in object-oriented languages that, instead of extending the object-oriented paradigm with session type features, focus on the amalgamation of session types with the object-oriented paradigm by unifying sessions and methods and basing choices on the transmitted objects. Like methods, sessions in Stoop are invoked from methods and its execution starts immediately but, just like sessions as presented in previous work, concurrently in a different thread. [11] only presents Stoop as a "language kernel" that concerns only with the amalgamation of object-oriented features with sessions. Stoop also guarantees type preservation and progress.

Gay et al. presented in [20] a new approach to session type integration in object-oriented languages where channels are modelled as classes and session types are aggregated to their class definitions and describe the behaviour of each intervenient in the protocol, more specifically the possible sequence of method calls on an object of the class. [20] also presents the idea of modularizing sessions types, allowing the implementation of session types through multiple methods instead of just one. Through static typing it is possible to guarantee that every sequence of method calls on every non-uniform object and, consequently, every sequence of messages transmitted by channels follow the specification described by the session types.

One of the basis of our work will be the Mool language, developed by Campos and Vasconcelos [3, 4]. Like in [20], the language allows to associate with each class a behavioural type specifying safe orderings of method calls but in contrast it does not include channels, leaving method calls as the only method of communication. Another difference between [20] and Mool is that [20] only deals with linear objects while in Mool objects can evolve from a linear state to shared one, and so the language is extended with qualifiers for aliasing control of objects.

2.5 Session type inference

There are several proposals that aim for session-type inference of programs created without session types.

Hüttel et al. presented in [21] an approach to session type inference for the π-calculus based on constraint generation and solving, but they argue that it should be possible to adapt the work for other programming languages since these constraints are present in other languages with binary session types. Our approach is based on propositional logic formula satisfiability and, while now the formulas are specified by the programmer through assertions, we expect in the future to generate automatically these formulas through syntactic analysis of the code.

Uchitel et al. propose a behavioural model approach [6] consisting on a algorithm that receives the source code of a program written in C, equipped with assertions representing invariants and requires clauses, and constructs automatically a enabledness-preserving behaviour model, which is similar to a typestate. These behaviour models are permissive, meaning that they include every possible operation sequence of the program. In [6] the authors argue that their technique ensures that the constructed behaviour models are always permissive independently of the library's internal state being finite or not.

Collinbourne and Kelly developed a session type inference algorithm for programs written in C [8]. This algoritm is composed by three stages: The first stage consists on converting every communication statement in the code to static single use (SSU) form [27]; In the second stage, a graph describing the session transitions is obtained from the communication statements in the SSU form of the code; in the third stage, the graph is translated into a session type.

Our approach to behavioural type inference is based on the techniques in [6, 8], with some changes necessary to adapt them to the context of our work. These changes are discussed during the presentation of our usage type inference algorithm in Chapter 4.

Chapter 3

A Revision of the Mool Language

This chapter presents an analysis of the Mool language, which is briefly described in Section 2.4.2. This analysis is a contribution to the development of the language, detecting bugs not only in the implementation, but also in the formalisation. We also propose revisions of aspects of the language we find too restrictive [1].

Section 3.1 presents correction proposals. We organise them in two categories: minor aspects (Section 3.1.1), which have little influence on the language or their correction is very straightforward; and major aspects in(Section 3.1.2), which heavily influence the behaviour of the language and are more complex to change.

We complement the analysis of the Mool language formal system with a small review of the Mool compiler (version 0.3, available in May 2016 from `gloss.di.fc.ul.pt/mool/download`). The purpose is to understand if the aspects we presented in Section 3.1 were solved in the implementation, and, if they were, how the compiler copes with them.

To test our analysis, we implemented the original formalisation of Mool using PLT-Redex [14], a module available in Racket [15] that allows us to implement and debug formal systems of programming languages. Section 3.4 presents our implementation and explains briefly the examples we used to demonstrate how the aspects in Section 3.1 affect the language.

Section 3.5 consists on our revision proposal for the Mool language. We present a full formal system, consisting on the revised operational semantics and a type system of the language, based on the original but with changes that try to solve the aspects identified in Section 3.1 plus the addition of new features such as constructors.

Again, to test our revision we implemented the revised formalisation using PLT-Redex. Section 3.6 presents the list of examples used to test this second implementation. Most of these examples are almost identical to the ones in Section 3.4, but now they are expected to have a different behaviour, while some are new examples that were used to test our changes a little further.

Chapter B contains the full syntax, operational semantics and typing rules of our revised version of Mool.

[1] For more details: `https://arxiv.org/abs/1604.06245`

3.1 The original Mool language

Like said before, the main objective is to understand where Mool can be too restrictive or even present incorrect behaviour. We did this by not only reviewing the original definitions [4], but also by implementing the language using PLT Redex and trying to falsify properties of the system (see Section 3.4). These aspects have been categorised in major and minor aspects, based on their complexity.

3.1.1 Minor errors and limitations

The following observations are minor errors and limitations found on Mool, i.e., they are very simple to solve:

1. The evaluation context for while is unnecessary. The evaluation contexts defined in the syntax of Mool specify that in a while expression the expression *e* that serves as the boolean condition must be evaluated before the while expression itself, but the reduction rule R-While specifies that a while expression should be immediately reduced to a if – else expression.

2. T-UsageVar returns a new typing environment but it is not clear why the final environment needs to be be different from the initial.

3. T-Assign restricts assignments to unrestricted variables and fields only, but assignment to linear variables can be possible since any case that can risk linearity can be prevented by a predicate that checks if a variable has a linear type when it should not (for example, that already happens in rule T-Class where is specified that all of the class fields should be unrestricted).

4. T-Call specifies that the parameter type should be the same as the method type, which is unnecessarily restricting.

3.1.2 Major errors and limitations

The following aspects are errors and limitations found on Mool that are more complex to solve:

1. Subtyping for variant types is not well defined. The correct definition, based on the sub-typing definition in [20], is as follows:

   $$\text{If } \langle u' + u'' \rangle <: u \text{ then } u = \langle u_t + u_f \rangle \text{ with } u' <: u_t \text{ and } u'' <: u_f$$

2. Subtyping seems to be unsafe. Consider the following expression:

   $$\mathsf{if}(f.eof())\ \{\ f.close();\ false;\ \}\ \mathsf{else}\ \{\ f.read();\ true;\ \}$$

In this expression, is the file has been fully read then it closes and returns $false$, informing the client that there is no more lines to read, otherwise it reads a line and returns true, informing the client that there is still lines to be read. Assume that we reverse the result output as follow:

$$\mathsf{if}(f.eof())\ \{\ f.close();\ true;\ \}\ \mathsf{else}\ \{\ f.read();\ false;\ \}$$

Mool accepts this, but it can cause a runtime error because the client can try to close an already closed file. In this revision we will not propose a fix for the subtyping since it is not in the context of our work.

3. The typing rule T-Spawn states that the expression e should have an unrestricted type, but that is not enough to prevent situations where the occurrence of statements being executed in different threads can result in the correct execution flow of a program being disrespected. For example, assuming a $File$ class with the usage

$$\mathsf{lin}\{\ open : \mathsf{lin}\{\ read : \mathsf{lin}\{\ close :\ \mathsf{un}\{\ \}\ \}\ \}\ \}$$

where methods *open*, *read* and *close* are all of type unit, the code

$$f.open();\ \mathsf{spawn}\ f.read();\ f.close()$$

which opens the file, creates a separate thread for the reading operation and closes the file, is wrong because after creating the new thread with the reading operation it is not possible to predict the next step, so the file can be read or closed. As defined, the type system will accept this because $f.read()$ has type unit, which is an unrestricted type, and so the typing rule T-Spawn will accept this expression, as the following partial derivation shows:

$$\dfrac{\dfrac{\dots}{\Gamma \rhd f.open() : unit \lhd \Gamma'}\ \text{T-Call} \qquad \text{T1}}{\Gamma \rhd f.open();\ \mathsf{spawn}\ f.read();\ f.close() : unit \lhd \Gamma'''}\ \text{T-Seq}$$

$$\text{T1}\quad \dfrac{\dfrac{\dfrac{\dots}{\Gamma' \rhd f.read() : unit \lhd \Gamma''}\ \text{T-Call}}{\Gamma' \rhd \mathsf{spawn}\ f.read() : unit \lhd \Gamma''}\ \text{T-Spawn} \qquad \text{T2}}{\Gamma' \rhd \mathsf{spawn}\ f.read();\ f.close() : unit \lhd \Gamma''}\ \text{T-Seq}$$

$$\text{T2}\quad \dfrac{\dots}{\Gamma'' \rhd f.close() : unit \lhd \Gamma'''}\ \text{T-Call}$$

$\Gamma = f : File[\mathsf{lin}\{\ open : \mathsf{lin}\{\ read : \mathsf{lin}\{\ close : \ \mathsf{un}\ \{\ \}\ \}\ \}\ \}]$
$\Gamma' = f : File[\mathsf{lin}\{\ read : \mathsf{lin}\{\ close : \mathsf{un}\{\ \}\ \}\ \}]$
$\Gamma'' = f : File[close : \ \mathsf{un}\{\ \}]$
$\Gamma''' = f : File[\mathsf{un}\{\ \}]$

4. Usages allow incorrect specifications of sequence of methods calls. Consider the following usage type:

$$\mathsf{lin}\{\ read : \mu Read.\mathsf{un}\{\ eof : \langle close : \mathsf{un}\{\} + read : Read\rangle\ \}\ \}$$

This usage describes a behaviour for a *File* class, where method *read* depends on variables initialized by a method *open* that is implemented as a private method and is never called, that allows to read a line from a file before opening it but the typechecker allows it.

5. The type checker does not check if a field is initialised or not, allowing these to be dereferenced even when they are not.

6. The type system does not have typing rules for self calls. Although the typing rules for self calls were deliberately omitted from [4], they are essential since in case of recursion, the type system will not terminate the program evaluation. For instance, the method *run* of the class *Seller* of the example presented in Chapter 2 of [4] is an example of a program that contains a self call that causes the type checker to go into an infinite loop.

7. Private methods are not evaluated since the type system, as defined, only checks methods in the class usage, which the system description considers public, and self calls are not included in the type system.

8. Typing rules for the control flow expressions with method calls as conditions are not applied when the method call is preceded of a negation, like

$$\mathsf{if}(!f.eof())\ \{\ f.read()\ \}\ \mathsf{else}\ \{\ f.close()\ \}$$

, treating these calls as regular expressions and so it does not operate the necessary usage changes.

9. The language formalisation does not allow unrestricted classes, i.e., classes without usages.

10. null cannot be used as a value, not allowing the programmer to set objects to null or check if they are null.

11. An usage can go from an unrestricted state into a different state. According to the system description, an usage cannot go from an unrestricted state into a linear state.

$$\mathsf{lin}\{\, open : \mu Read.\mathsf{un}\{\, eof : \langle close : \mathsf{un}\{\,\} + read : Read\rangle \,\}\,\}$$

This usage, presented in the configuration of the core language, is a slightly modified usage to the File class of the example presented in [4]. The type system, as defined, will accept this usage but it clearly represents a situation where the usage goes from unrestricted to linear since when executing the method *open* the usage goes from linear to unrestricted and when executing the method *eof* the usage goes back to being linear.

Although, the same concerns are valid when an usage is composed by several unrestricted states and it transits between unrestricted states. Consider a variation of the FileReader class that hosts a file whose reading access can be blocked or unblocked. A possible usage would be:

$$\mathsf{lin}\{\, open : \mu Blocked.\mathsf{un}\{\, unblock : \\ \mu Unblocked.\mathsf{un}\{\, block; Blocked + read : Unblocked \,\}\,\}\,\}$$

Consider also a situation where an instance of this $FileReader$ class, in state $Unblocked$, is shared between two clients. Since the usage allows concurrent interaction with the instance, it is possible for one client to execute *read* and the other client to execute *block* at the same time and the *block* operation terminates before the *read* operation. The client that is trying to read will do it while the usage is in state $Blocked$, which is not the expected behaviour.

When in an unrestricted state, not only it must no return to a linear state it also must only go to the same state or to an equivalent state (i.e., a state with the exact same actions), like the following example:

$$\mathsf{lin}\{\, open : \mu Blocked.\mathsf{un}\{\, push : \mu Unblocked.\mathsf{un}\{\, push : Blocked \,\}\,\}\,\}$$

Although the original definition [4] lacked the ability to declare local variables, it was mentioned that the implementation of Mool at the time had allowed it, so this aspect was omitted from this list.

3.2 Latest Mool implementation

The work developed and presented in the following sections is based on the Mool language presented in [4], but we also reviewed the current Mool implementation available [2] to check if the aspects noted in Sections 3.1.1 and 3.1.2 still remain or not and try to understand how the language copes with those aspects. The examples used in this section are based on the $FileAll.mool$ example.

[2]The latest Mool implementation is available at `gloss.di.fc.ul.pt/tryit/Mool`

To check if the subtyping in the current version is still unsafe, consider the following code:

Listing 3.1: *FileReader* subtyping example

```
if(f.eof()) {
   f.close();
   true;
} else {
   s = s ++ f.read();
   false;
}
```

While using this code as the body of the method *next* of the *FileReader* class, the compiler accepts it but running it will cause an infinite loop, which not only is a runtime error, it goes against the behaviour specified by the usage since the interaction with the file should be terminated after closing it, but in this example the *FileReader* will execute the methods *eof* and *close*. This proves that subtyping is still unsafe.

The compiler for the current Mool implementation checks if all of class fields are initialised, even if they are not used, instead of waiting for a runtime error, showing that the problem presented in item 6 of Section 3.1.2 seems to be fixed. The compiler also allows to assign values of linear type to variables, showing that the restriction mentioned in item 3 of Section 3.1.1 was dropped, allowing code like this:

Listing 3.2: *FileReader* linear attribution example

```
FileReader f; f = new FileReader();
f.open();

FileReader f2; f2 = new FileReader();
f2.open();

f = f2;
```

Moreover, it is possible to observe two aspects of the spawn construct: Mool does not allow *e* to be a sequential composition (it must only be a single expression) and not only it must be a method call, it must consume that variable's usage. This last aspect hints that the rule T-Spawn checks if all variables in the typing environment are unrestricted after executing *e*. Using the example presented in item 2 of Section 3.1.2, with a class *File* with the following usage:

Listing 3.3: *File* usage variation

```
class File {
   usage lin{open; Read} where
      Read = lin{read ; Close}
      Close = lin{close ; end};
   ...
}
```

The following code, which is identical to the one from the example, will not compile, with the compiler saying that it expected *f* to be *null* in the third line :

Listing 3.4: *FileReader* spawn example 1

```
File f; f = new File();
f.open();
spawn f.read();
f.close();
```

However, the following code will compile, because the method *close* finalises the consumption of *f*'s usage:

Listing 3.5: *FileReader* spawn example 2

```
File f; f = new File();
f.open();
f.read();
spawn f.close();
```

About the unsafe sequence of calls in item 4 of section 3.1.2, consider the following example:

Listing 3.6: *File* unsafe usage

```
class File {
   usage lin{read; Read} where
     Read = lin{eof;
     <lin{close; end} + lin{read; Read}>};
}
```

Replacing the original usage of the *FileAll.mool* example with the one presented above will result in the program entering an infinite loop, due to the fact that the method *open* is never called, meaning that both variables *linesRead* and *linesInFile* are never explicitly initialized and so both are initialized with the default value which is 0. It is valid to assume that, while the current version of Mool checks if a variable is initialized in the code, it seems to not check if that initialization happens during the execution of the program, leading to these type of situations.

About the use of negated calls as conditions in control flow expressions, the current compiler still has this limitation. The following example will not compile, saying that the method *read* must be called on a control flow expression:

Listing 3.7: *FileReader* negated call example 1

```
if(!f.eof()) {
   s = s ++ f.read();
   true;
} else {
   f.close();
   false;
}
```

The message given by the compiler is not very clear since the method *read* is being called inside a control flow expression but the reason for this error is due to the fact that, during the type-checking process, the rule T-IF is applied instead of the rule T-IFV, and it does not operate the necessary changes to the usage of the field f so that method *read* is available to be called inside the first branch and the method *close* inside the second. Another example is the following code where a while expression is used but the compiler does not accept the code for the same reason as the previous example:

Listing 3.8: *FileReader* negated call example 2

```
while(!f.eof()) {
   s = f.read() ++ s;
}
```

The current compiler allows classes to be unrestricted, as shown by the example *PetitionAll.mool* which has unrestricted classes such as *Main* and *PetitionServer*.

Furthermore, the current compiler does not allow an usage to go from unrestricted to linear. The following example will not compile:

Listing 3.9: *FileReader* bad usage example 1

```
class File {
   usage lin{open; Read} where
      Read = un{eof;
               <lin{close; end} + lin{read; Read}>};
   ...
}
```

Furthermore, the current compiler does not allow an usage to go from unrestricted to linear. The following example will not compile:

Listing 3.10: *FileReader* bad usage example 2

```
class File {
   usage lin{open; Read} where
      Read = un{eof;
               <lin{close; end} + lin{read; Read}>};
   ...
}
```

But the compiler can accept an usage that goes from an unrestricted state to another different unrestricted state, like the following one:

Listing 3.11: *FileReader* bad usage example 3

```
class FileReader {
   usage lin{open; Blocked} where
      Blocked = un{unblock; Unblocked}
      Unblocked = un{read; Unblocked + block; Blocked};
   ...
}
```

3.3 Testing the formalization

Testing the formalization can be helpful to confirm the issues we presented in Section 3.1. Since producing derivations of executions or of typing is tedious and error-prone, implementing the reduction rules and the type system is crucial to avoid the above mentioned difficulties but may be very time consuming.

To test the reduction rules and the type system of Mool we use the Racket language, a programming language that supports other programming languages, more specifically its module PLT-Redex.

3.3.1 Racket

Racket is a programming language in the Lisp family, meaning that while it can be used to create solutions like any conventional programming language, it also allows a language-oriented programming, i.e., allows creating new programming languages. To support this feature, Racket provides building blocks for protection mechanisms, which allows the programmers to protect individual components of the language from their clients, and the internalization of extra-linguistic mechanisms, such as project contexts and the delegation of program execution and inspection to external agents, by converting them into linguistic constructs, preventing programmers to resort to mechanism outside Racket [15].

3.3.2 PLT Redex

PLT Redex is a domain-specific language embedded in Racket that allows programmers to formalize and debug programming languages. The modeling of a programming language in Redex is done by writing down the grammar, reductions of the language along with necessary metafunctions. Since Redex is embedded in Racket, programming in Redex is just like programming in Racket, with all of the features and tools available for Racket being also available for Redex, including DrRacket, a integrated development environment for Racket. One of the most interesting advantages of using DrRacket is the automatically generated reduction graphs that allows programmers to visualize reductions step by step (examples presented in Section A). Redex also has other methods of testing, such as pattern matcher (for grammar testing) and judgment-form evaluation (which we use to test the type system) [14, 26].

3.4 PLT Redex implementation of the original formalization

We implemented Mool as presented in [4] using PLT Redex [3]. Due to the syntax of Racket, we had to make some modifications on the syntax of Mool, such as:

[3]Available at `https://sourceforge.net/p/mool-plt-redex/code/ci/master/tree/mool1.rkt`

- Every expression must be in parenthesis.
- **;** is reserved by Racket, so it cannot be used to separate expressions.
- **.** is also reserved by Racket, so it was replaced by **->**.
- To help implementing the type system, the usage variables **X** were replaced by **!X** so they could be distinguished from regular variables.
- A new construct, getref, was added to the runtime syntax. This new construct returns the last object identifier created so it can be assigned to a field.
- In the runtime syntax used by the type system, nonterminals *u* and *D* were added to *e* since there must be only one domain which, in this case, is *e*.

In addition to the language implementation, the code also contains a few examples to show some of the problems noted in Section 3.1.2. In order to implement more elaborate examples, some other changes were made:

- Items 1, 2 and 4 of Section 3.1.1 are already solved in the implementation.
- A typing rule for self calls was added. It is the same as T-Call but it does not change the usage, as the system description specifies.
- Arithmetic and boolean expressions were implemented.

Finally, since this does not exist in this version, the object identifier 0 was reserved to represent this, so every class field access and self call are done in 0. The examples are the following:

R-01 Implementation of the File example presented in [4], with an modification on how the program checks if it has reached the end of the file, due to the limitation presented in item 7 of section 3.1.2. This example serves to test the operational semantics of Mool and when running it the reduction graph of the program's reduction will be shown.

T-01 Typing example of the File example. When run the type system should be able to check the whole program with success.

T-02 Typing example that implements the situation expressed in item 2 of Section 3.1.2. The type checker verifies successfully when it should not.

T-03 Implementation of the example presented in item 3 of Section 3.1.2. The type checker evaluates the program successfully even though it is not desirable to have a situation where the file can be closed before being read.

T-04 Same thing as T-01 but the fields *f* from the *FileReader* class and from the *Main* class are not initialised, while both are dereferenced as in T-01. The program is evaluated successfully, allowing both fields to be dereferenced even though they are not initialised.

T-05 Same thing as T-01 but in the usage of the *File* class the method *open* is replaced by the method *read*, same as the usage presented in item 4 of Section 3.1.2. The usage allows to read the file without opening it but the typechecker verifies the program successfully.

T-06 A variation of the *File* example where the body of the method *count* is changed to *true*. The return type of the method is *unit* but the body of the method is of type *boolean* and the type checker verifies the program successfully since the body of the method is not verified, only its signature;

T-07 Implementation of the *File* and *FileReader* classes as presented in [4], including the using of a negated method call as a condition for a control flow expression in method *next* of *FileReader*. This example serves to demonstrate the limitation presented in item 8 of Section 3.1.2 and it should fail.

