University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Jarosław Dominik Kuśmierek

A Mixin Based Object-Oriented Calculus: True Modularity in Object-Oriented Programming

PhD dissertation

Supervisors

prof. dr. Viviana Bono
Department of Informatics
Torino University

prof. dr. hab. Paweł Urzyczyn
Institute of Informatics
University of Warsaw

April 2010

Author's declaration:
Aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

April 20, 2010	....
date	*Jarosław Dominik Kuśmierek*

Supervisors' declaration:
the dissertation is ready to be reviewed

April 20, 2010	....
date	*prof. dr. Viviana Bono*
April 20, 2010	....
date	*prof. dr. hab. Paweł Urzyczyn*

Abstract

Nowadays, most of the commonly used imperative programming languages fall into the category named object-oriented languages (or OO shortly). The domination of the OO languages is especially visible in the area of commercial projects. OO languages have evolved during last 20 years and, while becoming an industry standard, reached a mature state.

However, the mainstream OO languages (like Java [39], C♯ [40] and C++ [63]) still show some common limitations. The main problems one can meet using these languages are: (*i*) lack of expressiveness requiring additional programming during the declaration of new classes; (*ii*) limitations of possible reuse scenarios caused by the single inheritance mechanism (and the complications of semantics of the multiple inheritance); (*iii*) danger of random conflicts of class members.

When programmers develop bigger and bigger products (especially the ones combining code coming from different providers) these limitations are becoming more and more visible. Additionally, while some tools are being designed to fight these difficulties, those tools are becoming very cumbersome. We believe that most of programmers using C++ [63], Java [39] and C♯ [40] have difficulties with the understanding of some subtleties of the semantics of the language they use. As an effect, it implies more errors and less effective use of the language features.

To solve those problems, we present a new OO, mixin-based language called Magda, which is more expressive with respect to reuse than the above mentioned languages, by allowing much stronger and more flexible reuse of existing components. At the same time, the language is safer, because it completely protects the programmer from conflicts of class members, during the design and writing of the software, as well as during its further evolution.

To show that this language is safe, we formally define its semantics and type system, and then we prove its soundness.

Contents

List of Figures

Chapter 1

Introduction

Many software systems produced nowadays are implemented using the OO programming languages. Those languages have numerous advantages, which makes them so successful. However, even though many of them include sophisticated constructs (thus having also a complicated semantics), they still have some limitations, which restrict the modularity and composability of different components[1] of the program. In the rest of the thesis we assume that the reader is familiar with OO programming.

1.1 Problems we are concerned with

The limitations which we are particularly concerned with, and which have been the motivation for the design of our new language, are the following:

Non modular initialization protocol. In most OO languages newly created objects are initialized according to the specification present in constructors. Constructors are responsible for initialization of many different features coming from different places in the class hierarchy, yet they are monolithic, which means that they have to perform all this job in one block of code. While a constructor can call a super class' constructor to delegate some parts of the work, it still needs to keep all its parameters in its own signature. This increases the amount of work which has to be performed when the initialization process needs to be modified. Additionally, by creating unnecessary dependencies, it limits the possible combinations of otherwise independent parts of the code.

Clashes of method names. When classes are composed from different components, like in cases when they inherit from other classes, or implement some interfaces, it may occur that the same method name has different meanings in the components we want to use. This causes different kinds of problems, from non-compilation to an unexpected behavior during the program execution.

[1]We do not assume any specific meaning of "component". We use this notion to refer to any entity being a reusable part of the program.

Limitations of composition mechanisms. The most widely applied reuse and composition mechanism is inheritance. In its classical (single inheritance) setting this mechanism is simple and understood by many people, however often not sufficient. On the other hand, there were other solutions developed, like multiple inheritance, mixins [56, 22, 11, 36] and traits [60, 33] (in the order of appearance). The first of those is widely percepted as too complicated and too dangerous [30]. The two other solutions (mixins and traits) have been designed to solve this problem and managed to improve the reusability in many aspects. However, in their current implementations, some conflicts and non-predicted behavior can still occur.

All those limitations together with their consequences in different languages are described in more detail in Chapter 2.

1.2 General idea of our solutions

In order to solve the problems mentioned above, we have designed and implemented a new language called Magda. The core notion in Magda is the *mixin*, which defines a building block from which objects are created. Our mixin construct has many things in common with its previous revisions [56, 22, 11, 36] with one noticeable difference: In Magda there is no concept of class, so mixins are not interpreted as functions from classes to classes. Mixins are first-class entities, which are used to create new objects from, and also induce types in the nominal static type system of our language.

Additionally, to completely take advantage of mixin reusability, Magda contains two additional unique features. Thanks to those features, Magda avoids the problems with mixins, which we introduced in Section 1.1 (and studied in more details later).

The first of those features is the modularization of constructors. This approach allows one to define the initialization process of objects in a modular fashion, in such a way that part of the constructors which is responsible for the initialization of the state related to some feature is declared together with that feature. Then many mixins with independent definitions of constructors can be combined without the need to copying any code and without the risk of any clashes.

The second distinctive feature of Magda is the way of uniquely declaring and referencing identifiers. This approach modifies the way declarations of new methods, overriding of existing methods, and method calls are specified. Those modifications result in the property that no combinations of methods coming from different mixins and called by other mixins will cause any name clashes, accidental overridings or any other problems of that kind.

In this thesis, we present the syntax and the core features of Magda together with its semantics and formal properties. In order to keep this presentation as simple as possible we have focused on a subset of the language which contains most of the unique features, while avoiding features which are common to many other languages and are mostly orthogonal to other features. From now on, unless stated otherwise, when we use the word Magda, we mean this selected core subset of Magda.

1.3 Contributions of the thesis

The main contributions of this thesis are:

1. The introduction of a newly designed purely mixin-based language, which integrates the following features:

    - *Modular constructors approach*, which introduces the notion of *initialization module*, that corresponds to a piece of a constructor in classical OO languages. This approach allows arbitrary compositions of initialization modules coming from different mixins without the risk of accidental conflicts. A limited version of this feature have been previously published as a part of a Java extension called JavaMIP [18, 17, 55].
    - *Hygienic identification* of methods and fields, which guarantees that no clashes will occur when multiple mixins are combined. A version of this feature have been previously published as a foundation of the HygJava language [48].
    - Clean, purely mixin-based design. This design, thanks to its features, does not suffer from the ambiguity problems perceived as those which stop mixins from being more widely popularized (see the description of those problems in Section 2.4.4).

2. The formal semantics and the type system of the language.

3. A new approach of proving the type soundness of a language. This approach allows one to prove the strong type soundness property basing solely on the big-step semantics (where normally it requires a more tedious small-step formalization).

1.4 Outline of the thesis

This thesis is structured as follows. In Chapter 2 we discuss the problems present in existing OO languages, which we tried to solve. In Chapter 3 we introduce Magda by presenting different examples of code — starting from the simple “hello world” example, and then continuing with more sophisticated ones. In Chapter 4 we present in detail all syntactic constructs of Magda. In Chapter 5 we present a formal BNF definition of Magda. In Chapter 6 we introduce some definitions which formalize the contents of each program and allow us to formally define different properties of Magda. In Chapter 7 we present the formal big-step semantics of our language. In Chapter 8 we define the type checking system of Magda. In Chapter 9 we present a formal model of the computation steps performed by the programs written in Magda. This model relies heavily on the big-step semantics and allows us to state properties referring to the type soundness and subject reduction. In Chapter 10 we present a formulation and a proof of the subject reduction theorem for Magda. This proof uses the model of computation in order to obtain a stronger subject reduction property than the usual one when big-step semantics is exploited. In Chapter 11 we formulate and prove the type soundness property of Magda's semantics and type checking system. In Chapter 12 we briefly describe the actual implementation of Magda. In Chapter 13 we compare Magda's features to those available in other languages. In Chapter 14 we conclude our work.

Chapter 2

Weak points in the existing OO languages

Below we present an overview of three kinds of problems from which most of the current OO languages suffer. The problems listed motivated us to design from scratch the new language called Magda.

2.1 Non-modular initialization protocol

Most OO class-based languages are equipped with some form of specifications for the object initialization protocol. This protocol describes two aspects of the initialization:

- The kind of information that must be supplied to a class, to create and initialize an object. A class may support more than one variant of object initialization, which means that there may be more than one accepted set of such information;
- The code to be executed during this initialization. The sequence of instructions which should be executed depends on the kind of information supplied. Therefore, if the class supports distinct sets of information to be supplied during initialization, then for any such a set a different sequence of instructions should be executed.

Usually, the initialization protocol of a class is specified by a list of *constructors*. Each constructor corresponds to one accepted set of information and consists of:

- a list of parameters (names and types), specifying a set of information required to initialize an object;
- a body, containing a list of instructions which must be executed in order to initialize an object.

Such approach is present in most of the contemporary OO languages like, for instance, C++ [63], C♯ [40], Delphi [3], Java [39], and Visual Basic [2].

Unfortunately, this traditional initialization protocol (abbreviated TIP from now on) has some drawbacks, studied in details below. The main limitation of such an approach and the source of all the mentioned problems is the following: Every time a programmer wants to reference an initialization protocol of a class, he/she must define or reference explicitly the whole list of parameters, or, in the case of inheritance, sometimes even repeat the whole list of constructors. However, in many cases his/her actual intent may be different. The actual idea behind the code might be much simpler than the actual work which has to be done in the code, which causes various flaws. Below we present different scenarios in which TIP requires unnecessary additional code and creates additional dependencies. We perform the analysis by presenting some Java examples, however, most of the mentioned problems occur also in any other mainstream object oriented language.

2.1.1 Optional parameters

Unfortunately, traditional initialization protocol leads to an exponential number of constructors with respect to the number of optional parameters.

In Java, in classes like `java.awt.TextArea`, there is often one "complete" constructor declared, containing the largest set of parameters. However, when some of those parameters are optional, then additional constructors, with less parameters than the "complete" one, must be declared.

Moreover, most of the time, all desired constructors cannot be introduced. It happens so because it is not allowed to place, in one class, two constructors with parameter lists of the same length and compatible types of the corresponding parameters. Even though such two constructors can represent two different subsets of parameters of the complete one, the compiler will not recognize which constructor to call at object creation time (because the constructor is chosen via the types of its arguments). However, if the compiler could make such a choice, then most probably a lot of classes would contain all possible constructors, leading to an exponential number of constructors. This can be partially solved in languages containing named parameters (like Python [13]) and default parameter values (like Delphi [3], Python [13], C++ [63] and others).

2.1.2 Multiple initialization options

TIP leads also to an exponential number of constructors with respect to the number of object properties with different initialization options.

If a class contains some properties, where each of them can be initialized in more than one way, then the possible number of initialization options of a given class (thus the number of constructors) is a multiplication of the numbers of options of object properties. An example could be the combination of a property color (with two options, RGB and CMYK) with a property position (with three options, cartesian, polar, and complex) in a class `ColorPoint`, which induces six constructors (see Figure 2.1). The Java class `java.net.Socket` is a more sophisticated example, which due to this and the previous described problem contains presently nine constructors.

```
// Class of points definable by three different coordinate systems
class Point
{ float x, y; //object state variables

  Point (float x, float y)
  { this.x = x; this.y = y; }

  Point (Complex comp)
  { x = comp.x;  y = comp.y;  }

  //the third parameter is required only for the compiler
  //to distinguish between this one and the (x,y) constructor
  Point (float angle, float rad, boolean PolarDef)
  { x:=cos(angle)*rad; y:=sin(angle)*rad; }
}

// Class of colored points whose color is definable by two different
// color palettes
class ColorPoint extends Point
{ float r, g, b; //object state variable

  // There are the constructors (for three different coord. systems) with RGB
  ColorPoint (float x, float y, float r, float g, float b)
  { super(x,y);
    this.r = r; this.g = g; this.b = b;}
  ColorPoint (Complex comp, float r, float g, float b)
  { super(comp);
    this.r = r; this.g = g; this.b = b;}
  ColorPoint (float angle, float rad, boolean PolarDef,
              float r, float g, float b)
  { super(angle, rad, PolarDef);
    this.r = r; this.g = g; this.b = b;}

  // While there are the constructors with CMYK arguments.
  ColorPoint (float x, float y, float c, float m, float yc, float k)
  { super(x,y);
    r = somefun1(c,m,yc,k); g = somefun2(c,m,yc,k); b = somefun3(c,m,yc,k);}
  ColorPoint (Complex comp, float c, float m, float yc, float k)
  { super(comp);
    r = somefun1(c,m,yc,k); g = somefun2(c,m,yc,k); b = somefun3(c,m,yc,k);}
  ColorPoint (float angle, float rad, boolean PolarDef,
              float c, float m, float yc, float k)
  { super(angle, rad, PolarDef);
    r = somefun1(c,m,yc,k); g = somefun2(c,m,yc,k); b = somefun3(c,m,yc,k);}
}
```

Figure 2.1: `Point` and `ColorPoint` Java classes — presenting exponential growth of the number of constructors

2.1.3 Code duplication

Further problem with TIP is that excessive amount of constructors (as described in Section 2.1.1 and 2.1.2) very often contain duplicated code. To see what parts of code get duplicated, let us consider an example in which we have two attributes characterizing the state of an object, for example the mentioned position and color from a class `ColorPoint` (see Figure 2.1). If those attributes can be initialized in multiple ways (the position can by supplied by three different coordinate systems and the color by two different palettes), then we would need six constructors where most of pairs of those constructors share some common code. An example of such code duplication can be seen in the constructor bodies of `ColorPoint` class on Figure 2.1.

2.1.4 Work consuming subclassing

TIP makes the extension of a class by a new subclass cumbersome. In most cases, the designer of a subclass just wants to modify the initialization protocol of a parent class, not to redefine the parent's protocol completely. However, using the TIP approach, he/she must declare the whole resulting protocol in a subclass. Let us consider, for example, a class of blinking buttons, which blink for some time after clicking. It is possible to declare it by extending the class of ordinary buttons, for example `javax.swing.JButton`, which has five constructors. Some information is needed for such a button, for instance the frequency of blinking and the time for which the button must blink. Therefore, the subclass must contain five constructors, as the parent class. Additionally, the declaration of each of those constructors will begin with the identical parameters of the corresponding constructor in the parent class, but it will contain two additional parameters, time and frequency. This may lead also to code duplication.

2.1.5 Unnecessary constructor dependencies

In languages using the traditional initialization protocol, modifications of existing classes force unnecessary changes in subclasses. To present this problem in more detail, let us consider such a scenario:

- there exists a class C_1 with some set of constructors;
- another class C_2 is declared by another developer as a subclass of C_1. Class C_2 needs some additional initialization information, so it has all parent constructors redeclared by adding one parameter *Par'*, which is needed by subclass C_2. Then, all those constructors will most probably call the corresponding constructors in the base class C_1 and execute some sequence of code regarding *Par'*;
- another constructor is added to C_1. Unfortunately, the class C_2 will not inherit automatically the corresponding constructor. This class, depending on the language design, will either retain the constructor added in C_1 without the additional parameter *Par'*,

or just will not inherit it at all (as it is in case of Java). We think that neither of these options is good enough.

