A Revision of the Mool Language

Cláudio Vasconcelos    António Ravara

NOVA-LINCS and Dep. de Informática, FCT.
Universidade NOVA de Lisboa, Portugal

July 5, 2016

Abstract

We present here in a thorough analysis of the Mool language, covering not only its implementation but also the formalisation (syntax, operational semantics, and type system). The objective is to detect glitches in both the implementation and in the formal definitions, proposing as well new features and added expressiveness. To test our proposals we implemented the revision developed in the Racket platform.

1 Introduction

This article presents an analysis of the Mool language, a small object-oriented language similar to Java, developed by Campos and Vasconcelos [1, 2]. The language allows to associate with each class a behavioural type specifying safe orderings of method calls, along the lines of [5].

This analysis is a contribution to the development of the language, detecting bugs not only in the implementation, but also in formalisation. We also propose revisions of aspects of the language we find too restrictive.

Section 2 presents correction proposals. We organise them in two categories: minor aspects (Section 2.1), which have little influence on the language or their correction is very straightforward; and major aspects in (Section 2.2), 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 0.2, available in September 2015 from gloss.di.fc.ul.pt/mool/download). The purpose is to understand if the aspects we presented in section 2 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 [3], a module available in Racket [4] that allows us to implement and debug formal systems of programming languages. Section 4 present our implementation and explains briefly the examples we used to demonstrate how the aspects in Section 2 affect the language. The code of our implementation, along with the examples, is available at bitbucket.org/cvasconcelos/thesis/src/876fc254db78aca1bb058bb7e6ef069ee23c6c237/Mool/mool1.rkt.

Section 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 [2] plus the addition of new features such as constructors.

Again, to test our revision we implemented the revised formalisation using
PLT-Redex. Section 6 presents the list of examples used to test this second
implementation. Most of these examples are almost identical to the ones in
section 4 but now they are expected to have a different behaviour, while a
few new examples that were used to test our changes a little further. The code
of the implementation, along with the examples, is available at bitbucket.org/
cvasconcelos/thesis/src/876fc254db76aca1bb058b7e6ef069ee23c6c237/Mool/
mool2.rkt.

2 The original Mool language

Like said before, the main objective is to understand where Mool can be too re-
strictive or even present incorrect behaviour. We did this by not only reviewing
the original definitions [2], but also by implementing the language using PLT Re-
dex and trying to falsify properties of the system (see Section 4). These aspects
have been categorised in major and minor aspects, based on their complexity.

2.1 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 ex-
pression $e$ 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.

2.2 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 [5], is as follows:
If $\langle u' + u'' \rangle < u$ then $u = \langle u_t + u_f \rangle$ with $u' < u_t$ and $u'' < u_f$

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

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

In this expression, is the file has been fully read then it closes and returns \text{false}, informing the client that there is no more lines to read, otherwise it reads a line and returns true, informing the client it can try to read another line. Assume that we reverse the result output as follow:

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

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 existence 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

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

where methods \text{open}, \text{read} and \text{close} are all of type unit, the code

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

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 $f.\text{read}()$ 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:

$$\begin{align*}
\frac{
\begin{array}{c} 
\therefore \\
T1
\end{array}}{
\Gamma \vdash f.\text{open}() : \text{unit} \triangleleft \Gamma' \quad T-\text{CALL} \\
\Gamma \vdash f.\text{open}(); \text{spawn } f.\text{read}(); f.\text{close}() : \text{unit} \triangleleft \Gamma'' \quad T-\text{SEQ}
\end{align*}$$

$$\begin{align*}
\frac{
\begin{array}{c} 
\therefore \\
T2
\end{array}}{
\Gamma' \vdash \text{spawn } f.\text{read}() : \text{unit} \triangleleft \Gamma'' \quad T-\text{CALL} \\
\Gamma' \vdash \text{spawn } f.\text{read}(); f.\text{close}() : \text{unit} \triangleleft \Gamma'' \quad T-\text{SPAWN} \\
\Gamma' \vdash f.\text{read}(); f.\text{close}() : \text{unit} \triangleleft \Gamma'' \quad T-\text{SEQ}
\end{align*}$$

$$\begin{align*}
\frac{
\begin{array}{c} 
\therefore \\
T3
\end{array}}{
\Gamma'' \vdash f.\text{close}() : \text{unit} \triangleleft \Gamma'' \quad T-\text{CALL}
\end{align*}$$
\[
\Gamma = f : \text{File}\{\text{lin}\{\text{open} : \text{lin}\{\text{read} : \text{lin}\{\text{close} : \text{un}\{\}}\}\}\}\}
\]
\[
\Gamma' = f : \text{File}\{\text{lin}\{\text{read} : \text{lin}\{\text{close} : \text{un}\{\}}\}\}\}
\]
\[
\Gamma'' = f : \text{File}\{\text{close} : \text{un}\{\}}\}
\]
\[
\Gamma''' = f : \text{File}\{\text{un}\{\}}\}
\]

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

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

6. 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.

7. 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

\[
\text{if}(!\text{f.eof}()) \text{f.read()} \text{else} \text{f.close()}
\]

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

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

9. \text{null} cannot be used as a value, not allowing the programmer to set objects to \text{null} or check if they are \text{null}.

10. 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.

\[
\text{lin}\{\text{open} : \mu\text{Read}\{\text{eof} : (\text{close} : \text{un}\{\} + \text{read} : \text{Read})\}\}\}
\]

This usage, presented in the configuration of the core language, is a slightly modified usage to the \text{File} class of the example presented in [2]. 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 \text{open} the usage goes from linear to unrestricted and when executing the method \text{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 \text{FileReader} class that hosts a file whose reading access can be blocked or unblocked. A possible usage would be:
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:

\[
\text{lin}\{\text{open} : \mu\text{Blocked.un}\text{\{unblock}; \\
\mu\text{Unblocked.un}\text{\{block; Blocked + read : Unblocked}\}}\}\]

Although the original definition \cite{2} 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 Latest Mool implementation

The work developed and presented in the following sections is based on the Mool language presented in \cite{2}, but we also reviewed the current Mool implementation available\footnote{The latest Mool implementation is available at\url{gloss.di.fc.ul.pt/tryit/Mool}} to check if the aspects noted in Sections 2.1 and 2.2 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.

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

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 2.2 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 2.1 was dropped, allowing code like this:

Listing 2: FileReader linear attribution example

```java
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 2.2, with a class `File` with the following usage:

Listing 3: File usage variation

```java
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 4: FileReader spawn example 1

```java
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 5: FileReader spawn example 2

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

With respect to 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 6: FileReader negated call example 1

```java
if (!f.eof()) {
    s = s ++ f.read();
    true;
} else {
```
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 7: FileReader negated call example 2

```java
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 8: FileReader bad usage example 1

```java
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 9: FileReader bad usage example 2

```java
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 10: FileReader example 5

```java
class FileReader {
    usage FileReader (open; Blocked) where
    Blocked = un(unblock; Unblocked)
    Unblocked = un(read; Unblocked + block; Blocked);
    ...
}
```
4 PLT Redex implementation

We implemented Mool as presented in [2] using PLT Redex[3]. Due to the syntax of Racket, we had to make some modifications on the syntax of Mool, such as:

- 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 2.2. In order to implement more elaborate examples, some other changes were made:

- Items 1, 2 and 4 of Section 2.1 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 [2], 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 2.2. 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 2.2. The type checker verifies successfully when it should not.

T-03 Implementation of the example presented in item 3 of Section 2.2. 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.