T-08 A variation of the *File* and *FileReader* classes, where now the method *next* of *FileReader* reads the whole file at once. This example is to demonstrate again the limitation presented in item 8 of section 3.1.2 with the same result, but now in a while expression.

T-09 Another typing example that shows that the type system allows an usage to go from unrestricted to linear. This program only contains one class, *File*, but its usage is the same as the first example given in item 11 of Section 3.1.2.

T-10 A simplistic version of the *FileReader* where the methods do not do anything but the usage, which is the same as the second usage presented in item 11 of Section 3.1.2, is composed by two different unrestricted states and they change between them. This should not be allowed but the type checker allows it.

3.5 The revised Mool language

This section presents our revision of the Mool language that tries to solve the problems mentioned in Sections 3.1.1 and 3.1.2. Some of the modifications are based on the observations made in Section 3.2.

3.5.1 Revised syntax

Figure B.1 shows a modified syntax for the Mool language. This revised syntax contains the following new/changed elements:

1. Arithmetic and boolean expressions, represented by the nonterminals a and b respectively.

2. A new nonterminal r for value references, which contains local variables d and this (to help solving the problems noted in items 5 and 6 in Section 3.1.2).

3. Expressions e contain now only values and expressions, including calls, and put the rest of the constructs in a new nonterminal s that represents statements.

4. Constructs $g\ d = e$ and $d = e$ to s to allow local variable declaration and assignment.

5. Since we want to add the concept of constructor in the language, we modified the construct $\mathsf{new}\ C()$ to $\mathsf{new}\ C(e)$, allowing to pass parameters to the constructor.

6. We divided types into two nonterminals, g and t. g contains types that can be used to declare fields and variables, while t contains every type in g plus every other type such as and null

7. We divided the usages into two nonterminals, u and z. u contains the usage constructs that can be used right at the beginning of the usage while z contains the usage constructs used during compile time. In the runtime syntax we added z to u to avoid too many changes to the typing rules.

8. In the nonterminal u we added ϵ to indicate that it is possible to not define an usage, making the class an unrestricted class.

9. The term o, which are objects identifiers, is moved from the user syntax for the runtime syntax.

10. In the runtime syntax, a new type of value, null, is added and it is used to represent values for non initialised objects, and a new type $C[u;\vec{F}]$, where $\vec{F}$ are mappings from fields that are initialised to types, is added to solve the problems in items 4 and 5 of Section 3.1.2.

11. The evaluation context $\mathsf{while}\ \mathscr{E}\ e$ is removed for the reasons stated in item 1 of Section 3.1.1.

3.5.2 Revised operational semantics

Figure B.4 shows the modified reduction rules for this revised version of Mool. The rules differ from the original ones, as we add a new environment, *local*, for the local variables.

We modified the rule R-New so that it reduces to a sequential composition with the body of the constructor and the created object identifier.

We also add the new rules R-NewVar and R-AssignVar which are for local variable declaration and assignment.

User Syntax

(class declarations)	$D ::= \mathsf{class}\ C\ \{u;\vec{F};\vec{M}\}$
(field declaration)	$F ::= g\ f$
(method declarations)	$M ::= y\ t\ m(t'\ x)\ \{e\}$
(method qualifiers)	$y ::= \epsilon \mid \mathsf{sync}$
(values)	$v ::= \mathsf{unit} \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null}$
(local value references)	$r ::= d \mid \mathsf{this}$
(global value references)	$w ::= r \mid r.f$
(calls)	$c ::= \mathsf{new}\ C(e) \mid r.m(e) \mid r.f.m(e)$
(arithmetic operations)	$a ::= n \mid w \mid c \mid a+a \mid a-a$
	$\mid a*a \mid a/a$
(boolean operations)	$b ::= \mathsf{true} \mid \mathsf{false} \mid w \mid c \mid a == a \mid a\,!= a$
	$\mid a <= a \mid a >= a \mid a < a \mid a > a$
	$\mid b\ \&\&\ b \mid b \mathbin{\Vert} b \mid !b$
(expressions)	$e ::= v \mid a \mid b$
	$\mid c \mid w$
(statements)	$s ::= e \mid s;s'$
	$\mid r.f = e \mid g\ d = e \mid d = e$
	$\mid \mathsf{if}\ (b)\ s'\ \mathsf{else}\ s'' \mid \mathsf{while}\ (b)\{s'\}$
	$\mid \mathsf{spawn}\{s\}$
(types)	$t ::= \mathsf{void} \mid g \mid \mathsf{null}$
(declarable types)	$g ::= \mathsf{int} \mid \mathsf{bool} \mid C[z]$
(class usages)	$u ::= \epsilon \mid q\{m_i;z_i\}_{i \in I} \mid \mu X.u$
(usages)	$z ::= u \mid \langle u+u \rangle \mid X$
(usage types)	$q ::= \mathsf{un} \mid \mathsf{lin}$

Runtime Syntax

(values)	$v ::= \ldots \mid o$
(value references)	$r ::= \ldots \mid o$
(class usages)	$u ::= \ldots \mid z$
(types)	$t ::= \ldots \mid C[z;\vec{F}]$
(object records)	$R ::= (C, u, \overrightarrow{f = v}, l)$
(field value map)	$l ::= 0 \mid 1$
(heap)	$h ::= \emptyset \mid h, o = R$
(evaluation context)	$\mathscr{E} ::= [-] \mid \mathscr{E};s \mid o.f = \mathscr{E} \mid o.m(\mathscr{E}) \mid o.f.m(\mathscr{E})$
	$\mid \mathsf{if}\ (\mathscr{E})\ s\ \mathsf{else}\ s'$
(States)	$S ::= (h, local, s_1 \mid \ldots \mid s_n)$

Figure 3.1: Revised syntax

Object Record and Heap Operations

$$\langle C,u,\vec{V}\rangle.f \overset{\text{def}}{=} \vec{V}(f) \qquad\qquad \langle C,u,\vec{V}\rangle.\mathsf{usage} \overset{\text{def}}{=} u$$

$$\langle C,u,\vec{V}\rangle.\mathsf{class} \overset{\text{def}}{=} C$$

Operations for values and types

$$\mathsf{lin}(v) \overset{\text{def}}{=} \begin{cases} tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \langle u' + u'' \rangle \\ tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \mathsf{lin}\{m_i; z_i\}_{i\in I} \\ f\!f & \text{otherwise} \end{cases} \quad \mathsf{un}(v) \overset{\text{def}}{=} \begin{cases} tt & \text{if } v = \mathsf{unit} \\ tt & \text{if } v = n \\ tt & \text{if } v = \mathsf{true} \\ tt & \text{if } v = \mathsf{false} \\ tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \mathsf{un}\{m_i; z_i\}_{i\in I} \\ f\!f & \text{otherwise} \end{cases}$$

Class Definition Operations

$$C.\mathsf{methods} \overset{\text{def}}{=} \overrightarrow{M, eval} \quad \text{where class } C\ \{u;\vec{F};\vec{M}\} \in \vec{D} \text{ and } eval \in \{0,1\}$$

$$C.\mathsf{fields} \overset{\text{def}}{=} \vec{F} \quad \text{where class } C\ \{u;\vec{F};\vec{M}\} \in \vec{D}$$

$$C.\mathsf{usage} \overset{\text{def}}{=} u \quad \text{where class } C\ \{u;\vec{F};\vec{M}\} \in \vec{D}$$

Figure 3.2: Auxiliary definitions and Operations

$$\text{R-CONTEXT}\ \frac{(h, local, s_1 \mid \ldots \mid s \mid \ldots \mid s_n) \longrightarrow (h', local', s_1 \mid \ldots \mid s' \mid \ldots \mid s_n)}{(h, local, s_1 \mid \ldots \mid \mathscr{E}[s] \mid \ldots \mid s_n) \longrightarrow (h', local', s_1 \mid \ldots \mid \mathscr{E}[s'] \mid \ldots \mid s_n)}$$

$$\text{R-SPAWN}\ (h, local, s_1 \mid \ldots \mid \mathscr{E}[\mathsf{spawn}\{s\}] \mid \ldots \mid s_n) \longrightarrow (h, local, s_1 \mid \ldots \mid \mathscr{E}[\mathsf{unit}] \mid \ldots \mid s_n)$$

Figure 3.3: Reduction semantics for states

The rule R-ASSIGNFIELDNULL, allows to assign null values to fields, removing them from the object's record.

Figures B.5 and B.6 show the evaluation functions for the arithmetic and boolean expressions. These functions, based on the ones presented in [29], receive as arguments an expression and both the class field and local variable environment.

3.5.3 Revised type system

In this section we present a new set of typing rules. We omit the unchanged rules with respect to the original system [4].

Figure B.8 shows the proposed typing rules for programs:

1. Rule T-CLASS is a modified version of the rule with the same name that has a new premise that checks if the class usage is correct, i.e., it does not go from an unrestricted state to a linear one at any point.

   Moreover, evaluation of the usage has an object $C[u;\varnothing]$ for input, with no declared fields, and a object $C[u';\vec{F}]$ for output, forcing the method-level scope of the system.

$$\textsc{R-UnField}\ \frac{h(o).f = v \qquad \mathsf{un}(v,h)}{(h, local, o.f) \longrightarrow (h, local, v)} \qquad \textsc{R-LinField}\ \frac{h(o).f = v \qquad \mathsf{lin}(v,h)}{(h, local, o.f) \longrightarrow (h\{o.f \mapsto \mathsf{null}\}, local, v)}$$

$$\textsc{R-UnVar}\ \frac{h(d) = v \qquad \mathsf{un}(v,h)}{(h, local, d) \longrightarrow (h, local, v)} \qquad \textsc{R-LinVar}\ \frac{h(d) = v \qquad \mathsf{lin}(v,h)}{(h, local, d) \longrightarrow (h\{d \mapsto \mathsf{null}\}, local, v)}$$

$$\textsc{R-Seq}\ (h, local, v; s) \longrightarrow (h, local, s) \qquad \textsc{R-NewVar}\ (h, local, g\ d = v) \longrightarrow (h, local\{d \mapsto v\}, \mathsf{unit})$$

$$\textsc{R-AssignVar}\ (h, local, d = v) \longrightarrow (h, local\{d \mapsto v\}, \mathsf{unit})$$

$$\textsc{R-AssignField}\ \frac{v \neq \mathsf{null}}{(h, local, o.f = v) \longrightarrow (h\{o.f \mapsto v\}, local, \mathsf{unit})}$$

$$\textsc{R-AssignFieldNull}\ \frac{v = \mathsf{null}}{(h, local, o.f = v) \longrightarrow (h \setminus o.f, local, \mathsf{unit})}$$

$$\textsc{R-New}\ \frac{o\ \mathsf{fresh} \qquad (\_\ C(\_\ x)\ \{s\}, \_) \in C.\mathsf{methods} \qquad C.\mathsf{fields} = \overrightarrow{t\ f} \qquad C.\mathsf{usage} = u}{(h, local, \mathsf{new}\ C(v)) \longrightarrow ((h, o = \langle C, u, \overrightarrow{f = \mathsf{null}} \rangle), local, s\{^o/_{\mathsf{this}}\}\{^v/_x\}; o)}$$

$$\textsc{R-Call}\ \frac{(\_\ m(\_\ x)\ \{s\}, \_) \in (h(o).\mathsf{class}).\mathsf{methods}}{(h, local, o.m(v)) \longrightarrow (h, local, s\{^o/_{\mathsf{this}}\}\{^v/_x\})}$$

$$\textsc{R-FieldCall}\ \frac{(\_\ m(\_\ x)\ \{s\}, \_) \in (h(o).f.\mathsf{class}).\mathsf{methods}}{(h, \varnothing, o.f.m(v)) \longrightarrow (h, \varnothing, s\{^o/_{\mathsf{this}}\}\{^v/_x\})}$$

$$\textsc{R-While}\ (h, local, \mathsf{while}\ (b)\{s\}) \longrightarrow (h, local, \mathsf{if}\ (b)\ (s; \mathsf{while}\ (b)\{s\})\ \mathsf{else\ unit})$$

$$\textsc{R-IfTrue}\ (h, local, \mathsf{if\ (true)}\ s'\ \mathsf{else}\ s'') \longrightarrow (h, local, s')$$

$$\textsc{R-IfFalse}\ (h, local, \mathsf{if\ (false)}\ s'\ \mathsf{else}\ s'') \longrightarrow (h, local, s'')$$

Figure 3.4: Revised reduction semantics for statements

$$\mathcal{N}(n) = n \qquad \mathcal{A}(n, h, local) = \mathcal{N}(n)$$

$$\mathcal{A}(o.f, h, local) = h(o).f \qquad \mathcal{A}(d, h, local) = local(d)$$

$$\mathcal{A}(a_1 + a_2, h, local) = \mathcal{A}(a_1, h, local) + \mathcal{A}(a_2, h, local)$$

$$\mathcal{A}(a_1 - a_2, h, local) = \mathcal{A}(a_1, h, local) - \mathcal{A}(a_2, h, local)$$

$$\mathcal{A}(a_1 * a_2, h, local) = \mathcal{A}(a_1, h, local) * \mathcal{A}(a_2, h, local)$$

$$\mathcal{A}(a_1\ /\ a_2, h, local) = \mathcal{A}(a_1, h, local)\ /\ \mathcal{A}(a_2, h, local)$$

Figure 3.5: Evaluation functions for arithmetic values and expressions

$$\mathcal{B}(\mathsf{true}, h, local) = \mathsf{true} \qquad \mathcal{B}(\mathsf{false}, h, local) = \mathsf{false}$$

$$\mathcal{B}(o.f, h, local) = h(o).f \qquad \mathcal{B}(d, h, local) = local(d)$$

$$\mathcal{B}(a_1 == a_2, h, local) = \begin{cases} \mathsf{true} & \mathcal{A}(a_1, h, local) = \mathcal{A}(a_2, h, local) \\ \mathsf{false} & \mathcal{A}(a_1, h, local) \neq \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(a_1 != a_2, h, local) = \begin{cases} \mathsf{true} & \mathcal{A}(a_1, h, local) \neq \mathcal{A}(a_2, h, local) \\ \mathsf{false} & \mathcal{A}(a_1, h, local) = \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(a_1 < a_2, h, local) = \begin{cases} \mathsf{true} & \mathcal{A}(a_1, h, local) < \mathcal{A}(a_2, h, local) \\ \mathsf{false} & \mathcal{A}(a_1, h, local) >= \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(a_1 < a_2, h, local) = \begin{cases} \mathsf{true} & \mathcal{A}(a_1, h, local) < \mathcal{A}(a_2, h, local) \\ \mathsf{false} & \mathcal{A}(a_1, h, local) <= \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(a_1 <= a_2, h, local) = \begin{cases} \mathsf{true} & \mathcal{A}(a_1, h, local) <= \mathcal{A}(a_2, h, local) \\ \mathsf{false} & \mathcal{A}(a_1, h, local) > \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(a_1 >= a_2, h, local) = \begin{cases} \mathsf{true} & \mathcal{A}(a_1, h, local) >= \mathcal{A}(a_2, h, local) \\ \mathsf{false} & \mathcal{A}(a_1, h, local) < \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(b_1 \ \&\& \ b_2, h, local) = \begin{cases} \mathsf{true} & \mathcal{B}(b_1, h, local) = \mathsf{true} \wedge (b_2, h, local) = \mathsf{true} \\ \mathsf{false} & \mathsf{otherwise} \end{cases}$$

$$\mathcal{B}(b_1 \parallel b_2, h, local) = \begin{cases} \mathsf{true} & \mathcal{B}(b_1, h, local) = \mathsf{true} \vee (b_2, h, local) = \mathsf{true} \\ \mathsf{false} & \mathsf{otherwise} \end{cases}$$

$$\mathcal{B}(!b, h, local) = \begin{cases} \mathsf{true} & \mathcal{B}(b, h, local) = \mathsf{false} \\ \mathsf{false} & \mathcal{B}(b, h, local) = \mathsf{true} \end{cases}$$

Figure 3.6: Evaluation functions for boolean values and expressions

In the end it checks if all fields in $\vec{F}$ are unrestricted.

2. Rule T-UnClass is for unrestricted classes and, instead of verifying the usage, it verifies all of the methods of the class.

   We assume that in unrestricted classes every method is independent, i.e., the changes it introduces to the state of the object do not affect other methods (e.g. initialised fields), so every method is verified using the same typing environments.

Figure B.9 shows the proposed typing rules for usages:

1. Rule T-BranchEnd is a variation of T-Branch that is applicable when a usage branch terminates and so it does only evaluate the method, not the next usage (because there is none).

2. Rule T-UsageVar returns the same typing environment mapped to the usage variable, reflecting the observation made in the item 2 of 3.1.1.

Figures B.11 and B.12 show the typing rules for the arithmetic and boolean expressions. While the syntax itself already enforces the correct types, we need these rules because the operands can change the usage, e.g., a call made on a field as an operand.

Figure B.13 shows the proposed typing rules for field and variable dereference, where we added a new rule, T-NullField, for dereference of fields that have not initialized.

Figure B.14 shows the proposed typing rules for simple statements:

1. Two new rules, T-NewVar and T-AssignVar, for local variable declaration and assignment respectively, are added so the type checker can evaluate local variable declarations.

   Both T-AssignVar and T-AssignField allow linear type value assignment, solving the limitation in item 3 of Section 3.1.1.

2. The rule T-Spawn is modified based on the conclusions presented in Section 3.2, making the type checker checking that:

   a) all variables modified in *s* are unrestricted; and

   b) in case of variables that are objects, the usage is fully consumed and therefore cannot be called in any other expression outside the spawn;

   c) instead of checking if *s* has unrestricted type, it allows expressions other than method calls and it also allows *s* to be a sequential composition.

Figure B.15 shows the proposed typing rules for control flow expressions:

1. Rules T-IfCall and T-WhileCall are similar to the original rules T-IfV and T-WhileV, but we extended them so they can be applied to method calls made on local variables as the conditional expressions for these control flow expressions.

   Both these rules are replicated for unrestricted classes through rules T-UnIfCall and T-UnWhileCall, with the difference being that there is no usage modification because there is no usage, so they are essentially the rules T-If and T-While but instead of having a value as a condition they have a call on a object of a unrestricted class.

2. To solve the error in item 7 in section 3.1.2 we added the rules T-IfNotCall and T-WhileNotCall, which are similar to the rules T-IfCall and T-WhileCall but are for cases where the method call that serves as the condition is negated, resulting in the inverted attribution of the appropriate usage from the variant type given after the verification of the condition to the expressions that compose the control flow expression.

3. Rules T-If and T-While, which are for cases where the condition is simply a value and not a method call, are similar to the original rules with the same name, but the condition is a value v instead of an expression e.

   All four rules related to the while control flow expression were modified so that they allow modifications inside the loop, but to ensure that, in rules T-WhileCall and T-WhileNotCall, it is possible to execute the condition after executing the loop, both rules state that the type (and, consequently, the usage) of w after the loop must be the same as the type w has before executing the condition.

Figure B.17 shows the proposed typing rules for method calls:

1. We modified the rule T-New so that the constructor is evaluated has a call at the moment of initialisation and added the rule T-UnNew for unrestricted classes initialisation.

2. Rules T-SelfCall1 and T-SelfCall2, which are for method calls made on this, are added to solve the problem stated in item 5 in section 3.1.2.

   Unlike the other typing rules for method calls, these do not change the usage, like the system description in [4] specified, and check if the method was already evaluated or not, so that the type checker only checks a method body once in case of self calls, to prevent entering into a loop when the method is recursive. To check this, the *methods* definition presented in B.2 is changed so that every method is associated to a boolean operator that informs if the method was already evaluated or not. This operator is ignored in the other method call rules.

3. Rule T-Call is similar to the original rule with the same name, but it is extended for method calls made on local variables and also with the minor error mentioned in item 4 of 3.1.1 corrected.

   Moreover, due to the definition of the predicate *allows*, in particular the case when the usage is ϵ, i.e., the class is unrestricted, the predicate also returns ϵ, this rule can also be applied when the call is made on a object of an unrestricted class.

About the subtyping not being safe, one possible solution would be modifying every if-else typing rule to force both branches to be equivalent, i.e., to produce the same changes to the interacted objects. For example, consider the following derivation:

$$\dfrac{\Gamma_1 \rhd f.eof() : \mathsf{bool} \lhd \Gamma_2 \qquad T1 \qquad T2}{\Gamma_1 \rhd \mathsf{if}\ (f.eof())\ \{\ f.close();\ \mathsf{true};\ \}\ \mathsf{else}\ \{\ f.read();\ \mathsf{false};\ \} : t \lhd \Gamma_6}\ \textsc{T-IfCall}$$

$$T1 \quad \dfrac{\dfrac{\dfrac{}{\Gamma_3 \rhd f.close() : \ \lhd \Gamma_5}\ \textsc{T-Call} \qquad \dfrac{}{\Gamma_5 \rhd \mathsf{true} : \ \lhd \Gamma_5}\ \textsc{T-True}}{\Gamma_3 \rhd f.close();\ \mathsf{true} : \ \lhd \Gamma_5}\ \textsc{T-Seq}}{\Gamma_3 \rhd f.close();\ \mathsf{true} : \ \lhd \Gamma_6}\ \textsc{T-InjR}$$

$$T2 \quad \dfrac{\dfrac{\dfrac{}{\Gamma_4 \rhd f.read() : \ \lhd \Gamma_1}\ \textsc{T-Call} \qquad \dfrac{}{\Gamma_1 \rhd \mathsf{true} : \ \lhd \Gamma_1}\ \textsc{T-False}}{\Gamma_4 \rhd f.read();\ \mathsf{false} : \ \lhd \Gamma_1}\ \textsc{T-Seq}}{\Gamma_4 \rhd f.read();\ \mathsf{false} : \ \lhd \Gamma_6}\ \textsc{T-InjL}$$

$\Gamma_1 = f : File[Read]$
$\Gamma_2 = f : File[\langle \mathsf{lin}\{\ close : \mathsf{un}\{\ \}\ \} + \mathsf{lin}\{\ read : Read\ \}\rangle]$
$\Gamma_3 = f : File[\mathsf{lin}\{\ close : \mathsf{un}\{\ \}\ \}]$
$\Gamma_4 = f : File[\mathsf{lin}\{\ read : Read\ \}]$
$\Gamma_5 = f : File[un\{\ \}]$
$\Gamma_6 = f : \langle \Gamma_1 + \Gamma_5 \rangle$

This derivation is of the same example we used to show that subtyping can be unsafe in item 2 of section 3.1.2, and demonstrates that the type system allows it to be verified. If we remove the subtyping this example would not pass, but neither would any other correct example.