2.1.6 Fragile overloaded constructors

Additionally, overloaded constructors in TIP make safe-looking changes non-conservative. When a Java (or C++ or C♯) class contains many different options of initialization implemented as many different constructors, the choice of the constructor actually called during object creation is done in the same way as the choice of an overloaded method variant. Thus, it suffers the same problems, as this example shows:

```
interface I1 {...}
interface I2 {...}
class C1 implements I1, I2 {...}
class C2 implements I2 {...}

class ClassWithOptions
{   ClassWithOptions (I1 a, I2 b);
    ClassWithOptions (I2 a, I1 b);
}
 ...
 new ClassWithOptions( new C1(), new C2() );
```

This code will compile, because only one of constructors of class `ClassWithOptions` matches the last `new` expression. But if the class `C2` is enriched in order to make it implement also the interface `I1`, then this "safe-looking" change will make the previous code not working (because the last `new` will be ambiguous). Notice that this is not a problem in languages in which it is possible to name constructors (e.g., in Delphi [3]), therefore the overloading can be avoided.

2.1.7 Unnecessary redeclarations of checked exceptions

In Java, if a constructor of a parent class is declared to throw some exceptions and is called by a constructor of a subclass, the subclass constructor must repeat the whole list of exception declarations. While the repetition of such declarations in normal methods is important, because methods have choices (to catch the exceptions, or throw them further), the situation with constructors is different. A constructor cannot catch the exceptions thrown be a superclass' constructors, therefore the repetition of the declarations is an additional work for the programmer which cannot be justified.

Additionally, when a superclass constructor is refined by replacing one exception with two sub-exceptions of the original one (which is a conservative, thus a safe, extension), the subclasses will still contain the information about the original exception. As a result, clients of the subclass will not get as complete type information about the thrown exceptions as they could.

2.1.8 Problems with traditional mixins

A mixin is a class parameterized over a superclass, that was introduced to model some forms of multiple inheritance and improve code modularization and reusability, [56, 22, 20, 11, 36, 10, 15]. There are usually two operations we can perform over mixins: (*i*) application, which applies a mixin to a class to obtain a fully-fledged subclass (the class argument plays the role of the parent); (*ii*) composition, which makes a more specialized mixin by composing two existing ones. Notice that an indirect form of composition is possible even in the presence of application only, via the application of a chain of mixins to a class (which is the way a linearized multiple inheritance is obtained). A mixin declaration, like any other subclass declaration, may contain declarations of new constructors[1], which, in turn, may reference the superclass constructors.

There are cases in which it would be desirable to compose independently designed mixins. Let us consider the following example: A class `Button` and two mixins `Blinking` and `Ringing`, that when applied separately to `Button` will result in, respectively, a class of blinking buttons and a class of ringing buttons. Then, it would be good to be able to compose them (the order should not matter), in order to obtain a class of blinking and ringing buttons. Unfortunately, if those mixins modify the interface of an initialization protocol (that is, the parameters of a constructor) of the parent class (e.g., `Button`), then they are non-composable.

To better understand why, let us detail our example. Let us assume that `Button` class has a constructor with x parameters. Let us assume also that a properly initialized object of the class `BlinkingButton` must know its blinking color. Therefore, in the TIP approach, the mixin `Blinking` must define a new constructor (that replaces the old one), having $x + 1$ parameters (the additional one is the color) and calling the superclass constructor inside by passing the other x parameters. Similarly, the `Ringing` mixin may need, for example, some information about the frequency of the sound, therefore it will have an $x + 1$-parameter constructor as well, containing an additional frequency parameter and calling the superclass constructor inside by passing the other x parameters. Now, if we apply one of those mixins to the `Button` class, then the resulting class will have a constructor with $x + 1$ parameters, and it will not be an appropriate argument for the other mixin.

Those problems have, in fact, a similar nature as those occurring with classical subclassing. However, mixins are designed for a wider reuse than subclasses, therefore those problems may occur more often and are more difficult to foresee. Notice that also the designers of Jam [10] have noticed this problem. In order to simplify the matter, they decided to disallow the declarations of constructors in mixins, thus forcing programmers to write constructors manually in all classes resulting from a mixin application.

2.2 Risk of accidental name clashes in different scenarios

The base object-oriented concept of "method" is realized by three different actions:

(*i*) the introduction of a new method;

[1]In fact, many mixin-related proposals do not allow any form of constructors (see for example [10]).

(ii) the implementation/override of an existing method;

(iii) the method call.

The bindings between (*ii*) and (*i*), as well as between (*iii*) and (*i*) are typically made using a method name, which is not guaranteed to be unique, thus such bindings might cause some ambiguities. Additionally, in many popular languages (like Java, C♯ and C++), the distinction between (*i*) and (*ii*) is also based upon names.

Therefore, modifications of existing classes (even modifications designed as conservative extensions of some functionality) may cause errors in some other parts of the code referencing such classes. Such problems can occur even more frequently, and are more difficult to predict, when the modification of a class is performed in a library written by some third party developer, and used by other parties. In general, a programmer cannot predict the moment in the execution when such ambiguities may occur.

Moreover, in languages containing the *mixin* construct (first introduced in a dynamically type checked language called Flavors [56], then developed in a statically typed language by Gilad Bracha and others — see [22, 20, 36, 19, 10, 15]), the set of allowed combinations of modules is much bigger, thus all these ambiguity problems are more probable to occur.

Recall that a mixin is a subclass parameterized with respect to a superclass, and mixin inheritance is obtained by applying a mixin (or a "chain" of mixins) to a class. A mixin can introduce new methods, request some methods to be supplied by its superclass, and override some of the superclass methods as well. Therefore, during mixin application, name clashes can occur. Problems regarding name clashes during mixin application has already been pointed out in the literature, e.g. by Schärli et al., in their work on traits [60], followed by Ducasse et al. in [33] and also by the designers of MixedJava language [36], as well as Eric Allen et al. in his work on first class genericity and the MixGen language [6].

In this section we present three kinds of ambiguity problems which can occur when programming in a Java-like language, and which we have hopefully solved when designing the Magda language. The first two of these problems occur within the Java language (and also, with some differences, in other languages, see Section 13), while the last one occurs only in statically-typed languages containing the mixin construct.

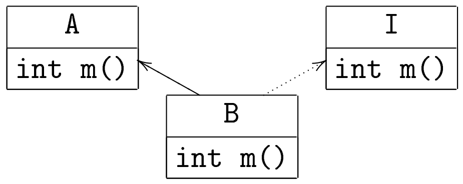

Figure 2.2: Name clash during interface implementation

2.2.1 Name clash caused by the implementation of an interface

Let us assume that class `A` and interface `I` are defined independently in different libraries (see Figure 2.2). Assume also that both of these contain a declaration of the method `m`.

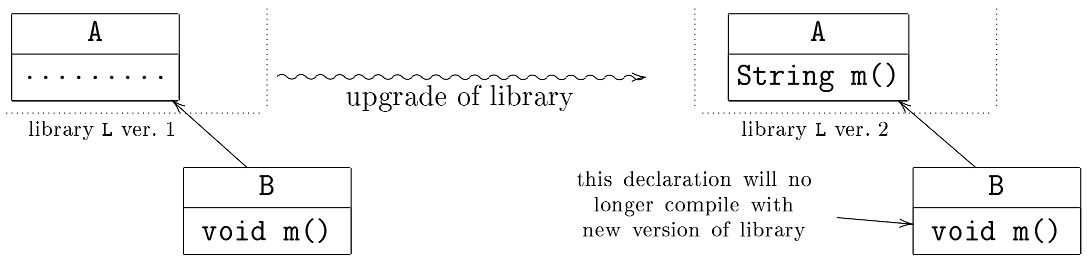

Figure 2.3: Upgrade of the library causing the code not to compile

Now, let us imagine that a developer needs to create class `B` as a subclass of class `A`, and also as an implementation of the interface `I`. Then it might happen that, in order to match the interface `I`, the implementation of method `m` must be completely different from the one inherited from `A`. Therefore, the change of the implementation of method `m` in order to have the behavior expected by `I` might make the functionalities inherited from class `A` behave unexpectedly. When class `A` also has a few ancestors, it might be the case that method `m` was introduced in an ancestor, and the developer is not aware of the existence of method `m` in class `A`. Then this override can even be unnoticed.

On the other hand, if the result types of the methods introduced in `A` and interface `I` are incompatible, then the class `B` will not even compile.

2.2.2 Name clash caused by the addition of a new method

Let us assume that there exists a library `L` containing a class `A` (see Figure 2.3). Assume also that there exists a class `B` created by a different developer as a subclass of class `A`, containing a declaration of the method `m`. Additionally, let us assume that the developer of library `L` knows nothing about class `B`. Now let us assume that the developer of `L` decides to modify the functionality of `A` by adding the method `m` (for example, by implementing a refactoring-based method extraction supported by a tool), and referencing it from existing methods. Then, unfortunately, class `B` used with the new version of `L` can suffer from two kinds of problems:

- If the result type of method `m` in `B` is not compatible with the one declared in `A`, then `B` will not compile anymore.

- If the result type is compatible, then method `m` in `B` will unexpectedly override method `m` from `A`, changing the behavior of the class `B` in a potentially undesired way. Consider a more detailed example, where class `A` is upgraded in the way presented on Figure 2.4 and class `B` is declared as in the Figure 2.3.

  Here, the newly added method `m` is referenced from the existing method `oldmet`. Therefore, an accidental overriding of `m` will not only make class `B` not have the functionality expected from `m`, but also will change the behavior of another method (`oldmet`, in this

```
class A
{ void oldmet()
  { $\overrightarrow{I_1}$;
    if (...)
    { $\overrightarrow{I_2}$;}
  }
}
```

$\underset{\text{upgrade of library}}{\rightsquigarrow}$

```
class A
{ void oldmet()
  { $\overrightarrow{I_1}$;
    if (...)
      m();
  }
  void m()
  { $\overrightarrow{I_2}$; }
}
```

Figure 2.4: Upgrade of the library causing accidental override

case). Moreover, this dependency between `m` and `oldmet` is not visible in the external interface of any class.

One common situation in which those problems may occur is the one when a core system is sold to many customers and modified on the customer site, as well as upgraded during its lifetime.

Such problems are, in fact, the result of conflicting specifications of newly added and inherited implementations of method `m`. Therefore, those can be checked (dynamically or statically), if the specifications are formally defined as assertions and verified. This can be done, for example, in the Java environment with the use of the tool *JML* [25], or in Eiffel via the *Design by Contract* [52]. Both warn the programmer with a "specification-not-fulfilled" error, Eiffel at run time, JML both at verification time and at run time. Such approach allows the detection of conflicts that must be fixed in order to make the program work properly (and therefore eliminate the warnings). However still, a safe looking change in one library might enforce many changes in other libraries. This can be problematic when such method (like `m` in class `B` from Figure 2.3) is used in many places.

2.2.3 Name clash caused by mixin application

Let us assume that there exist two independently developed mixins `M` and `N`, both adding method $\mathtt{m}_1$ with the same name and types of parameters and result. Assume also that there exists class `A` to which both of these mixins can be applied. Next, if we will build the class `M(N(A))`, then, depending on the implementation of mixins in the language, we will have either:

1. a conflict raised by the compiler, or
2. a class where an implementation from `M` overrides the one from `N` (such an approach is the one of Jam, see [10]), or
3. a class with both methods available, but only with one of them at a certain moment, depending on the context, given by the type of an object expression. If the variable has

a type containing `M`, then method from that mixin will be accessible, and analogously for `N`. Such approach can be found in the MixedJava language [36] and MixGen [6].

We think that the third solution is the best of the above, however it is still not completely satisfactory, because: (*i*) in contexts where the receiving object expression has both types `M` and `N`, the choice of the method is still ambiguous; (*ii*) in some situations a programmer might need the access to both methods in one single place.

Also designers of traits [60] present this "accidental override" problem as the main drawback of mixin approach (see [33] for the detailed motivation). In order to solve this, they invented traits. Traits are similar to mixins in the sense that they are reusable sets of methods. However, traits are more loosely coupled than mixins, and instead of being directly used as mixins to obtain a new class from existing ones, traits are combined with existing classes using additional "glue code" which specifies which method of the given trait overrides which method in class. A deeper analysis of traits approach is present in Section 2.4.5 and some comparison to our approach in Section 13.3.3.

2.2.4 Name clashes of field declarations

Note that some of the described problems can also occur with regard to public and protected field declarations. However, fields are less often declared with a public visibility, and cause ambiguity problems less often. Additionally, problems with fields are generally easier to solve (because fields cannot be redefined), hence these can be solved with basically the same techniques exploited to solve ambiguity problems concerning methods. On the other hand, our solution to this problem is general enough to cover also ambiguities of field references.

2.3 Problems in the Java base library

In order to give some evidence for the relevance of the ambiguity-caused problems, we present an analysis of the source code of Java APIs (ver. 1.5). This analysis shows name conflicts that might lead to the problems described in the previous section, when using Java's API. The Java API's of course compile and work; thus, the ambiguity problems which possibly occurred during the development of the Java API's themselves were dealt with traditional techniques, such as the renaming of some methods and by discarding some changes, which would be unnecessary if Java had hygienic identifiers (see Sections 1.3 and 3.1). However, still different combinations of Java's API classes and interfaces can cause the mentioned problems.

First, on Figure 2.5, we show some numbers representing the occurrences of the introduction of the same method name in different classes and interfaces (notice that we did not count overridden definitions, or implementations of methods declared in the interfaces). Then we present some simple but representative examples to underline the nature of the problems caused by the lack of hygiene in Java API.

Below we present three examples of possible problems deriving from the presence of the ambiguities described above:

method	occurrences in		
	interf.	classes	total
getName()	59	148	207
getType()	36	71	107
close()	30	38	68
getLength()	28	33	61
getValue()	24	45	69
item(int)	19	12	31
getId()	20	32	52
reset()	14	95	109

method	occurrences in		
	interf.	classes	total
setName(String)	15	7	22
getAttributes()	13	33	46
remove(int)	6	19	25
setValue(String)	9	2	11
setType()	10	1	11
getWidth()	13	17	30
clear()	8	61	69
isEmpty()	8	32	40

Figure 2.5: List of names of methods used repeatedly in different contexts

- The `Set` interface contains the method `isEmpty` with the obvious meaning. The class `Hashtable` (which is a class of partial function objects that assign a value to a value) contains a method `isEmpty` which checks if the dictionary contains any assignment. Unfortunately, if one wants to implement a class of subsets of some set X as a characteristic function of this subset (which means function from X to *bool*), by subclassing the `Hashtable` class, then it cannot be done. This happens, because the meaning of `isEmpty` from the point of view of `Set` must mean that the `Dictionary` assigns false to everything, which does not mean that the Dictionary itself is empty.

- The `Map` interface (which represents the concept of a partial mapping or function from one set to another) contains a method `clear`, which empties the mapping. The typical graphical component `List` (available in the `java.awt` package) represents a list of items, and contains a method `clear`, which makes this list empty. Now, let us assume we want to represent a mapping from some small domain, such that all of its elements fit on the screen. In other words we want to write a visual component which implements the interface `Map`. Assume also that this component lists all the mappings of the form `X -> Y` for each value `X` in our domain and displays `X -> ?` if no value is assigned to `X` in this particular mapping. We might then choose to make our component a subclass of `List`. Then, the meaning of the method `clear` which will behave accordingly to the "contract" of interface `Map` should not delete all the items on the visible list, but just replace them with `X -> ?`. This, unfortunately, is completely incompatible with the notion of clearing the visual list.

- The `java.sql.Connection` interface, which represents the concept of connection to an SQL database, contains a method `close`. Similarly, the class `java.net.Socket`, representing network communication sockets, contains a method `close`. Now, let us assume that someone would like to create a subclass of class `Socket` representing a socket designed especially for communication with some specific SQL database. Then we might want to implement the interface `Connection`. Unfortunately, closing the actual logical connection with the database does not have to imply closing the physical connection with the database. This might happen, when an application wants to keep a pool of open physical connections, which can be used at any time when the logical communication with the database is needed, to make the connection quicker. This can also happen when closing of logical connection requires some additional operations like asynchronous flushing of some internal cache into the physical connection, and the physical connection itself will be closed after those operations finish.