\footnote{Available at bitbucket.org/cvasconcelos/thesis/src/875c254db76aca1bb058b7e6ef069ee23c6c237/Mool/mool1.rkt}
T-04 Same thing as T-01 but the fields File from the FileReader class and FileReader 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 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-06 Implementation of the File and FileReader classes as presented in [2], 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 7 of section 2.2 and it should fail.

T-07 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 7 of section 2.2 with the same result, but now in a while expression.

T-08 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 10 of Section 2.2.

T-09 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 10 of Section 2.2 is composed by two different unrestricted states and they change between them. This should not be allowed but the type checker allows it.

5 The revised Mool language

This section presents our revision of the Mool language that tries to solve the problems mentioned in Sections 2.1 and 2.2. Some of the modifications are based on the observations made in Section 3.

5.1 Revised syntax

Figure 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 2.2).

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 \( t \cdot d = e \) 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 \texttt{new } \( C() \) to \texttt{new } \( C(e) \), allowing to pass parameters to the constructor.

6. 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.

7. In the nonterminal \( u \) we added \( \epsilon \) to indicate that it is possible to not define an usage, making the class an unrestricted class.

8. The term \( o \), which are objects identifiers, is moved from the user syntax for the runtime syntax.

9. In the runtime syntax, a new type of value, \texttt{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 problem in item 4 of Section 2.2.

10. The evaluation context while \( \mathcal{E} \) \( e \) is removed for the reasons stated in item 1 of Section 2.1.

5.2 Revised operational semantics

Figure 3 shows the modified reduction rules for this revised version of Mool. The rules differ from the original ones, as we add a new environment, \texttt{local}, for the local variables.

We modified the rule \texttt{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 \texttt{R-NewVar} and \texttt{R-AssignVar} which are for local variable declaration and assignment.

The rule \texttt{R-AssignFieldNull}, allows to assign \texttt{null} values to fields, removing them from the object’s record.

Figures 4 and 5 show the evaluation functions for the arithmetic and boolean expressions. These functions, based on the ones presented in [6], receive as arguments an expression and both the class field and local variable environment.

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 [2].

Figure 7 shows the proposed typing rules for programs:

1. Rule \texttt{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.
User Syntax

(class declarations) \[ D ::= \text{class} \ C \{u; \vec{F}; \vec{M}\} \]

(field declaration) \[ F ::= t \ f \]

(method declarations) \[ M ::= y \ t \ m(t') \{e\} \]

(method qualifiers) \[ y ::= e | \text{sync} \]

(values) \[ v ::= \text{unit} | n | \text{true} | \text{false} | \text{null} \]

(local value references) \[ r ::= d | \text{this} \]

(global value references) \[ w ::= r | r.f \]

(calls) \[ c ::= \text{new} \ C(e) | r.m(e) | r.f.m(e) \]

(arithmetic operations) \[ a ::= n | w | c | a + a | a - a \]

(boolean operations) \[ b ::= \text{true} | \text{false} | w | c | a == a | a != a \]

(expressions) \[ e ::= v | a | b \]

(statements) \[ s ::= e | s; s' \]

(types) \[ t ::= \text{void} | \text{bool} | C[z] \]

(class usages) \[ u ::= e | q\{m; z_i\}_{i \in I | \mu X.u} \]

(usage types) \[ q ::= \text{un} | \text{lin} \]

Runtime Syntax

(values) \[ v ::= \ldots | o \]

(value references) \[ r ::= \ldots | o \]

(expressions) \[ e ::= \ldots | s; e | \text{if} (b) e' \text{else} e'' \]

(class usages) \[ u ::= \ldots | z \]

(types) \[ t ::= \ldots | C[z; \vec{F}] | \text{null} \]

(object records) \[ R ::= (C, u, f = v, l) \]

(field value map) \[ l ::= 0 | 1 \]

(heap) \[ h ::= \emptyset | h, o = R \]

(evaluation context) \[ \mathcal{E} ::= [-] | \mathcal{E}; s | o.f = \mathcal{E} | o.m(\mathcal{E}) | o.f.m(\mathcal{E}) \]

| if (\mathcal{E}) s \text{else} s' 

Figure 1: Revised syntax
Object Record and Heap Operations

\[(C, u), V \vdash f \overset{\text{def}}{=} V(f)\]
\[(C, u), V, \overrightarrow{V} \vdash \text{usage} \overset{\text{def}}{=} u\]

Class Definition Operations

\[(C, u), V \vdash \text{lin}(v, h) \overset{\text{def}}{=} h(v) \text{.usage} = l\]
\[(C, u), V \vdash \text{un}(v, h) \overset{\text{def}}{=} v \notin \text{dom}(h) \lor \lnot \text{lin}(v, h)\]

Figure 2: Auxiliary definitions and Operations

\[
\begin{align*}
\text{R-UnField} & \quad (h, o, f = v) \quad \text{un}(v, h) \\
& \quad \text{lin}(v, h) \\
& \quad (h, o, f \rightarrow \text{null}) \quad (h, \text{local}, v) \\
\text{R-LinField} & \quad (h, o, f = v) \quad \text{lin}(v, h) \\
& \quad (h, o, f \rightarrow \text{null}) \quad (h, \text{local}, v) \\
\text{R-UnVar} & \quad (h, d = v) \quad \text{un}(v, h) \\
& \quad (h, \text{local}, d) \quad (h, \text{local}, v) \\
\text{R-LinVar} & \quad (h, d = v) \quad \text{lin}(v, h) \\
& \quad (h, \text{local}, d) \quad (h, \{d \rightarrow \text{null}\}, \text{local}, v) \\
\text{R-Seq} & \quad (h, \text{local}, v; s) \quad (h, \text{local}, s) \\
\text{R-NewVar} & \quad (h, \text{local}, t = d) \quad (h, \text{local}, t = d) \quad (h, \text{local}, \{d \rightarrow v\}, \text{unit}) \\
\text{R-AssignVar} & \quad (h, \text{local}, d = v) \quad (h, \text{local}, \{d \rightarrow v\}, \text{unit}) \\
\text{R-AssignField} & \quad (h, o, f = v) \quad \text{null} \\
& \quad (h, \text{local}, o, f \rightarrow \text{null}) \quad (h, \text{local}, \text{unit}) \\
\text{R-AssignFieldNull} & \quad (h, o, f = v) \quad \text{null} \\
& \quad (h, \text{local}, o, f \rightarrow \text{null}) \quad (h, \text{local}, \text{unit}) \\
\text{R-New} & \quad \text{o fresh} \quad \text{C}(\_x) \{s, \ldots\} \in \text{C.methods} \quad \text{C.fields} = t \\
& \quad \text{C.usage} = u \\
& \quad (h, \text{local}, \text{new} C(v)) \quad (h, o = (C, u, f = \text{null})), \text{local}, s\{o/\text{this}\}\{o/\_s\}; o) \\
\text{R-Call} & \quad \text{m}(\_x) \{s, \ldots\} \in \text{C}(o).\text{class}.\text{methods} \\
& \quad (h, \text{local}, o.m(v)) \quad (h, \text{local}, s\{o/\text{this}\}\{o/\_s\}) \\
\text{R-FieldCall} & \quad \text{m}(\_x) \{s, \ldots\} \in \text{C}(o.f.class).\text{methods} \\
& \quad (h, \text{local}, o.f.m(v)) \quad (h, \text{local}, s\{o/\text{this}\}\{o/\_s\}) \\
\text{R-While} & \quad (h, \text{local}, \text{while} (b)\{s\}) \quad (h, \text{local}, \{\text{if} (b) (s) \text{while} (b)\{s\} \text{else} \text{unit}\}) \\
\text{R-IfTrue} & \quad (h, \text{local}, \{\text{if} (true) s' \text{ else} s''\}) \quad (h, \text{local}, s') \\
\text{R-IfFalse} & \quad (h, \text{local}, \{\text{if} (false) s' \text{ else} s''\}) \quad (h, \text{local}, s'')
\end{align*}
\]