To show how the type system would behave without subtyping For example consider the following usage:

$$\mathsf{lin}\{\ open : \mu Read.\mathsf{un}\{\ eof : \langle close : \mathsf{un}\{\ \} + read : \mathsf{un}\{\ \}\rangle\ \}\ \}$$

This usage is a variation of the *File* usage, with the difference being that after executing *read* the file is fully read and closes automatically. With this usage, the previous example would work without subtyping:

$$\dfrac{\Gamma_1 \rhd f.eof() : \mathsf{bool} \lhd \Gamma_2 \qquad T1 \qquad T2}{\Gamma_1 \rhd \mathsf{if}\ (f.eof())\ \{\ f.close();\ \mathsf{true};\ \}\ \mathsf{else}\ \{\ f.read();\ \mathsf{false};\ \} : t \lhd \Gamma_5}\ \textsc{T-IfCall}$$

$$T1 \quad \dfrac{\dfrac{}{\Gamma_3 \rhd f.close() : \ \lhd \Gamma_5}\ \textsc{T-Call} \qquad \dfrac{}{\Gamma_5 \rhd \mathsf{true} : \ \lhd \Gamma_5}\ \textsc{T-True}}{\Gamma_3 \rhd f.close();\ \mathsf{true} : \ \lhd \Gamma_5}\ \textsc{T-Seq}$$

$$T2 \quad \dfrac{\dfrac{}{\Gamma_4 \rhd f.read() : \ \lhd \Gamma_1}\ \textsc{T-Call} \qquad \dfrac{}{\Gamma_1 \rhd \mathsf{true} : \ \lhd \Gamma_5}\ \textsc{T-False}}{\Gamma_4 \rhd f.read();\ \mathsf{false} : \ \lhd \Gamma_5}\ \textsc{T-Seq}$$

$\Gamma_1 = f : File[Read]$
$\Gamma_2 = f : File[\langle \mathsf{lin}\{\ close : \mathsf{un}\{\ \}\ \} + \mathsf{lin}\{\ read : \mathsf{un}\{\ \}\}\rangle]$
$\Gamma_3 = f : File[\mathsf{lin}\{\ close : \mathsf{un}\{\ \}\ \}]$
$\Gamma_4 = f : File[\mathsf{lin}\{\ read : \mathsf{un}\{\ \}\ \}]$
$\Gamma_5 = f : File[un\{\ \}]$
$\Gamma_6 = f : \langle \Gamma_1 + \Gamma_5 \rangle$

Although this would work, it can be too restrictive to force both branches to leave the interacted object with the same usage in both returned environments. Because of this and the fact that proposing a more appropriate solution requires a deeper study that is out of the context of our work, we choose to ignore from now on.

3.6 PLT Redex implementation of the revised Mool formalization

To test our revision we implemented our formal system in PLT Redex.[4] Some of the examples in this version, aside from local variables and the use of this as an value reference, are equal to the ones in the PLT Redex implementation of Mool presented in Section 3.4. The examples presented in this version are the following:

R-01 Implementation of the *File* example presented in [4] to test the operational semantics of Mool. Running it will result in the reduction graph of the program's reduction being shown.

R-01 Implementation of the *File* example presented in [20].

R-03 Example of a small program that uses an unrestricted class. The program contains the class *Folder* which contains three methods independent from each other and a *Main* class where a object of *Folder* is created and interacted with. The *Main* class could also be unrestricted but we defined it as linear to show the interaction of an unrestricted class through a linear one.

R-04 Implementation of the *Auction* example presented in [4] that serves as a more complex test to the operational semantics of Mool.

T-01 Typing example of the File example presented in [4]. Should evaluate successfully.

T-02 With the changes made to the T-Spawn rule, the type checker notices that executing the *read* operation will modify the variable f but will not consume its usage, which goes against what is pretended, so the evaluation should fail.

[4]Available at `https://sourceforge.net/p/mool-plt-redex/code/ci/master/tree/mool2.rkt`

Type Operations

$$\mathsf{lin}(t) \overset{\text{def}}{=} \begin{cases} tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \langle u' + u'' \rangle \\ tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \mathsf{lin}\{m_i; z_i\}_{i\in I} \\ ff & \text{otherwise} \end{cases}$$

$$\mathsf{un}(t) \overset{\text{def}}{=} \begin{cases} tt & \text{if } v = \mathsf{void} \\ tt & \text{if } v = \mathsf{int} \\ tt & \text{if } v = \mathsf{bool} \\ tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \mathsf{un}\{m_i; z_i\}_{i\in I} \\ ff & \text{otherwise} \end{cases}$$

$$\mathsf{lin}(\Gamma) \overset{\text{def}}{=} \forall\ (t\ f) \in \Gamma : \mathsf{lin}(t)$$

$$\mathsf{un}(\Gamma) \overset{\text{def}}{=} \forall\ (t\ f) \in \Gamma : \mathsf{un}(t)$$

$$\mathsf{lin}(\vec{F}) \overset{\text{def}}{=} \forall\ (t\ f) \in \vec{F} : \mathsf{lin}(t)$$

$$\mathsf{un}(\vec{F}) \overset{\text{def}}{=} \forall\ (t\ f) \in \vec{F} : \mathsf{un}(t)$$

$$\mathsf{check}(\Phi, u) \overset{\text{def}}{=} \begin{cases} \mathsf{check}(\Phi, u_i) & u = \mathsf{lin}\{m_i; u_i\}_{i\in I} \\ \mathsf{check}(\Phi, u_t) \wedge \mathsf{check}(\Theta, u_f) & u = \langle u_t + u_t \rangle \\ \mathsf{check}((\Phi, X : u), u) & u = \mu X.u \\ ff & u = \mathsf{un}\{m_i; u_i\}_{i\in I} \wedge \exists u_n \in u_i : u_n = \mu X.u_X \\ ff & u = \mathsf{un}\{m_i; u_i\}_{i\in I} \wedge \exists u_n \in u_i : u = \mathsf{lin}\{m_j; u_j\}_{j\in J} \\ ff & u = \mathsf{un}\{m_i; u_i\}_{i\in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \quad \Phi(X) = \langle u_t + u_t \rangle \\ ff & u = \mathsf{un}\{m_i; u_i\}_{i\in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \quad \Phi(X) = \mathsf{lin}\{m_j; u_j\}_{j\in J} \\ ff & u = \mathsf{un}\{m_i; u_i\}_{i\in I} \wedge \exists u_n \in u_i : u = \langle u_t + u_t \rangle \\ ff & u = \mathsf{un}\{m_i; u_i\}_{i\in I} \wedge \exists u_n \in u_i : u = \mathsf{un}\{m_j; u_j\}_{j\in J} \wedge m_i \neq m_j \\ ff & u = \mathsf{un}\{m_i; u_i\}_{i\in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \quad \Phi(X) = \mathsf{un}\{m_j; u_j\}_{j\in J} \wedge m_i \neq m_j \\ tt & \text{otherwise} \end{cases}$$

$$u.\mathsf{allows}(m_j) \overset{\text{def}}{=} \begin{cases} \epsilon & u = \epsilon \\ \mathsf{un} & u = \mathsf{un} \\ u_j & u = \{m_i; u_i\}_{i\in I} \text{ and } j \in I \\ u'\{^{\mu X.u'}/_X\}.\mathsf{allows}(m_j) & u = \mu X.u' \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{agree}(t, t') \overset{\text{def}}{=} \begin{cases} tt & \text{if } t = t' = \mathsf{bool} \\ tt & \text{if } t = t' = \mathsf{int} \\ tt & \text{if } t = t' = \mathsf{void} \\ tt & \text{if } t = C[u] \text{ and } t' = C[u; \vec{F}] \\ tt & \text{if } t = \mathsf{null} \text{ or } t' = \mathsf{null} \end{cases}$$

$$\mathsf{modified}(\Gamma, \Gamma') \overset{\text{def}}{=} \forall r \in \Gamma' :\ r \notin \Gamma \vee \Gamma(r) \neq \Gamma'(r)$$

$$\mathsf{completed}(\Gamma, \Gamma') \overset{\text{def}}{=} \forall r \in \Gamma' : (r \notin \Gamma \vee \Gamma(r) \neq \Gamma'(r)) \wedge (r = C[u; \vec{F}] \implies u = \mathsf{un}\{\})$$

Figure 3.7: Types, Type Definitions and Operations

$$\textsc{T-Class}\ \dfrac{\mathsf{check}(\emptyset, u) \qquad C[u;\emptyset] \blacktriangleright u \blacktriangleleft C[u;\vec{F}] \qquad \mathsf{un}(\vec{F})}{\vdash \mathsf{class}\ C\ \{u;\vec{F};\vec{M}\}}$$

$$\textsc{T-UnClass}\ \dfrac{\begin{array}{c}\forall i \in I \cdot \\ (y\ t\ m_i(t'\,x)\ \{s\},\_) \in \vec{M} \qquad C[\emptyset], x : t' \blacktriangleright s \blacktriangleleft C[\vec{F}] \qquad \mathsf{un}(\vec{F})\end{array}}{\vdash \mathsf{class}\ C\{\vec{F};\vec{M}\}}$$

Figure 3.8: Revised typing rules for programs

$$\textsc{T-Branch}\ \dfrac{\begin{array}{c}\forall i \in I \cdot \\ (y\ t\ m_i(t'\,x)\ \{s\},\_) \in C.\mathsf{methods} \qquad \mathsf{this} : C[u;\vec{F}], x : t' \triangleright s : t \triangleleft \Gamma \\ \Gamma \triangleright x : t'' \qquad \mathsf{un}(t'') \qquad \Gamma \triangleright \mathsf{this} : C[u_i;\vec{F_i}] \qquad \Theta;\Gamma \blacktriangleright u_i \blacktriangleleft \Gamma'\end{array}}{\Theta; C[u;\vec{F}] \blacktriangleright \_\{m_i;u_i\}_{i\in I} \blacktriangleleft \Gamma'}$$

$$\textsc{T-BranchEnd}\ \ \Theta;\Gamma \blacktriangleright un\{\} \blacktriangleleft \Gamma \qquad\qquad \textsc{T-UsageVar}\ \ (\Theta, X : \Gamma);\Gamma \blacktriangleright X \blacktriangleleft \Gamma$$

Figure 3.9: Revised typing rules for usages

$$\textsc{T-Add}\ \dfrac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 + a_2 : \mathsf{int} \triangleleft \Gamma''} \qquad \textsc{T-Sub}\ \dfrac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 - a_2 : \mathsf{int} \triangleleft \Gamma''}$$

$$\textsc{T-Mult}\ \dfrac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 * a_2 : \mathsf{int} \triangleleft \Gamma''} \qquad \textsc{T-Div}\ \dfrac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 \,/\, a_2 : \mathsf{int} \triangleleft \Gamma''}$$

Figure 3.10: Revised typing rules for arithmetic expressions

$$\textsc{T-Eq}\ \dfrac{\Gamma \triangleright e_1 : t \triangleleft \Gamma' \qquad \Gamma' \triangleright e_2 : t' \triangleleft \Gamma'' \qquad \mathsf{agree}(t', t)}{\Gamma \triangleright e_1 == e_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-Diff}\ \dfrac{\Gamma \triangleright e_1 : t \triangleleft \Gamma' \qquad \Gamma' \triangleright e_2 : t' \triangleleft \Gamma'' \qquad \mathsf{agree}(t', t)}{\Gamma \triangleright e_1 != e_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-Greater}\ \dfrac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 > a_2 : \mathsf{bool} \triangleleft \Gamma''} \qquad \textsc{T-Less}\ \dfrac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 < a_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-GtEqual}\ \dfrac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 >= a_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-LeEqual}\ \dfrac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 <= a_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-And}\ \dfrac{\Gamma \triangleright b_1 : \mathsf{bool} \triangleleft \Gamma' \qquad \Gamma' \triangleright b_2 : \mathsf{bool} \triangleleft \Gamma''}{\Gamma \triangleright b_1\ \&\&\ b_2 : \mathsf{bool} \triangleleft \Gamma''} \qquad \textsc{T-Or}\ \dfrac{\Gamma \triangleright b_1 : \mathsf{bool} \triangleleft \Gamma' \qquad \Gamma' \triangleright b_2 : \mathsf{bool} \triangleleft \Gamma''}{\Gamma \triangleright b_1 \,||\, b_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-Not}\ \dfrac{\Gamma \triangleright b : \mathsf{bool} \triangleleft \Gamma'}{\Gamma \triangleright !b : \mathsf{bool} \triangleleft \Gamma''}$$

Figure 3.11: Revised typing rules for boolean expressions

$$\textsc{T-LinVar}\ \dfrac{\mathsf{lin}(g)}{(\Gamma, r : g) \rhd r : g \lhd \Gamma} \qquad \textsc{T-UnVar}\ \dfrac{\mathsf{un}(g)}{(\Gamma, r : t) \rhd r : g \lhd (\Gamma, r : g)}$$

$$\textsc{T-LinField}\ \dfrac{\Gamma \rhd \mathit{this} : C[u;\vec{F}] \qquad \vec{F}(f) = g \qquad \mathsf{lin}(g)}{\Gamma \rhd \mathsf{this}.f : t \lhd \Gamma\{\mathsf{this} \mapsto C[u;(\vec{F} \setminus f)]\}}$$

$$\textsc{T-UnField}\ \dfrac{\Gamma \rhd \mathsf{this} : C[u;\vec{F}] \qquad \vec{F}(f) = t \qquad \mathsf{un}(t)}{\Gamma \rhd \mathsf{this}.f : t \lhd \Gamma}$$

$$\textsc{T-NullField}\ \dfrac{\Gamma \rhd \mathsf{this} : C[u;\vec{F}] \qquad (\_\ f) \notin \vec{F}}{\Gamma \rhd \mathsf{this}.f : \mathsf{null} \lhd \Gamma}$$

Figure 3.12: Revised typing rules for field and variable dereference

$$\textsc{T-AssignVar}\ \dfrac{e \neq \mathsf{null} \qquad \Gamma \rhd d : g \lhd \Gamma' \qquad \Gamma \rhd e : g' \lhd \Gamma' \qquad \mathsf{agree}(g', g)}{\Gamma \rhd d = e : \mathsf{void} \lhd \Gamma'}$$

$$\textsc{T-AssignField}\ \dfrac{\begin{array}{c} e \neq \mathsf{null} \qquad \Gamma \rhd e : g \lhd \Gamma' \\ \Gamma' \rhd \mathsf{this} : C[u;\vec{F}] \qquad C.\mathsf{fields}(f) = g' \qquad (\_\ f) \notin \vec{F} \vee \vec{F}(f) = g \qquad \mathsf{agree}(g', g) \end{array}}{\Gamma \rhd \mathsf{this}.f = e : \mathsf{void} \lhd \Gamma'\{\mathsf{this} \mapsto C[u;(\vec{F} \cup (g\ f))]\}}$$

$$\textsc{T-AssignFieldNull}\ \dfrac{\Gamma' \rhd \Gamma \rhd e : \mathsf{null} \lhd \Gamma \qquad \mathsf{this} : C[u;\vec{F}]}{\Gamma \rhd \mathsf{this}.f = e : \mathsf{void} \lhd \Gamma'\{\mathsf{this} \mapsto C[u;(\vec{F} \setminus (\_\ f))]\}}$$

$$\textsc{T-NewVar}\ \dfrac{d \notin \Gamma \qquad \Gamma \rhd e : g' \lhd \Gamma' \qquad \mathsf{agree}(g', g)}{\Gamma \rhd g\ d = e : \mathsf{void} \lhd \Gamma'\{d \mapsto g\}}$$

$$\textsc{T-Spawn}\ \dfrac{\Gamma \rhd s : t \lhd \Gamma' \qquad \mathsf{un}(\mathsf{modified}(\Gamma, \Gamma')) \qquad \mathsf{completed}(\Gamma, \Gamma')}{\Gamma \rhd \mathsf{spawn}\ \{s\} : \mathsf{void} \lhd \Gamma'}$$

Figure 3.13: Revised typing rules for simple statements

T-03 This program is similar to the one from T-05 but instead of creating a new thread for a reading operation, two separate threads are created for opening, reading and close separate files. This example, which evaluates successfully, shows that it is possible to use the construct spawn with several expressions.

T-04 A similar example to T-01 but now the body of the method *main* of class *Main* is executed using spawn. Although in the end the variable *f* is unrestricted, it still can be used after the spawn expression, so it should fail because we changed the T-Spawn so that every usage modified inside a spawn expression should be at a end, making it impossible to call any method from the object after the spawn expression.

T-05 The field file of class *FileReader* is not initialised but it is used, so the type checker will fail to evaluate because it checks if the field has already been initialised before using it.

$$\textsc{T-IfCall}\quad \frac{\begin{array}{c} w \neq \mathsf{this} \qquad \Gamma \rhd w.m(e) : \mathsf{bool} \lhd \Gamma' \\ \Gamma' \rhd w : C[\langle u_t + u_f \rangle; \vec{F}] \qquad \Gamma'\{w \mapsto C[u_t;\vec{F}]\} \rhd s' : t \lhd \Gamma'' \qquad \Gamma'\{w \mapsto C[u_f;\vec{F}]\} \rhd s'' : t \lhd \Gamma'' \end{array}}{\Gamma \rhd \mathsf{if}\ (w.m(e))\ s'\ \mathsf{else}\ s'' : t \lhd \Gamma''}$$

$$\textsc{T-IfNotCall}\quad \frac{\begin{array}{c} w \neq \mathsf{this} \qquad \Gamma \rhd w.m(e) : \mathsf{bool} \lhd \Gamma' \qquad \Gamma' \rhd w : C[\langle u_t + u_f \rangle; \vec{F}] \\ \Gamma'\{w \mapsto C[u_f;\vec{F}]\} \rhd s' : t \lhd \Gamma'' \qquad \Gamma'\{w \mapsto C[u_t;\vec{F}]\} \rhd s'' : t \lhd \Gamma'' \end{array}}{\Gamma \rhd \mathsf{if}\ (!w.m(e))\ s'\ \mathsf{else}\ s'' : t \lhd \Gamma''}$$

$$\textsc{T-IfUnCall}\quad \frac{\Gamma \rhd w.m(e) : \mathsf{bool} \lhd \Gamma' \qquad \Gamma' \rhd w : C[\epsilon;\vec{F}] \qquad \Gamma' \rhd s' : t \lhd \Gamma'' \qquad \Gamma' \rhd s'' : t \lhd \Gamma''}{\Gamma \rhd \mathsf{if}\ (w.m(e))\ s'\ \mathsf{else}\ s'' : t \lhd \Gamma''}$$

$$\textsc{T-If}\quad \frac{\Gamma \rhd b : \mathsf{bool} \lhd \Gamma \qquad \Gamma \rhd s' : t \lhd \Gamma'' \qquad \Gamma \rhd s'' : t \lhd \Gamma''}{\Gamma \rhd \mathsf{if}\ (b)\ s'\ \mathsf{else}\ s'' : t \lhd \Gamma''}$$

$$\textsc{T-WhileCall}\quad \frac{\begin{array}{c} w \neq \mathsf{this} \qquad \Gamma \rhd w.m(e) : \mathsf{bool} \lhd \Gamma' \\ \Gamma' \rhd w : C[\langle u_t + u_f \rangle; \vec{F}] \qquad \Gamma'\{w \mapsto C[u_t;\vec{F}]\} \rhd s' : t \lhd \Gamma'' \qquad \Gamma(w) = \Gamma''(w) \end{array}}{\Gamma \rhd \mathsf{while}\ (w.m(e))\{s'\} : t \lhd \Gamma''\{w \mapsto C[u_f;\vec{F}]\}}$$

$$\textsc{T-WhileNotCall}\quad \frac{\begin{array}{c} w \neq \mathsf{this} \qquad \Gamma \rhd w.m(e) : \mathsf{bool} \lhd \Gamma' \\ \Gamma' \rhd w : C[\langle u_t + u_f \rangle; \vec{F}] \qquad \Gamma'\{w \mapsto C[u_f;\vec{F}]\} \rhd s' : t \lhd \Gamma'' \qquad \Gamma(w) = \Gamma''(w) \end{array}}{\Gamma \rhd \mathsf{while}\ (!w.m(e))\{s'\} : t \lhd \Gamma''\{w \mapsto C[u_t;\vec{F}]\}}$$

$$\textsc{T-WhileUnCall}\quad \frac{\Gamma \rhd w.m(e) : \mathsf{bool} \lhd \Gamma' \qquad \Gamma' \rhd w : C[\epsilon;\vec{F}] \qquad \Gamma' \rhd s : t \lhd \Gamma}{\Gamma \rhd \mathsf{while}\ (w.m(e))\{s\} : t \lhd \Gamma'}$$

$$\textsc{T-While}\quad \frac{\Gamma \rhd b : \mathsf{bool} \lhd \Gamma \qquad \Gamma \rhd s : t \lhd \Gamma}{\Gamma \rhd \mathsf{while}\ (b)\{s\} : t \lhd \Gamma}$$

Figure 3.14: Revised typing rules for control flow expressions

$$\textsc{T-New}\quad \frac{\Gamma \rhd e : t' \lhd \Gamma' \qquad C.\mathsf{usage} = \mathsf{lin}\{C;u\} \qquad (t\ C(t'\ x)\ \{s\}, \_) \in C.\mathsf{methods} \qquad \mathsf{un}(\Gamma' \backslash \Gamma)}{\Gamma \rhd \mathsf{new}\ C(e) : C[u] \lhd \Gamma'}$$

$$\textsc{T-UnNew}\quad \frac{\Gamma \rhd e : t' \lhd \Gamma' \qquad C.\mathsf{usage} = \epsilon \qquad (t\ C(t'\ x)\ \{s\}, \_) \in C.\mathsf{methods} \qquad \mathsf{un}(\Gamma' \backslash \Gamma)}{\Gamma \rhd \mathsf{new}\ C(e) : C[u] \lhd \Gamma'}$$

$$\textsc{T-SelfCall1}\quad \frac{\begin{array}{c} \Gamma \rhd e : t' \lhd \Gamma' \\ \Gamma' \rhd \mathsf{this} : C[u;\vec{F}] \qquad (t\ m(t'\ x)\ \{s\}, 0) \in C.\mathsf{methods} \qquad \Gamma' \rhd s : t \lhd \Gamma'' \qquad \mathsf{un}(\Gamma' \backslash \Gamma) \end{array}}{\Gamma \rhd \mathsf{this}.m(e) : t \lhd \Gamma''}$$

$$\textsc{T-SelfCall2}\quad \frac{\Gamma \rhd e : t' \lhd \Gamma' \qquad \Gamma' \rhd \mathsf{this} : C[u;\vec{F}] \qquad (t\ m(t'\ x)\ \{s\}, 1) \in C.\mathsf{methods}}{\Gamma \rhd \mathsf{this}.m(e) : t \lhd \Gamma'}$$

$$\textsc{T-Call}\quad \frac{\begin{array}{c} w \neq \mathsf{this} \qquad \Gamma \rhd e : t' \lhd \Gamma' \\ \Gamma' \rhd w : C[u;\vec{F}] \qquad u.\mathsf{allows}(m) = u' \qquad (t\ m(t'\ x)\ \{s\}, \_) \in C.\mathsf{methods} \qquad \mathsf{un}(\Gamma' \backslash \Gamma) \end{array}}{\Gamma \rhd w.m(e) : t \lhd \Gamma'\{w \mapsto C[u';\vec{G}]\}}$$

Figure 3.15: Revised typing rules for calls

T-06 Same as T-01 but the constructor of the class *File*, where a number of variables are initialized, is replaced by the incrementation of the field *linesRead*, just like the method *read*. The typechecker does not accept this program;

T-07 The type system now goes inside the body of private methods and verifies them, so in this example the verification will fail because the type checker notices that the return type of method *count* is *void* but the type of the body is *boolean*;

T-08 Since now the type system is aware of which methods were already evaluated, this time the type checker will not enter in a infinite loop because it will only evaluate the the body of the recursive method *read* of the class File once, ignoring its body when reaching the self call and thus evaluating the program successfully.