An alternative to inheritance which can be used to solve such problems is to keep the class containing the desired implementation as an internal component, and in some cases it might be a better design decision. However, it is bad if the lack of hygiene itself limits the possible uses of the inheritance mechanism in a language.

2.4 Limitations of reuse mechanisms

Another problem with existing OO languages lies in their ability to decompose components in small, reusable pieces and then compose them arbitrarily when creating a class. This problem manifests itself in various ways depending on the way new classes are build in a given language, however in each language it still persists in some form.

We will present what form this problem takes in many languages using a simple example consisting of several classes. Those classes represent the hierarchy of different variants of streams, namely:

- `BaseStream` - an abstract class (or interface as it is called in other languages), which contains abstract methods `read` and `write`.
- `FileStream` - a class representing streams which read and write contents to and from file.
- `NetworkStream` - a class representing streams which read and write the data across the network.
- `BufferedStream` - a class representing streams which buffer data before sending them in a one big batch.
- `CompressedStream` - a class representing streams which compress and decompress data on the fly.
- `StatsStream` - a class representing streams which calculate different statistics like: total amount of data read and written, maximal and average throughput, etc.
- `EncryptedStream` - a class representing streams which encrypt data when those are written and decrypt during reading.
- `DatabaseStream` - a class representing streams, which write and read data from some specific table in some database.

Let us assume that class `BufferedStream` has a method `SetBufferSize`, which sets the amount of data which is kept in memory before being sent to the underlying communication device. At the same time, class `CompressedStream` uses the method named also `SetBufferSize` to decide about the amount of data which is compressed at the time (the bigger this number, the more CPU-intensive the process is, however it also increases the compression ratio).

Additionally, we expect to be able to obtain many combinations of the above features, like: compressed and encrypted network stream, or buffered file stream with statistics enabled. Unfortunately, as we discuss it below, this is difficult to achieve with languages used currently.

2.4.1 Simple inheritance

In simple inheritance languages, every class can have no more than one ancestor. As a result it can reuse only one set of features. When we have a case when some class needs features declared in two or more distinct classes, then the only way is to perform the following operation: inherit from the class which contains the most of what is needed and then obtain the rest of needed functionality by either:

- copying manually the code from the remaining classes. Thus, in our example, we would have `FileStream` and `NetworkStream` as classes inheriting directly from `BaseStream`. Then `CompressedFileStream` would be declared as a subclass of `FileStream`. However, to obtain `CompressedNetworkStream` we would need to inherit from the class `NetworkStream` and copy all parts of the code specified in `CompressedFileStream`;

- using object-composition, by declaring local fields of the types of remaining classes and then implementing remaining methods by delegating them to appropriate delegate objects. This approach can however cause additional problems, when all those classes we need to reuse share some common ancestor with some state. Then every change of the common state of the main object needs to be propagated to the delegate objects also. This results in behavior which is in many aspects similar to the virtual multiple inheritance in C++ (see Section 2.4.3).

2.4.2 Object composition/Decorator design pattern

As mentioned above, one of the mechanisms used to modularize the code is object composition. One of the approaches which heavily exploit the object composition is the one defined as *Decorator pattern* by Gamma et al. in their book on Design Patterns [37]. In this approach, when programmer needs to create an `EncryptedStream`, he or she creates a new class, which contains a reference to another stream object (called delegate). Then, this class has declared all the methods of the `BaseStream` class. In case of methods whose behavior needs to be modified (like `read` and `write` in stream hierarchy mentioned above), they perform their tasks and then call the corresponding methods in the delegate. All other methods are implemented to just call their counterparts in the delegate object.

This approach very often allows one to obtain the required compositional flexibility, and is sometimes preferred over inheritance. It is especially useful when additional features of objects should be enabled and disabled dynamically during the life of an object. However it has numerous disadvantages when used instead of inheritance:

- Requires declarations of many methods which will only repeat the same method calls to the delegate object, which means more coding.
- Creates more unwanted dependencies in the code, because now any modification or addition of a method in the delegate object's class requires repetition of the same operation in other classes.
- When a class `A` is composed with a class `B` and redefines some of the methods from `B` by providing a new implementation in its declaration (what corresponds to method overriding in inheritance), then this form of overriding is visible only from the point of view of external clients. Unfortunately, other methods in the same object calling this method will call the original implementation and not see the overridden one. Therefore it works the same way as the inheritance mechanism used without the virtual method construct.

  The virtual method construct can be simulated in this pattern by passing the decorator object `A` as the additional parameter to all the methods in `B` and using `A` as a target of all the method calls performed by `B` (rather than the decorated object `B`). However it makes the code more complicated and requires modifications in every decorated class in order to allow composition with decorators.

2.4.3 Multiple inheritance

Multiple inheritance, whose most widely known implementation is the one in C++, is a much more powerful feature, however its complicated semantics make it difficult and dangerous to use on a wider scale, as summarized by Cook [30]:

> "Multiple inheritance is good, but there is no good way to do it."

The main problem people find with multiple inheritance is the way it deals with ambiguous and conflicting features, which happens when a class inherits from superclasses which have fields or methods with the same name. In the example mentioned on the beginning of Section 2.4, such problems will occur in a class inheriting from both `CompressedStream` and `BufferedStream`, because both of those classes have method `SetBufferSize`. Multiple inheritance has a complicated semantics and still cannot offer satisfying results in some cases. In particular, it does not allow the user to keep in the resulting class both of the methods with the same name. Moreover, when the same method is defined or overridden in multiple superclasses which share a common ancestor, then the actual order in which overridden variants of the method are called is not always obvious.

There exists a few other languages which adopted different flavors of multiple inheritance, like Loglan [45] and Python [13]. Those languages use linearization algorithms to change the graph of ancestors into a list over which dynamic dispatch is performed. However, every such linearization leads to cases when the ordering of the feature overriding is: (*i*) non-trivial to understand; (*ii*) fragile to innocent-looking changes in the hierarchy.

As a result of all the mentioned problems, many mainstream languages developed after C++ (like Java and C♯), borrowed numerous features of that language, however not the multiple inheritance.

2.4.4 Mixin inheritance

Mixins [56, 22, 11, 36] allow the program to decompose features into many pieces, which can then be composed in different ways to obtain required combinations of features. Thanks to the linear composition used to obtain the resulting class, the mixin inheritance avoids the problem of the ambiguous multiple inheritance semantics.

However, in the case of many methods of the same name occurring in different mixins (like the aforementioned `SetBufferSize`), even though mixins allow direct control of the order in which those methods will override each other, they might not allow the resulting class to have both variants of the inherited methods with the same name. The approach which allows the class to have both variants is presented in the work on MixGen by Eric Allen et al. [6] and in the MixedJava language [36]. In the MixGen language, the class obtained from applying both `CompressedStream` and `BufferedStream` mixins will keep both of those methods, while giving access to one of them at one moment, depending on the static type of the variable keeping the actual object. However, this approach does not allow the user to access both of them at the same time and in some cases it is not obvious for the user which implementation will be called in the given expression. We have analyzed this problem in more detail in Section 13.2.8.

In general this problem, occurring also in classical inheritance but especially often in the presence of mixins, is called “fragile class hierarchies”, as pointed also by Ducasse et al. [33, 60]. The actual problem can be summarized as follows: The addition of a new method in some superclass or some mixin might accidentally break other subclasses or classes obtained using that mixin because of name conflicts. And since classes obtained from a mixin or a superclass can be in fact present in the source code of some other party reusing the original superclass, this problem cannot be foreseen at the moment of new method declaration. More detailed description of the problems which can occur in such cases is given in Section 2.2.3.

2.4.5 Traits

Traits [60, 33] have been designed as an alternative mechanism of code reuse. Their first design as well as the first implementation has been developed to extend a Smalltalk dialect called Squeak. One of the design goals of this approach was to overcome problems with mixins mentioned above. Each trait is a minimal unit of reuse, and it:

- contains some set of methods,
- requires another set of methods to be supplied, in order to work properly. The methods required by some trait are those which are used inside the methods added by the trait, however not contained within this trait.

Then, a class can be build basing on some existing class, by composing traits. However, in contrast to the mixins approach, when a programmer uses a trait to create a new class, he/she can use an additional operator to arbitrarily choose which methods from this trait he or she wants to add to this newly created class (by hiding some of them). Additionally the programmer can also use additional constructs to make methods coming from traits accessible under new names. When a trait specifies that it needs some methods to work, then the programmer can specify during the composition which methods from the base class will be used to fulfill those requirements. If the base class does not contain methods which can be used as the ones required by the trait, then the programmer can supply an implementation of such methods during this composition. Such additional code supplied to make this composition work is called "glue code".

Then, in our example, one can specify traits `CompressedStream`, `EncryptedStream`, where each of those traits would require methods `read` and `write` and contain their overriding variants supporting compressed and encrypted data. Finally, using those traits, classes like `CompressedFileStream`, or `EncryptedNetworkStream` can be created.

Thanks to this flexible method manipulation mechanism, the programmer has the choice of how to deal with the conflicts of methods with the same name coming from different places (from different traits in this case). The traits approach was first developed in the untyped languages like Smalltalk, however later on it was studied in typed settings (see Section 13.3.3).

However, those features do not come without the price. First of all, traits cannot contain public or protected field declarations, while private fields has been added recently in [14], and are not yet available in most implementations. Similarly, traits cannot contain constructors, which are often useful when combined with fields. All the constructors of the created classes need to be specified in their declaration, not in traits.

Additionally, the mentioned flexibility has the following consequence: The fact that a given class was build from a given trait, does not imply any guarantees about the methods of objects created from this class. It happens so because even if we know that the class was constructed using the trait containing a method `m`, then the method might have been renamed, removed or replaced by other method with the same name.

This lack of guarantee has also some more direct implications (present regardless of any type system in the language) on the design of bigger libraries. Let us imagine a situation where traits are used to design a library which contains some specific functionality spanning multiple collaborating objects created from different classes and traits. For example method $\mathtt{m}_1$ in trait $\mathtt{T}_1$ calls method $\mathtt{m}_2$ of some other object declared in some other trait $\mathtt{T}_2$. Notice that such dependency is not reflected in the specification of methods required by a given trait, because the latter lists methods required to be present in the same object. Then consider an application of such library functionality in some existing system, which means applying those traits to a set of existing classes. Unfortunately, when during this application, the programmer will perform method renaming or hiding on methods from $\mathtt{T}_2$, then the code in $\mathtt{m}_1$ will stop working in this scenario, because it will still expect method $\mathtt{m}_2$ of the collaborating object to exist under the original name. To better understand this problem let us analyze the below scenario.

Consider the trait `EncryptedStream` on Figure 2.6, which encrypts the data using some certificate, and which comes together with some `CertificateManager` trait or class. This manager, depending on the user currently logged to the system, automatically sets certificates on some or all created encrypted streams. To do this, the `CertificateManager` calls method `setCertificate` on the `EncryptedStream` and also some getters to choose the certificate needed for specific stream. Unfortunately, when the new `CryptFileStream` class will be defined using the `EncryptedStream` trait, and the `setCertificate` method will be renamed, then the certificate manager will stop working with such a stream.

In general, when traits are used to create functionalities spanning many collaborating objects, it might be hard to predict which dependency will be broken, when any method supplied in the trait will be renamed or hidden.

This problem has already been spotted (and worked-around in a various ways) by a few independent groups using different solutions. The first solution was developed by Charles Smith et al., when designing Chai, an extension of statically typed Java with traits [62], and independently by Allen et al. when designing the Fortress language [9, 7, 8]. The second solution was developed by Oscar Niestrasz et al., when they worked on adapting traits to statically typed languages [57]. Finally it was also spotted by Ducasse et al. in their work on Freezable Traits [34].

In the first mentioned solution, the authors decided to restrict hiding and renaming of methods in a way in which when a method is removed, then another implementation of the method with the same signature must be added to the same class. This way the compiler ensures that the class build from trait X will always have all the methods of names and types specified in X. However, those methods can be sometimes completely different methods, not those introduced in X. Additionally, this restriction implies that conflicts of two different methods with the same name and different result type cannot be resolved. Similarly, in the Fortress language, traits also induce types. Furthermore, Fortress language is even more restrictive than Chai. The designers of this language, to avoid the mentioned problems with typing of traits, decided not to allow method renaming and hiding. As a consequence, they also decided that, when two traits with methods of the same name are composed, then one method overrides the other one (in the order reverse to the one of declarations of composed traits).

In the second of the mentioned solutions, the authors decided that traits themselves will not be visible in the nominal type system of a statically type checked language like Java. Therefore, to be able to implement the functionality which refers to some method of objects created using some given trait, a programmer needs to declare an additional interface containing some of the methods of that trait, and make sure that all classes using that mixin

```
// A library containing trait EncryptedConnection, and some helper class
// called CertificateManager
trait named: #EncryptedStream
  instanceVariableNames: 'aCertificate'
  setCertificate: certificate
    ...do checks on the certificate...
    aCertificate := certificate.

  autoFindCertificate
    | manager |
    manager = getGlobalCertificateManager.
    manager findCertificateFor: self.

Object subclass: #CertificateManager
  findCertificateFor: encrypted_connection
    | some_certificate |
    some_certificate := ... do something to establish certificate....
    encrypted_connection setCertificate: some_certificate.


// Some client code written by other programmer, basing the above library.
// This code contains declaration of class CryptFileStream and its usage.
FileStream subclass: #CryptFileStream
  uses: {EncryptedStream @ {#setCertificate: -> #setEncryptCert:} -
         {#setCertificate:}} + otherTrait
     ... some class declaration ....

// Usage of the above declared CryptFileStream class.
// The call to method autoFindCertificate fails, because the renamed method
// setCertificate is used by internally created CertificateManager object.
// However, programmer writing this client code could not predict it, since
// this dependency is not visible outside in the signature.
  stream := CryptFileStream new.
  stream autoFindCertificate.
```

Figure 2.6: Broken dependency during trait method renaming (using Smalltalk-like syntax)

implement that interface. This means additional work for the programmer. Moreover, when some method is renamed in the declaration of some class, then it either requires the class to stop implementing that interface or requires some additional changes in the code.

In the third solution based on the notion of method freezing [34], the authors allowed the programmer to keep two versions of conflicting methods: one as private, and other one as public. The private version is executed when so called self call is performed, which means a call from the trait in which this private, or frozen method has been declared. On the other hand, the public version of method can be called externally - through the so called object calls. However, this approach can only be used to solve the problem when the conflicting method is called internally, by another method declared in the same trait. Unfortunately, it cannot be used when this method is required by code present in another class/trait, as the method `setCertificate:` called in method `findCertificateFor:` of class `CertificateManager` visible on Figure 2.6.

Additionally, such problems show that the trait approach breaks encapsulation in some sense. It breaks encapsulation in the sense that the programmer using the trait not only has to know what methods it provides, but also needs to know their actual implementation to be aware of such dependencies between the methods and thus of consequences of method renaming or hiding.

Chapter 3

The Magda language

In this chapter we present a statically typed, OO language called Magda. Magda, thanks to its unique constructs, enjoys the property of the expressive and safe modularity discussed in Section 1.2. We introduce this language by presenting a series of examples, each with a set of comments. We start from a simple "Hello-world" example, then we follow with more complicated examples in order to make the reader familiar with most of the features of the language. In Chapter 4 we provide a complete description of the syntax. In Chapter 7 we formalize the semantics of Magda and in Section 8 we present all the type checking rules of Magda.