Figure 3: Revised reduction Semantics
\[ N(n) = n \quad A(n, h, \text{local}) = N(n) \]
\[ A(o.f, h, \text{local}) = h(o).f \quad A(d, h, \text{local}) = \text{local}(d) \]
\[ A(a_1 + a_2, h, \text{local}) = A(a_1, h, \text{local}) + A(a_2, h, \text{local}) \]
\[ A(a_1 - a_2, h, \text{local}) = A(a_1, h, \text{local}) - A(a_2, h, \text{local}) \]
\[ A(a_1 \cdot a_2, h, \text{local}) = A(a_1, h, \text{local}) \cdot A(a_2, h, \text{local}) \]
\[ A(a_1 / a_2, h, \text{local}) = A(a_1, h, \text{local}) / A(a_2, h, \text{local}) \]

Figure 4: Evaluation functions for arithmetic values and expressions

\[ B(\text{true}, h, \text{local}) = \text{true} \quad B(\text{false}, h, \text{local}) = \text{false} \]
\[ B(o.f, h, \text{local}) = h(o).f \quad B(d, h, \text{local}) = \text{local}(d) \]
\[ B(a_1 == a_2, h, \text{local}) = \begin{cases} \text{true} & A(a_1, h, \text{local}) = A(a_2, h, \text{local}) \\ \text{false} & A(a_1, h, \text{local}) \neq A(a_2, h, \text{local}) \end{cases} \]
\[ B(a_1 \neq a_2, h, \text{local}) = \begin{cases} \text{true} & A(a_1, h, \text{local}) \neq A(a_2, h, \text{local}) \\ \text{false} & A(a_1, h, \text{local}) = A(a_2, h, \text{local}) \end{cases} \]
\[ B(a_1 < a_2, h, \text{local}) = \begin{cases} \text{true} & A(a_1, h, \text{local}) < A(a_2, h, \text{local}) \\ \text{false} & A(a_1, h, \text{local}) \geq A(a_2, h, \text{local}) \end{cases} \]
\[ B(a_1 < a_2, h, \text{local}) = \begin{cases} \text{true} & A(a_1, h, \text{local}) < A(a_2, h, \text{local}) \\ \text{false} & A(a_1, h, \text{local}) \leq A(a_2, h, \text{local}) \end{cases} \]
\[ B(a_1 <= a_2, h, \text{local}) = \begin{cases} \text{true} & A(a_1, h, \text{local}) \leq A(a_2, h, \text{local}) \\ \text{false} & A(a_1, h, \text{local}) > A(a_2, h, \text{local}) \end{cases} \]
\[ B(a_1 >= a_2, h, \text{local}) = \begin{cases} \text{true} & A(a_1, h, \text{local}) \geq A(a_2, h, \text{local}) \\ \text{false} & A(a_1, h, \text{local}) < A(a_2, h, \text{local}) \end{cases} \]
\[ B(b_1 \&\& b_2, h, \text{local}) = \begin{cases} \text{true} & B(b_1, h, \text{local}) \land (b_2, h, \text{local}) = \text{true} \\ \text{false} & \text{otherwise} \end{cases} \]
\[ B(b_1 || b_2, h, \text{local}) = \begin{cases} \text{true} & B(b_1, h, \text{local}) \lor (b_2, h, \text{local}) = \text{true} \\ \text{false} & \text{otherwise} \end{cases} \]
\[ B!(h, h, \text{local}) = \begin{cases} \text{true} & B(h, h, \text{local}) = \text{false} \\ \text{false} & B(h, h, \text{local}) = \text{true} \end{cases} \]

Figure 5: Evaluation functions for boolean values and expressions
Moreover, evaluation of the usage has an object $C[u; \emptyset]$ for input, with no declared fields, and a object $C[u'; \vec{F}]$ for output, forcing the method-level scope of the system. 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.

This rule can be used to verify a Main class of a program.

Figures 8 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 2.1.

Figures 9 and 10 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 12 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 2.1.

2. The rule T-Spawn is modified based on the conclusions presented in Section 3, 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 13 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 an object of an unrestricted class.

2. To solve the error in item 7 in section 2.2, 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 14 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 2.2. Unlike the other typing rules for method calls, these do not change the usage, like the system description in [2] 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 [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 2.1 corrected.

Moreover, due to the definition of the predicate allows, in particular the case when the usage is \( \epsilon \), i.e., the class is unrestricted, the predicate also returns \( \epsilon \), this rule can also be applied when the call is made on an object of an unrestricted class.
Type Operations

\[
\begin{align*}
\text{check}(\emptyset, u) & \quad u = \text{lin}\{m_i; u_i\}_{i \in I} \\
\text{check}(\emptyset, u) \land \text{check}(\Theta, u_j) & \quad u = (u + u_i) \\
\text{check}((\Phi, X : u), u) & \quad u = \mu X.u \\
\text{ff} & \quad u = \mu\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u_n = mX.u_X \\
\text{ff} & \quad u = \mu\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u = \text{lin}\{m_j; u_j\}_{j \in J} \\
\text{ff} & \quad u = \mu\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u_n = X \land \Phi(X) = (u + u_i) \\
\text{ff} & \quad u = \mu\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u = (u + u_i) \\
\text{ff} & \quad u = \mu\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u = \mu\{m_j; u_j\}_{j \in J} \land m_i \neq m_j \\
\text{ff} & \quad u = \mu\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u_n = X \land \Phi(X) = \mu\{m_j; u_j\}_{j \in J} \land m_i \neq m_j \\
\text{tt} & \quad \text{otherwise} \\
\end{align*}
\]

\[ u.\text{allows}(m_j) \overset{\text{def}}{=} \begin{cases} \\
\epsilon & \quad u = \epsilon \\
\text{un} & \quad u = \text{un} \\
\{m_i; u_i\}_{i \in I} \land j \in I & \quad u = \text{un}\{m_i; u_i\}_{i \in I} \land j \in I \\
\text{undef} & \quad \text{otherwise} \\
\end{cases} \]

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

\[ \text{modified}(\Gamma, \Gamma') \overset{\text{def}}{=} \forall r \in \Gamma' : r \notin \Gamma \lor \Gamma(r) \neq \Gamma'(r) \]

\[ \text{completed}(\Gamma, \Gamma') \overset{\text{def}}{=} \forall r \in \Gamma' : (r \notin \Gamma \lor \Gamma(r) \neq \Gamma'(r)) \land (r = C[u; \overrightarrow{F}] \implies u = \text{un}(\{\}) ) \]

Figure 6: Types, Type Definitions and Operations

\[
\begin{align*}
\text{T-CLASS} & \quad \frac{\text{check}(\emptyset, u)}{C[u; \emptyset] \vdash u < C[u; \overrightarrow{F}]} \quad \text{un}(\overrightarrow{F}) \\
\vdash \text{class } C\{u; \overrightarrow{F}; \overrightarrow{M}\} \\
\text{T-UNCLASS} & \quad \frac{(y \ t \ m_i(t' x) \ \{s\}, \_ \in \overrightarrow{M}) \in \overrightarrow{I}}{C[\emptyset].x : t' \vdash s \Diamond C[\overrightarrow{F}]} \quad \text{un}(\overrightarrow{F}) \\
\vdash \text{class } C[\overrightarrow{F}; \overrightarrow{M}] \\
\end{align*}
\]