T-09 Typing example of the File example presented in [20]. Should evaluate successfully.

T-10 A variation of the File example where in the method *next* of the *FileReader* class, after closing the file the field *file* is set to null. The type checker verifies the program successfully.

T-11 With the new rule T-Class the type checker will detect that the usage goes from unrestricted to linear when executing the method *eof*, so the evaluation should fail.

T-12 Again, the new rule T-Class also prevents an usage changing between different unrestricted states, so the program verification should fail;

T-13 This example is similar to the one in T-12 but now the usage can change between equivalent unrestricted states.

T-14 Typing example of the simple program introduced in R-03. Should evaluate successfully.

T-15 Typing example of the *Auction* example presented in [4]. Should evaluate successfully.

Chapter 4

Behavioural type inference

This chapter presents the behavioural type inference algorithm. This algorithm will work over a variation of the Mool language, called Mool^{-}, which we present in Section 4.1. Section 4.2 shows an example based on a blog scenario composed by its specification and an implementation proposal using Mool^{-}. This example has the purpose of exemplifying the behaviour type inference algorithm presented in Section 4.3. In this section we specify each step of the algorithm and exemplify the input and output of each one using the example of Section 4.2.

4.1 Mool^{-}

The target language for our behavioural type inference tool is a variation of Mool which we call Mool^{-}. It differs from the original version in three aspects: (1) it is based on a revised version of Mool [35], (2) does not have usages and (3) allows to annotate classes with assertions. With these assertions, programmers can specify the expected state of an object during its existence through invariants, and also specify the state of the object before and after a method execution. Figure 4.1 shows the syntax for the Mool^{-}, which is very similar to the syntax of the revised Mool but with the removal of usages and addition of assertions reflected, with the nonterminal b_a representing the assertion language, which is composed by every expression already in the language plus the term |result|, that represents the result of the method, and implication expression. The syntax only consists on the user syntax since we do not want to execute the language. For the same reason we do not provide reduction semantics nor typing rules for it.

4.2 Example: Blog

This example is based on a blog scenario. Due to the lack of collections in Mool we assume that this blog can contain only one post. There are two types of users:

Admin Can create, remove, and publish a post. The admin can remove a post only

$$
\begin{array}{rrcl}
\text{(class declarations)} & D & ::= & [\textsf{unrestricted}]\ \textsf{class}\ C\{\overrightarrow{F}; M_c; \overrightarrow{M}\} \\
\text{(field declaration)} & F & ::= & g\ f \\
\text{(constructor declaration)} & M_c & ::= & \textsf{inv}\ b_a\ [\textsf{req}\ b_a]\ \textsf{init}\ b_a\ t\ C(t'\ x)\ \{e\} \\
\text{(method declarations)} & M & ::= & \textsf{req}\ b_a\ \textsf{ens}\ b_a\ y\ t\ m(t'\ x)\ \{e\} \\
\text{(method qualifiers)} & y & ::= & \epsilon \mid \textsf{sync} \\
 & & & \\
\text{(values)} & v & ::= & \textsf{unit} \mid n \mid \textsf{true} \mid \textsf{false} \mid \textsf{null} \\
\text{(local value references)} & r & ::= & d \mid \textsf{this} \\
\text{(global value references)} & w & ::= & r \mid r.f \\
\text{(calls)} & c & ::= & \textsf{new}\ C(e) \mid r.m(e) \mid r.f.m(e) \\
\text{(arithmetic operations)} & a & ::= & n \mid w \mid c \mid a + a \mid a - a \\
 & & & \mid a * a \mid a/a \\
\text{(boolean operations)} & b & ::= & \textsf{true} \mid \textsf{false} \mid w \mid c \mid a == a \mid a\,! = a \\
 & & & \mid a <= a \mid a >= a \mid a < a \mid a > a \\
 & & & \mid b\ \&\&\ b \mid b \,\|\, b \mid !b \\
\text{(assertion language)} & b_a & ::= & b \mid |\textsf{result}| \mid b_a \rightarrow b_a \\
\text{(expressions)} & e & ::= & v \mid a \mid b \\
 & & & \mid c \mid w \\
\text{(statements)} & s & ::= & e \mid s; s' \\
 & & & \mid r.f = e \mid g\ d = e \mid d = e \\
 & & & \mid \textsf{if}\ (b)\ s'\ \textsf{else}\ s'' \mid \textsf{while}\ (b)\{s'\} \\
 & & & \mid \textsf{spawn}\{s\} \\
\text{(types)} & t & ::= & \textsf{void} \mid g \mid \textsf{null} \\
\text{(declarable types)} & g & ::= & \textsf{int} \mid \textsf{bool} \mid C
\end{array}
$$

Figure 4.1: Mool$^-$ syntax

if it has not been published before. Figure 4.2 shows a sequence diagram that exemplifies a communication between the admin and the system;

Viewer Can view a published post. Figure 4.3 shows a sequence diagram that exemplifies a communication between the viewer and the system.

Listing 4.1 shows the *Post* class. Since the access control to a post object is done by an instance of *Blog* we can have all three methods available at any time, which explains why every assertion is *true*. The reason this class is not specified as an unrestricted class is because we want to separate the initialisation process of the remaining operations, which we cannot do with an unrestricted class. Listing 4.2 shows the *Blog* class. When a blog is initialised it does not have a post so a new one must be created. After creating a new post it can be removed or it can be published, making it public to the viewers and blocking any writing operations on the post.

Listing 4.3 shows the *Viewer* class. When initialised a session is created and it can be closed any time. The viewer must request the post, which will save the post locally so that

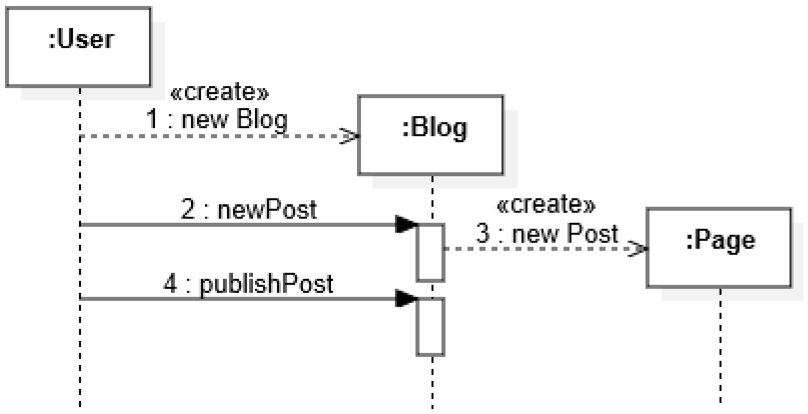


Figure 4.2: Admin interaction sequence diagram

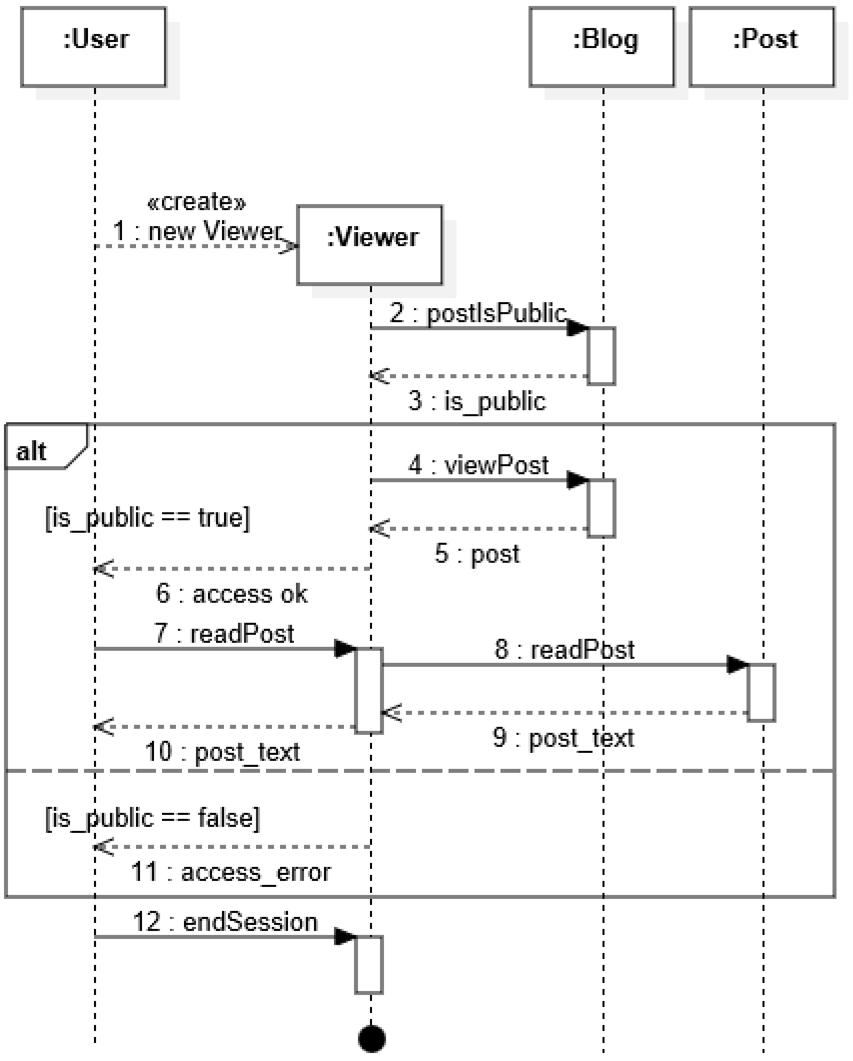


Figure 4.3: Viewer interaction sequence diagram

Listing 4.1: Code for the *Post* class

```
class Post {

    string title; string body;

    //@ invariant true;
    //@ initial true;
    void Post(string t, string t) {
        title = t;
        body = b;
    }

    //@ requires true;
    //@ ensures true;
    string readPost() {
        title ++ " - " ++ body;
    }
}
```

the viewer can read it. The viewer must be able to request a post right after beginning the session.

Finally, Listing 4.4 shows the Main class. The scenario it implements is as follow: It starts by creating a *Blog* object and then executes the *newPost* method which will create a *Post* object inside the *Blog* object. With the blog and post set it then creates three *Viewer* objects, representing three sessions that will interact with the *Blog* object and consequently the *Post* object in parallel. In each session the viewer requests the post and, if successful, reads it. In the end the viewer terminates the session.

4.3 Usage inference algorithm

In this section we describe the full usage inference algorithm adapted to our target language. This algorithm should be able to receive a program composed by classes written in Mool^{-} and return them as classes in Mool annotated with its respective usages. The algorithm works through three stages, with the first two based on already existing work which we adapted to fulfil the requirements of our target language. The first stage will extract the typestates of each Mool^{-} class, which will then be translated into usages in the second stage, and then the third stage will define the usage state each object is in the moment of its declaration. For each stage we provide the output for the example presented in section 4.2 to exemplify what to expect from each.

4.3.1 Stage 1: Typestate generation

For this first stage of the usage inference algorithm we use the behavioural model approach presented in [6] and briefly described in Section 2.5.

We made a few changes to the algorithm to fit it to our target language. The first concerns checking the validity of the assertions. In their presentation of the algorithm [6], Listing 4.2: Code for the *Blog* class

```
class Blog {

     Post post;
     boolean is_public;

     //@ invariant true;
     //@ initial !is_public && post == null;
     void Blog() {
          is_public = true;
     }

     //@ requires !is_public && post == null;
     //@ ensures !is_public && post != null;
     void newPost(string title, string body) {
          post = new Post(title, body);
     }

     //@ requires !is_public && post != null;
     //@ ensures !is_public && post == null;
     void deletePost() {
          post = null;
     }

     //@ requires true;
     //@ ensures |result| -> post != null;
     boolean hasPost() {
          post != null;
     }

     //@ requires true;
     //@ ensures |result| -> is_public;
     boolean postIsPublic() {
          is_public;
     }

     //@ requires is_public && post != null;
     //@ ensures is_public && post != null;
     sync Post viewPost() {
          no_views = no_views + 1;
          printStr("This post has ");
          printInt(no_views);
          printStr(" visualizations.");
          post;
     }

     //@ requires !is_public && post != null;
     //@ ensures is_public && post != null;
     void publishPost() {
          is_public = false;
     }
}
```

Listing 4.3: Code for the *Viewer* class

```
class Viewer {

    Blog blog;
    Post post;
    boolean session;

    //@ invariant true;
    //@ requires b.postIsPublic();
    //@ initial post == null && blog != null && session;
    void Viewer(Blog b) {
        blog = b;
        session = true;
    }

    //@ requires post == null && blog != null && session;
    //@ ensures (blog != null && session) && (|result| -> post != null);
    boolean requestPost() {
        if(blog.postIsPublic()) {
            post = blog.viewPost();
            true;
        } else {
            printStr("Access denied");
            false;
        }
    }

    //@ requires post != null && blog != null && session;
    //@ ensures post != null && blog != null && session;
    void readPost() {
        printStr(post.viewPost());
    }

    //@ requires blog != null && session;
    //@ ensures blog != null && !session;
    void endSession() {
        session = false;
    }
}
```

the authors suggest using code reachability to do the required validity checks, arguing that their implementation does not use a theorem prover due to the lack of postconditions. We, however, decided to include postconditions in our language and the init clause that represents the state of the object after its initialization. Therefore, we are able to use a theorem prover (Z3[1]) to check the validity of the assertions in first-order logic.

The second important change is related to non-determinism. The original algorithm allows non-deterministic transitions, depending on the internal state of the program. Since, as usual in a programming language, Mool does not support non-determinism, we also adapted the original algorithm in this respect. Mool allows transitions to have, at most, two target states, with these being based on choice, which are represented in usages through variant types. These transitions can only be triggered by a boolean method and

[1] https://github.com/Z3Prover/z3/wiki

Listing 4.4: Code for the *Main* class

```
unrestricted class Main {

    //@ invariant true;
    //@ initial true;
    void Main() {
        Blog blog; Blog = new Blog();

        blog.newPost("Sodales", "Lorem ipsum dolor sit amet");
        blog.submitPost();

        spawn {
            Viewer v1; v1 = new Viewer(blog);
            if(v1.requestPost()) {
                v1.readPost();
            }
            v1.endSession();
        }

        spawn {
            Viewer v2; v2 = new Viewer(blog);
            if(v2.requestPost()) {
                v2.readPost();
            }
            v2.endSession();
        }

        spawn {
            Viewer v3; v3 = new Viewer(blog);
            if(v3.requestPost()) {
                v3.readPost();
            }
            v3.endSession();
        }
    }
}
```

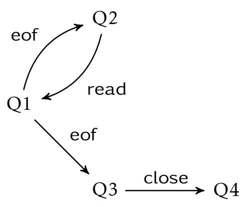

$$\begin{tikzcd} & Q2 \arrow[dl, bend left, "read"] & \\ Q1 \arrow[ur, bend left, "eof"] \arrow[dr, "eof"] & & \\ & Q3 \arrow[r, "close"] & Q4 \end{tikzcd}$$

Figure 4.4: Non-deterministic behaviour a *File* class

they rely on its result to choose the next state, hence the two target state limit for each transition.

For the algorithm to allow transitions based on choice (according to the result of a boolean method), it needs to know what state to choose if the result is true and what state to choose if the result is false. For example, consider a File class that follows the protocol defined in Figure 1.1. The transition relation δ is expected to include the triples (Q1,eof,Q2) and (Q1,eof,Q3), since δ needs to include a transition corresponding to

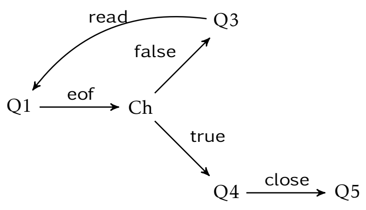


Figure 4.5: Behaviour a *File* class

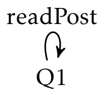


Figure 4.6: Typestate for the *Post* class

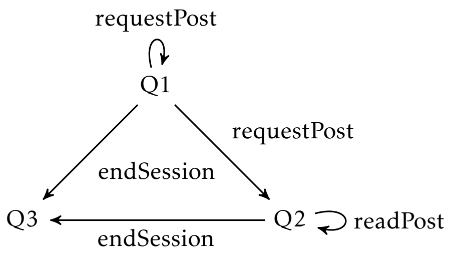


Figure 4.7: Typestate for the *Viewer* class

the case when the method eof returns true and another transition corresponding to the case when the method eof returns false. We explicitly need to know which target state corresponds to what result since we cannot rely on the order of how they are presented. In our algorithm the transition relation is as function defined as in Figure 4.5.

Such transition function is defined by the object state that causes the method to return true and the object state that causes the method to return false. So, in these situations, the post-condition of the method that triggers the transition must specify both states.

Therefore, our algorithm returns, for each class, a state machine representing its typestate. Figures 4.6, 4.7 and 4.8 show a graphical representation of the generated typestates for the classes that compose the example presented in 4.2.

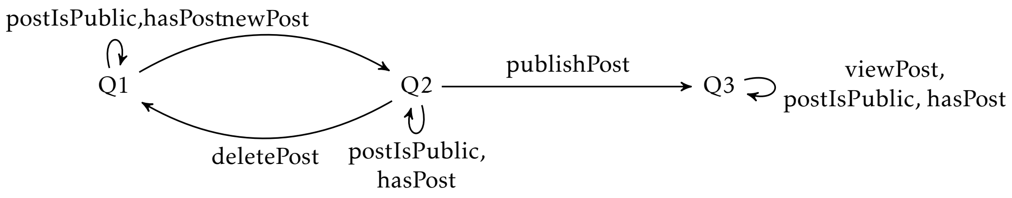


Figure 4.8: Typestate for the *Blog* class

4.3.2 Stage 2: Usage generation

The second stage of our algorithm consists thus on obtaining usages from the typestates obtained from the first phase. To do this we will use the idea presented by Collingbourne and Kelly [8], which consists on a three-stage algorithm that receives code from a very simple language similar to C and produces a session type from it.

Although the technique itself infers session types directly from code, it is defined to work on a limited language that lacks object and concurrency support, two important aspects of our target language. If we were to adapt it to our target language we would need to extend it with the missing features. Instead of doing such extension, we adapted the algorithm of Caso *et al.* [6].