3.1 The structure of a program

Every program in Magda consists of two parts. The first part (and in most cases the biggest one) is the list of *mixin declarations*. The second part is a list of instructions, called *main instructions*. These instructions are the ones which will be executed when the program is started. Each mixin declaration placed in the first part of a program contains declarations of methods, fields and additional members responsible for the initialization of new objects. In most cases the main instructions use the mixin declarations to perform the task of the whole program.

The execution process of every program written in Magda is performed by the execution of all main instruction. When the last main instruction finishes, then the whole program also finishes. Every main instruction can be one of the following:

- The creation of a new object. Each object in Magda is created from a non-empty sequence of mixins. In this sense, the sequence of mixins plays a role similar to the one of `class` in other OO languages like Java [39], C++ [63] and C♯ [40];

- Execution of a method on behalf of an existing object (called also method call or message sending). In Magda, as in most OO languages, the set of methods "understood" by the object depends on the "template" from which it was created. Therefore, the set of understood methods depends on the sequence of mixins used to create the object.

The whole further execution of the program (including the execution of instructions placed within the bodies of methods) follows the same pattern. The only difference is that instructions placed within the method bodies can also use local variables to store values within, and can also modify the state of an object (see Section 3.2 and Section 3.3). All the values used in Magda programs are either `null` values or objects created from mixins. Each not initialized variable and field has the value `null`.

The simple example of a program printing "Hello World" written in Magda can be seen on Figure 3.1.

```
mixin HelloWorld of Object =
  new Object MainMatter()
  begin
    "Hello world".String.print();
  end;
end;
//
(new HelloWorld []).HelloWorld.MainMatter();
```

Figure 3.1: A simple "Hello World" example in Magda

The above program consists of the declaration of one mixin named `HelloWorld` and one main instruction.

The declaration of mixin `HelloWorld` contains a declaration of one method named `MainMatter`. The declaration of the `MainMatter` method begins with the keyword `new` which indicates the introduction of a new method identifier, as opposed to method redefinition described in Section 3.6, and abstract methods described in Section 3.7. Then the declaration of the method contains the type of a value returned by the method. The `MainMatter` method does not explicitly return any value (which means that it implicitly returns `null` value), therefore it uses the `Object` return type — which is a type of all values. The above presented header of the method is followed by the body of the method. The body of each method consists of a list of instructions, in case of this method it means one instruction only.

Additionally, this program contains one main instruction. This instruction starts from the creation of an object from the above declared mixin `HelloWorld`, using the `new HelloWorld[]` expression. Then this instruction calls the method `MainMatter` of the newly created object. The only job performed by the `MainMatter` method is the call of the method `print` (declared in mixin `String`) on the string literal `"Hello world"`. In Magda, `String` is a built-in mixin used by text literals. The `String` mixin contains a few methods, like `print` which prints the value on the screen, or `add` which concatenates two string values. Similarly, the Magda language contains a few other built-in mixins like `Integer` and `Boolean` (see Section 3.10).

As it can be seen in the above example, in each method call, the method name is prefixed with the name of the mixin in which it was introduced. In this example `MainMatter` is prefixed with the `HelloWorld` mixin name. Similarly the `print` method call is prefixed with the `String` mixin name. We have developed the above approach to identifier referencing called *Hygienic identifiers* in order to resolve ambiguities, and to avoid any accidental name clashes. We first introduced this approach and studied carefully in separation of other Magda features in [48].

In the current version of Magda all methods are visible to calls from all classes. This results in a similar behavior as the one of `public` methods in Java. In the future versions of Magda we plan to allow one to include additional information about the visibility of methods in a way similar to other languages. This aspect is however orthogonal to Magda's specific features presented below, therefore it is not discussed in this thesis.

3.2 Object fields

Similarly to other OO languages, in Magda an object can have a mutable state, which can change during the life of the object. The current state of an object is stored in the object's fields. The list of all fields used by an object is declared in the mixin/mixins from which the given object has been created. The value of each field of an object can be either `null` value or an object. The initial value of each field is `null`.

Each field declaration consists of a field name followed by its type. The type of the field (as well as any other type in a Magda program) is a sequence of names of mixins. See Section 3.9 for further comments on the types, subtyping and the type checking.

For simplicity we have chosen that each field in Magda can be only accessed by methods of that object. This access is performed using the following syntax: `this.`*FieldId*. As a result we obtain the behavior similar to the one of `protected` fields in other languages. Furthermore, similarly to references to method identifiers, all references to fields are prefixed with the name of the mixin in which the referenced fields have been declared.

In Figure 3.2 one can see a declaration of mixin `Point2D` containing two field declarations: `x` and `y`. In this example, the mixin `Point2D` is used to create objects representing mutable, two-dimensional points. Method `setCoords` allows other objects to modify the state of objects created from mixin `Point2D`. This method modifies the state of an object by setting values of `x` and `y` fields of the local object. This modification is performed using the field assignment instruction `this.Point2D.x := ax`. Similarly, using `this.Point2D.x` syntax, method `getX` allows other objects to query the state of the given object.

3.3 Local variables

Similarly to other languages, each method in Magda can contain declarations of local variables. As one can see in the `MainClass.MainMatter` method on Figure 3.2, variable declarations are placed in the header of the method, after the name of the method and the declaration of parameters, before the keyword `begin`, denoting the start of method body (similarly to the way in which variables are declared in Pascal [12]). Each such declaration consists of a variable name followed by a colon and the type of the variable. The type of the variable is a sequence of mixin names. In our example one can see the declaration of

```
mixin Point2D of Object =
  x:Integer;
  y:Integer;

  new Object setCoords( ax:Integer; ay:Integer)
  begin
    this.Point2D.x := ax;
    this.Point2D.y := ay;
  end;

  new Integer getX()
  begin
    return this.Point2D.x;
  end;
end;
//
mixin MainClass of Object =
  new Object MainMatter()
    p1:Point2D;
    p2:Point2D;
  begin
    p1 := new Point2D[];
    p2 := p1;
    p1.Point2D.setCoords( 11, 10);
    p2.Point2D.getX().Integer.print();  //this line prints out 11
  end;
end;
//
(new MainClass []).MainClass.MainMatter();
```

Figure 3.2: Fields and variables in Magda

the variable `p1` with type `Point2D` present within method `MainMatter` in mixin `MainClass`. Similarly to fields, local variables initially contain `null` value, and can be assigned with object values.

As in many other languages (like Java and C♯), the value of each field and variable is a reference to an object. The value of a field cannot be an object itself (as it can for example in C++ [63]). In Java there is one exception from this rule: Fields of primitive types like `int` and `float` are not references, however in Magda, for simplicity, we decided that simple numerical types are objects too.

As a result, in Magda, the assignment instruction does not copy the object, only copies the reference to it. Therefore, when two variables (or fields) contain references to the same object, and the state of the object accessed through one of the variables is modified, then the state of the object accessed through the second one reflects all the changes. The example of such behavior can be seen in the last instruction of method `MainClass.MainMatter` on Figure 3.2.

3.4 Inheritance

The inheritance mechanism in Magda works in a slightly different way than in typical single-inheritance OO languages (like Java, C♯) or multiple-inheritance ones (like C++).

In traditional OO languages, a new class which is declared to extend an existing class or classes, automatically includes all of their members (like fields and methods). In Magda, a mixin `C` can be declared in a way in which it specifies that it "uses" (or "requires") another mixin `A`. In this case we say that mixin `A` is a *base mixin* of the mixin `C`. Then, every object created from mixin `C` must be also created from mixin `A`.

As a result, all the code referring to an object created from mixin `C` can also safely use methods declared in mixin `A`. This applies also to the code of the methods declared within mixin `C` itself.

However, unlike in classical OO languages, mixin `A` as well as all the methods and the fields declared in that mixin, are not implicitly included in mixin `C`. All the members (fields and methods) declared within mixin `A` are not visible in the same way as if they had been declared in mixin `C`. In particular it means that:

- every expression creating a new object using mixin `C` needs also to explicitly mention mixin `A` in order to create a working object. Moreover, in the sequence of mixins used to create an object from, `A` needs to occur at an earlier position than mixin `C`.
- every method call (and field dereference as well) targeting an object created from the set of mixins {`A`, `C`} is prefixed with the name of the mixin from which the chosen method originates. If the method was declared originally within the mixin `A` then the method call needs to specify it.

On Figure 3.3 one can see a simple example of a program containing declarations of two mixins: `Point2D`, and `Point3D`. In this case the *base mixin* of `Point3D` is `Point2D`. In other words: `Point3D` extends `Point2D`.

```
mixin Point2D of Object =
  x:Integer;
  y:Integer;

  new Object setCoords2D( ax:Integer; ay:Integer)
  begin
    this.Point2D.x := ax;
    this.Point2D.y := ay;
  end;

  new Integer getX()
  begin
    return this.Point2D.x;
  end;
end;

mixin Point3D of Point2D =
  z:Integer;

  new Object setCoords3D( ax:Integer; ay:Integer; az:Integer)
  begin
    this.Point2D.setCoords2D(ax, ay); //a call to the method coming from
    this.Point3D.z := az;             //another mixin is prefixed with its name
  end;
end;

mixin MainClass of Object =
  new Object MainMatter()
    x:Point3D;
  begin
    x := new Point2D, Point3D[];
    x.Point3D.setCoords3D( 10, 11, 12); //this method comes from Point3D
    x.Point2D.getX().Integer.print();   //while that one from    Point2D
  end;
end;
(new MainClass []).MainClass.MainMatter();
```

Figure 3.3: Inheritance example in Magda

The first instruction of the method `MainClass.MainMatter()` assigns the object created from mixins `Point2D` and `Point3D` to the variable `x`. Then, as one can see in further instructions of the method `MainClass.MainMatter()`, each method call targeting the value of variable `x` is prefixed with the name of a mixin in which the referenced method was declared. For example, the call to `setCoords3D` is prefixed with `Point3D` mixin name, while the call to `getX` is prefixed with `Point2D`. Similarly, method `setCoords3D` setting the state of the object itself needs to specify whether it references some member declared locally (like `this.Point3D.z`) or inherited from the base mixin (like `this.Point2D.setCoords2D`).

Each mixin which is not declared to explicitly extend any specific mixin, implicitly extends mixin `Object` which is a base mixin of every mixin.

3.5 Multiple inheritance

In this section we present how mixins declared in a Magda program can be combined to obtain results comparable to those which, in classical OO languages, require multiple inheritance. Magda has two features which allow the programmer to modularize the code in a flexible way and compose components from other existing components:

- The declaration of base mixins of a mixin can contain multiple names of mixins. In this way the mixin can extend multiple mixins. As a result, in each object creation expression using the given mixin, all of its base mixins have to be included explicitly to create an object. Consider a mixin `A` containing `B` and `C` as its base mixins. Then each `new ... A ... [...]` expression must also include `B` and `C` mixins. Those mixins need to occur earlier than `A` in the given sequence, however their relative ordering is arbitrary (see Figure 3.6 for example which presents different results depending on such relative ordering).

- The declaration of base mixins of some mixin `A` represents only the minimal set of mixins required for given mixin to be combined with. Therefore, the actual object creation expression can combine more mixins with the mixin `A` than specified in its base mixin expression, obtaining even more functionality in one object. As a result, the programmer can combine features of distinct mixins, even when they are declared to extend one mixin only (therefore looking like a "single inheritance hierarchy") and have been declared independently.

The example visible on Figures 3.4 and 3.5 contains a program which uses the multiple inheritance. In this program there are declarations of five mixins. The first two mixins (`DisplayableObject`, `Point2D`) are built from scratch (not inheriting from any other mixin). Next two mixins (`Point3D`, `ColorPoint`) extend mixin `Point2D` in a single inheritance way. Then, the example contains a declaration of a mixin `Displayable3DColorPoint`, which extends three independent mixins. Finally, the example contains an object creating expression, which uses three independent mixins to create an object from.

```
// Declarations of two mixins (DisplayableObject, Point2D) built from scratch
mixin DisplayableObject of Object = ...
end;

mixin Point2D of Object =  ...
end;

// Two other mixins extending the Point2D mixin
mixin Point3D of Point2D =  ...
end;

mixin ColorPoint of Point2D = ...
end;

// One mixin extending three independent mixins
mixin Displayable3DColorPoint of DisplayableObject, Point3D, ColorPoint =  ...
end;

// Composition of independent mixins in object creation
new Point2D, Point3D, ColorPoint [...]
```

Figure 3.4: Multiple inheritance in Magda: Code snippets

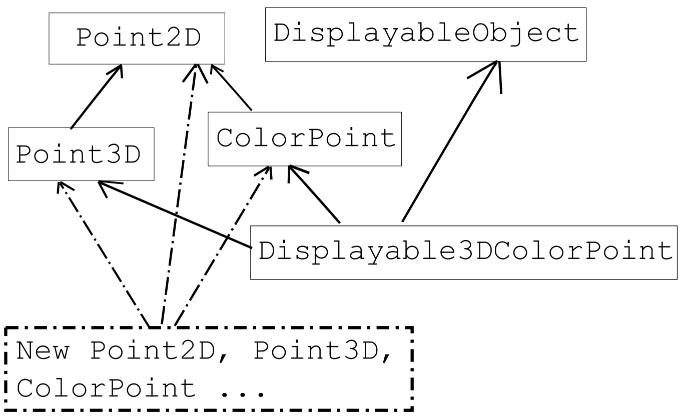

Figure 3.5: Multiple inheritance in Magda: mixin hierarchy graph example

3.6 Virtual methods

In this section we describe how methods declared in base mixins can be redefined in extending mixins. As a result of the below described mechanism, all methods in Magda work like virtual methods in other languages [63].

We have already presented how new methods can be declared in mixins — in Section 3.1. We have also presented how new mixins can be build using existing ones, using the inheritance presented in Section 3.4. However, a declaration of a mixin extending other mixins can also contain redefinitions of methods introduced in the base mixins. Then, when the object created from a mixin containing the method redefinition is requested to execute the method introduced in the base mixin, the body of the redefined method is used. Then, as a part of its implementation, the body of the redefined method can contain a call to the original implementation of the method.

However, unlike in Java language, the method redefinition uses the syntax different from the method introduction. The redefinition begins with `override` keyword instead of `new` keyword used by the introduction. Additionally, each declaration of a method redefinition contains the name of the mixin in which the redefined method was first introduced.

The body of a method redefinition can use `super(...)` expression to call the "previous" implementation of the redefined method. When `super(...)` expression is evaluated in the method mt of the object created from mixins $\overrightarrow{M}$, the mentioned "previous" implementation of mt is chosen from the bodies of mt occurring within the declarations of mixins $\overrightarrow{M}$. The body called by a `super(...)` expression present in the mixin M_c, is picked from the last mixin in $\overrightarrow{M}$ preceding M_c, which contains a definition/redefinition of the method mt.

As a result of the above defined mechanism, one `super(...)` call expression placed within a given method redefinition used in different objects can call different method implementations. The actual method body called by the `super(...)` expression depends on the mixins (and their order) which have been used to create the current object.

An example of a program using method redefinitions in which the behavior of `super(...)` call changes depending on the ordering of mixins can be seen on Figure 3.6.

This example begins with the declaration of mixin `BaseMixin` with one introduced method `GetActualName`, which returns string value `"Base "`. Then the example contains declarations of two mixins: `Extension1` and `Extension2`. Each of those mixins extends mixin `BaseMixin` and contains a redefinition of the method introduced within the declaration of the mixin `BaseMixin`. The implementation of each of those redefinitions contains one instruction, which first calls the original implementation of the method and then adds some suffix (`"Extension1 "` and `"Extension2 "`) to the obtained string.