Figure 7: Revised typing rules for programs

\[
\begin{align*}
\text{T-BRANCH} & \quad \frac{\forall i \in I. \\
(y \ t \ m_i(t' x) \ \{s\}, \_ \in C.\text{methods} \quad \text{this } : C[u; \overrightarrow{F}], x : t' \vdash s : t \triangleleft \Gamma) \\
\Gamma \vdash x : t'' \quad \text{un}(t'') \quad \text{this } : C[u; \overrightarrow{F}_i] \\
\Theta_i \vdash \text{this } : \Gamma' \quad \Theta_i \vdash u_i \triangleleft \Gamma'}{\Theta_i; C[u; \overrightarrow{F}] \triangleright\_ \{m_i; u_i\}_{i \in I} \triangleleft \Gamma'} \\
\end{align*}
\]

\[
\begin{align*}
\text{T-BRANCHEND} \quad \Theta_i \vdash \text{un}(\{\}) \triangleleft \Gamma & \quad \text{T-USAGEVAR } (\Theta_i, X : \Gamma) \vdash X \triangleleft \Gamma \\
\end{align*}
\]

Figure 8: Revised typing rules for usages

16
Figure 9: Revised typing rules for arithmetic expressions

| Rule   | Precondition                              | Postcondition                              |
|--------|-------------------------------------------|--------------------------------------------|
| T-ADD  | $\Gamma \vdash a_1 : \text{int} \triangleright \Gamma'$ $\Gamma' \vdash a_2 : \text{int} \triangleright \Gamma''$ | $\Gamma \vdash a_1 + a_2 : \text{int} \triangleright \Gamma''$ |
| T-SUB  | $\Gamma \vdash a_1 : \text{int} \triangleright \Gamma'$ $\Gamma' \vdash a_2 : \text{int} \triangleright \Gamma''$ | $\Gamma \vdash a_1 - a_2 : \text{int} \triangleright \Gamma''$ |
| T-MULT | $\Gamma \vdash a_1 : \text{int} \triangleright \Gamma'$ $\Gamma' \vdash a_2 : \text{int} \triangleright \Gamma''$ | $\Gamma \vdash a_1 \times a_2 : \text{int} \triangleright \Gamma''$ |
| T-DIV  | $\Gamma \vdash a_1 : \text{int} \triangleright \Gamma'$ $\Gamma' \vdash a_2 : \text{int} \triangleright \Gamma''$ | $\Gamma \vdash a_1 \div a_2 : \text{int} \triangleright \Gamma''$ |
| T-EQ   | $\Gamma \vdash e_1 : t \triangleright \Gamma'$ $\Gamma' \vdash e_2 : t' \triangleright \Gamma''$ | $\text{agree}(t', t)$ $\Gamma \vdash e_1 == e_2 : \text{bool} \triangleright \Gamma''$ |
| T-DIFF | $\Gamma \vdash e_1 : t \triangleright \Gamma'$ $\Gamma' \vdash e_2 : t' \triangleright \Gamma''$ | $\text{agree}(t', t)$ $\Gamma \vdash e_1! = e_2 : \text{bool} \triangleright \Gamma''$ |
| T-GREATER | $\Gamma \vdash a_1 : \text{int} \triangleright \Gamma'$ $\Gamma' \vdash a_2 : \text{int} \triangleright \Gamma''$ | $\Gamma \vdash a_1 < a_2 : \text{bool} \triangleright \Gamma''$ |
| T-LESS | $\Gamma \vdash a_1 : \text{int} \triangleright \Gamma'$ $\Gamma' \vdash a_2 : \text{int} \triangleright \Gamma''$ | $\Gamma \vdash a_1 < a_2 : \text{bool} \triangleright \Gamma''$ |
| T-GTEQUAL | $\Gamma \vdash a_1 : \text{int} \triangleright \Gamma'$ $\Gamma' \vdash a_2 : \text{int} \triangleright \Gamma''$ | $\Gamma \vdash a_1 >= a_2 : \text{bool} \triangleright \Gamma''$ |
| T-LEEQUAL | $\Gamma \vdash a_1 : \text{int} \triangleright \Gamma'$ $\Gamma' \vdash a_2 : \text{int} \triangleright \Gamma''$ | $\Gamma \vdash a_1 <= a_2 : \text{bool} \triangleright \Gamma''$ |
| T-AND  | $\Gamma \vdash b_1 : \text{bool} \triangleright \Gamma'$ $\Gamma' \vdash b_2 : \text{bool} \triangleright \Gamma''$ | $\Gamma \vdash b_1 \&\& b_2 : \text{bool} \triangleright \Gamma''$ |
| T-OR   | $\Gamma \vdash b_1 : \text{bool} \triangleright \Gamma'$ $\Gamma' \vdash b_2 : \text{bool} \triangleright \Gamma''$ | $\Gamma \vdash b_1 || b_2 : \text{bool} \triangleright \Gamma''$ |
| T-NOT  | $\Gamma \vdash b : \text{bool} \triangleright \Gamma'$ | $\Gamma \vdash \neg b : \text{bool} \triangleright \Gamma''$ |
| T-LINVAR | $\Gamma, r : t \triangleright (t : \Gamma)$ | $\Gamma \vdash \text{lin}(t)$ |
| T-UNVAR  | $\Gamma, r : t \triangleright (r : \Gamma)$ | $\Gamma \vdash \text{un}(t)$ |
| T-LINFIELD | $\Gamma \vdash \text{this:} C[u; F]$ $F(f) = t$ | $\Gamma \vdash \text{lin}(t)$ |
| T-UNFIELD | $\Gamma \vdash \text{this:} C[u; F]$ $F(f) = t$ | $\Gamma \vdash \text{lin}(t)$ |

Figure 10: Revised typing rules for boolean expressions

Figure 11: Revised typing rules for field and variable dereference
T-AssignVar: $\frac{\Gamma \vdash d : t < \Gamma' \quad \Gamma \vdash e : t' < \Gamma' \quad \text{agree}(t', t)}{\Gamma \vdash d = e : \text{void} < \Gamma'}$

T-AssignField: $\frac{\Gamma \vdash e : t < \Gamma' \quad \Gamma' \vdash \text{this} : C[u; F] \quad \text{C.fields}(f) = t' \quad \text{agree}(t', t)}{\Gamma \vdash \text{this}.f = e : \text{void} < \Gamma' \{ \text{this} \mapsto C[u; (F \cup (t f))] \}}$

T-AssignFieldNull: $\frac{\Gamma' \vdash \text{this} : C[u; F] \quad \text{C.fields}(f) = t}{\Gamma \vdash \text{this}.f = \text{null} : \text{void} < \Gamma' \{ \text{this} \mapsto C[u; (F \setminus (t f))] \}}$

T-NewVar: $\frac{d \notin \Gamma \quad \Gamma \vdash e : t' < \Gamma' \quad \text{agree}(t', t)}{\Gamma \vdash d = e : \text{void} < \Gamma' \{ d \mapsto t \}}$

T-Spawn: $\frac{\Gamma \vdash s : t < \Gamma' \quad \text{un(modified}(\Gamma, \Gamma')) \quad \text{completed}(\Gamma, \Gamma')}{\Gamma \vdash \text{spawn } \{ s \} : \text{void} < \Gamma'}$

Figure 12: Revised typing rules for simple statements

T-IfCall: $\frac{w \notin \text{this} \quad \Gamma \vdash w.m(e) : \text{bool} < \Gamma' \quad \Gamma' \vdash w : C[(u_i + u_f); \bar{F}]}{\Gamma' \{ w \mapsto C[u_i; \bar{F}] \} \vdash s' : t < \Gamma'' \quad \Gamma' \{ w \mapsto C[u_f; \bar{F}] \} \vdash s'' : t < \Gamma''}$