Therefore, we are only going to use the third stage of Collingbourne and Kelly's algorithm, where it receives a state machine, converts it into a deterministic state machine, and applies a function that translates it into a session type. Since Mool has usages and not session types we need to adapt the function so that it translates state machines to usages instead. However, the conversion of non-deterministic state machines to deterministic ones, in our case that, instead of non-determinism in supports choice, would allow unwanted behaviour. For example, consider the following state machine:

$$\begin{tikzcd}
\{read\} \arrow[r, bend left, "\text{read}"] & \{eof\} \arrow[l, bend left, "\text{eof}"] \arrow[r, "\text{eof}"] & \{close\} \arrow[r, "\text{close}"] & \{\,\}
\end{tikzcd}$$

This state machine is nondeterministic. If we translate it into a deterministic one we get the following:

$$\begin{tikzcd}
\{eof\} \arrow[r, bend left, "\text{eof}"] & \{read, close\} \arrow[l, bend left, "\text{read}"] \arrow[r, "\text{close}"] & \{\,\}
\end{tikzcd}$$

This state machine allows us to read a file past its end because the method *read* is available after the method *eof* returns *true*, meaning that a file can be read past its end.

Instead of following Collingbourne and Kelly's approach in this respect, we need to introduce variant types. Determining which target transition corresponds to which boolean value is done using the information given by the new transition function presented in the previous stage.

Finally, since Mool supports shared and linear objects, the latter with behaviour captured in usage types and the former with behaviour captured in "standard" class types, we added a mechanism that ensures the generated usage does not have transitions that go from a shared state to a non-shared state or a non-equivalent non-shared state, *i.e.*, a non-shared state that offers a different set of methods.

Listing 4.5: Code for the *Post* class usage

```
usage lin{Post; Q1} where
     Q1 = un{readPost; Q1};
```

Listing 4.6: Code for the *Viewer* class usage

```
usage lin{Viewer; Q1} where
     Q1 = lin{endSession; end + requestPost; <Q2 + Q1>}
     Q2 = lin{endSession; end + readPost; Q2};
```

Listing 4.7: Code for the *Blog* class usage

```
usage lin{Blog; Q1} where
     Q1 = lin{postIsPublic; Q1 + hasPost; Q1 + newPost; Q2}
     Q2 = lin{publishPost; Q3 + postIsPublic; Q2 + hasPost; Q2 + deletePost; Q1}
     Q3 = un{viewPost; Q3 + postIsPublic; Q3 + hasPost; Q3};
```

In short, this second phase of our algorithm is loosely inspired by that of Collingbourne and Kelly, but has important modifications. To illustrate this phase of our algorithm, consider listings 4.5, 4.6 and 4.7. Each presents the usage type corresponding, respectively, to the typestates in Figures 4.6, 4.7, and 4.8. These usages were obtained by the algorithm from the typestates generated in the previous phase of the algorithm.

4.3.3 Stage 3: Object usage state inference

Mool offers the possibility of indicating the usage state an object has in its declaration. Consider the following excerpt of the $Viewer$ class

Listing 4.8: Excerpt of $Viewer$ class with field $blog$ without an usage state

```
class Viewer {
     ...
     Blog blog;
     ...
}
```

We want the *blog* field to be initialised in a state where the viewer can request a post right away which, according to the generated usage for the *Blog* class in listing 4.2, corresponds to the usage state $Q3$. In Mool, we can specify this as follow:

Listing 4.9: Excerpt of $Viewer$ class with field $blog$ annotated with an usage state

```
class Viewer {
     ...
     Blog[Q3] blog;
     ...
}
```

In the context of our work, we do not expect the programmer to know beforehand the states that will compose the generated usage, so the programmer does not have a way to indicate the usage state of an object when initialised. Although, we can expect the programmer to know the overall state of an object when initialised and be able to express it through assertions.

It is possible to express the expected state of the instance received as a parameter in the precondition of the method. In this case, since the field *blog* is initialised in the constructor with an object passed as an argument, we can specify the usage state of *blog* by defining the constructor as follow:

Listing 4.10: Constructor of the *Viewer* class equipped with assertions

```
//@ invariant true;
//@ requires b.postIsPublic();
//@ initial !has_post && session;
void Viewer(Blog b) {
    blog = b;
    ...
}
```

In the *requires* clause we call the method *postIsPublic* on the parameter *b*, stating that *b* must have the post available for the viewer.

In Mool one can also define the usage state of an object returned by a method. The method *viewPost* of the *Blog* class returns an object of the *Post* class

Listing 4.11: Method *viewPost* with a return type without an usage state

```
sync Post viewPost() {
    ...
    post;
}
```

Since we want the object *post* to be initialised, we should specify the return type of the method as follow:

Listing 4.12: Method *viewPost* with a return type annotated an usage state

```
sync Post[Q1] viewPost() {
    ...
    post;
}
```

This usage state can be inferred using the method postcondition, where it is possible to express the expected state of the returned object. For this example, since the usage state $Q1$ corresponds to the state where the object of type *Post* is initialized, we can just specify that the returned object is not *null*.

Listing 4.13: Method *viewPost* equipped with assertions

```
//@ requires is_public && post != null;
//@ ensures is_public && post != null;
sync Post viewPost() {
     ...
     post;
}
```

All these tasks are done by an algorithm that goes through all the methods of each class and does the following:

1. Checks for parameters containing objects. The algorithm uses the precondition of the method to determine their state. In this context only the premises related to the parameters being verified are considered.

2. Checks the return type of the method. If it is a class, the algorithm uses the postcondition of the method to determine the usage state of the return type. Again, only the premises related to the class field or local variable being returned by the method are considered.

3. Analyses the code of the method and checks where are the fields initialised. For every initialisation value, the algorithm sets the usage state of the field with the same usage state of the value. Moreover, since the object used as the initialising value can be manipulated before being assign to the field, the algorithm also checks the calls on that object and keeps track of its current usage state.

4.4 ML implementation

To present the details of the full algorithm we implemented it in ML. The implementation can be seen in Chapter C.

Section C.1 shows the types and structures used by the algorithm. Section C.2 shows the prover module that contains the functions used for validity checks.

Section C.3 shows the ML code for the first stage of the algorithm. The algorithm starts by determining the initial states, which is done by determining the set of methods which the precondition is implied by the constructor's initial condition. From the initial state, the algorithm starts to determine the transitions of the typestate and does it until every state of the typestate has been determined and explored. If a state has a boolean method that manipulates the object, leaving it in one of two possible states, the algorithm determines the states to which the typestate transits to depending of the returned result. The algorithm does this by determining the set of methods which the precondition is implied by the *true* side and the set of methods which the precondition is implied by the *false* side of the postcondition of the boolean method

Section C.4 shows the ML code for the second stage of the algorithm. The algorithm starts by creating a set of pairs $(id, state)$, with id being the identifier of $state$. Using this set and the transition relation returned by the previous stage, the algorithm translates each state of the typestate into an usage state. After the usage type is obtained, the algorithm goes through it and checks for irregularities such as an shared usage state going to an linear usage state or an non-equivalent shared state. In both situations the algorithm terminates with an error.

Section C.5 shows the ML code for the third stage of the algorithm, whose overall description was already presented in Section 4.3.3.

Chapter 5

Conclusions

5.1 Summary

The first contribution of this thesis is a detailed analysis of the formal definition and implementation of the Mool programming language, followed by the formalization of a new version of the language with corrections of errors and broader approaches to aspects where the language is too restrictive. One aspect we left out in our formalization of the language is the subtyping, which we argue that it is unsafe, since solving it would require work that is out of the context of this thesis. Along with the formalization we also provide the implementation of an interpreter of both the original and the revised versions using the Racket programming language, more specificaly its PLT Redex module, both complemented with examples to helps understanding the evolution between versions.

The second contribution is a behaviour type inference tool that, from code with assertions, generates the usage types necessary for the program to have its behaviour statically checked by a type system. The tool works on code written in a variation of Mool, called Mool^{-}, which is based on our revised version of Mool but instead of having usage annotations it has assertions. The reason we we choose to work with a small language instead of standard Java because right now behavioural type do not cope with features such as generics and collections. The algorithm takes a program fully annotated with assertions and either fails: the code is not well-typed (in the standard sense) or it may produce a run-time error due to calling methods in an incorrect order ; or returns a new version of the code with the classes annotated with behavioural types (called usages). Usage types can then be statically checked to verify if the code is data-safe and flow-safe: there will be no (null pointer) exceptions, no methods called when they are not supposed to, and moreover, that the protocols of critical resources are fully executed. In short, usage types ensure safe interoperability, which in this case means object compatibility: all inter-object method calls are valid and happen at a time where the state of the object allows those calls.

We implemented the algorithms described herein[1]. The tool starts by generating a

[1]Available at `http://usinfer.sourceforge.net/`

Listing 5.1: Code for the *File* class with weak assertions

```
class File {

     int linesInFile; int linesRead; boolean closed;

     //@ invariant linesRead >= 0 && linesRead ≤ linesInFile;
     //@ initial linesRead == 0 && linesInFile == 5 && !closed;
     void File() {
     linesRead = 0; linesInFile = 5; closed = false;
     }

     //@ requires linesRead < linesInFile && !closed;
     //@ ensures linesRead + 1 ≤ linesInFile && !closed;
     String read() {
          linesRead = linesRead + 1;
          "reading line... \n";
     }

     //@ requires linesRead ≤ linesInFile && !closed;
     //@ ensures (|result| −> linesRead == linesInFile) && !closed;
     boolean eof() {
          linesInFile == linesRead;
     }

     //@ requires linesRead == linesInFile && !closed;
     //@ ensures linesRead == linesInFile && closed;
     void close() {
          closed = true;
     }
}
```

permissive state machine representing a typestate, based on the assertions on the code. The tool then translates the generated typestates into usages. In the end, it defines the usage state where each object is when declared by using the assertions and the usages obtained in the previous stage. This tool is composed by three algorithms, with the first two adapted from algorithms presented in other works [6, 8], and the third one being original.

During the whole process the tool assumes the assertions are correct. If not, it can cause the tool to fail to infer the usage types or to produce usage types that do not specify the expected behaviour, which can result in allowing unwanted behaviour or invalidating correct behaviour. For example, consider the Mool$^-$ for the $File$ class presented in Listing 5.1. Using the behavioural inference tool to extract the usage for this class would result in the usage presented in Listing 5.2. The generated usage is accepted by the Mool compiler, allowing programs such as the one in Listing 5.3 to be accepted by the compiler when it should not because the method *read* is executed twice for each execution of method eof, allowing to read one extra line that does not exist and thus generating an runtime error.

By analysing the usage in 5.2, the programmer can see that the assertions specify a behaviour that allows the method eof to be executed at any point of the program. While the code of the method eof does not manipulate the object, the usage would allow code

Listing 5.2: Usage of the *File* class with weak assertions

```
usage lin{File; Q1} where
     Q1 = lin{eof; <Q2 + Q1> + read; Q1}
     Q2 = lin{eof; <Q2 + Q1> + close; end};
```

Listing 5.3: Code with incorrect interaction with the *File* class

```
usage lin{File; Q1} where
     Q1 = lin{eof; <Q2 + Q1> + read; Q1}
     Q2 = lin{close; end};

f = new File();
f.open();

if(f.eof()) {
     f.close();
     false;
} else {
     s = s ++ f.read();
     f.read(); //The reader tries to read two lines from the file in one go.
     true;
}
```

where, after reaching the end of the file, the object would transit to a state where it could execute the method *read* again. A possible solution is the *File* class in Mool$^-$ in Listing 5.4. This new class uses two new variables: $lineInBuffer$, which indicates that there is one line that can be consumed, and eof, that is $true$ when the end of the file has been reached. Every time the method eof is executed, if we have not reached the end of the file, the variable $lineInBuffer$ will turn $true$, and when we execute the method $read$, which requires $lineInBuffer$ to be $true$, we read the file and turn $lineInBuffer$ to $false$, indicating that the line verified by eof was consumed. This will continue until eof returns $true$, and by then $lineInBuffer$ will remain $false$, not allowing the execution of $read$ anymore. The variable eof represents the comparison between variables $linesRead$ and $linesInFile$, which seems redundant but we need it for the tool to separate the state specified by the precondition and both states specified in the postcondition of method eof.

Using the behavioural inference tool we get the usage in Listing 5.5, which does not allow code such as the one in Listing 5.3.

5.2 Future work

The difficulty of programming with assertions in comparison to programming with behaviour types is a valid topic of discussion because it is harder for the programmer to specify the expected behaviour of the program using assertions and also more error-prone since it is possible to write assertions that produce a valid (i.e., accepted by the type system) but ill-behaved usage. The $File$ class example presented in section 5.1 is an example

Listing 5.4: Code for the *File* class with stronger assertions

```
class File {

    int linesInFile; int linesRead; boolean closed;
    boolean lineInBuffer; boolean eof;

    //@ invariant linesRead >= 0 && linesRead ≤ linesInFile;
    //@ initial linesRead == 0 && linesInFile == 5 && !closed && !lineInBuffer && !eof;
    void File() {
        linesRead = 0;
        linesInFile = 5;
        closed = false;
        lineInBuffer = false;
    }

    //@ requires linesRead < linesInFile && !closed && lineInBuffer && !eof;
    //@ ensures linesRead + 1 ≤ linesInFile && !closed && !lineInBuffer && !eof;
    String read() {
        linesRead = linesRead + 1;
        lineInBuffer = false;
        "reading line... \n";
    }

    //@ requires linesRead ≤ linesInFile && !closed && !lineInBuffer && !eof;
    //@ ensures (linesRead == linesInFile -> (!lineInBuffer && eof)) && !closed;
    boolean eof() {
        lineInBuffer = !(linesRead == linesInFile);
        !lineInBuffer;
    }

    //@ requires linesRead == linesInFile && !closed && eof;
    //@ ensures linesRead == linesInFile && closed && eof;
    void close() {
        closed = true;
    }
}
```

Listing 5.5: Usage of the *File* class with stronger assertions

```
usage lin{File; Q1} where
    Q1 = lin{eof; <Q2 + Q3>}
    Q3 = lin{read; Q1}
    Q2 = lin{close; end};
```

of how easy is to produce wrong assertions, and also an example of the extra work the programmer might have to do just to produce assertions that allows the tool to infer the correct usage types by showing a situation where the programmer had to use extra variables that would not use otherwise.

At the moment the tool requires the specification of the behaviour of the code through assertions written by the programmer, but in the end we want it to infer the behaviour from the code itself, removing as much as possible the need for assertions, so future work will consist on automatically inferring as much assertions from the code as possible, with the next step consisting on postcondition inference using Hoare logic.

Future work will also include developing correctness proofs for the algorithms.

Bibliography

[1] D. Ancona, V. Bono, M. Bravetti, J. Campos, P.-M. Deniélou, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, et al. "BETTY WG3–Languages: State of the Art Report". In: *Report of the EU COST Action IC1201–Behavioural Types for Reliable Large-Scale Software Systems.* (2014).

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN: 026202649X, 9780262026499.

[3] J. Campos and V. T. Vasconcelos. "Channels as Objects in Concurrent Object-Oriented Programming". In: *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software*. Vol. 69. EPTCS. 2010, pp. 12–28.

[4] J. C. Campos. "Linear and shared objects in concurrent programming". MA thesis. University of Lisbon, 2010.

[5] L. Cardelli. "Type Systems". In: *The Computer Science and Engineering Handbook*. Ed. by A. B. Tucker. CRC Press, 1997. Chap. 103, pp. 2208–2236.

[6] G. D. Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. "Enabledness-based Program Abstractions for Behavior Validation". In: *ACM Trans. Softw. Eng. Methodol.* 22.3 (July 2013), 25:1–25:46. ISSN: 1049-331X. DOI: 10.1145/2491509.2491519. URL: http://doi.acm.org/10.1145/2491509.2491519.

[7] E. M. Clarke and E. A. Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logic of Programs, Workshop*. London, UK, UK: Springer-Verlag, 1982, pp. 52–71. ISBN: 3-540-11212-X. URL: http://dl.acm.org/citation.cfm?id=648063.747438.

[8] P. Collingbourne and P. H. J. Kelly. "Inference of Session Types From Control Flow". In: *Electron. Notes Theor. Comput. Sci.* 238.6 (June 2010), pp. 15–40. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2010.06.003. URL: http://dx.doi.org/10.1016/j.entcs.2010.06.003.

[9] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. "A Distributed Object-Oriented Language with Session Types". In: *Lecture Notes in Computer Science* 3705 (2005), p. 299. DOI: 10.1007/11580850_16.

[10] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. “Session Types for Object-Oriented Languages”. In: *ECOOP 2006 – Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings*. Ed. by D. Thomas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 328–352. ISBN: 978-3-540-35727-8. DOI: 10.1007/11785477_20. URL: http://dx.doi.org/10.1007/11785477_20.

[11] S. Drossopoulou, D. Dezani-Ciancaglini, and M. Coppo. “Amalgamating the Session Types and the Object Oriented Programming Paradigms”. In: *Multiparadigm Programming with Object-Oriented Languages 2007 (an ECOOP workshop)*. 2007. URL: http://pubs.doc.ic.ac.uk/sessionsAmalgamate00/.

[12] *Edsger W. Dijkstra - A.M. Turing Award Winner*. URL: http://amturing.acm.org/award_winners/dijkstra_1053701.cfm (visited on 01/29/2016).

[13] *E.W.Dijkstra Archive: The Humble Programmer (EWD 340)*. URL: https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html (visited on 01/30/2016).

[14] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. 1st. The MIT Press, 2009. ISBN: 0262062755, 9780262062756.

[15] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. A. McCarthy, and S. Tobin-Hochstadt. “The Racket Manifesto.” In: *SNAPL*. Ed. by T. Ball, R. Bodík, S. Krishnamurthi, B. S. Lerner, and G. Morrisett. Vol. 32. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 113–128. ISBN: 978-3-939897-80-4. URL: http://dblp.uni-trier.de/db/conf/snapl/snapl2015.html#FelleisenFFKBMT15.

[16] J.-C. Filliâtre. “Deductive software verification”. In: *International Journal on Software Tools for Technology Transfer* 13.5 (2011), pp. 397–403. ISSN: 1433-2787. DOI: 10.1007/s10009-011-0211-0. URL: http://dx.doi.org/10.1007/s10009-011-0211-0.

[17] J.-C. Filliâtre and A. Paskevich. “Why3 – Where Programs Meet Provers”. In: *ESOP’13 22nd European Symposium on Programming*. Vol. 7792. Rome, Italy: Springer, Mar. 2013. URL: https://hal.inria.fr/hal-00789533.

[18] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich. “Foundations of Typestate-Oriented Programming”. In: *ACM Trans. Program. Lang. Syst.* 36.4 (Oct. 2014), 12:1–12:44. ISSN: 0164-0925. DOI: 10.1145/2629609. URL: http://doi.acm.org/10.1145/2629609.

[19] S. Gay and V. T. Vasconcelos. “Linear Type Theory for Asynchronous Session Types”. In: *Journal of Functional Programming* 20.1 (2010), pp. 19–50. DOI: http://dx.doi.org/10.1017/S0956796809990268. URL: http://www.di.fc.ul.pt/~vv/papers/gay.vasconcelos_linear-sessions.pdf.

[20] S. J. Gay, N. Gesbert, A. Ravara, and V. T. Vasconcelos. "Modular Session Types for Objects". In: *Logical Methods in Computer Science* 11.4 (2015). DOI: 10.2168/LMCS-11(4:12)2015. URL: http://dx.doi.org/10.2168/LMCS-11(4:12)2015.

[21] E. F. Graversen, J. B. Harbo, H. Hüttel, M. O. Bjerregaard, N. S. Poulsen, and S. Wahl. "Type Inference for Session Types in the Pi-calculus". Aalborg University, Department of Computer Science. 2015.

[22] K. Honda. "Types for Dyadic Interaction". In: *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*. Vol. 715. Lecture Notes in Computer Science. Springer, 1993, pp. 509–523.

[23] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language Primitives and Type Discipline for Structured Communication-Based Programming". In: *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*. ESOP '98. London, UK, UK: Springer-Verlag, 1998, pp. 122–138. ISBN: 3-540-64302-8. URL: http://dl.acm.org/citation.cfm?id=645392.651876.

[24] H. Hüttel et al. "Foundations of Behavioural Types". In: *ACM Comput. Surv.* (To appear).

[25] E. Kindler. *Safety and Liveness Properties: A Survey*.

[26] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. "Run Your Research: On the Effectiveness of Lightweight Mechanization". In: *SIGPLAN Not.* 47.1 (Jan. 2012), pp. 285–296. ISSN: 0362-1340. DOI: 10.1145/2103621.2103691. URL: http://doi.acm.org/10.1145/2103621.2103691.

[27] R. Lo, F. Chow, R. Kennedy, S. ming Liu, and P. Tu. "Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores". In: *In Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. 1998, pp. 26–37.

[28] M. Neubauer and P. Thiemann. "An Implementation of Session Types". In: *Practical Aspects of Declarative Languages: 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004. Proceedings*. Ed. by B. Jayaraman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 56–70. ISBN: 978-3-540-24836-1. DOI: 10.1007/978-3-540-24836-1_5. URL: http://dx.doi.org/10.1007/978-3-540-24836-1_5.

[29] H. R. Nielson and F. Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN: 1846286913.

[30] B. C. Pierce. *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002. ISBN: 0-262-16209-1.

[31] J.-P. Queille and J. Sifakis. “Specification and Verification of Concurrent Systems in CESAR”. In: *Proceedings of the 5th Colloquium on International Symposium on Programming*. London, UK, UK: Springer-Verlag, 1982, pp. 337–351. ISBN: 3-540-11494-7. URL: http://dl.acm.org/citation.cfm?id=647325.721668.

[32] R. E. Strom and S Yemini. “Typestate: A Programming Language Concept for Enhancing Software Reliability”. In: *IEEE Trans. Softw. Eng.* 12.1 (Jan. 1986), pp. 157–171. ISSN: 0098-5589. DOI: 10.1109/TSE.1986.6312929. URL: http://dx.doi.org/10.1109/TSE.1986.6312929.