Finally, after all these three mixin declarations, the program contains two analogous instructions (separated by the instruction printing end of the line). Each of them: (*i*) creates an object from all three declared mixins, (*ii*) calls the method `BaseMixin.GetActualName`, (*iii*) prints the results of that method call. The only difference between those two instructions is in the order of mixins used to create an object. As a result of different ordering of mixins, the bodies of redefined methods are also executed in different order. As a consequence, the

```
mixin BaseMixin of Object=
  new String GetActualName()
  begin
    return "Base ";
  end;
end;

mixin Extension1 of BaseMixin =
  override String BaseMixin.GetActualName()
  begin
    return super().String.add("Extension1 ");
  end;
end;

mixin Extension2 of BaseMixin =
  override String BaseMixin.GetActualName()
  begin
    return super().String.add("Extension2 ");
  end;
end;

(new BaseMixin, Extension1, Extension2[]).BaseMixin.GetActualName().String.print();
"\n".String.print();                    // this prints the "end of line" character
(new BaseMixin, Extension2, Extension1[]).BaseMixin.GetActualName().String.print();
// Program generates output of the form:
// Base Extension1 Extension2
// Base Extension2 Extension1
```

Figure 3.6: Virtual method redefinitions in Magda

strings printed by those instructions differ (as mentioned in the comments at the end of the program).

Notice that, as a result of the above syntax of the method redefinition, there are no problems in cases when two base mixins of a given mixin contain introductions of methods with the same name. The programmer declaring a method redefinition in a mixin explicitly chooses which method from which mixins he/she wants to redefine. Additionally, when a new method introduction is added to the existing mixin extended by other mixins, there is no risk of accidental name clashes. Such a risk of accidental name clashes and conflicts exists in most other OO languages, as described in Section 2.2. We have analyzed this problem in detail, and presented numbers of possible name ambiguities in the standard Java library in [48].

3.7 Abstract methods

As presented in the previous section, a declaration of a mixin can contain declaration of new methods (marked with `new` directive) together with their implementations, as well as redefinitions of methods declared in base mixins of the mixin (marked with `override` directive). As we have also seen, redefined bodies can use a `super(...)` expression to call previous implementations of methods.

However, a mixin declaration can also contain a declaration of a new method identifier without the body, by marking the method declaration with the keyword `abstract` (similarly to abstract methods in Java). Then, other mixins can supply an implementation of that method, by marking the declaration with the keyword `implement`. If a mixin containing a declaration of a method marked as `abstract` is used in the object creation expression, then at the same time another mixin with the `implement` version of the same method is required. As a result, each object created using a mixin with an `abstract` method must contain at least one body of that method.

The declaration of a method marked with the `implement` keyword differs from the one marked with as `override` in two aspects:

- the body of an `implement` method cannot contain `super(...)` call;
- in contrast to `implement` versions, a mixin with an `override` version of the method can only be used in the sequence of mixins of an object creation expression if, before that mixin, there is another mixin with a `new` or an `implement` version of the same method. This restriction is necessary in order to ensure that each time `super(...)` expression placed in the `override` version of method is executed, there is some implementation to be called.

One might note that, in classical single-inheritance OO languages, the explicit distinction in syntax between `implement` and `override` variant is not required. This happens so because, in such languages, by using the fixed inheritance hierarchy the compiler can verify if there is another implementation of the method in the superclasses of the class. On other hand, in Magda, a mixin can be used in different scenarios of object creation and Magda supports separate unit compilation, thus the compiler cannot verify all possible usages of the mixin at the moment of the compilation of its declaration.

The example on Figure 3.7 shows the mixin `M1` with the declaration of the abstract method `Met1` as well as the mixin `M2` with the declaration of implementation of the method `Met1` and mixin `M3` with a redefinition of that method.

Finally the program contains three main instructions: one containing a proper object creation from three mixins and two other which are incorrect (and need to be removed from the program in order to make it compile). One of the erroneous instructions contains an object creation in which the `override` method does not have any other implementation to redefine (and reference using `super(...)` call), while the second one contains a mixin with `abstract` method without any actual implementation.

```
mixin M1 of Object =
  abstract String Met1();
end;

mixin M2 of M1 =
  implement String M1.Met1()
  begin
    return "Implementation from M2";
  end;
end;

mixin M3 of M1 =
  override String M1.Met1()
  begin
    return super().String.add(" with redefinition from M3");
  end;
end;

// The below line is OK and prints:
"Implementation from M2 with redefinition from M3"
(new M1, M2, M3 []).M1.Met1().String.print();

// The below line will not compile since the override method
// in M3, has no implementation to redefine
(new M1, M3, M2 []);

// This will not compile due to lack of implementation of M1.Met1
(new M1 []);
```

Figure 3.7: Abstract methods in Magda

3.8 Object initialization

In this section we describe how the initialization process of newly created objects is performed in Magda.

We decided not to integrate in Magda the classical constructors approach, which causes some problems, especially in the presence of mixins, as discussed in Section 2.1. Instead we developed a “modular initialization protocol” approach which allows one to split declarations of the initialization protocol into smaller, composable pieces called *initialization modules* (or *ini modules*). Each such ini module contains a declaration of a list of *input parameters*, which can be supplied to a mixin during object creation, together with a set of instructions which will be executed when the values of input parameters are supplied. The signature of an ini module also carries information whether its usage is optional or mandatory (then its input parameters need to be supplied). It also specifies a list of *output parameters* (referring to input parameters declared in other modules). This list specifies input parameters of other modules, which will be computed by the given ini module and supplied to the modules from which those parameters originate.

Additionally, the body of each ini module contains at some place the `super[...]` instruction. This instruction is responsible for calling further ini modules and for supplying values of the input parameters of other modules, which have been declared as output parameters of the given ini module. Therefore, inside the square brackets there is one assignment for each declaration of output parameter of the ini module in which this instruction occurred.

On the other hand, every object creation expression is accompanied with the list (possibly empty) of initialization parameters, together with their values. When such an expression is evaluated, the list of ini modules within the mixins used is traversed. Each module for which we have input parameters supplied is executed and, when the execution of a given module reaches the `super[...]` instruction, the traversal procedure is resumed. A module is executed when its input parameters are supplied either directly in the object creation expression, or are supplied indirectly by some other module, which was executed before.

In Figure 3.8 we present an example of two mixins containing simple ini module declarations.

The first mixin in this example (`Point2D`) contains the declaration of one ini module `mod1`. This ini module begins with the keyword `required` and then contains the name of the mixin in which it is declared (we include the name of the mixin in the declaration of ini module, in order to make ini modules look similar to constructors in Java [39] and C♯ [40]). Then, inside the parentheses, it contains the list of two input parameters: `x` and `y` together with their types. It means that this ini module requires parameters `x` and `y`. Then, after the keyword `initializes` there is a list of output parameters, which in this case is empty. Finally, the ini module declaration ends with its body containing the list of instructions which will be executed when the input parameters are supplied. The last instruction of the ini module is `super[]`. This instruction does not contain any parameter assignments inside the brackets because the module has no output parameters declared. The only task of this instruction is to call the next ini modules (if they exist and are used in the given object creation expression) at the end of the body of that module. If there are no other modules to execute, the `super[]` instruction does nothing.

The second mixin (`Point3D`) contains declarations of two more ini modules. The first one (`mod2`) expects one input parameter, which is used internally to initialize fields of the object. Then, the second ini module marked as `mod3` contains the declaration of one input parameter named `other` and three output parameters. Those output parameters refer to input parameters of the module `mod2` declared in the same mixin, as well to the parameters of the module `mod1` declared in the base mixin. At the same time, the body of that ini module contains the `super[...]` instruction, which computes the values of those three output parameters.

Finally, the `MainClass.MainMatter` method contains two object creation expressions. The first expression supplies values of three initialization parameters: `z` of mixin `Point3D` and `x` and `y` of mixin `Point2D`. The first parameter `Point3D.z` is supplied to module `mod2`, and that module is the first one executed. When the execution of that module reaches

```
mixin Point2D of Object =
  fx:Integer; fy:Integer;

  required Point2D(x:Integer; y:Integer) initializes ()         //module mod1
  begin
    this.Point2D.fx := x;
    this.Point2D.fy := y;
    super[];
  end;

  new Integer getX() ....;
  new Integer getY() ....;
end;

mixin Point3D of Point2D =
  fz:Integer;

  required Point3D(z:Integer) initializes ()                    //module mod2
  begin
    this.Point3D.fz := z;
    super[];
  end;

  optional Point3D(other:Point3D) initializes
      (Point2D.x, Point2D.y, Point3D.z)                         //module mod3
  begin
    super[Point2D.x:= other.Point2D.getX(), Point2D.y:= other.Point2D.getY(),
          Point3D.z:= other.Point3D.getZ()];
  end;

  new Integer getZ() ....;
end;

mixin MainClass of Object =
  new Object MainMatter()
    p1:Point3D; p2:Point2D;
  begin
    p1:= new Point2D, Point3D[ Point3D.z:= 12, Point2D.x:=10, Point2D.y:=11 ];
    p2:= new Point2D, Point3D[ Point3D.other := p1 ];
  end;
end;
(new MainClass []).MainClass.MainMatter();
```

Figure 3.8: Object initialization in Magda

`super[]` instruction, the program searches for the next module to be executed. The next module to be executed means a module (occurring in this mixin, or in other mixins from which the object have been created) for which we have values of input parameters supplied or computed. In this case, the next module to be executed means `mod1`, because parameters `x` and `y` are still not consumed. We say that some parameter *par* is *not consumed* at a given point of initialization if it was either supplied in the object creation expression itself, or calculated as an output parameter of some already executed ini module, while no module has been executed which takes *par* as an input parameter.

When the execution of the module `mod1` reaches the last instruction (which is the `super[]` call), then this instruction checks if there are some ini modules to execute and does nothing, since there are no modules to execute. Then module `mod1` finishes and the control returns to the module `mod2` at the point after the `super[]` call. There are no further instructions in the module `mod2`, therefore the whole initialization process finishes with all the `required` modules executed and the initialized object is returned in order to be assigned to variable `p1`. In such cases we say that modules `mod1` and `mod2` are *activated* by those three parameters.

The second object creation expression supplies the value of one initialization parameter: `Point3D.other`. That parameter is supplied to the optional module `mod3`. When this module is executed, it computes the set of three parameters `x`,`y`,`z`. The third parameter is passed to the module `mod2` and then, when the execution reaches `super[...]` call within it, the remaining two parameters are supplied to module `mod1`. As a result all the ini modules declared in the program are executed during creation of that object, in the "bottom-up" order which in this case means: `mod3`, `mod2`, `mod1`. Notice that in case when two or more ini modules declared in one mixin are used during one object creation (like modules `mod2` and `mod3` in the last object creation from our example), their respective ordering decides. The one which is *textually below* (like `mod3`) is executed first and those placed *textually above* (like `mod2`) afterwards. As a consequence, in every mixin declared in a Magda program, if some module outputs a parameter which is then consumed by another module in the same mixin (as `Point3D.z` produced by `mod3` and consumed by `mod2`), then the consuming module (like `mod2`) needs to be placed textually above the one which outputs it (like `mod3`). This condition is verified by the type checker.

The motivation for such an unusual approach (different from declaring lists of constructors with parameters) is the following. We want each mixin to be as composable with other mixins as possible and we want to minimize the amount of the code a programmer has to write (thus also to minimize the code copying). In traditional OO approach, each constructor is a monolithic entity which completely describes one variant of the initialization process. When the designer of some subclass of an existing class (or mixin) wants to extend the initialization protocol of the superclass (for example, wants to add one additional parameter) it needs to copy the whole list of constructors with all their parameters. Furthermore, such a mixin explicitly refers to the list of constructors of the parent class, which limits the possibilities of future mixin combinations. See section 2.1 for a more detailed description of the traditional approach.

On the other hand, in our approach of initialization modules, if one needs to add to some

```
mixin ColorPoint of Point2D =
  fcr:Integer;
  fcg:Integer;
  fcb:Integer;

  required ColorPoint(cr:Integer, cg:Integer, cb:Integer) initializes ()
  begin
    this.Point3D.fcr := cr;
    this.Point3D.fcg := cg;
    this.Point3D.fcb := cb;
    super[];
  end;
end;

new Point2D, Point3D, ColorPoint[Point2D.x := 1, Point2D.y := 2, Point3D.z := 10,
                                 ColorPoint.cr := 255, ....];
```

Figure 3.9: `ColorPoint` mixin - continuation of the example from Figure 3.8

mixin some additional initialization information, he/she adds new initialization module with that input parameter and does not need to refer to any other parameters. Such a declaration is independent from the list and types of constructors/initialization modules of base mixins, as the initialization module in mixin `ColorPoint` on Figure 3.9 which is independent from the parameters of the mixin `Point2D`. A module contains references to other parameters only when it replaces them, as the initialization module from Figure 3.8 with the parameter `other:Point3D`, which refers to replaced parameters `x`, `y`, and `z`. Also, when the new object is created from multiple independent mixins, their initialization parameters and modules do not interfere one with each other. For example, one can create a new object from mixins `ColorPoint` and `Point3D`, which do not share common ancestor yet add new initialization parameters. As a result the programmer has greater flexibility of combining the mixins than in the constructor-based approach.

We have introduced this approach in [18], where we presented the results of an addition of the modular initialization protocol to Java, together with possible implementation techniques for such Java extension. Later on, in [17], we have proved the type soundness of the Featherweight Java [42] (which is a functional subset of Java) extended with the modular initialization protocol.

3.9 Types and type expressions

In this section we describe how the type system works in Magda.

First of all, Magda is a statically typed language. It means that the types of all the elements in the program are checked during the compilation (statically). As a result, if the program successfully passes the type checking procedure, we have the guarantee that program will not fail due to type errors. This property of Magda is called type soundness and is precisely defined and proved in Section 11. In particular, the type soundness of Magda ensures the following: Whenever the execution of a type checked program reaches a point of some method call on a particular object value o (different from `null`), it is guaranteed that the object o was created from mixins which contain at least one implementation of the method invoked in that call. In other words, "message not understood" errors will never occur during the program execution (in contrast to dynamically typed languages like Smalltalk [38] or Python [13, 27]).

Secondly, Magda uses a Church-style type system [28], which means that types of method results, variables, fields and parameters are explicitly written in a program. Both of the above properties are similar to most industry-standard OO languages, like C++, Java and C♯. What distinguishes Magda from other languages is the way the type expressions are formed and the way the subtyping is verified.

Each type in a Magda program is a set of mixin names, written as a sequence of mixin names separated by commas. The ordering of mixin names in a type expression is insignificant. In the simplest case the type of variable is a single mixin name. When some variable or field is declared of type T it means that the actual value of the variable can be either `null` or an object created from a sequence of mixins, which contains at least the mixin names present in T (even though it can contain more), except the mixin `Object` which is always used implicitly during each object creation. Similarly, if the formal method parameter is declared with the type T, it means that in each method call the actual value must be an object created from all the mixins in T (and maybe more).