T-IfNotCall: $\frac{w \notin \text{this} \quad \Gamma \vdash w.m(e) : \text{bool} < \Gamma' \quad \Gamma' \vdash w : C[(u_i + u_f); \bar{F}]}{\Gamma' \{ w \mapsto C[u_i; \bar{F}] \} \vdash s' : t < \Gamma'' \quad \Gamma' \{ w \mapsto C[u_f; \bar{F}] \} \vdash s'' : t < \Gamma''}$

T-IfUncall: $\frac{\Gamma \vdash w.m(e) : \text{bool} < \Gamma'}{\Gamma \vdash (w.m)(e) s' \text{ else } s'' : t < \Gamma''}$

T-If: $\frac{\Gamma \vdash b : \text{bool} < \Gamma \quad \Gamma \vdash s' : t < \Gamma'' \quad \Gamma \vdash s'' : t < \Gamma''}{\Gamma \vdash (b) s' \text{ else } s'' : t < \Gamma''}$

T-WhileCall: $\frac{w \notin \text{this} \quad \Gamma \vdash w.m(e) : \text{bool} < \Gamma' \quad \Gamma' \vdash w : C[(u_i + u_f); \bar{F}]}{\Gamma' \{ w \mapsto C[u_i; \bar{F}] \} \vdash s' : t < \Gamma'' \quad \Gamma(w) = \Gamma''(w)}$

T-WhileNotCall: $\frac{w \notin \text{this} \quad \Gamma \vdash w.m(e) : \text{bool} < \Gamma' \quad \Gamma' \vdash w : C[(u_i + u_f); \bar{F}]}{\Gamma' \{ w \mapsto C[u_i; \bar{F}] \} \vdash s' : t < \Gamma'' \quad \Gamma(w) = \Gamma''(w)}$

T-WhileUncall: $\frac{\Gamma \vdash w.m(e) : \text{bool} < \Gamma' \quad \Gamma \vdash w : C[e; \bar{F}]}{\Gamma \vdash \text{while } (w.m)(e) \{ s \} : t < \Gamma'}$

T-While: $\frac{\Gamma \vdash b : \text{bool} < \Gamma \quad \Gamma \vdash s : t < \Gamma}{\Gamma \vdash \text{while } (b) \{ s \} : t < \Gamma'}$

Figure 13: Revised typing rules for control flow expressions
6 PLT Redex implementation

To test our revision we implemented our formal system in PLT Redex\footnote{Available at bitbucket.org/cvasconcelos/thesis/src/876fc254db76aca1bb058b7e6ef069ee23c6c237/Mool/mool2.rkt}. 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 4. The examples presented in this version are the following:

R-01 Implementation of the File example presented in [2] 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 [5].

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 [2] that serves as a more complex test to the operational semantics of Mool.

T-01 Typing example of the File example presented in [2]. 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.

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.

T-06 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-07 Since now the type system is aware of which methods were already evaluated, this time the type checker will not enter in an 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-08 Typing example of the File example presented in [5]. Should evaluate successfully.

T-09 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-10 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-11 Again, the new rule T-CLASS also prevents an usage changing between different unrestricted states, so the program verification should fail;

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

T-13 Typing example of the simple program introduced in R-03. Should evaluate successfully.

T-14 Typing example of the Auction example presented in [2]. Should evaluate successfully.
7 Conclusions and further work

Following a detailed analysis of the formal definition and of the implementation of the Mool programming language, we provide the formalisation of a new version of the language with corrections of errors and broader approaches to aspects where the language is too restrictive. We also provide the implementation of the formalisation of both the original and the revised versions using the Racket programming language, more specifically its PLT Redex module, both complemented with examples to help understanding the evolution between versions.

The next stage of our work will be about the inference of usages from programs written in a variation of Mool based on our revised formalisation but it will not have usage annotations. Instead, the programs will be equipped with assertions that we will use to infer the usages.

References

[1] Joana Campos and Vasco 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.
[2] Joana Correia Campos. “Linear and shared objects in concurrent programming”. MA thesis. University of Lisbon, 2010.
[3] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. Semantics Engineering with PLT Redex. 1st. The MIT Press, 2009. ISBN: 0262062755, 9780262062756.
[4] Matthias Felleisen et al. “The Racket Manifesto.” In: SNAPL. Ed. by Thomas Ball et al. 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.
[5] Simon J. Gay et al. “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.
[6] Hanne Riis Nielson and Flemming Nielson. Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science), Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN: 1846286913.
A  Mool- syntax

A.1 User syntax

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

A.2 Runtime syntax

(values) \[ v ::= \ldots \mid o \]
(value references) \[ r ::= \ldots \mid o \]
(expressions) \[ e ::= \ldots \mid s; e \mid \text{if} (b) \, e' \text{ else } e'' \]
(class usages) \[ u ::= \ldots \mid z \]
(types) \[ t ::= \ldots \mid C[z; \vec{F}] \mid \text{null} \]
(object records) \[ R ::= (C, u, \vec{f} = \vec{v}, l) \]
(field value map) \[ l ::= 0 \mid 1 \]
(heap) \[ h ::= \emptyset \mid h, o = R \]
(evaluation context) \[ \mathcal{E} ::= [-] \mid \mathcal{E}; s \mid o.f = \mathcal{E} \mid o.m(\mathcal{E}) \mid o.f.m(\mathcal{E}) \]
\[ \mid \text{if} (\mathcal{E}) \, s \text{ else } s' \]
B Mool revised operational semantics

B.1 Object Record and Heap Operations

\[ \langle C, u, \vec{V} \rangle . f \overset{\text{def}}{=} \vec{V}(f) \]
\[ \langle C, u, \vec{V} \rangle . \text{usage} \overset{\text{def}}{=} u \]
\[ \langle C, u, \vec{V} \rangle . \text{class} \overset{\text{def}}{=} C \]
\[ \text{lin}(v, h) \overset{\text{def}}{=} h(v).\text{usage} = l \]
\[ \text{un}(v, h) \overset{\text{def}}{=} v \not\in \text{dom}(h) \lor \neg \text{lin}(v, h) \]

B.2 Class Definition Operations

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

B.3 Evaluation functions for arithmetic values and expressions

\[ \mathcal{N}(n) = n \quad \mathcal{A}(n, h, \text{local}) = \mathcal{N}(n) \]
\[ \mathcal{A}(o.f, h, \text{local}) = h(o).f \quad \mathcal{A}(d, h, \text{local}) = \text{local}(d) \]
\[ \mathcal{A}(a_1 + a_2, h, \text{local}) = \mathcal{A}(a_1, h, \text{local}) + \mathcal{A}(a_2, h, \text{local}) \]
\[ \mathcal{A}(a_1 - a_2, h, \text{local}) = \mathcal{A}(a_1, h, \text{local}) - \mathcal{A}(a_2, h, \text{local}) \]
\[ \mathcal{A}(a_1 \ast a_2, h, \text{local}) = \mathcal{A}(a_1, h, \text{local}) \ast \mathcal{A}(a_2, h, \text{local}) \]
\[ \mathcal{A}(a_1 / a_2, h, \text{local}) = \mathcal{A}(a_1, h, \text{local}) / \mathcal{A}(a_2, h, \text{local}) \]
B.4 Evaluation functions for boolean values and expressions