[33] K. Takeuchi, K. Honda, and M. Kubo. “An Interaction-based Language and its Typing System”. In: *In PARLE’94, volume 817 of LNCS*. Springer-Verlag, 1994, pp. 398–413.

[34] *The Plaid Programming Language*. URL: http://www.cs.cmu.edu/~aldrich/plaid/ (visited on 02/05/2016).

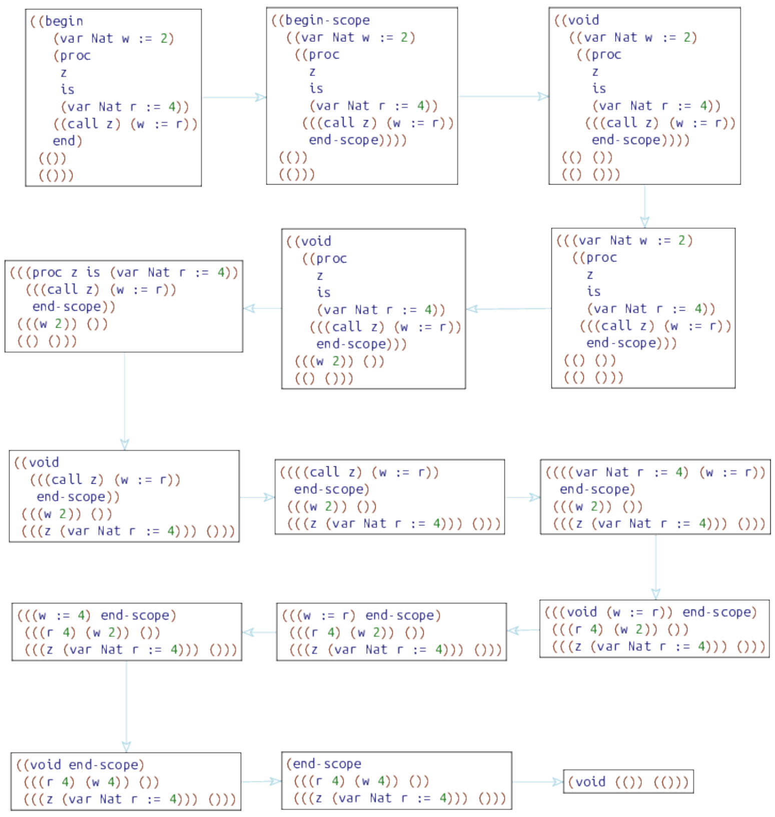
[35] C. Vasconcelos and A. Ravara. “Revision Proposal for the Mool Language”. 2016. URL: http://arxiv.org/abs/1604.06245.

[36] V. T. Vasconcelos, S. Gay, and A. Ravara. “Typechecking a Multithreaded Functional Language with Session Types”. In: 368.1–2 (2006), pp. 64–87. URL: http://www.di.fc.ul.pt/~vv/papers/vasconcelos.gay.ravara_tychecking-session-types.pdf.

Appendix A

Reduction graphs in Racket

Figures A.1, A.2 and A.3 show the reduction graphs for simple program examples for the While language and serve as examples of how PLT-Redex allows to execute the implemented languages and visualize it.



```
((begin
   (var Nat w := 2)
   (proc
    z
    is
    (var Nat r := 4))
   ((call z) (w := r))
   end)
 (())
 (()))
```

```
((begin-scope
  ((var Nat w := 2)
   ((proc
     z
     is
     (var Nat r := 4))
    (((call z) (w := r))
     end-scope))))
 (())
 (()))
```

```
((void
  ((var Nat w := 2)
   ((proc
     z
     is
     (var Nat r := 4))
    (((call z) (w := r))
     end-scope))))
 (() ())
 (() ()))
```

```
(((var Nat w := 2)
  ((proc
    z
    is
    (var Nat r := 4))
   (((call z) (w := r))
    end-scope)))
 (() ())
 (() ()))
```

```
((void
  ((proc
    z
    is
    (var Nat r := 4))
   (((call z) (w := r))
    end-scope)))
 (((w 2)) ())
 (() ()))
```

```
(((proc z is (var Nat r := 4))
  (((call z) (w := r))
   end-scope))
 (((w 2)) ())
 (() ()))
```

```
((void
  (((call z) (w := r))
   end-scope))
 (((w 2)) ())
 (((z (var Nat r := 4))) ()))
```

```
((((call z) (w := r))
  end-scope)
 (((w 2)) ())
 (((z (var Nat r := 4))) ()))
```

```
((((var Nat r := 4) (w := r))
  end-scope)
 (((w 2)) ())
 (((z (var Nat r := 4))) ()))
```

```
(((void (w := r)) end-scope)
 (((r 4) (w 2)) ())
 (((z (var Nat r := 4))) ()))
```

```
(((w := r) end-scope)
 (((r 4) (w 2)) ())
 (((z (var Nat r := 4))) ()))
```

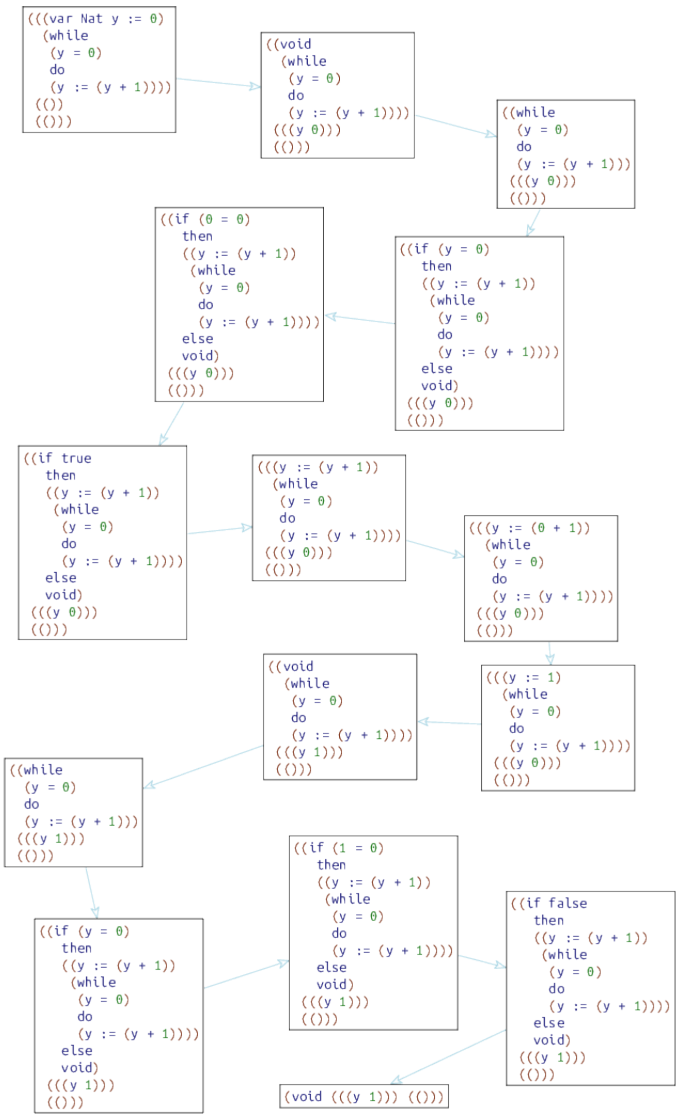
```
(((w := 4) end-scope)
 (((r 4) (w 2)) ())
 (((z (var Nat r := 4))) ()))
```

```
((void end-scope)
 (((r 4) (w 4)) ())
 (((z (var Nat r := 4))) ()))
```

```
(end-scope
 (((r 4) (w 4)) ())
 (((z (var Nat r := 4))) ()))
```

```
(void (()) (()))
```

Figure A.1: PLT-Redex reduction graph example 2



```
(((var Nat y := 0)
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (())
 (()))
```

```
((void
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (((y 0)))
 (()))
```

```
((while
  (y = 0)
  do
  (y := (y + 1)))
 (((y 0)))
 (()))
```

```
((if (y = 0)
  then
  ((y := (y + 1))
   (while
    (y = 0)
    do
    (y := (y + 1))))
  else
  void)
 (((y 0)))
 (()))
```

```
((if (0 = 0)
  then
  ((y := (y + 1))
   (while
    (y = 0)
    do
    (y := (y + 1))))
  else
  void)
 (((y 0)))
 (()))
```

```
((if true
  then
  ((y := (y + 1))
   (while
    (y = 0)
    do
    (y := (y + 1))))
  else
  void)
 (((y 0)))
 (()))
```

```
(((y := (y + 1))
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (((y 0)))
 (()))
```

```
(((y := (0 + 1))
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (((y 0)))
 (()))
```

```
(((y := 1)
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (((y 0)))
 (()))
```

```
((void
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (((y 1)))
 (()))
```

```
((while
  (y = 0)
  do
  (y := (y + 1)))
 (((y 1)))
 (()))
```

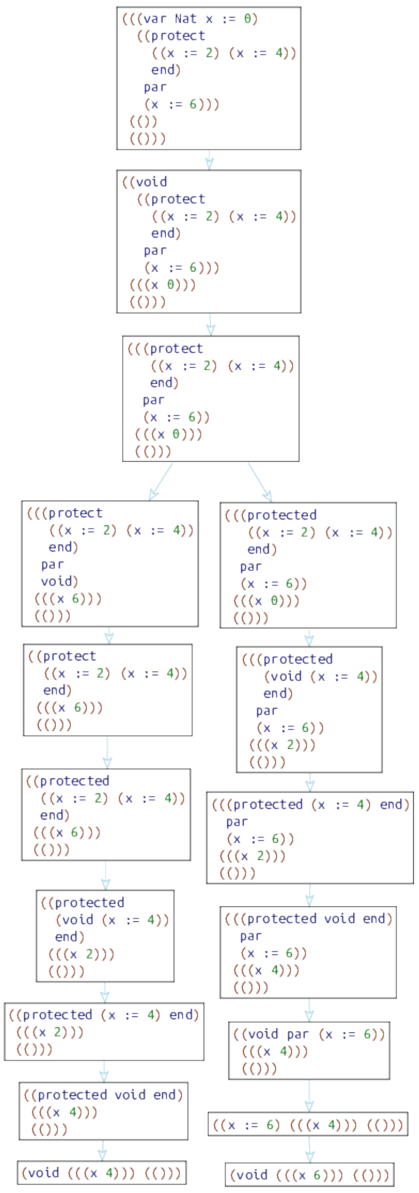
```
((if (y = 0)
  then
  ((y := (y + 1))
   (while
    (y = 0)
    do
    (y := (y + 1))))
  else
  void)
 (((y 1)))
 (()))
```

```
((if (1 = 0)
  then
  ((y := (y + 1))
   (while
    (y = 0)
    do
    (y := (y + 1))))
  else
  void)
 (((y 1)))
 (()))
```

```
((if false
  then
  ((y := (y + 1))
   (while
    (y = 0)
    do
    (y := (y + 1))))
  else
  void)
 (((y 1)))
 (()))
```

```
(void (((y 1))) (()))
```

Figure A.2: PLT-Redex reduction graph example 1



```
(((var Nat x := 0)
  ((protect
    ((x := 2) (x := 4))
    end)
   par
   (x := 6)))
 (())
 (()))
```

```
((void
  ((protect
    ((x := 2) (x := 4))
    end)
   par
   (x := 6)))
 (((x 0)))
 (()))
```

```
(((protect
    ((x := 2) (x := 4))
    end)
   par
   (x := 6))
 (((x 0)))
 (()))
```

```
(((protect
    ((x := 2) (x := 4))
    end)
   par
   void)
 (((x 6)))
 (()))
```

```
((protect
   ((x := 2) (x := 4))
   end)
 (((x 6)))
 (()))
```

```
((protected
   ((x := 2) (x := 4))
   end)
 (((x 6)))
 (()))
```

```
((protected
   (void (x := 4))
   end)
 (((x 2)))
 (()))
```

```
((protected (x := 4) end)
 (((x 2)))
 (()))
```

```
((protected void end)
 (((x 4)))
 (()))
```

```
(void (((x 4))) (()))
```

```
(((protected
    ((x := 2) (x := 4))
    end)
   par
   (x := 6))
 (((x 0)))
 (()))
```

```
(((protected
    (void (x := 4))
    end)
   par
   (x := 6))
 (((x 2)))
 (()))
```

```
(((protected (x := 4) end)
   par
   (x := 6))
 (((x 2)))
 (()))
```

```
(((protected void end)
   par
   (x := 6))
 (((x 4)))
 (()))
```

```
((void par (x := 6))
 (((x 4)))
 (()))
```

```
((x := 6) (((x 4))) (()))
```

```
(void (((x 6))) (()))
```

Figure A.3: PLT-Redex reduction graph example 3

Appendix B

Revised Mool syntax and rules

User Syntax

(class declarations)	$D ::= \mathsf{class}\ C\ \{u;\vec{F};\vec{M}\}$
(field declaration)	$F ::= g\ f$
(method declarations)	$M ::= y\ t\ m(t'\ x)\ \{e\}$
(method qualifiers)	$y ::= \epsilon \mid \mathsf{sync}$
(values)	$v ::= \mathsf{unit} \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null}$
(local value references)	$r ::= d \mid \mathsf{this}$
(global value references)	$w ::= r \mid r.f$
(calls)	$c ::= \mathsf{new}\ C(e) \mid r.m(e) \mid r.f.m(e)$
(arithmetic operations)	$a ::= n \mid w \mid c \mid a+a \mid a-a$
	$\mid a*a \mid a/a$
(boolean operations)	$b ::= \mathsf{true} \mid \mathsf{false} \mid w \mid c \mid a == a \mid a\,! = a$
	$\mid a <= a \mid a >= a \mid a < a \mid a > a$
	$\mid b\ \&\&\ b \mid b \,\|\, b \mid\ !b$
(expressions)	$e ::= v \mid a \mid b$
	$\mid c \mid w$
(statements)	$s ::= e \mid s;s'$
	$\mid r.f = e \mid g\ d = e \mid d = e$
	$\mid \mathsf{if}\ (b)\ s'\ \mathsf{else}\ s'' \mid \mathsf{while}\ (b)\{s'\}$
	$\mid \mathsf{spawn}\{s\}$
(types)	$t ::= \mathsf{void} \mid g \mid \mathsf{null}$
(declarable types)	$g ::= \mathsf{int} \mid \mathsf{bool} \mid C[z]$
(class usages)	$u ::= \epsilon \mid q\{m_i;z_i\}_{i\in I} \mid \mu X.u$
(usages)	$z ::= u \mid \langle u+u \rangle \mid X$
(usage types)	$q ::= \mathsf{un} \mid \mathsf{lin}$

Runtime Syntax

(values)	$v ::= \dots \mid o$
(value references)	$r ::= \dots \mid o$
(class usages)	$u ::= \dots \mid z$
(types)	$t ::= \dots \mid C[z;\vec{F}]$
(object records)	$R ::= (C,u,\overrightarrow{f=v},l)$
(field value map)	$l ::= 0 \mid 1$
(heap)	$h ::= \emptyset \mid h,o = R$
(evaluation context)	$\mathscr{E} ::= [-] \mid \mathscr{E};s \mid o.f = \mathscr{E} \mid o.m(\mathscr{E}) \mid o.f.m(\mathscr{E})$
	$\mid \mathsf{if}\ (\mathscr{E})\ s\ \mathsf{else}\ s'$
(States)	$S ::= (h, local, s_1 \mid \dots \mid s_n)$

Figure B.1: Revised syntax

Object Record and Heap Operations

$$\langle C, u, \vec{V}\rangle.f \overset{\text{def}}{=} \vec{V}(f) \qquad \langle C, u, \vec{V}\rangle.\mathsf{usage} \overset{\text{def}}{=} u$$

$$\langle C, u, \vec{V}\rangle.\mathsf{class} \overset{\text{def}}{=} C$$

Operations for values and types

$$\mathsf{lin}(v) \overset{\text{def}}{=} \begin{cases} tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \langle u' + u''\rangle \\ tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \mathsf{lin}\{m_i; z_i\}_{i\in I} \\ ff & \text{otherwise} \end{cases} \qquad \mathsf{un}(v) \overset{\text{def}}{=} \begin{cases} tt & \text{if } v = \mathsf{unit} \\ tt & \text{if } v = n \\ tt & \text{if } v = \mathsf{true} \\ tt & \text{if } v = \mathsf{false} \\ tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \mathsf{un}\{m_i; z_i\}_{i\in I} \\ ff & \text{otherwise} \end{cases}$$

Class Definition Operations

$$C.\mathsf{methods} \overset{\text{def}}{=} \overrightarrow{M, eval} \quad \text{where class } C\ \{u; \vec{F}; \vec{M}\} \in \vec{D} \text{ and } eval \in \{0,1\}$$

$$C.\mathsf{fields} \overset{\text{def}}{=} \vec{F} \quad \text{where class } C\ \{u; \vec{F}; \vec{M}\} \in \vec{D}$$

$$C.\mathsf{usage} \overset{\text{def}}{=} u \quad \text{where class } C\ \{u; \vec{F}; \vec{M}\} \in \vec{D}$$

Figure B.2: Auxiliary definitions and Operations

$$\text{R-CONTEXT}\ \frac{(h, local, s_1 \mid \ldots \mid s \mid \ldots \mid s_n) \longrightarrow (h', local', s_1 \mid \ldots \mid s' \mid \ldots \mid s_n)}{(h, local, s_1 \mid \ldots \mid \mathscr{E}[s] \mid \ldots \mid s_n) \longrightarrow (h', local', s_1 \mid \ldots \mid \mathscr{E}[s'] \mid \ldots \mid s_n)}$$

$$\text{R-SPAWN}\ (h, local, s_1 \mid \ldots \mid \mathscr{E}[\mathsf{spawn}\{s\}] \mid \ldots \mid s_n) \longrightarrow (h, local, s_1 \mid \ldots \mid \mathscr{E}[\mathsf{unit}] \mid \ldots \mid s_n)$$

Figure B.3: Reduction semantics for states

$$\textsc{R-UnField}\quad \frac{h(o).f = v \qquad \mathsf{un}(v,h)}{(h, local, o.f) \longrightarrow (h, local, v)} \qquad \textsc{R-LinField}\quad \frac{h(o).f = v \qquad \mathsf{lin}(v,h)}{(h, local, o.f) \longrightarrow (h\{o.f \mapsto \mathsf{null}\}, local, v)}$$

$$\textsc{R-UnVar}\quad \frac{h(d) = v \qquad \mathsf{un}(v,h)}{(h, local, d) \longrightarrow (h, local, v)} \qquad \textsc{R-LinVar}\quad \frac{h(d) = v \qquad \mathsf{lin}(v,h)}{(h, local, d) \longrightarrow (h\{d \mapsto \mathsf{null}\}, local, v)}$$

$$\textsc{R-Seq}\quad (h, local, v; s) \longrightarrow (h, local, s) \qquad \textsc{R-NewVar}\quad (h, local, g\ d = v) \longrightarrow (h, local\{d \mapsto v\}, \mathsf{unit})$$

$$\textsc{R-AssignVar}\quad (h, local, d = v) \longrightarrow (h, local\{d \mapsto v\}, \mathsf{unit})$$

$$\textsc{R-AssignField}\quad \frac{v \neq \mathsf{null}}{(h, local, o.f = v) \longrightarrow (h\{o.f \mapsto v\}, local, \mathsf{unit})}$$

$$\textsc{R-AssignFieldNull}\quad \frac{v = \mathsf{null}}{(h, local, o.f = v) \longrightarrow (h \setminus o.f, local, \mathsf{unit})}$$

$$\textsc{R-New}\quad \frac{o \text{ fresh} \qquad (\_\ C(\_\ x)\ \{s\}, \_) \in C.\mathsf{methods} \qquad C.\mathsf{fields} = \overrightarrow{t\ f} \qquad C.\mathsf{usage} = u}{(h, local, \mathsf{new}\ C(v)) \longrightarrow ((h, o = \langle C, u, \overrightarrow{f = \mathsf{null}} \rangle), local, s\{^{o}/_{\mathsf{this}}\}\{^{v}/_{x}\}; o)}$$

$$\textsc{R-Call}\quad \frac{(\_\ m(\_\ x)\ \{s\}, \_) \in (h(o).\mathsf{class}).\mathsf{methods}}{(h, local, o.m(v)) \longrightarrow (h, local, s\{^{o}/_{\mathsf{this}}\}\{^{v}/_{x}\})}$$

$$\textsc{R-FieldCall}\quad \frac{(\_\ m(\_\ x)\ \{s\}, \_) \in (h(o).f.\mathsf{class}).\mathsf{methods}}{(h, \varnothing, o.f.m(v)) \longrightarrow (h, \varnothing, s\{^{o}/_{\mathsf{this}}\}\{^{v}/_{x}\})}$$

$$\textsc{R-While}\quad (h, local, \mathsf{while}\ (b)\{s\}) \longrightarrow (h, local, \mathsf{if}\ (b)\ (s; \mathsf{while}\ (b)\{s\})\ \mathsf{else\ unit})$$

$$\textsc{R-IfTrue}\quad (h, local, \mathsf{if}\ (\mathsf{true})\ s'\ \mathsf{else}\ s'') \longrightarrow (h, local, s')$$

$$\textsc{R-IfFalse}\quad (h, local, \mathsf{if}\ (\mathsf{false})\ s'\ \mathsf{else}\ s'') \longrightarrow (h, local, s'')$$