For example, consider the program on Figure 3.8. The declared type of variable `p1` in method `MainClass.MainMatter` is `Point2D`, while the declared type of variable `p2` is `Point2D`. The second declaration means that the variable `p2` can hold only `null` value, or a reference to an object created using some sequence of mixins containing `Point2D`. However, notice that `Point3D` mixin has `Point2D` as its base mixins. As a result, every object created from `Point3D` is also created from `Point2D`. Therefore the type `Point2D`, `Point3D` is equivalent to the type `Point3D` as well as to the type `Point3D`, `Point2D`. Notice that the type expressions differ in this matter from the sequences of mixins used within the object creation expressions. In an object creation expression, the base mixins cannot be skipped because the order in which they are placed within the object creation expression is significant. See Figure 3.6 for an example in which the order matters. On the other hand, in declarations of formal types, the ordering is insignificant, as well as the removal of base mixins.

We say that type T_2 is a *fully expanded form* of type T_1 if T_2 is the biggest type equivalent to T_1. In other words, it is a type obtained by starting from T_1 and adding all the base mixins of mixins present in the T_1, base mixins of base mixins etc. In the example present on Figures 3.4 and 3.5, the fully expanded form of type `Displayable3DColorPoint` is: `Object, Point2D, DisplayableObject, Point3D, ColorPoint, Displayable3DColorPoint`.

Now consider the program from Figure 3.4, extended with the mixin present on Figure 3.10. In the method `SomeMethod` there are two variables: `v1`, `v2`. The requirements enforced by the type of variable `v2` are more strict than the requirements enforced by the

```
mixin test of Object =
  new Object SomeMethod ()
    v1: Point3D;
    v2: Point3D, ColorPoint;
  begin
    ...
    v1 := v2; //OK
    v2 := v1; //not OK
  end;
end
```

Figure 3.10: An example of subtyping in Magda

type of variable `v1`. As a result, each value of variable `v2` can be also a value of variable `v1`. However, the opposite does not hold. As a result, the first assignment present in this method is type correct, however the second one is not. Thus this program will not compile.

In general, we say that type T_2 is a *subtype* of type T_1 when the fully expanded form of T_1 is a subset of the fully expanded form of T_2. The fact that type T_2 is subtype of type T_1 will be denoted as $T_2 \preceq T_1$.

As a consequence of this definition, we define the type of `null` value as the set of all mixin names used within the program.

3.10 Control instructions and built-in Boolean type

Boolean values are also object values in Magda. There exists a `Boolean` mixin which is the type of each boolean value. The only specific property of `Boolean` mixin is that it cannot be used directly in object creation expressions. Consequently, this mixin cannot be used as a base mixin of a user-defined mixin. The only way to obtain a fresh `Boolean` value is to use one of the `Boolean` constants: `true` and `false`. On the other hand, since booleans are object values, `null` is also a proper `Boolean` value. Moreover, `null` is also an initial value of each variable and object field of the type `Boolean`.

Additionally, each program in Magda can contain two control instructions: conditional instruction `if` and loop instruction `while`. Those instructions, as in other languages, begin with a condition expression. In Magda, this expression is required to be of the built-in type `Boolean`. Moreover, those instructions require the condition expression to evaluate to `true` or `false`. If this condition evaluates to `null` value, the program stops with “null pointer exception”.

The syntax of those instructions can be seen on the example in Figure 3.11.

```
mixin test of Object =

  // negation implemented using if condition
  new Boolean Not(a:Boolean)
  begin
    if (a) then
      return false;
    else
      return true;
    end;
  end;

  new Boolean Xor(a:Boolean; b:Boolean)
  begin
    if (a) then
      return test.Not(b);
    else
      return b;
    end;
  end;

  new Object SomeMethodWithLoop ()
    x:Boolean;
  begin
    x := true;
    while (x)
      ...
      ...
    end;
  end;
end
```

Figure 3.11: Boolean type and control instructions

3.11 Summary of Magda's features

After going through this short introduction, the reader should notice that the design of Magda language, while sometimes using a little more lengthy syntax than classical OO languages, has a few unique properties, which (we hope) significantly improve the programming process:

- constructors can be composed from many independent initialization modules, coming from different mixins, thus avoiding all the problems described in Section 2.1;
- all identifier references are performed using fully qualified names, which guarantee that programs will never behave unexpectedly, or fail to compile as a result of some random change causing some name clash. This way we avoid all the problems described in Section 2.2;
- the language has a simple mechanism for reuse basing on mixin inheritance, while avoiding all the known problems of other implementations of the mixin construct (see Section 2.4.4).
- the reuse mechanism, in conjunction with the hygienic identifiers approach and the modular constructors, allows the programmer to compose almost any two mixins. This property is expressed precisely in Section 3.11.1.

An in-depth comparison of Magda's features with solutions present in other languages can be found in Chapter 13.

3.11.1 Safety of mixin combination

In this section we state a property of Magda, which captures the safety of composition mechanisms in Magda.

We say that the parameter `p` is an *effective output* of the module `m`, when `p` occurs in the output parameters of module `m`, or module `m` has some output parameter `p'` declared as input parameter in some module which has `p` as its effective output.

We say that two mixins are *explicitly exclusive*, when they both contain declarations of an initialization module, which is required and both have the same initialization parameter as their effective output, thus both modules replace the same parameter. It is important to notice that all declarations of optional initialization modules are always safe, and never make any mixins explicitly exclusive.

We say that mixins M_1 and M_2 can be *combined together*, when the following holds: For any sequence $\overrightarrow{M}$ of mixins from which an object can be created, and which does not contain those two mixins, there exists a sequence $\overrightarrow{M'}$, which contains $\overrightarrow{M}$, M_1, M_2 (and possibly other mixins required to supply implementations of abstract methods in M_1 and M_2), such that $\overrightarrow{M'}$ can be used to create new objects from.

We say that a mixin is *correct* if there exists a sequence of mixins $\overrightarrow{M'}$ containing this mixin, such that $\overrightarrow{M'}$ can be used to create new objects from. And example of mixin which is not correct would be a mixin which contains as its base mixins, two explicitly exclusive mixins.

Finally, we state one of the most important properties of Magda:

Property 1 (Safety of mixin combination) *Any two correct mixins, which are not explicitly exclusive, can be combined together.*

This property is a direct consequence of the three properties mentioned in Section 3.11:

3.12 A note on the modularization of mixin declarations

In practice, for a better code organization, the declarations of mixins should be split into different modules. All those modules should create their own namespaces. Access to a declaration of a mixin from other modules should be available by prefixing a mixin name with a module name. The modules should be structured into a tree hierarchy and name qualification also should follow this path. However, this is a pattern similar to the one used in many existing languages (like Java and $C\sharp$). Therefore, for simplicity, we skip this aspect of the language, just assuming that every name of mixin is unique (by proper qualification of module names).

Chapter 4

Detailed syntax description

In this section we present detailed syntax of all parts of the language. We do this by listing and describing the main syntactic domains of Magda. For an easier understanding, the syntax description of each language construct is accompanied by an explanation of its actual meaning.

4.1 Syntactic domains

The syntax of Magda consists of the following categories:

- programs, as described in Section 4.2,
- mixin declarations, as described in Section 4.3,
- mixin member declarations, as described in Section 4.3.1,
- instructions (for example: assignment, `if`, and `while` statements), as described in Section 4.4,
- object expressions (for example: method call, variable evaluation, object construction from a sequence of mixins, etc), as described in Section 4.5,
- mixin/type expressions, as described in Section 4.6.

4.2 Syntax of a program

Every program in Magda consists of a list of mixin declarations separated by semicolons, followed by a list of instructions, called main instructions, also separated by semicolons. Both of these constructs are described below.

4.3 Syntax of a mixin declaration

A *mixin declaration* consists of:

- the keyword `mixin` followed by the name of the declared mixin;

- the keyword `of` followed by a *mixin expression*;

  This mixin expression denotes a minimal sequence of mixins, with which this mixin must be composed during an object creation. As a result, it plays a role similar to the one of *superclass* in classical OO languages. A mixin expression used in this specific context in the declaration of mixin M is called a *base mixin expression* of mixin M;

- an equals sign followed by the list of *mixin member* declarations, separated by semicolons;

- the keyword `end`.

4.3.1 A mixin member declaration

A *mixin member declaration* is one of the following:

- A declaration of an object field. Each such declaration consists of two parts (separated by a colon): (*i*) field name, (*ii*) the *mixin expression*, which represents the declared type of the given field (called also *formal type*, as opposed to the *runtime type*);

- A declaration of a method as described in Section 4.3.2;

- A declaration of an initialization module as described in Section 4.3.3.

Summarizing, the formal syntax of a mixin declaration is the following:

```
mixin MixinName of MixinExpression
  field:MixinExpression; ...
  MethodDeclaration; ...
  IniModuleDeclaration; ...
end;
```

An example is given on Figure 3.3. In this example, mixin `Point3D` has a base mixin expression `Point2D` and a field declaration, as well as a method declaration.

Notice also that the ordering of most of the members in mixin is not significant except for initialization modules, because the textual ordering of them in mixin influences the order in which those are executed (see Section 3.8).

4.3.2 Syntax of a method declaration

Every *method declaration* consists of:

- The *signature* of the method, which in turn consists of:
  - One of the four keywords: `new`, `abstract`, `implement` or `override`. Those keywords are called from now on *method specifiers.*
  - The return type of the given method denoted by a mixin expression.
  - The method name identifier (in case of `new` and `abstract` methods), or: (*i*) the mixin name, (*ii*) dot character, (*iii*) method name (in case of `implement` and `override`). In the second case the mixin name denotes the name of a mixin in which the given method was originally introduced. This means that within the referenced mixin declaration there must be a declaration of the given method with a `new` or `abstract` annotation.
  - The list of the method's parameter declarations enclosed in parentheses and separated with semicolons. Each parameter declaration consists of a parameter name followed by a colon and by the mixin expression representing its type.
- The *body* of the method. This part is present in declarations of methods, whose header begins with a keyword other than `abstract`. The body of a method consists of:
  - a list of local variable declarations separated by semicolons. Each variable declaration consists of the name of the variable, followed by the colon character, and the mixin expression which denotes the formal type of the given local variable;
  - the `begin` keyword;
  - a list of instructions to be executed when the method is called. Method redefinitions can use special expression `super(...)`, which calls the previous implementation of this method;
  - the keyword `end`.

Thus the syntax of the method declaration is as follows:

```
new ResultType MethodName
  ( param:Type ;  ...;  param:Type  )
  VarName:Type ;  ...;  VarName:Type ;
begin
  Instr;
  ...
  Instr;
end;
```

or

```
abstract ResultType MethodName
  ( param:Type; ...; param:Type );

   or

(implement / override) ResultType Mixin.Method
  ( param:Type; ...; param:Type )
  VarName:Type; ...; VarName:Type;
begin
  Instr;
  ...
  Instr;
end;
```

The keywords occurring on the very beginning of every method declaration have the following meaning: The keyword `new` denotes that the given method declaration is an introduction of a new method identifier together with its implementation, in the form of the method body. The keyword `abstract` denotes that the given method declaration is an introduction of a fresh method identifier, however without supplying its body.

Keywords `implement` and `override` denote that this method declaration only supplies a new implementation of method declared in another mixin. The mixin containing the introduction of the implemented method identifier needs to be one of the mixins present in the base mixin expression of the given mixin. Such a declaration of a method does not introduce a new identifier, and therefore does not change the "interface" of an object.

We use the keyword `implement` when the method implementation is intended to be an independent implementation of the given method. It can be used to supply the implementation of an abstract method declared in one of the base mixins. As a result, the body of such a method declaration cannot refer to the "previous" method implementation using `super(...)` call, because it is not guaranteed that such implementation exists.

On the other hand, we use the keyword `override` to denote that the given implementation is intended as an extension of the existing implementation of the given method. As a result, the body of such a method declaration starting with `override` can use the special `super(...)` call expression (see Section 4.5.4), which refers to existing overridden implementations. On the other side, a mixin containing an `override` method declaration can only be combined with mixins containing other implementations of the given method.

4.3.3 Syntax of an initialization module declaration

Each initialization module declaration consists of:

- keyword `required` or keyword `optional`;
- the name of the mixin within which the initialization module has been declared;

- a list of the *input parameter* declarations contained within parentheses and separated by semicolons. Each such input parameter declaration consists of a parameter name, a colon and a mixin expression representing the formal type of given input parameter.

- keyword `initializes`;

- the list of *output parameter* declarations contained within parentheses and separated by semicolons. Each output parameter declaration consists of: the name of mixin, dot character and the name of the output parameter.

  The output parameter declaration of the form $M1.P1$ refers to an input parameter $P1$ declared within some ini module of mixin $M1$.

- the body of the initialization module. Each such body is similar to a method body and consists of: (*i*) a list of local variable declarations; (*ii*) the keyword `begin`; (*iii*) a list of instructions separated by semicolons, containing exactly one `super[`*par1* `:=` exp_1`, ...]` instruction (see below); (*iv*) keyword `end`.

  The `super[`*par1* `:=` exp_1`, ...]` instruction used within the body of an initialization module cannot be contained within some nested instruction like `if` or `while`. It must be one of the top-level instructions. Thus, it is guaranteed that: if the module finishes, this instruction was executed exactly once.

  This instruction consists of the `super` keyword and the square brackets containing a list of *parameter assignments* separated by the semicolon. The parameter assignments list contains exactly one parameter assignment for each output parameter declaration of the ini module within which it occurs. Each parameter assignment corresponding to the output parameter declaration *p* consists of three parts: (*i*) mixin name, dot character and the parameter name of the output parameter *p*; (*ii*) assignment symbol (`:=`); (*iii*) an *expression*.

Thus the formal syntax of an initialization module is as follows:

```
(required / optional ) Mixin (Iparam1: TypeI1; ...; IparamN: TypeIN)
    initializes (Mixin1.Oparam1; ...; MixinM.OparamM)
   VarName: Type ; ...  VarName: Type ;
begin
   Instr; ...
   super[Mixin1.Oparam1 := exp; ...; MixinM.OparamM := exp];
   Instr; ..
end;
```

Each declaration of an initialization module describes two aspects (see also Section 3.8 for description with examples):

- whether the given module is mandatory to be executed when the object is created from the given mixin (then it begins with the `required` keyword) or is optional (`optional`);

- what set of information needs to be supplied to a mixin during the object creation in order to execute the given initialization module. The set of information to be supplied is declared in the form of input parameters. We say that a given set of parameters is *supplied* in the given object creation expression if each parameter `m1.p1` in this set is either: (i) supplied directly in the expression, which means that `m1.p1 :=` *exp* occurs in the expression, or (ii) supplied indirectly, which means that it occurs within the declarations of the output parameters of another module whose parameters have been supplied;

- what further initialization information are computed by the given module when it is executed. When a new object is initialized using that module, those informations are supplied to other modules in the form of the values of their input parameters. The list of input parameters of other modules computed by the given module is declared in the form of output parameters of the given module. Type system enforces that for each object creation expression, and each initialization module used during that object creation, if the input parameters of that module are supplied, then it supplies its output parameters to input parameters of other modules and those input parameters cannot be supplied by any other means. Those parameters cannot be supplied neither by output parameters of other executed modules, nor directly in the object creation expression;

- what instructions are executed when the input parameters of module are supplied. Those instructions are declared in the form of the body of initialization module.

The execution of the `super[`*Mixin1*`.`*Oparam1* `:=` *exp*`; ...]` instruction calls the next activated initialization module (see below for a detailed definition). Furthermore, the execution of that instruction supplies values of output parameters (*Mixin1 . Oparam1...*) of that module, by evaluating expressions assigned to them. Those expressions are evaluated just before the control is passed to next initialization modules in order to supply their values of input parameters.