\[
\begin{align*}
B(\text{true}, h, \text{local}) &= \text{true} & B(\text{false}, h, \text{local}) &= \text{false} \\
B(o.f, h, \text{local}) &= h(o).f & B(d, h, \text{local}) &= \text{local}(d) \\
B(a_1 == a_2, h, \text{local}) &= \begin{cases} 
\text{true} & A(a_1, h, \text{local}) = A(a_2, h, \text{local}) \\
\text{false} & A(a_1, h, \text{local}) \neq A(a_2, h, \text{local}) 
\end{cases} \\
B(a_1 != a_2, h, \text{local}) &= \begin{cases} 
\text{true} & A(a_1, h, \text{local}) \neq A(a_2, h, \text{local}) \\
\text{false} & A(a_1, h, \text{local}) = A(a_2, h, \text{local}) 
\end{cases} \\
B(a_1 < a_2, h, \text{local}) &= \begin{cases} 
\text{true} & A(a_1, h, \text{local}) < A(a_2, h, \text{local}) \\
\text{false} & A(a_1, h, \text{local}) \geq A(a_2, h, \text{local}) 
\end{cases} \\
B(a_1 <= a_2, h, \text{local}) &= \begin{cases} 
\text{true} & A(a_1, h, \text{local}) \leq A(a_2, h, \text{local}) \\
\text{false} & A(a_1, h, \text{local}) > A(a_2, h, \text{local}) 
\end{cases} \\
B(a_1 >= a_2, h, \text{local}) &= \begin{cases} 
\text{true} & A(a_1, h, \text{local}) \geq A(a_2, h, \text{local}) \\
\text{false} & A(a_1, h, \text{local}) < A(a_2, h, \text{local}) 
\end{cases} \\
B(b_1 \&\& b_2, h, \text{local}) &= \begin{cases} 
\text{true} & B(b_1, h, \text{local}) = \text{true} \wedge (b_2, h, \text{local}) = \text{true} \\
\text{false} & \text{otherwise} 
\end{cases} \\
B(b_1 || b_2, h, \text{local}) &= \begin{cases} 
\text{true} & B(b_1, h, \text{local}) = \text{true} \lor (b_2, h, \text{local}) = \text{true} \\
\text{false} & \text{otherwise} 
\end{cases} \\
B(b, h, \text{local}) &= \begin{cases} 
\text{true} & B(b, h, \text{local}) = \text{false} \\
\text{false} & B(b, h, \text{local}) = \text{true} 
\end{cases}
\end{align*}
\]
B.5 Reduction rules

\[ \frac{h.o.f = v}{(h, \text{local}, o.f) \rightarrow (h, \text{local}, v)} \]  
\textbf{R-UnField}  

\[ \frac{h.o.f = v}{(h, \text{local}, o.f) \rightarrow (h[\text{local} \mapsto \text{null}], \text{local}, v)} \]  
\textbf{R-LinField}  

\[ \frac{h[d] = v}{(h, \text{local}, d) \rightarrow (h, \text{local}, v)} \]  
\textbf{R-UnVar}  

\[ \frac{h[d] = v}{(h, \text{local}, d) \rightarrow (h[\text{local} \mapsto \text{null}], \text{local}, v)} \]  
\textbf{R-LinVar}  

\[ \frac{(\text{C}(x) \{s\}, \_ ) \in \text{C}.\text{methods} \quad \text{C.fields} = f \quad \text{C.usage} = u}{(h, \text{local}, \text{new} C(v)) \rightarrow ((h, o = \langle C, u, f = \text{null} \rangle, \text{local}, s(\text{\_}/\text{this})\{s'/x\}; o)}} \]  
\textbf{R-New}  

\[ \frac{(\_ m(\_ x) \{s\}, \_ ) \in (h(o).\text{class}).\text{methods}}{(h, o, m(v)) \rightarrow (h, \text{local}, s(\_)/\\text{this})\{s'/x\})} \]  
\textbf{R-Call}  

\[ \frac{(\_ m(\_ x) \{s\}, \_ ) \in (h(o).f.\text{class}).\text{methods}}{(h, o, m(v)) \rightarrow (h, \text{local}, s(\_)/\\text{this})\{s'/x\})} \]  
\textbf{R-FieldCall}  

\[ \frac{(h, \text{local}, \text{while} (b) \{s\}) \rightarrow (h, \text{local}, \text{if} (b) \{s\}; \text{while} (b) \{s\}) \text{ else unit)}{(h, \text{local}, \text{while} (b) \{s\}) \rightarrow (h, \text{local}, \text{if} (b) \{s\}; \text{while} (b) \{s\}) \text{ else unit)}} \]  
\textbf{R-While}  

\[ \frac{(h, \text{local}, \text{if} (\text{true}) s' \text{ else } s'') \rightarrow (h, \text{local}, s')}{(h, \text{local}, \text{if} (\text{true}) s' \text{ else } s'') \rightarrow (h, \text{local}, s')} \]  
\textbf{R-IfTrue}  

\[ \frac{(h, \text{local}, \text{if} (\text{false}) s' \text{ else } s'') \rightarrow (h, \text{local}, s')}{(h, \text{local}, \text{if} (\text{false}) s' \text{ else } s'') \rightarrow (h, \text{local}, s')} \]  
\textbf{R-IfFalse}
C Mool revised type system

C.1 Type operations

\[
\begin{align*}
\text{check}(\Phi, u) & \quad u = \text{lin}\{m_i; u_i\}_{i \in I} \\
\text{check}(\Phi, u) \land \text{check}(\Theta, u_f) & \quad u = (u_t + u_u) \\
\text{check}(\Phi, X : u, u) & \quad u = \mu X. u \\
\text{ff} & \quad u = \text{un}\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u_n = \mu X. u_X \\
\text{ff} & \quad u = \text{un}\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u = \text{lin}\{m_j; u_j\}_{j \in J} \\
\text{ff} & \quad u = \text{un}\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u_n = X \land \\
\Phi(X) & \quad (u_t + u_u) \\
\text{ff} & \quad u = \text{un}\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u_n = X \land \\
\Phi(X) & \quad \text{lin}\{m_i; u_j\}_{j \in J} \\
\text{ff} & \quad u = \text{un}\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u = (u_t + u_u) \\
\text{ff} & \quad u = \text{un}\{m_i; u_i\}_{i \in I} \land \exists u_n \in u_i : u = \text{un}\{m_j; u_j\}_{j \in J} \land m_i \neq m_j \\
\text{tt} & \quad u = \text{un}\{m_j; u_j\}_{j \in J} \land m_i \neq m_j \\
\end{align*}
\]

\[
\begin{align*}
\text{check}(\Phi, u) & \quad \text{if } u = \epsilon \\
\text{check}(\Phi, u) & \quad \text{un} \\
\text{check}(\Phi, u) & \quad \text{u'}{\{\mu X. u'/X\}}.\text{allows}(m_j) \quad u = \mu X. u' \\
\text{tt} & \quad \text{tt} \quad \text{if } t = t' = \text{bool} \\
\text{tt} & \quad \text{tt} \quad \text{if } t = t' = \text{void} \\
\text{tt} & \quad \text{tt} \quad \text{if } t = C[u] \text{ and } t' = C[u; F] \\
\text{tt} & \quad \text{tt} \quad \text{if } t = \text{null or } t' = \text{null} \\
\end{align*}
\]

\[
\begin{align*}
\text{un}(\Phi) & \quad \forall r \in \Gamma' : r \not\in \Gamma \lor \Gamma(r) \neq \Gamma'(r) \\
\text{completed}(\Gamma, \Gamma') & \quad \forall r \in \Gamma' : (r \not\in \Gamma \lor \Gamma(r) \neq \Gamma'(r)) \land (r = C[u; F]) \implies u = \text{un}\{j\}
\end{align*}
\]