Figure B.4: Revised reduction Semantics

$$\mathcal{N}(n) = n \qquad \mathcal{A}(n, h, local) = \mathcal{N}(n)$$

$$\mathcal{A}(o.f, h, local) = h(o).f \qquad \mathcal{A}(d, h, local) = local(d)$$

$$\mathcal{A}(a_1 + a_2, h, local) = \mathcal{A}(a_1, h, local) + \mathcal{A}(a_2, h, local)$$

$$\mathcal{A}(a_1 - a_2, h, local) = \mathcal{A}(a_1, h, local) - \mathcal{A}(a_2, h, local)$$

$$\mathcal{A}(a_1 * a_2, h, local) = \mathcal{A}(a_1, h, local) * \mathcal{A}(a_2, h, local)$$

$$\mathcal{A}(a_1\ /\ a_2, h, local) = \mathcal{A}(a_1, h, local)\ /\ \mathcal{A}(a_2, h, local)$$

Figure B.5: Evaluation functions for arithmetic values and expressions

$$\mathscr{B}(\mathsf{true}, h, local) = \mathsf{true} \qquad \mathscr{B}(\mathsf{false}, h, local) = \mathsf{false}$$

$$\mathscr{B}(o.f, h, local) = h(o).f \qquad \mathscr{B}(d, h, local) = local(d)$$

$$\mathscr{B}(a_1 == a_2, h, local) = \begin{cases} \mathsf{true} & \mathscr{A}(a_1, h, local) = \mathscr{A}(a_2, h, local) \\ \mathsf{false} & \mathscr{A}(a_1, h, local) \neq \mathscr{A}(a_2, h, local) \end{cases}$$

$$\mathscr{B}(a_1 ! = a_2, h, local) = \begin{cases} \mathsf{true} & \mathscr{A}(a_1, h, local) \neq \mathscr{A}(a_2, h, local) \\ \mathsf{false} & \mathscr{A}(a_1, h, local) = \mathscr{A}(a_2, h, local) \end{cases}$$

$$\mathscr{B}(a_1 < a_2, h, local) = \begin{cases} \mathsf{true} & \mathscr{A}(a_1, h, local) < \mathscr{A}(a_2, h, local) \\ \mathsf{false} & \mathscr{A}(a_1, h, local) >= \mathscr{A}(a_2, h, local) \end{cases}$$

$$\mathscr{B}(a_1 < a_2, h, local) = \begin{cases} \mathsf{true} & \mathscr{A}(a_1, h, local) < \mathscr{A}(a_2, h, local) \\ \mathsf{false} & \mathscr{A}(a_1, h, local) <= \mathscr{A}(a_2, h, local) \end{cases}$$

$$\mathscr{B}(a_1 <= a_2, h, local) = \begin{cases} \mathsf{true} & \mathscr{A}(a_1, h, local) <= \mathscr{A}(a_2, h, local) \\ \mathsf{false} & \mathscr{A}(a_1, h, local) > \mathscr{A}(a_2, h, local) \end{cases}$$

$$\mathscr{B}(a_1 >= a_2, h, local) = \begin{cases} \mathsf{true} & \mathscr{A}(a_1, h, local) >= \mathscr{A}(a_2, h, local) \\ \mathsf{false} & \mathscr{A}(a_1, h, local) < \mathscr{A}(a_2, h, local) \end{cases}$$

$$\mathscr{B}(b_1 \mathbin{\&\&} b_2, h, local) = \begin{cases} \mathsf{true} & \mathscr{B}(b_1, h, local) = \mathsf{true} \land (b_2, h, local) = \mathsf{true} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathscr{B}(b_1 \parallel b_2, h, local) = \begin{cases} \mathsf{true} & \mathscr{B}(b_1, h, local) = \mathsf{true} \lor (b_2, h, local) = \mathsf{true} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathscr{B}(!b, h, local) = \begin{cases} \mathsf{true} & \mathscr{B}(b, h, local) = \mathsf{false} \\ \mathsf{false} & \mathscr{B}(b, h, local) = \mathsf{true} \end{cases}$$

Figure B.6: Evaluation functions for boolean values and expressions

Type Operations

$$\mathsf{lin}(t) \overset{\text{def}}{=} \begin{cases} tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \langle u' + u'' \rangle \\ tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \mathsf{lin}\{m_i ; z_i\}_{i \in I} \\ ff & \text{otherwise} \end{cases}$$

$$\mathsf{un}(t) \overset{\text{def}}{=} \begin{cases} tt & \text{if } v = \mathsf{void} \\ tt & \text{if } v = \mathsf{int} \\ tt & \text{if } v = \mathsf{bool} \\ tt & \text{if } v = o \wedge h(v).\mathsf{usage} = \mathsf{un}\{m_i ; z_i\}_{i \in I} \\ ff & \text{otherwise} \end{cases}$$

$$\mathsf{lin}(\Gamma) \overset{\text{def}}{=} \forall\ (t\ f) \in \Gamma : \mathsf{lin}(t)$$

$$\mathsf{un}(\Gamma) \overset{\text{def}}{=} \forall\ (t\ f) \in \Gamma : \mathsf{un}(t)$$

$$\mathsf{lin}(\vec{F}) \overset{\text{def}}{=} \forall\ (t\ f) \in \vec{F} : \mathsf{lin}(t)$$

$$\mathsf{un}(\vec{F}) \overset{\text{def}}{=} \forall\ (t\ f) \in \vec{F} : \mathsf{un}(t)$$

$$\mathsf{check}(\Phi, u) \overset{\text{def}}{=} \begin{cases} \mathsf{check}(\Phi, u_i) & u = \mathsf{lin}\{m_i ; u_i\}_{i \in I} \\ \mathsf{check}(\Phi, u_t) \wedge \mathsf{check}(\Theta, u_f) & u = \langle u_t + u_t \rangle \\ \mathsf{check}((\Phi, X : u), u) & u = \mu X.u \\ ff & u = \mathsf{un}\{m_i ; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = \mu X.u_X \\ ff & u = \mathsf{un}\{m_i ; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u = \mathsf{lin}\{m_j ; u_j\}_{j \in J} \\ ff & u = \mathsf{un}\{m_i ; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \quad \Phi(X) = \langle u_t + u_t \rangle \\ ff & u = \mathsf{un}\{m_i ; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \quad \Phi(X) = \mathsf{lin}\{m_j ; u_j\}_{j \in J} \\ ff & u = \mathsf{un}\{m_i ; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u = \langle u_t + u_t \rangle \\ ff & u = \mathsf{un}\{m_i ; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u = \mathsf{un}\{m_j ; u_j\}_{j \in J} \wedge m_i \neq m_j \\ ff & u = \mathsf{un}\{m_i ; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \quad \Phi(X) = \mathsf{un}\{m_j ; u_j\}_{j \in J} \wedge m_i \neq m_j \\ tt & \text{otherwise} \end{cases}$$

$$u.\mathsf{allows}(m_j) \overset{\text{def}}{=} \begin{cases} \epsilon & u = \epsilon \\ \mathsf{un} & u = \mathsf{un} \\ u_j & u = \{m_i ; u_i\}_{i \in I} \text{ and } j \in I \\ u'\{^{\mu X.u'}/_X\}.\mathsf{allows}(m_j) & u = \mu X.u' \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{agree}(t, t') \overset{\text{def}}{=} \begin{cases} tt & \text{if } t = t' = \mathsf{bool} \\ tt & \text{if } t = t' = \mathsf{int} \\ tt & \text{if } t = t' = \mathsf{void} \\ tt & \text{if } t = C[u] \text{ and } t' = C[u; \vec{F}] \\ tt & \text{if } t = \mathsf{null} \text{ or } t' = \mathsf{null} \end{cases}$$

$$\mathsf{modified}(\Gamma, \Gamma') \overset{\text{def}}{=} \forall r \in \Gamma' :\ r \notin \Gamma \vee \Gamma(r) \neq \Gamma'(r)$$

$$\mathsf{completed}(\Gamma, \Gamma') \overset{\text{def}}{=} \forall r \in \Gamma' : (r \notin \Gamma \vee \Gamma(r) \neq \Gamma'(r)) \wedge (r = C[u; \vec{F}] \implies u = \mathsf{un}\{\})$$

Figure B.7: Types, Type Definitions and Operations

$$\textsc{T-Class}\ \frac{\mathsf{check}(\emptyset, u) \qquad C[u;\emptyset] \blacktriangleright u \blacktriangleleft C[u;\vec{F}] \qquad \mathsf{un}(\vec{F})}{\vdash \mathsf{class}\ C\ \{u;\vec{F};\vec{M}\}}$$

$$\textsc{T-UnClass}\ \frac{\begin{array}{c}\forall i \in I \cdot \\ (y\ t\ m_i(t'\,x)\ \{s\},\_) \in \vec{M} \qquad C[\emptyset], x : t' \blacktriangleright s \blacktriangleleft C[\vec{F}] \qquad \mathsf{un}(\vec{F})\end{array}}{\vdash \mathsf{class}\ C\{\vec{F};\vec{M}\}}$$

Figure B.8: Revised typing rules for programs

$$\textsc{T-Branch}\ \frac{\begin{array}{c}\forall i \in I \cdot \\ (y\ t\ m_i(t'\,x)\ \{s\},\_) \in C.\mathsf{methods} \qquad \mathsf{this} : C[u;\vec{F}], x : t' \rhd s : t \lhd \Gamma \\ \Gamma \rhd x : t'' \qquad \mathsf{un}(t'') \qquad \Gamma \rhd \mathsf{this} : C[u_i;\vec{F_i}] \qquad \Theta;\Gamma \blacktriangleright u_i \blacktriangleleft \Gamma'\end{array}}{\Theta; C[u;\vec{F}] \blacktriangleright \_\{m_i;u_i\}_{i \in I} \blacktriangleleft \Gamma'}$$

$$\textsc{T-BranchEnd}\quad \Theta;\Gamma \blacktriangleright un\{\} \blacktriangleleft \Gamma \qquad\qquad \textsc{T-UsageVar}\quad (\Theta, X : \Gamma);\Gamma \blacktriangleright X \blacktriangleleft \Gamma$$

$$\textsc{T-Variant}\ \frac{\Theta;\Gamma' \blacktriangleright u_t \blacktriangleleft \Gamma \qquad \Theta;\Gamma'' \blacktriangleright u_f \blacktriangleleft \Gamma}{\Theta;\langle\Gamma' + \Gamma''\rangle \blacktriangleright \langle u_t + u_f\rangle \blacktriangleleft \Gamma} \qquad \textsc{T-Rec}\ \frac{(\Theta, X : \Gamma);\Gamma \blacktriangleright u \blacktriangleleft \Gamma'}{\Theta;\Gamma \blacktriangleright \mu X.u \blacktriangleleft \Gamma'}$$

Figure B.9: Revised typing rules for usages

$$\textsc{T-Unit}\quad \Gamma \rhd \mathsf{unit} : \mathsf{void} \lhd \Gamma \qquad \textsc{T-Int}\quad \Gamma \rhd n : \mathsf{int} \lhd \Gamma \qquad \textsc{T-True}\quad \Gamma \rhd \mathsf{true} : \mathsf{boolean} \lhd \Gamma$$

$$\textsc{T-False}\quad \Gamma \rhd \mathsf{false} : \mathsf{boolean} \lhd \Gamma \qquad \textsc{T-Null}\quad \Gamma \rhd \mathsf{null} : \mathsf{null} \lhd \Gamma$$

Figure B.10: Revised typing rules for values

$$\textsc{T-Add}\ \frac{\Gamma \rhd a_1 : \mathsf{int} \lhd \Gamma' \qquad \Gamma' \rhd a_2 : \mathsf{int} \lhd \Gamma''}{\Gamma \rhd a_1 + a_2 : \mathsf{int} \lhd \Gamma''} \qquad \textsc{T-Sub}\ \frac{\Gamma \rhd a_1 : \mathsf{int} \lhd \Gamma' \qquad \Gamma' \rhd a_2 : \mathsf{int} \lhd \Gamma''}{\Gamma \rhd a_1 - a_2 : \mathsf{int} \lhd \Gamma''}$$

$$\textsc{T-Mult}\ \frac{\Gamma \rhd a_1 : \mathsf{int} \lhd \Gamma' \qquad \Gamma' \rhd a_2 : \mathsf{int} \lhd \Gamma''}{\Gamma \rhd a_1 * a_2 : \mathsf{int} \lhd \Gamma''} \qquad \textsc{T-Div}\ \frac{\Gamma \rhd a_1 : \mathsf{int} \lhd \Gamma' \qquad \Gamma' \rhd a_2 : \mathsf{int} \lhd \Gamma''}{\Gamma \rhd a_1 \,/\, a_2 : \mathsf{int} \lhd \Gamma''}$$

Figure B.11: Revised typing rules for arithmetic expressions

$$\textsc{T-Eq}\ \frac{\Gamma \triangleright e_1 : t \triangleleft \Gamma' \qquad \Gamma' \triangleright e_2 : t' \triangleleft \Gamma'' \qquad \mathsf{agree}(t', t)}{\Gamma \triangleright e_1 == e_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-Diff}\ \frac{\Gamma \triangleright e_1 : t \triangleleft \Gamma' \qquad \Gamma' \triangleright e_2 : t' \triangleleft \Gamma'' \qquad \mathsf{agree}(t', t)}{\Gamma \triangleright e_1 != e_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-Greater}\ \frac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 > a_2 : \mathsf{bool} \triangleleft \Gamma''} \qquad \textsc{T-Less}\ \frac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 < a_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-GtEqual}\ \frac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 >= a_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-LeEqual}\ \frac{\Gamma \triangleright a_1 : \mathsf{int} \triangleleft \Gamma' \qquad \Gamma' \triangleright a_2 : \mathsf{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 <= a_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-And}\ \frac{\Gamma \triangleright b_1 : \mathsf{bool} \triangleleft \Gamma' \qquad \Gamma' \triangleright b_2 : \mathsf{bool} \triangleleft \Gamma''}{\Gamma \triangleright b_1 \ \&\&\ b_2 : \mathsf{bool} \triangleleft \Gamma''} \qquad \textsc{T-Or}\ \frac{\Gamma \triangleright b_1 : \mathsf{bool} \triangleleft \Gamma' \qquad \Gamma' \triangleright b_2 : \mathsf{bool} \triangleleft \Gamma''}{\Gamma \triangleright b_1 \parallel b_2 : \mathsf{bool} \triangleleft \Gamma''}$$

$$\textsc{T-Not}\ \frac{\Gamma \triangleright b : \mathsf{bool} \triangleleft \Gamma'}{\Gamma \triangleright !b : \mathsf{bool} \triangleleft \Gamma''}$$

Figure B.12: Revised typing rules for boolean expressions

$$\textsc{T-LinVar}\ \frac{\mathsf{lin}(g)}{(\Gamma, r : g) \triangleright r : g \triangleleft \Gamma} \qquad \textsc{T-UnVar}\ \frac{\mathsf{un}(g)}{(\Gamma, r : t) \triangleright r : g \triangleleft (\Gamma, r : g)}$$

$$\textsc{T-LinField}\ \frac{\Gamma \triangleright \mathit{this} : C[u; \vec{F}] \qquad \vec{F}(f) = g \qquad \mathsf{lin}(g)}{\Gamma \triangleright \mathsf{this}.f : t \triangleleft \Gamma\{\mathsf{this} \mapsto C[u; (\vec{F} \setminus f)]\}}$$

$$\textsc{T-UnField}\ \frac{\Gamma \triangleright \mathsf{this} : C[u; \vec{F}] \qquad \vec{F}(f) = t \qquad \mathsf{un}(t)}{\Gamma \triangleright \mathsf{this}.f : t \triangleleft \Gamma}$$

$$\textsc{T-NullField}\ \frac{\Gamma \triangleright \mathsf{this} : C[u; \vec{F}] \qquad (\_\ f) \notin \vec{F}}{\Gamma \triangleright \mathsf{this}.f : \mathsf{null} \triangleleft \Gamma}$$

Figure B.13: Revised typing rules for field and variable dereference

$$\textsc{T-Seq}\ \dfrac{\Gamma \triangleright s : t \triangleleft \Gamma' \qquad \Gamma' \triangleright t' : g' \triangleleft \Gamma''}{\Gamma \triangleright s;s' : t' \triangleleft \Gamma''}$$

$$\textsc{T-AssignVar}\ \dfrac{e \neq \mathsf{null} \qquad \Gamma \triangleright d : g \triangleleft \Gamma' \qquad \Gamma \triangleright e : g' \triangleleft \Gamma' \qquad \mathsf{agree}(g',g)}{\Gamma \triangleright d = e : \mathsf{void} \triangleleft \Gamma'}$$

$$\textsc{T-AssignField}\ \dfrac{\begin{array}{c} e \neq \mathsf{null} \qquad \Gamma \triangleright e : g \triangleleft \Gamma' \\ \Gamma' \triangleright \mathsf{this} : C[u;\vec{F}] \qquad C.\mathsf{fields}(f) = g' \qquad (\_\ f) \notin \vec{F} \vee \vec{F}(f) = g \qquad \mathsf{agree}(g',g) \end{array}}{\Gamma \triangleright \mathsf{this}.f = e : \mathsf{void} \triangleleft \Gamma'\{\mathsf{this} \mapsto C[u;(\vec{F} \cup (g\ f))]\}}$$

$$\textsc{T-AssignFieldNull}\ \dfrac{\Gamma' \triangleright \Gamma \triangleright e : \mathsf{null} \triangleleft \Gamma \qquad \mathsf{this} : C[u;\vec{F}]}{\Gamma \triangleright \mathsf{this}.f = e : \mathsf{void} \triangleleft \Gamma'\{\mathsf{this} \mapsto C[u;(\vec{F} \setminus (\_\ f))]\}}$$

$$\textsc{T-NewVar}\ \dfrac{d \notin \Gamma \qquad \Gamma \triangleright e : g' \triangleleft \Gamma' \qquad \mathsf{agree}(g',g)}{\Gamma \triangleright g\ d = e : \mathsf{void} \triangleleft \Gamma'\{d \mapsto g\}}$$

$$\textsc{T-Spawn}\ \dfrac{\Gamma \triangleright s : t \triangleleft \Gamma' \qquad \mathsf{un}(\mathsf{modified}(\Gamma,\Gamma')) \qquad \mathsf{completed}(\Gamma,\Gamma')}{\Gamma \triangleright \mathsf{spawn}\ \{s\} : \mathsf{void} \triangleleft \Gamma'}$$

Figure B.14: Revised typing rules for simple statements

$$\textsc{T-IfCall}\ \dfrac{\begin{array}{c} w \neq \mathsf{this} \qquad \Gamma \triangleright w.m(e) : \mathsf{bool} \triangleleft \Gamma' \\ \Gamma' \triangleright w : C[\langle u_t + u_f \rangle;\vec{F}] \qquad \Gamma'\{w \mapsto C[u_t;\vec{F}]\} \triangleright s' : t \triangleleft \Gamma'' \qquad \Gamma'\{w \mapsto C[u_f;\vec{F}]\} \triangleright s'' : t \triangleleft \Gamma'' \end{array}}{\Gamma \triangleright \mathsf{if}\ (w.m(e))\ s'\ \mathsf{else}\ s'' : t \triangleleft \Gamma''}$$

$$\textsc{T-IfNotCall}\ \dfrac{\begin{array}{c} w \neq \mathsf{this} \qquad \Gamma \triangleright w.m(e) : \mathsf{bool} \triangleleft \Gamma' \qquad \Gamma' \triangleright w : C[\langle u_t + u_f \rangle;\vec{F}] \\ \Gamma'\{w \mapsto C[u_f;\vec{F}]\} \triangleright s' : t \triangleleft \Gamma'' \qquad \Gamma'\{w \mapsto C[u_t;\vec{F}]\} \triangleright s'' : t \triangleleft \Gamma'' \end{array}}{\Gamma \triangleright \mathsf{if}\ (!w.m(e))\ s'\ \mathsf{else}\ s'' : t \triangleleft \Gamma''}$$

$$\textsc{T-IfUnCall}\ \dfrac{\Gamma \triangleright w.m(e) : \mathsf{bool} \triangleleft \Gamma' \qquad \Gamma' \triangleright w : C[\epsilon;\vec{F}] \qquad \Gamma' \triangleright s' : t \triangleleft \Gamma'' \qquad \Gamma' \triangleright s'' : t \triangleleft \Gamma''}{\Gamma \triangleright \mathsf{if}\ (w.m(e))\ s'\ \mathsf{else}\ s'' : t \triangleleft \Gamma''}$$

$$\textsc{T-If}\ \dfrac{\Gamma \triangleright b : \mathsf{bool} \triangleleft \Gamma \qquad \Gamma \triangleright s' : t \triangleleft \Gamma'' \qquad \Gamma \triangleright s'' : t \triangleleft \Gamma''}{\Gamma \triangleright \mathsf{if}\ (b)\ s'\ \mathsf{else}\ s'' : t \triangleleft \Gamma''}$$

$$\textsc{T-WhileCall}\ \dfrac{\begin{array}{c} w \neq \mathsf{this} \qquad \Gamma \triangleright w.m(e) : \mathsf{bool} \triangleleft \Gamma' \\ \Gamma' \triangleright w : C[\langle u_t + u_f \rangle;\vec{F}] \qquad \Gamma'\{w \mapsto C[u_t;\vec{F}]\} \triangleright s' : t \triangleleft \Gamma'' \qquad \Gamma(w) = \Gamma''(w) \end{array}}{\Gamma \triangleright \mathsf{while}\ (w.m(e))\{s'\} : t \triangleleft \Gamma''\{w \mapsto C[u_f;\vec{F}]\}}$$

$$\textsc{T-WhileNotCall}\ \dfrac{\begin{array}{c} w \neq \mathsf{this} \qquad \Gamma \triangleright w.m(e) : \mathsf{bool} \triangleleft \Gamma' \\ \Gamma' \triangleright w : C[\langle u_t + u_f \rangle;\vec{F}] \qquad \Gamma'\{w \mapsto C[u_f;\vec{F}]\} \triangleright s' : t \triangleleft \Gamma'' \qquad \Gamma(w) = \Gamma''(w) \end{array}}{\Gamma \triangleright \mathsf{while}\ (!w.m(e))\{s'\} : t \triangleleft \Gamma''\{w \mapsto C[u_t;\vec{F}]\}}$$

$$\textsc{T-WhileUnCall}\ \dfrac{\Gamma \triangleright w.m(e) : \mathsf{bool} \triangleleft \Gamma' \qquad \Gamma' \triangleright w : C[\epsilon;\vec{F}] \qquad \Gamma' \triangleright s : t \triangleleft \Gamma}{\Gamma \triangleright \mathsf{while}\ (w.m(e))\{s\} : t \triangleleft \Gamma'}$$

$$\textsc{T-While}\ \dfrac{\Gamma \triangleright b : \mathsf{bool} \triangleleft \Gamma \qquad \Gamma \triangleright s : t \triangleleft \Gamma}{\Gamma \triangleright \mathsf{while}\ (b)\{s\} : t \triangleleft \Gamma}$$