One of the things that distinguishes the `super[...]` instruction in an ini module from the `super(...)` expression used within overriding method bodies is the fact that the parameters supplied in `super[...]` do not have to be the parameters which will be passed to the module executed by this `super[...]` instruction. Those parameters can be in fact passed to other modules which will be executed later on. To understand this phenomenon better, consider the example on Figure 3.8. The `super[...]` instruction placed in the module `mod3` supplied values of three initialization parameters and calls the module `mod2`. However only one of these parameters (`z`) is supplied to the called module `mod2`, while the remaining parameters (`x`, `y`) are kept to be supplied to next modules. On the other hand, `super[...]` instruction placed within the module `mod2` does not contain any parameters, however it calls module `mod1` which takes 2 parameters which have been computed by module `mod3`. The detailed semantics of those instructions is described in Section 7.

According to the procedure introduced in Section 3.8, the *next activated ini module* called by `super[...]` is a module chosen in the following way: The execution of the `super[...]`

instruction starts from evaluating the values of expressions present within the instruction. Then the values of those expressions (together with identifiers of their corresponding parameters) are added to the set of all initialization parameters supplied for the initialization of the current object. Note that, at this point, the set can also contain some other parameters, when not all parameters supplied so far have been consumed by the current initialization module. Then, the next activated module is chosen by traversing the list of ini module declarations present within the mixin in which the current ini module is declared, occurring textually above the declaration of the current ini module (see Section 3.8). The module to be executed is chosen as the first one such that all of its input parameters are contained in the extended set of parameters. If we cannot find such module in the current mixin, then the search continues amongst the ini modules declared in the next mixin. The next mixin is chosen from those mixins in the sequence of mixins from which the object is created, which precede the currently analyzed mixin. The mixins are scanned bottom-up for matching modules and if there is no such module found then the `super[...]` instruction does not call any module.

4.4 Syntax of instructions

We have the following kinds of instructions, similar to analogous instructions in mainstream object oriented languages. Those instructions correspond to nonterminal `INSTRUCTION` in the BNF grammar in Section 5.

- assignment of a value of an expression to a local identifier. It has the following syntax:
  VarName`:=` *exp*

- assignment of a value of an expression to a field of a target object. It has the following syntax:
  `this.`*Mixin*`.`*Field* `:=` *exp*

- `return` instruction, having the following syntax:
  `return` *exp*

- loop instruction, having the following syntax:
  `while (`*exp*`)` *Instr* `end`

- conditional instruction, having the following syntax:
  `If (`*exp*`) then` *Instr* `else` *Instr* `end`

- expression evaluation instruction, having the following syntax:
  exp

- call to next initialization module (occurring within some initialization module), having the following syntax:
  `super[` *Mix*`.`*par* `:=` *exp*`, ...]`
  This instruction is described in details in Section 4.3.3.

4.5 Syntax of object expressions

Each object expression[1] has one of the below forms. All those expressions correspond to nonterminal `EXPRESSION` in the BNF grammar in Section 5.

- one of the three constants: `null`, `true` or `false`;
- local identifier, which is either:
  - a local variable name,
  - a method parameter name,
  - an input parameter in the ini module declaration;
  - or identifier `this`;
- field dereference, as described in Section 4.5.1;
- object creation, as described in Section 4.5.2.
- ordinary method call, as described in Section 4.5.3;
- `super(`*exp, ..., exp*`)` method call, as described in Section 4.5.4.

4.5.1 Syntax of a field dereference

Each field dereference expression consists of:

- expression, denoting the target object, followed by the dot character;
- the name of the mixin, followed by the dot character;
- the name of the field to be picked from the target object.

4.5.2 Syntax of an object creation

Each object creation expression consists of:

- the keyword `new`;
- a mixin expression, which denotes a sequence of mixins from which the object will be created;
- a sequence of parameter assignments (see below) included in square brackets and separated by commas. Each parameter assignment consists of a mixin name, a parameter name and an expression. It has the following form: $Mix.Par$ `:=` $Expr$.

[1]We use the notion "object expression" for all expressions which denote first class values, to distinguish them from mixin expressions.

Therefore, a formal syntax of an object creation expression is the following:

`new` *MixinExpr* `[`*Mixin1*`.`*parName1* `:=` exp_1`, ..., `*MixinN*`.`*parNameN* `:=` exp_n`]`

Within each parameter assignment, the expression denotes the value of the input parameter, which will be supplied to an initialization module within which the given input parameter was declared (see Section 3.8).

The evaluation of such an object creation expression proceeds as follows. First, the memory is allocated for a new object created from the sequence of mixins *MixinExpr*. In other words, the allocated memory holds the state of an object, which means non-initialized (*null*) values of all fields declared in mixins *MixinExpr*, together with its runtime type (being a sequence of mixin names from which the object was created). Then, the expressions exp_1... exp_N are evaluated. Next, depending on the set of parameters supplied directly (*Mixin1*`.`*parName1*, ..., *MixinN*`.`*parNameN*) the sequence of the initialization modules to execute is chosen. Finally the initialization modules are executed, taking their input parameter values from the results of evaluation of exp_1... exp_n.

The algorithm of the choice and execution of initialization modules works in the following way:

- It picks the set X of all parameters supplied directly within the object creation expression.
- The sequence of mixins *MixinExpr* is processed from the right to the left.
- For each processed mixin, its declaration is searched for the initialization modules. Initialization modules found are traversed from the bottom to the top of the mixin declaration (i.e., from the last declaration to the first one).
- Each found module is executed if and only if all its input parameters occur in the set X.
- If the module is executed, then its set of input parameters is removed from X, and all the output parameters of that module are added to X, together with their values evaluated by `super[...]` expression.

For every such object creation expression, the set of parameters supplied within this expression is verified during the compilation, to check if:

- During the execution of the given expression, all the modules marked as `required` will be executed. This condition refers to `required` modules of mixins used in the given object creation expression.
- At the end of the execution of that process no not-consumed parameters will be left. In other words, it is verified if at the end of execution of the above algorithm the set X will be empty.

4.5.3 Syntax of a method call

Each ordinary method call expression consists of:

- An object expression (as defined in Section 4.5), which denotes the object which is the target of the method call. In other words, this expression represents the object on behalf of which the method will be executed. This expression is followed by the dot character;
- Mixin name, followed by the dot character;
- Method name.
- A sequence of object expressions separated by commas and enclosed in parentheses. Those expressions denote values of the actual parameters of the method. Therefore the number of those expressions is equal to the number of formal parameters of the method.

Therefore the formal syntax of an ordinary method call is the following:
  exp `.` *Mix* `.` ***mt*** `(` ***exp***$_1$ `, ..., ` ***exp***$_n$ `)`
We use the notion *ordinary method call* to refer to the above defined method call as opposed to the `super(...)` method call (see Section 4.5.4). For every such expression to be type correct, three conditions must be fulfilled:

- It needs to be guaranteed that the value of a "method target" (denoted by ***exp***) was created from a sequence of mixins containing mixin *Mix*.
- The declaration of mixin *Mix* needs to contain an introduction of the method named ***mt***, which means a declaration which begins with the keyword `new` or `abstract`.
- This method declaration needs to contain a declaration of a list of parameters with number and types matching the types of expressions *exp*$_1$ `, ..., ` *exp*$_n$.

4.5.4 Syntax of a `super` method call

Each `super` method call expression consists of:

- A `super` keyword.
- A sequence of object expressions separated by commas and enclosed in parentheses. Those expressions are used to evaluate values of the actual parameters of the method. Their number is equal to the number of formal parameters of the method.

Therefore the formal syntax of a `super` method call is the following:
  `super(` *exp*$_1$ `, ..., ` *exp*$_n$`)`
For every such expression to be type correct, two conditions must be fulfilled:

- It needs to be placed within the body of the method declaration marked with the `override` keyword.

- This method declaration needs to contain a declaration of a list of parameters with number and types matching the types of expressions exp_1, ..., exp_n.

4.6 Mixin expression syntax

Each mixin expression is a nonempty sequence of mixin names separated by commas.

Mixin expressions are used in two different roles:

- As type expressions, denoting types of declared fields, variables, return types of methods, types of method parameters etc.
- As expression denoting mixins from which the new object is created. In this context mixin expressions play the role of classes in classical OO languages.

Chapter 5

Formal definition of the syntax

Below we present the formal syntax of Magda in the Backus-Naur form. To increase the readability, all keywords (terminal symbols) in grammar are written in non-capital letters and in quotes, like: `'implements'`. One general non-terminal symbol (denoting all acceptable identifiers) is denoted as `<ID>`. Two additional non-terminals are `<STRING_LITERAL>` and `<NUMBER_LITERAL>`, which denote any constant values which can be used in a program, representing strings and numbers respectively. All nonterminals are written in capital letters with underscores.

```
PROGRAM ::= ( MIXIN_DECLARATION )* INSTRUCTIONS
```

Figure 5.1: Formal grammar of program

```
FIELD        ::= 'this' '.' <ID> '.' <ID>
VARIABLE     ::= <ID>
LVALUE       ::= FIELD | VARIABLE
WHILE_LOOP   ::= 'while' '(' EXPRESSION ')' INSTRUCTIONS 'end'

IF_COND      ::= 'if' '(' EXPRESSION ')' 'then' INSTRUCTIONS
                 [ 'else' INSTRUCTIONS ] 'end'

INSTRUCTION  ::= LVALUE ':=' EXPRESSION |
                 'return' EXPRESSION    |
                 WHILE_LOOP             |
                 IF_COND                |
                 EXPRESSION             |
                 INSTRUCTION ';' INSTRUCTION

INSTRUCTIONS ::= INSTRUCTION | ϵ
```

Figure 5.2: Formal grammar of instructions

```
ACTUAL_PARAMETERS  ::= '('  [ EXPRESSION (',' EXPRESSION )* ] ')'

METHOD_CALL        ::= EXPRESSION '.' <ID> '.' <ID> ACTUAL_PARAMETERS
FIELD_SELECT       ::= EXPRESSION '.' <ID> '.' <ID>
EXPRESSION         ::= 'this' | 'null' | 'false' | 'true ' |
                       <STRING_LITERAL> |
                       <NUMBER_LITERAL> |
                       <ID> |
                       FIELD_SELECT |
                       OBJECT_CREATION |
                       METHOD_CALL |
                       'super' ACTUAL_PARAMETERS |
                       '(' EXPRESSION ')'

PARAM_ASSIGNMENT   ::= <ID> '.' <ID> ':=' EXPRESSION
PARAM_ASSIGNMENTS  ::= [ PARAM_ASSIGNMENT (',' PARAM_ASSIGNMENT )*]
OBJECT_CREATION    ::= 'new' MIXIN_EXPRESSION '[' PARAM_ASSIGNMENTS ']'

MIXIN_EXPRESSION   ::= '(' MIXIN_EXPRESSION ')' |
                      <ID> (',' ( <ID>  | '(' MIXIN_EXPRESSION ')' ) )*
```

Figure 5.3: Formal grammar of expressions

```
OUTPUT_PARAMETER      ::= <ID> '.' <ID>
OUTPUT_PARAMETERS     ::= '('[ OUTPUT_PARAMETER (',' OUTPUT_PARAMETER)* ] ')'
PARAMETER_DECL        ::= <ID> ':' MIXIN_EXPRESSION
PARAMETER_DECLS       ::= '(' [ PARAMETER_DECL (';' PARAMETER_DECL )* ] ')'

VARIABLE_DECLARATIONS ::=  ( <ID> ':' MIXIN_EXPRESSION ';' )*

MODULE_SUPER_CALL     ::= 'super' '[' PARAMS_ASSIGNMENTS ']' ';'

INI_MODULE_BODY       ::= VARIABLE_DECLARATIONS
                            'begin'
                              INSTRUCTIONS
                              MODULE_SUPER_CALL
                              INSTRUCTIONS
                            'end'

INI_MODULE_DECLARATION ::=
    ( 'required' | 'optional' ) <ID>
     PARAMETER_DECLS 'initializes' OUTPUT_PARAMETERS
     INI_MODULE_BODY

METHOD_BODY          ::= VARIABLE_DECLARATIONS
                            'begin'
                              INSTRUCTIONS
                            'end'

METHOD_DECLARATION   ::=
   'abstract'  MIXIN_EXPRESSION <ID> PARAMETER_DECLS |
   'new'       MIXIN_EXPRESSION <ID> PARAMETER_DECLS METHOD_BODY |
   'override'  MIXIN_EXPRESSION <ID> '.' <ID>  PARAMETER_DECLS METHOD_BODY |
   'implement' MIXIN_EXPRESSION <ID> '.' <ID>  PARAMETER_DECLS METHOD_BODY

FIELD_DECLARATION ::= <ID> ':' MIXIN_EXPRESSION

MIXIN_DECLARATION ::= 'mixin' <ID> 'of' MIXIN_EXPRESSION '='
                      ( FIELD_DECLARATION     ';' |
                        METHOD_DECLARATION    ';' |
                        INI_MODULE_DECLARATION';'
                      )* 'END'
```

Figure 5.4: Formal grammar of mixin declarations

Chapter 6

Syntax-related definitions

In this section we introduce some syntax related notions which will make the forthcoming definitions of semantics and type checking rules easier to specify and understand.

To do this, we fix some program P consisting of a sequence of mixin declarations and a sequence of main instructions. From now on, if not stated otherwise, when we refer to the set of declarations it will mean the set of declarations occurring in program P.

6.1 Notation

Here we introduce some notational conventions which will be used through the remainder of this thesis.

For each pair p we use $p|_1$ to denote the first element of p, and $p|_2$ to denote the second element of p. In general, for an arbitrary tuple t we use $t|_n$ to denote the n-th element of t.

We say that f returns b when applied to a, when $f(a) = b$. For each function or partial function f we use $f\{a \mapsto b\}$ to denote a function which when applied to x has value b when $x = a$ and $f(x)$ otherwise. In other words $f\{a \mapsto b\}$ is a function which differs from f only on position a. We also use $f\{a_1 \mapsto b_1; ...; a_n \mapsto b_n\}$ to denote $f\{a_1 \mapsto b_1\}...\{a_n \mapsto b_n\}$, when all a_1 ... a_n are different elements.

We use also the convention that every function f is a set of all pairs (a, b), such that $f(a) = b$.

For any two sets A and B, such that $A \subseteq B$, we will use $A \setminus^* B$ to denote $A \setminus B$. However $A \setminus^* B$ is not defined when $B \subseteq A$ does not hold.

A set of elements $\{p_1, ..., p_n\}$ is often denoted as $\overline{p}$. Additionally, a sequence of elements $(p_1, ..., p_n)$ is occasionally abbreviated as $\overrightarrow{p}$. We use notation $a \cdot \overrightarrow{b}$ to denote a sequence $\overrightarrow{b}$ with element a added at the beginning. Similarly, we use notation $\overrightarrow{b} \cdot a$ to denote a sequence $\overrightarrow{b}$ with element a added at the end.

For simplicity, we often treat a sequence as the set of its values.