C.2 Revised typing rules for programs

\[
\begin{align*}
\text{T-CLASS} & \quad \text{check}(\emptyset, u) \quad C[u; \emptyset] \triangleright u \triangleq C[u; F] \quad \text{un}(F) \\
\text{\quad} & \quad \triangleright \text{class } C \{u; F, M\} \\
\text{tt} & \quad \forall i \in I. \\
\text{T-UNCLASS} & \quad (y \ t \ m_i(t' x) \ \{s\}, \_ ) \in M \quad C[\emptyset], x : t' \triangleright s \triangleq C[F] \quad \text{un}(F) \\
\text{\quad} & \quad \triangleright \text{class } C\{F, M\} \\
\end{align*}
\]

C.3 Revised typing rules for usages

\[
\begin{align*}
\text{T-BRANCH} & \quad \forall i \in I. \\
\text{\quad} & \quad (y \ t \ m_i(t' x) \ \{s\}, \_ ) \in \text{C.methods} \quad \text{this : } C[\emptyset, F], x : t' \triangleright s : t \triangleq \Gamma \\
\text{\quad} & \quad \Theta; C[u; F] \triangleright \_ \{m_i; u_i\}_{i \in I} \triangleq \Gamma' \\
\text{T-VARIANT} & \quad \Theta; \Gamma \triangleright u_i \triangleq \Gamma \\
\text{\quad} & \quad \Theta; \Gamma'' \triangleright u_j \triangleq \Gamma \\
\text{T-REC} & \quad \Theta; \Gamma \triangleright \mu X. u \triangleq \Gamma'' \\
\end{align*}
\]

T-BranchEnd \quad \Theta; \Gamma \triangleright \text{un}\{\} \triangleq \Gamma
C.4 Revised typing rules for values

\[
\begin{align*}
&\text{T-UNIT} \quad \Gamma \triangleright \text{unit} : \text{void} < \Gamma \\
&\text{T-INT} \quad \Gamma \triangleright n : \text{int} < \Gamma \\
&\text{T-TRUE} \quad \Gamma \triangleright \text{true} : \text{boolean} < \Gamma \\
&\text{T-FALSE} \quad \Gamma \triangleright \text{false} : \text{boolean} < \Gamma \\
&\text{T-NULL} \quad \Gamma \triangleright \text{null} : \text{null} < \Gamma \\
\end{align*}
\]

C.5 Revised typing rules for arithmetic expressions

\[
\begin{align*}
&\text{T-ADD} \quad \Gamma \triangleright a_1 : \text{int} < \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} < \Gamma'' \quad \Gamma \triangleright a_1 + a_2 : \text{int} < \Gamma'' \\
&\text{T-SUB} \quad \Gamma \triangleright a_1 : \text{int} < \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} < \Gamma'' \quad \Gamma \triangleright a_1 - a_2 : \text{int} < \Gamma'' \\
&\text{T-MULT} \quad \Gamma \triangleright a_1 : \text{int} < \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} < \Gamma'' \quad \Gamma \triangleright a_1 \times a_2 : \text{int} < \Gamma'' \\
&\text{T-DIV} \quad \Gamma \triangleright a_1 : \text{int} < \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} < \Gamma'' \quad \Gamma \triangleright a_1 / a_2 : \text{int} < \Gamma'' \\
\end{align*}
\]

C.6 Revised typing rules for boolean expressions

\[
\begin{align*}
&\text{T-EQ} \quad \Gamma \triangleright e_1 : t < \Gamma' \quad \Gamma' \triangleright e_2 : t' < \Gamma'' \quad \text{agree}(t', t) \quad \Gamma \triangleright e_1 == e_2 : \text{bool} < \Gamma'' \\
&\text{T-DIFF} \quad \Gamma \triangleright e_1 : t < \Gamma' \quad \Gamma' \triangleright e_2 : t' < \Gamma'' \quad \text{agree}(t', t) \quad \Gamma \triangleright e_1 \neq e_2 : \text{bool} < \Gamma'' \\
&\text{T-GREATER} \quad \Gamma \triangleright a_1 : \text{int} < \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} < \Gamma'' \quad \Gamma \triangleright a_1 > a_2 : \text{bool} < \Gamma'' \\
&\text{T-LESS} \quad \Gamma \triangleright a_1 : \text{int} < \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} < \Gamma'' \quad \Gamma \triangleright a_1 < a_2 : \text{bool} < \Gamma'' \\
&\text{T-GTEQUAL} \quad \Gamma \triangleright a_1 : \text{int} < \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} < \Gamma'' \quad \Gamma \triangleright a_1 >= a_2 : \text{bool} < \Gamma'' \\
&\text{T-LTEQUAL} \quad \Gamma \triangleright a_1 : \text{int} < \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} < \Gamma'' \quad \Gamma \triangleright a_1 <= a_2 : \text{bool} < \Gamma'' \\
&\text{T-AND} \quad \Gamma \triangleright b_1 : \text{bool} < \Gamma' \quad \Gamma' \triangleright b_2 : \text{bool} < \Gamma'' \quad \Gamma \triangleright b_1 \&\& b_2 : \text{bool} < \Gamma'' \\
&\text{T-OR} \quad \Gamma \triangleright b_1 : \text{bool} < \Gamma' \quad \Gamma' \triangleright b_2 : \text{bool} < \Gamma'' \quad \Gamma \triangleright b_1 || b_2 : \text{bool} < \Gamma'' \\
&\text{T-NOT} \quad \Gamma \triangleright b : \text{bool} < \Gamma' \quad \Gamma \triangleright \neg b : \text{bool} < \Gamma'' \\
\end{align*}
\]
C.7 Revised typing rules for field and variable dereference

| Rule          | Pre-Condition | Post-Condition |
|--------------|---------------|----------------|
| T-LinVar     | $\Gamma, r : t \vdash r : t \in \Gamma$ | $\Gamma, r : t \vdash r : t \in \Gamma$ |
| T-UnVar      | $\Gamma, r : t \vdash r : t \in \Gamma$ | $\Gamma, r : t \vdash r : t \in \Gamma$ |
| T-LinField   | $\Gamma \vdash \text{this} : C[u; \bar{F}]$ | $\bar{F}(f) = t$ | $\text{lin}(t)$ |
|              | $\Gamma \vdash \text{this} : t \in \Gamma \{\text{this} \rightarrow C[u; (\bar{F} \setminus f)]\}$ |
| T-UnField    | $\Gamma \vdash \text{this} : C[u; \bar{F}]$ | $\bar{F}(f) = t$ | $\text{un}(t)$ |
|              | $\Gamma \vdash \text{this} : t \in \Gamma$ |