Figure B.15: Revised typing rules for control flow expressions

$$\textsc{T-InjL}\ \frac{\Gamma \rhd e : t \lhd \Gamma'}{\Gamma \rhd e : t \lhd \langle \Gamma' + \Gamma'' \rangle} \qquad \textsc{T-InjR}\ \frac{\Gamma \rhd e : t \lhd \Gamma''}{\Gamma \rhd e : t \lhd \langle \Gamma' + \Gamma'' \rangle}$$

$$\textsc{T-Sub}\ \frac{\Gamma \rhd e : C[u] \lhd \Gamma' \qquad C[u] <: C[u']}{\Gamma \rhd e : C[u'] \lhd \Gamma'} \textsc{T-SubEnv}\ \frac{\Gamma \rhd e : t \lhd \Gamma' \qquad \Gamma' <: \Gamma''}{\Gamma \rhd e : t \lhd \Gamma'}$$

Figure B.16: Typing rules for subtyping

$$\textsc{T-New}\ \frac{\Gamma \rhd e : t' \lhd \Gamma' \qquad C.\mathsf{usage} = \mathsf{lin}\{C; u\} \qquad (t\ C(t'\ x)\ \{s\}, \_) \in C.\mathsf{methods} \qquad \mathsf{un}(\Gamma' \backslash \Gamma)}{\Gamma \rhd \mathsf{new}\ C(e) : C[u] \lhd \Gamma'}$$

$$\textsc{T-UnNew}\ \frac{\Gamma \rhd e : t' \lhd \Gamma' \qquad C.\mathsf{usage} = \epsilon \qquad (t\ C(t'\ x)\ \{s\}, \_) \in C.\mathsf{methods} \qquad \mathsf{un}(\Gamma' \backslash \Gamma)}{\Gamma \rhd \mathsf{new}\ C(e) : C[u] \lhd \Gamma'}$$

$$\textsc{T-SelfCall1}\ \frac{\begin{array}{c}\Gamma \rhd e : t' \lhd \Gamma' \\ \Gamma' \rhd \mathsf{this} : C[u; \vec{F}] \qquad (t\ m(t'\ x)\ \{s\}, 0) \in C.\mathsf{methods} \qquad \Gamma' \rhd s : t \lhd \Gamma'' \qquad \mathsf{un}(\Gamma' \backslash \Gamma)\end{array}}{\Gamma \rhd \mathsf{this}.m(e) : t \lhd \Gamma''}$$

$$\textsc{T-SelfCall2}\ \frac{\Gamma \rhd e : t' \lhd \Gamma' \qquad \Gamma' \rhd \mathsf{this} : C[u; \vec{F}] \qquad (t\ m(t'\ x)\ \{s\}, 1) \in C.\mathsf{methods}}{\Gamma \rhd \mathsf{this}.m(e) : t \lhd \Gamma'}$$

$$\textsc{T-Call}\ \frac{\begin{array}{c} w \neq \mathsf{this} \qquad \Gamma \rhd e : t' \lhd \Gamma' \\ \Gamma' \rhd w : C[u; \vec{F}] \qquad u.\mathsf{allows}(m) = u' \qquad (t\ m(t'\ x)\ \{s\}, \_) \in C.\mathsf{methods} \qquad \mathsf{un}(\Gamma' \backslash \Gamma)\end{array}}{\Gamma \rhd w.m(e) : t \lhd \Gamma'\{w \mapsto C[u'; \vec{G}]\}}$$

Figure B.17: Revised typing rules for calls

Appendix C

Algorithm implementation in ML

C.1 Algorithm types structures

Listing C.1: ML code for the types and structures used by the algorithm

```
type stmt_type =
  AND
   | ASSIGN
   | CALL
   | CONCAT
   | DIFF
   | DIV
   | EQUAL
   | FALSE
   | FIELD
   | GREATER
   | GREATEROREQUAL
   | IF
   | IFELSE
   | INTEGER
   | LESS
   | LESSOREQUAL
   | MINUS
   | MULT
   | NEW
   | NOT
   | NULL
   | OR
   | PLUS
   | PRINTINT
   | PRINTSTR
   | SEQ
   | SPAWN
   | STRING
   | TRUE
   | UNIT
  | WHILE

type statement = {
  stmt_type : stmt_type;
  left_side : statement list;
  right_side : statement list;
  value : string
}

type var = {
  var_name : string;
  var_type : string;
  var_usage_state : int
}

type action = {
  action_name : string;
  action_usage_state : int;
  action_type : string;
  action_requires : string;
  action_ensures : string;
  is_sync : bool;
  parameters : var list;
  action_body : statement list
}

type state = {
  actions : action list;
  is_choice_state : bool
}

type transition = {
  state_a : state;
  transition_label : string;
  state_b : state
}

type usage_branch = {
  action : action;
  next_states : int list
} and usage_state = {
  usage_state_id : int;
  usage_state_shared : bool;
  branches : usage_branch list
}

type mool_class = {
  class_name : string;
  class_usage : usage_state list;
  class_fields : var list;
  class_inv : string;
  class_init : string;
  actions : action list
}

type analyzed_state = {
  id : int;
  state : state
}
```

C.2 Prover

Listing C.2: ML code for the prover module

```
open Printf
open Types
open Input

open Unix

let file = "assertions.z3"
let file2 = "assertions2.z3"
let file3 = "assertions3.z3"

let print_assertions_to_file file_channel assertions =
  (*Declare fields*)
  List.iter
  (fun field -> fprintf file_channel "%s\n" field)
  fields;
  (*Declare assertions*)
  List.iter
  (fun assertion -> fprintf file_channel "%s\n" assertion)
  assertions;
  fprintf file_channel "%s\n" "(check-sat)"

let prove assertions =
  let file_channel = open_out file in
    print_assertions_to_file file_channel assertions; close_out file_channel;
  let res = (input_line (Unix.open_process_in "z3 assertions.z3")) in
    if res = "sat" then true else false

let prove_variant assertions =
  let file_channel = open_out file2 in
    print_assertions_to_file file_channel assertions; close_out file_channel;
  let res = (input_line (Unix.open_process_in "z3 assertions2.z3")) in
    Sys.remove "assertions2.z3"; if res = "sat" then true else false

let rec build_assertion channel next_line =
  match next_line with
```

```
    " :precision precise :depth 1)" -> ""
    | "(error \"tactic failed: split-clause tactic failed, goal does not contain any
         clause\")
" -> print_string "ERROR\n"; ""
    | line -> line^(build_assertion channel (input_line channel))

let get_true assertion =
  let file_channel = open_out file2 in
    List.iter
    (fun field -> fprintf file_channel "%s\n" field)
    fields;
    fprintf file_channel "%s\n" ("(assert "^assertion^")");

    fprintf file_channel "%s\n" "(apply split-clause)"; close_out file_channel;
    let channel = (Unix.open_process_in "z3 assertions2.z3") in
      input_line channel; input_line channel;
      "(and "^(build_assertion channel "")^")"

let get_false assertion =
  let file_channel = open_out file2 in
    List.iter
    (fun field -> fprintf file_channel "%s\n" field)
    fields;
    fprintf file_channel "%s\n" ("(assert "^assertion^")");

    fprintf file_channel "%s\n" "(apply split-clause)"; close_out file_channel;
    let channel = (Unix.open_process_in "z3 assertions2.z3") in
      input_line channel; input_line channel; (build_assertion channel "");
          input_line channel;
      "(and "^(build_assertion channel "")^")"

let get_variant assertion state =
  let file_channel = open_out file2 in
    List.iter
    (fun field -> fprintf file_channel "%s\n" field)
    fields;
    fprintf file_channel "%s\n" assertion;

    fprintf file_channel "%s\n" "(apply split-clause)"; close_out file_channel;
    let channel = (Unix.open_process_in "z3 assertions2.z3") in
      input_line channel; input_line channel;

      let assertion_left = "(assert (and "^(build_assertion channel "")^"))\n" in
        input_line channel;
        let assertion_right = "(assert (and "^(build_assertion channel "")^"))\n" in
          List.exists
          (fun action -> (prove_variant [assertion_left; "(assert "^action.
              action_requires^")"]) && (not (prove_variant [assertion_right; "(assert
               "^action.action_requires^")"])))
          state
```

```

let has_two_states action =
  String.sub action.action_ensures 0 3 = "(or"

let changes_state action =
  if(action.action_requires = action.action_ensures)
    then false
    else
      (let file_channel = open_out file3 in
        List.iter
        (fun field -> fprintf file_channel "%s\n" field)
        fields;
        fprintf file_channel "%s\n" ("(assert "^action.action_ensures^")");

        fprintf file_channel "%s\n" "(apply split-clause)"; close_out file_channel;
        let channel = (Unix.open_process_in "z3 assertions3.z3") in
          if ((input_line channel) = "(error \"tactic failed: split-clause tactic
              failed, goal does not contain any clause\")")
          then not (prove ["(assert "^(action.action_requires)^")"; "(assert "^action.
              action_ensures^")"])
          else (input_line channel;
          let assertion_left = "(assert (and "^(build_assertion channel "")^"))\n" in
            input_line channel;
            let assertion_right = "(assert (and "^(build_assertion channel "")^"))\n"
                in
              not (prove ["(assert "^(action.action_requires)^")"; assertion_left])
                  || not (prove ["(assert "^(action.action_requires)^")";
                  assertion_right])))
```

C.3 Stage 1 algorithm

Listing C.3: ML code for the first stage of the algorithm

```
let build_state class_inv a actions =
  let initial_actions = List.filter
    (fun action -> Prover.prove ["(assert "^class_inv^")"; "(assert "^a^")"; "(
        assert "^action.action_requires^")"])
    actions in
      {actions = initial_actions; is_choice_state = false}

let generate_choice_state action =
  let dummy_action = {action_name = (action.action_name^"_choice");
    action_type = "unit";
    action_usage_state = -1;
    action_requires = "";
    action_ensures = "";
    is_sync = false;
    parameters = [];
    action_body = []} in {actions = [dummy_action]; is_choice_state = true}
```

```


let build_transitions class_inv class_init actions action_a state_a =
  if action_a.action_type = "boolean" && Prover.has_two_states action_a
  then
    let next_state_true = (build_state class_inv (Prover.get_true action_a.
         action_ensures) actions) and
        next_state_false = (build_state class_inv (Prover.get_false action_a.
             action_ensures) actions) in
      [{state_a = state_a; transition_label = action_a.action_name; state_b =
           generate_choice_state action_a};{state_a = generate_choice_state action_a;
            transition_label = "true"; state_b = next_state_true}; {state_a =
           generate_choice_state action_a; transition_label = "false"; state_b =
           next_state_false}]
  else
    if (not (Prover.changes_state action_a))
    then [{state_a = state_a; transition_label = action_a.action_name; state_b =
         state_a}]
    else let next_state = (build_state class_inv action_a.action_ensures actions) in
      [{state_a = state_a; transition_label = action_a.action_name; state_b =
           next_state}]


let rec analyse_state class_inv class_init actions w states delta =
  match w with
    [] -> delta
    | state::w ->
      if (not (List.exists (fun s -> s = state) states) && not state.is_choice_state
           )
        then let transitions = List.flatten (List.map
          (fun a -> build_transitions class_inv class_init actions a state)
        state.actions) in
          analyse_state class_inv class_init actions (List.append w (List.map (fun t
               -> t.state_b) transitions)) (List.append states [state]) (List.append
               delta transitions)
        else analyse_state class_inv class_init actions w states delta



let generate_typestate class_inv class_init actions =
  let initial_state = (build_state class_inv class_init actions) in
    analyse_state class_inv class_init actions [initial_state] [] []
```

C.4 Stage 2 algorithm

Listing C.4: ML code for the second stage of the algorithm

```
let rec build_state_set states analysed_states =
  match states with
```

```
    [] -> analysed_states
    | state::states -> let is_analysed = List.exists (fun s -> s.state = state)
        analysed_states in
      if is_analysed then build_state_set states analysed_states
      else build_state_set states (List.append analysed_states [{id = (List.length
          analysed_states); state = state}])

let rec get_shared_status usage_state_id branches =
  List.for_all
  (fun b -> List.length b.next_states < 2 && List.hd b.next_states = usage_state_id)
  branches

let get_next_state state action delta =
  List.hd (List.map (fun t -> t.state_b) (List.filter (fun t -> t.state_a = state &&
       t.transition_label = action.action_name) delta))

let get_decision_transition state_a label delta =
  List.find (fun t -> t.state_a = state_a && t.transition_label = label) delta

let generate_branches analysed_state analysed_states state delta =
  List.map
  (fun a -> let next_state = get_next_state state a delta in
    if next_state.is_choice_state
      then let true_transition = get_decision_transition next_state "true" delta and
            false_transition = get_decision_transition next_state "false" delta in
          ({action = a; next_states = [(List.find (fun s -> s.state = true_transition.
              state_b) analysed_states).id; (List.find (fun s -> s.state =
              false_transition.state_b) analysed_states).id]} )
      else if next_state.actions = [] then {action = a; next_states = [-1]}
      else {action = a; next_states = [(List.find (fun s -> s.state = next_state)
          analysed_states).id]}) state.actions

let rec generate_usage analysed_states states delta usage =
  match states with
    [] -> usage
    | state::states -> let analysed_state = List.find (fun s -> s.state = state)
        analysed_states in
      let branches = generate_branches analysed_state analysed_states state delta in
      List.append
      [{
        usage_state_id = analysed_state.id;
        usage_state_shared = get_shared_status analysed_state.id branches;
        branches = branches
      }]
      (generate_usage analysed_states states delta usage)

let rec check_usage usage_1 usage_2 =
  match usage_1 with
    [] -> usage_2
    | u::usage_1 -> if (u.usage_state_shared = true && (List.exists (fun b -> List.length b.next_states = 2 || (List.find (fun next_state -> next_state.usage_state_id = List.hd b.next_states) usage_2).usage_state_shared = false) u.branches)) then
        raise (Failure "Incorrect usage")
      else check_usage usage_1 usage_2

let rec annotate_classes input_classes output_classes =
  match input_classes with
    [] -> output_classes
    | c::input_classes ->
      let delta = generate_typestate c.class_inv c.class_init (List.filter (fun action -> action.action_name <> c.class_name) c.actions) in
        let states = List.filter (fun s -> not s.is_choice_state) (List.map (fun t -> t.state_a) delta) in
          let analysed_states = (build_state_set states []) in
            let usage = (generate_usage analysed_states (List.map (fun s -> s.state) analysed_states) delta []) in
              let new_class = {class_name = c.class_name; class_usage = check_usage usage usage; class_fields = c.class_fields; class_inv = c.class_inv; class_init = c.class_init; actions = c.actions} in
                annotate_classes input_classes (List.append output_classes [new_class])
```

C.5 Stage 3 algorithm

Listing C.5: ML code for the third stage of the algorithm

```
let is_primitive_type t =
  match t with
    "void" -> true
    | "boolean" -> true
    | "int" -> true
    | "string" -> true
    | _ -> false

let rec get_parameter_usage_state_2 usage action =
  match usage with
      [] -> -1
      | usage_state::usage -> if (List.for_all (fun b -> Prover.prove [("(assert "^action.action_requires^")"); ("(assert "^b.action.action_requires^")")]) usage_state.branches) then usage_state.usage_state_id else get_parameter_usage_state_2 usage action

let rec get_parameter_usage_state_1 classes parameter action =
  match is_primitive_type parameter.var_type with
    true -> -1
    | false -> let usage = (List.find (fun c -> c.class_name = parameter.var_type) classes).class_usage in
    get_parameter_usage_state_2 usage action

let rec annotate_parameters classes action input_parameters output_parameters =
  match input_parameters with
    [] -> output_parameters
    | p::input_parameters ->
      let new_parameter = {var_name = p.var_name; var_type = p.var_type;
          var_usage_state = (get_parameter_usage_state_1 classes p action)} in
        annotate_parameters classes action input_parameters (List.append
            output_parameters [new_parameter])

let rec get_action_usage_state_2 usage action =
  match usage with
      [] -> -1
      | usage_state::usage -> if (List.for_all (fun b -> Prover.prove [("(assert "^
          action.action_ensures^")"); ("(assert "^b.action.action_requires^")")])
          usage_state.branches) then usage_state.usage_state_id else
          get_action_usage_state_2 usage action

let rec get_action_usage_state_1 classes action =
  match is_primitive_type action.action_type with
    true -> -1
    | false -> let usage = (List.find (fun c -> c.class_name = action.action_type)
        classes).class_usage in
      get_action_usage_state_2 usage action

let rec annotate_actions classes input_actions output_actions =
  match input_actions with
    [] -> output_actions
    | a::input_actions ->
      let new_parameters = annotate_parameters classes a a.parameters [] in
        let new_action = {action_name = a.action_name; action_usage_state = (
            get_action_usage_state_1 classes a); action_type = a.action_type;
            action_requires = a.action_requires; action_ensures = a.action_ensures;
            is_sync = a.is_sync; parameters = new_parameters; action_body = a.
            action_body} in
      annotate_actions classes input_actions (List.append output_actions [new_action
          ])

let get_field_usage_state_from_param parameters param_name =
  let p = List.hd (List.filter (fun f -> f.var_name = param_name) parameters) in
    p.var_usage_state

let get_field_usage_state_from_call classes actions class_name action_name =
  let c = List.find (fun c -> c.class_name = class_name) classes in
    let a = List.hd (List.filter (fun a -> a.action_name = action_name) c.actions)
        in
      a.action_usage_state

let rec detect_attributions_2 classes fields actions parameters s output_fields=
  match s with
    {stmt_type = ASSIGN; left_side = [s1]; right_side = [{stmt_type = FIELD; left_side = []; right_side = []; value = v}]} ->
      let field = List.find (fun f -> f.var_name = s1.value) fields in
        output_fields@[{var_name = field.var_name; var_type = field.var_type; var_usage_state = (get_field_usage_state_from_param parameters v)}]
    | {stmt_type = ASSIGN; left_side = [s1]; right_side = [{stmt_type = CALL; left_side = [s2]; right_side = [s3]; value = v}]} ->
      let field = List.find (fun f -> f.var_name = s1.value) fields in
        output_fields@[{var_name = field.var_name; var_type = field.var_type; var_usage_state = (get_field_usage_state_from_call classes actions s2.value s3.value)}]
    | {stmt_type = CALL; left_side = [s1]; right_side = [s2]} ->
      let old_parameter = List.find (fun p -> p.var_name = s1.value) parameters in
        let new_parameter = {var_name = old_parameter.var_name; var_type = old_parameter.var_type; var_usage_state = old_parameter.var_usage_state} in
          let new_parameters = (List.filter (fun p -> p = old_parameter) parameters)@[new_parameter] in
            detect_attributions_2 classes fields actions new_parameters s1 output_fields
    | {stmt_type = IF; left_side = [s1]; right_side = []} -> detect_attributions_2 classes fields actions parameters s1 output_fields
    | {stmt_type = IFELSE; left_side = [s1]; right_side = [s2]} -> detect_attributions_2 classes fields actions parameters s2 (detect_attributions_2 classes fields actions parameters s1 output_fields)
    | {stmt_type = SEQ; left_side = [s1]; right_side = [s2]} -> detect_attributions_2 classes fields actions parameters s2 (detect_attributions_2 classes fields actions parameters s1 output_fields)
    | {stmt_type = SPAWN; left_side = [s1]} -> detect_attributions_2 classes fields actions parameters s1 output_fields
    | {stmt_type = WHILE; left_side = [s1]; right_side = []} -> detect_attributions_2 classes fields actions parameters s1 output_fields
    | _ -> output_fields


let rec detect_attributions_1 classes fields actions parameters stmts output_fields =
  match stmts with
    [] -> output_fields
    | s::stmts -> let new_output_fields = detect_attributions_2 classes fields actions parameters s [] in
      detect_attributions_1 classes fields actions parameters stmts new_output_fields

let rec annotate_fields classes fields actions output_fields =
  match actions with
    [] -> output_fields
    | a::actions -> detect_attributions_1 classes fields actions a.parameters a.action_body [];
```

```
      output_fields

let rec get_object_usage_states input_classes output_classes =
  match input_classes with
    [] -> output_classes
    | c::input_classes ->
      let new_actions = annotate_actions (input_classes@output_classes@[c]) c.
          actions [] in
        let new_fields = annotate_fields (input_classes@output_classes@[c]) c.
            class_fields new_actions [] in
          let new_class = {class_name = c.class_name; class_usage = c.class_usage;
              class_fields = new_fields; class_inv = c.class_inv; class_init = c.
              class_init; actions = new_actions} in
            get_object_usage_states input_classes (List.append output_classes [
                new_class])
```