C.8 Revised typing rules for simple statements

| Rule          | Pre-Condition | Post-Condition |
|--------------|---------------|----------------|
| T-Seq        | $\Gamma \vdash e : t \in \Gamma'$ | $\Gamma \vdash e' : t' \in \Gamma''$ |
| T-AssignVar  | $e \neq \text{null}$ | $\Gamma \vdash d : t \in \Gamma'$ | $\Gamma \vdash e : t' \in \Gamma'$ | $\text{agree}(t', t)$ |
|              | $\Gamma \vdash d = e : \text{void} \in \Gamma'$ |
| T-AssignField| $\Gamma \vdash e : t \in \Gamma'$ | $\Gamma' \vdash \text{this} : C[u; \bar{F}]$ | $\text{fields}(f) = t'$ | $\text{agree}(t', t)$ |
|              | $e \neq \text{null}$ | $\Gamma \vdash \text{this}.f = e : \text{void} \in \Gamma'' \{\text{this} \rightarrow C[u; (\bar{F} \cup (t, f))]\}$ |
| T-AssignFieldNull | $\Gamma' \vdash \text{this} : C[u; \bar{F}]$ | $\text{fields}(f) = t$ |
|              | $\Gamma \vdash \text{this}.f = \text{null} : \text{void} \in \Gamma'' \{\text{this} \rightarrow C[u; (\bar{F} \setminus (t, f))]\}$ |
| T-NewVar     | $d \notin \Gamma$ | $\Gamma \vdash e : t' \in \Gamma'$ | $\text{agree}(t', t)$ |
|              | $\Gamma \vdash t d = e : \text{void} \in \Gamma'' \{d \rightarrow t\}$ |
| T-Spawn      | $\Gamma \vdash s : t \in \Gamma'$ | $\text{un}(\text{modified}(\Gamma, \Gamma'))$ | $\text{completed}(\Gamma, \Gamma')$ |
|              | $\Gamma \vdash \text{spawn} \{s\} : \text{void} \in \Gamma''$ |
C.9 Revised typing rules for control flow expressions

**T-IfCall**
\[
\begin{array}{c}
\begin{array}{c}
w \neq \text{this} \\
\Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \\
\Gamma' \triangleright w : C[(u_1 + u_2) ; \vec{F}] \\
\Gamma' \{ w \mapsto C[u_1 ; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \\
\end{array} \\
\Gamma \triangleright (w.m(e)) s' \text{ else } s'' : t \triangleleft \Gamma''
\end{array}
\]

**T-IfNotCall**
\[
\begin{array}{c}
\begin{array}{c}
w \neq \text{this} \\
\Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \\
\Gamma' \triangleright w : C[(u_1 + u_2) ; \vec{F}] \\
\Gamma' \{ w \mapsto C[u_1 ; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \\
\Gamma' \{ w \mapsto C[u_1 ; \vec{F}] \} \triangleright s'' : t \triangleleft \Gamma''
\end{array} \\
\Gamma \triangleright \text{if (w.m(e)) s' else s'' : t} \triangleleft \Gamma''
\end{array}
\]

**T-IfUnCall**
\[
\begin{array}{c}
\begin{array}{c}
\Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \\
\Gamma' \triangleright w : C[e ; \vec{F}] \\
\Gamma' \triangleright s' : t \triangleleft \Gamma'' \\
\Gamma' \triangleright s'' : t \triangleleft \Gamma''
\end{array} \\
\Gamma \triangleright \text{if (w.m(e)) else s'' : t} \triangleleft \Gamma''
\end{array}
\]

**T-If**
\[
\begin{array}{c}
\begin{array}{c}
\Gamma \triangleright b : \text{bool} \triangleleft \Gamma \\
\Gamma \triangleright s' : t \triangleleft \Gamma'' \\
\Gamma \triangleright s'' : t \triangleleft \Gamma''
\end{array} \\
\Gamma \triangleright \text{if (b) s' else s'' : t} \triangleleft \Gamma''
\end{array}
\]

**T-WhileCall**
\[
\begin{array}{c}
\begin{array}{c}
w \neq \text{this} \\
\Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \\
\Gamma' \triangleright w : C[(u_1 + u_2) ; \vec{F}] \\
\Gamma' \{ w \mapsto C[u_1 ; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \\
\Gamma(w) = \Gamma''(w)
\end{array} \\
\Gamma \triangleright \text{while (w.m(e)} \{ s' \} : t \triangleleft \Gamma'' \{ w \mapsto C[u_1 ; \vec{F}] \}
\end{array}
\]

**T-WhileNotCall**
\[
\begin{array}{c}
\begin{array}{c}
w \neq \text{this} \\
\Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \\
\Gamma' \triangleright w : C[(u_1 + u_2) ; \vec{F}] \\
\Gamma' \{ w \mapsto C[u_1 ; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \\
\Gamma(w) = \Gamma''(w)
\end{array} \\
\Gamma \triangleright \text{while (w.m(e)} \{ s' \} : t \triangleleft \Gamma'' \{ w \mapsto C[u_1 ; \vec{F}] \}
\end{array}
\]

**T-WhileUnCall**
\[
\begin{array}{c}
\begin{array}{c}
\Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \\
\Gamma' \triangleright w : C[e ; \vec{F}] \\
\Gamma' \triangleright s : t \triangleleft \Gamma
\end{array} \\
\Gamma \triangleright \text{while (w.m(e))} \{ s \} : t \triangleleft \Gamma
\end{array}
\]

**C.10 Typing rules for subtyping**

**T-InjL**
\[
\begin{array}{c}
\Gamma \triangleright e : t \triangleleft \Gamma'
\end{array}
\]

**T-InjR**
\[
\begin{array}{c}
\Gamma \triangleright e : t \triangleleft (\Gamma + \Gamma')
\end{array}
\]

**T-Sub**
\[
\begin{array}{c}
\Gamma \triangleright e : C[u] \triangleleft \Gamma' \\
\Gamma \triangleright e : C[u'] \triangleleft \Gamma' \\
\Gamma \triangleright e : t \triangleleft \Gamma'
\end{array}
\]

**T-SubEnv**
\[
\begin{array}{c}
\Gamma \triangleright e : t \triangleleft \Gamma' \\
\Gamma' \triangleright e : t \triangleleft \Gamma'
\end{array}
\]
C.11 Revised typing rules for calls

| Rule | Premise | Conclusion |
|------|---------|------------|
| T-NEW | $\Gamma \vdash e : t' < \Gamma'$ $\Gamma \vdash C(e) : C[u] < \Gamma'$ | $\Gamma \vdash \text{new } C(e) : C[u] < \Gamma'$ $\text{un}(\Gamma' \setminus \Gamma)$ |
| T-UNNEW | $\Gamma \vdash e : t' < \Gamma'$ $\Gamma \vdash (t C(t' x) \{s\}, \_ \_ \_ \_ \_ \gamma) \in C.$ methods $\text{un}(\Gamma' \setminus \Gamma)$ | $\Gamma \vdash \text{new } C(e) : C[u] < \Gamma'$ $\text{un}(\Gamma' \setminus \Gamma)$ |
| T-SELFCALL1 | $\Gamma \vdash e : t' < \Gamma'$ $\Gamma' \triangleright this : C[u; F]$ $\Gamma' \vdash s : t < \Gamma'' \text{ un}(\Gamma'' \setminus \Gamma)$ | $\Gamma \triangleright \text{this.m(e)} : t \in \Gamma''$ |
| T-SELFCALL2 | $\Gamma \vdash e : t' < \Gamma'$ $\Gamma' \triangleright this : C[u; \tilde{F}]$ $\Gamma' \vdash (t m(t' x) \{s\}, 0) \in C.$ methods $\text{un}(\Gamma'' \setminus \Gamma)$ | $\Gamma \triangleright \text{this.m(e)} : t \in \Gamma''$ |
| T-CALL | $\Gamma \triangleright w : C[\tilde{F}]$ $\Gamma \triangleright \text{allows}(m) = u'$ $\Gamma \vdash (t m(t' x) \{s\}, \_ \_ \_ \_ \_ \gamma) \in C.$ methods $\text{un}(\Gamma'' \setminus \Gamma)$ | $\Gamma \triangleright w.m(e) : t \in \Gamma'' \{w \rightarrow C[u'; \tilde{G}]\}$ |

\[ w \neq \text{this} \]

\[ u.\text{allows}(m) = u' \]

\[ t m(t' x) \{s\}, \_ \_ \_ \_ \_ \gamma \in C.$ methods \]