6.2 Identifiers

We use the below notions for the sets of names and identifiers occurring in program P:

- *MixNames* — the set of names of all mixins declared in the program plus the names of the built-in mixins (at the moment we assume that there exists only one built-in mixin — `Boolean`). We use symbol *Mix*, M or m, with some optional indices, to denote an element of this set.
- *FieldNames* — the set of all names of object fields declared within the program. We use symbol *fl*, with some optional indices, to denote an element of this set.
- *FieldIDs* — the set of all identifiers of object fields declared in the program. Each identifier of an object field is a pair: the mixin name and the field name, where the mixin name is the name of the mixin within which the field is declared.
- *ParamIDs* — the set of all identifiers of input parameters declared in the program. Each input parameter identifier is a pair: a mixin name and an input parameter name. In every such identifier, the mixin name is the name of the mixin containing the declaration of the ini module with the given input parameter.
- *MethodNames* — the set of all names of the methods declared in the program. We use the symbol *mt*, with some optional indices, to denote an element of this set.
- *MethodIDs* — the set of all method identifiers of the program. Each method identifier is a pair consisting of a mixin name and a method name. Each pair $(\mathit{Mix}, \mathit{mt})$ in this set corresponds to a declaration of method *mt* present in mixin *Mix*, such that this declaration begins with the keyword `new` or `abstract`.
- *LocalIdentifiers* — the set of all locally visible identifiers (inside the bodies of methods and modules). This set contains `this` identifier, local variables declared at the beginning of the bodies and all names of parameters of methods and initialization modules. We use v with some indices to denote a local variable, and p or *par* with some optional indices to denote a parameter name.

6.3 Mixin declarations

For each $m \in \mathit{MixNames}$ we use:

- MixinD_m to denote the declaration of the mixin m within the program P.
- $\mathit{Methods}_m$ to denote the set of all method declarations present within MixinD_m.
- $\mathit{IModules}_m$ to denote the set of all ini module declarations present within MixinD_m.
- Fields_m to denote the set of all field declarations present within MixinD_m.

- $\mathit{MethodIDs}_m$ to denote the set of all method identifiers of the method declarations present within MixinD_m.

- $\mathit{IntrMethods}_m$ to denote the set of all names of methods introduced in mixin m. In other words, it is a set of names of method declarations present in $\mathit{Methods}_m$, such that the declaration of the method begins with the keyword `new` or `abstract`.

- $\mathit{ImplMethodIDs}_m$ to denote the set of all method identifiers of the declarations present within $Methods_m$, which do not begin with the keyword `abstract`.

For each $m \in \mathit{MixNames}$ and $(m2, mt) \in \mathit{MethodIDs}_m$ we use:

- $\mathit{MetDecl}_m^{(m2,mt)}$ to denote the declaration of the method $(m2, mt)$ in mixin m.

- $\mathit{RetType}_m^{(m2,mt)}$ to denote the return type occurring in $MetDecl_m^{(m2,mt)}$.

- $\mathit{MetParams}_m^{(m2,mt)}$ to denote the sequence of declarations of method parameters in $\mathit{MetDecl}_m^{(m2,mt)}$. Each element of this sequence is a parameter name followed by a colon and a mixin expression denoting the type of the given parameter.

- $\mathit{MetSpec}_m^{(m2,mt)}$ to denote the method specifier (see Section 4.3.2) occurring in the declaration $\mathit{MetDecl}_m^{(m2,mt)}$, i.e., one of the four keywords: `new`, `abstract`, `override`, or `implement`.

For each $m \in \mathit{MixNames}$ and $(m2, mt) \in \mathit{ImplMethodIDs}_m$ we use:

- $\mathit{MetLocals}_m^{(m2,mt)}$ to denote the sequence of local variable declarations in $\mathit{MetDecl}_m^{(m2,mt)}$.

- $\mathit{MetInstr}_m^{(m2,mt)}$ to denote the sequence of instructions within $\mathit{MetDecl}_m^{(m2,mt)}$ with one instruction added at the end: `return null;`. This instruction is added to ensure that every method returns `null` if it happens not to execute any `return` instruction (as described in Section 3.1).

To explain better the meaning of the above notions, we present below their values for the program used as an example on Figure 3.2 in Section 3:

$$\begin{aligned}
\mathit{Methods}_{\texttt{Point2D}} &= \{\texttt{new Object setCoords ...end}, \texttt{new Integer getX ...end}\}\\
\mathit{IModules}_{\texttt{Point2D}} &= \emptyset\\
\mathit{Fields}_{\texttt{Point2D}} &= \{\texttt{x:Integer, y:Integer}\}\\
\mathit{MethodIDs}_{\texttt{Point2D}} &= \{(\texttt{Point2D}, \texttt{setCoords}), (\texttt{Point2D}, \texttt{getX})\}\\
\mathit{IntrMethods}_{\texttt{Point2D}} &= \{\texttt{setCoords, getX}\}\\
\mathit{ImplMethodIDs}_{\texttt{Point2D}} &= \{(\texttt{Point2D}, \texttt{setCoords}), (\texttt{Point2D}, \texttt{getX})\}
\end{aligned}$$

$$\begin{aligned}
MetDecl^{(\texttt{Point2D, setCoords})}_{\texttt{Point2D}} &= \texttt{new Object setCoords... end}\\
RetType^{(\texttt{Point2D, setCoords})}_{\texttt{Point2D}} &= \texttt{Object}\\
MetParams^{(\texttt{Point2D, setCoords})}_{\texttt{Point2D}} &= (\texttt{ax:Integer}, \texttt{ay:Integer})\\
MetSpec^{(\texttt{Point2D, getX})}_{\texttt{Point2D}} &= \texttt{new}\\
MetLocals^{(\texttt{Point2D, getX})}_{\texttt{Point2D}} &= ()\\
MetInstr^{(\texttt{Point2D, getX})}_{\texttt{Point2D}} &= \texttt{return this.Point2D.x; return null;}
\end{aligned}$$

6.4 Function *IniModules*

The function *IniModules* takes as an argument a sequence of mixin names. When applied to a sequence $\overrightarrow{M}$, this function returns a sequence with one element for each initialization module declaration *mod* occurring in mixins whose names are in $\overrightarrow{M}$. Each such element of $IniModules(\overrightarrow{M})$ is a tuple consisting of:

1. The sequence of input parameter identifiers occurring in *mod*. Each parameter name within this sequence is prefixed with the name of mixin in which the module was declared.

2. The sequence of output parameter identifiers occurring in *mod*, accompanied with their types. Each element of this sequence is a pair of: (*i*) a parameter name prefixed with the target mixin name (as declared in the module) and (*ii*) its type. The type of the output parameter comes from the declaration of the module in which this parameter was introduced as its input parameter (not from the module declaration in which it was used as an output parameter).

3. The body of the ini module *mod*.

4. The name of the mixin within which *mod* has been declared.

Therefore, each such tuple is an element of the set *ModDecls* (see the definition below).

The elements of the sequence $IniModules(\overrightarrow{M})$ occur in the order of mixins in $\overrightarrow{M}$, and within modules from one mixin, in the order of textual ordering of the modules within this mixins (see Section 3.8).

This implies that all the ini modules from the first mixin within the sequence $\overrightarrow{M}$ will have its representants placed before the ones of the second mixin, and when one mixin declaration contains two module declarations, the representant of the ini module placed lower within the mixin declaration, will be placed later in the sequence $IniModules(\overrightarrow{M})$.

Therefore, we have:

$$\begin{array}{rcl} \mathit{IniModules} & : & \mathit{MixNames}^* \to \mathit{ModDecls}^* \\ \mathit{ModDecls} & = & \mathit{ParamIDs}^* \times \mathit{TypedParamIDs}^* \times \mathit{ModuleBodies} \times \mathit{MixNames} \\ \mathit{TypedParamIDs} & = & \mathit{ParamIDs} \times \mathit{Types} \end{array}$$

where *ModuleBodies* is the set of all possible bodies of bodies of initialization modules, which means, all sequences of terminal symbols derivable from the `INI_MODULE_BODY` nonterminal (see Section 5).

6.5 Functions *base* and *baseExt*

The function *base* takes as a argument a non-empty sequence of mixin names and returns a set of mixin names. This function, when applied to a sequence consisting of one mixin name, returns the set of mixins included in the base mixin expression used in the declaration of the given mixin. Therefore, in the example on Figure 3.4 we have:

$$\begin{array}{c} \mathit{base}((\texttt{Point3D})) = \{\texttt{Point2D}\} \\ \mathit{base}((\texttt{Displayable3DColorPoint})) = \{\texttt{DisplayableObject}, \texttt{Point3D}, \texttt{ColorPoint}\} \end{array}$$

The function *base* when applied to a sequence $\overrightarrow{X}$ of mixin names returns the set containing all the base mixins of all mixins in $\overrightarrow{X}$ except the names of mixins occurring in $\overrightarrow{X}$. In other words, the *base* function can be defined recursively in the following way:

$$\mathit{base}(\mathit{Mix} \cdot \overrightarrow{\mathit{tail}}) = \mathit{base}((\mathit{Mix})) \cup \mathit{base}(\overrightarrow{\mathit{tail}}) - (\{\mathit{Mix}\} \cup \overline{\mathit{tail}})$$

In the example on Figure 3.4 we have:

$$\mathit{base}((\texttt{Displayable3DColorPoint}, \texttt{Point3D})) = \{\texttt{DisplayableObject}, \texttt{ColorPoint}, \texttt{Point2D}\}$$

For every set of mixin names $\overline{\mathit{mixins}}$ we also define function *baseExt* as follows:

$$\mathit{baseExt}(\overline{\mathit{mixins}}) = \overline{X} \cup \bigcup_{\mathit{Mix} \in \overline{\mathit{mixins}}} \mathit{base}((\mathit{Mix}))$$

6.6 Mixin sequence manipulation: *LastMix*, *LastMixBef*

Below we define two partial functions: *LastMix* and *LastMixBef*. Those functions take as their parameters a sequence of distinct mixin names and an identifier of a method. The second function has one additional parameter: a mixin name.

Each of those functions, when applied to a sequence of mixins $\overrightarrow{\mathit{mixins}}$ and method identifier *mtID*, returns a mixin name occurring in $\overrightarrow{\mathit{mixins}}$ such that the declaration of that mixin contains an implementation of method *mtID*. As a result, those functions are defined only for such values of $\overrightarrow{\mathit{mixins}}$ which do contain an implementation of method *mtID*.

The functions are defined in the following way:

- *LastMix* — takes as parameters a sequence of distinct mixin names $\overrightarrow{mixins}$ and the method identifier *mtID*. As value of $\mathit{LastMixBef}(\overrightarrow{mixins}, \mathit{mtID}, \mathit{mix})$ is the last mixin name in the sequence $\overrightarrow{mixins}$, such that the declaration of this mixin contains an implementation of the method *mtID*.

- $\mathit{LastMixBef}(\overrightarrow{mixins}, \mathit{mtID}, \mathit{mix})$ — similarly to the above function takes as parameters the sequence $\overrightarrow{mixins}$, and the method identifier *mtID*. However it takes one additional parameter: mixin name *mix*. A value of $\mathit{LastMixBef}(\overrightarrow{mixins}, \mathit{mtID}, \mathit{mix})$ is the last mixin name in $\overrightarrow{mixins}$ preceding *mix*, such that the declaration of returned mixin contains an implementation of the method *mtID*.

Therefore, for these functions we have:

$$\begin{aligned} \mathit{LastMix} &: \mathit{MixNames}^* \times \mathit{MethodIDs} \rightharpoonup \mathit{MixNames} \\ \mathit{LastMixBef} &: \mathit{MixNames}^* \times \mathit{MethodIDs} \times \mathit{MixNames} \rightharpoonup \mathit{MixNames} \end{aligned}$$

These functions are defined in order to choose the implementation of method *mtID* during the dynamic dispatch of the method call. The first one is used during the ordinary method call, while the second is used in the `super(...)` method call, as defined in the operational semantics rules in Section 7.2.5.

6.7 A consistent mixin sequence

We say that the sequence of mixin names Mix_1, ..., Mix_n *is consistent* when the following conditions are met:

- For each method declared as `abstract` in a mixin within the sequence, there exists another mixin in the sequence, which contains the implementation of the method marked as `abstract`.

  This condition can be expressed in the following way:

$$\bigcup_i \mathit{MethodIDs}_{Mix_i} = \bigcup_i \mathit{ImplMethodIDs}_{Mix_i}$$

- For each `override` declaration of a method in a mixin occurring in the sequence, there exists another mixin placed earlier in this sequence, which contains the declaration of the same method marked with `new` or `implement` keyword. As a result this guarantees that a `super(...)` call placed in such `override` method will have another implementation to call. This is formalized as the below condition:

$$\forall_{mtID,i} \quad (MetSpec^{mtID}_{Mix_i} = \texttt{override} \quad \Rightarrow \quad \exists_{j<i}\, MetSpec^{mtID}_{Mix_j} \in \{\texttt{new}, \texttt{implement}\})$$

- this sequence does not contain any name of a built-in mixin (like `Boolean`).

In short, the fact that a sequence of mixins is consistent means that this sequence can be safely used in an object creation expressions to create an object from (see Section 8.6). The above definition guarantees the following property.

Property 2 (Consistent mixin sequence) *For each consistent mixin sequence Mix_1, ..., Mix_n two conditions hold:*

- $LastMix(\,(Mix_1, ..., Mix_n), mtID)$ is defined for each $mtID \in \bigcup_i MethodIDs_{Mix_i}$, and
- $LastMixBef(\,(Mix_1, ..., Mix_n), mtID, Mix_j)$ is defined for each $mtID \in \bigcup_i MethodIDs_{Mix_i}$ and for each j, such that $MetSpec^{mtID}_{Mix_j} = \texttt{override}$

Chapter 7

Formal semantics

In this section we present the formal semantics of Magda. To do this, we use the big-step operational semantics formulation (as presented by Kahn in [43]), which as many people believe is a more intuitive choice than small-step semantics [51, 41, 5, 4]. The superiority of such formulation is especially visible in imperative languages with recursion, nested expressions etc. Moreover, a small-step semantics formulation (as introduced by Plotkin [59]) of such a language in many cases requires one to define: (*i*) a second language used internally in the reduction process; (*ii*) some additional rules which enforce a specific order of the evaluation. An example of such small-step formulation of the semantics of an OO language can be found in [36].

Therefore we have chosen to describe Magda semantics in a more intuitive human-readable way. We believe that readability is an important aspect of the formal definition of semantics. This choice of big-step semantics shows its drawbacks when we state the type soundness theorem, in Section 9.1. Luckily those problems can be dealt with, as explained in Section 9.2 and Section 11.

The below definitions describe only successful executions. Therefore, those rules do not model any errors. In particular there are no rules which express null-pointer dereference errors. For each program which runs into such an error, in our semantics there is no derivation for any judgment saying that such program evaluates to some state. Similarly, there is no derivation for a program which runs into the point in which there is an attempt to execute a method on the object which does not support this method. There is also no derivation for programs which attempt to dereference a field of object which does not have such field. However, the programs which would suffer from the last two cases, will not pass the type checking procedure. On the other hand, similarly as with other languages, type checking does not prevent null-pointer dereference errors.

We assume also an infinite amount of memory available. Every object allocated stays in the memory forever. The implementation of Magda uses some form of garbage collection, however such garbage collection, under the assumption of an infinite amount of memory should be transparent, except for some form of finalization of external resources, as present in Java, however not in our core part of Magda. Recall also that we assume a fixed program P (see Section 6) and we implicitly refer to its elements where needed.

We believe that all the above choices will make the semantics easy to understand and make it easy to analyze its formal properies.

7.1 Preliminary definitions

7.1.1 Representation of the memory state

We begin describing the semantics of Magda, by presenting how the dynamic memory state and the object values are represented in this semantics.

To do this we use the below defined sets:

- *Addresses'* is a fixed, infinite, linearly ordered set of *memory addresses*. Each address is used later on (in the definition of the set *States*) to point to an object value. There are two distinguished addresses: tt, ff which will be later used to point to boolean values. This set does not contain a *null* address. We may assume that this set is the set of all natural numbers, $ff = 0$ and $tt = 1$.

- *Addresses* is the set of all possible values of expressions. The value of each expression can be either an address of an object or a *null* value. As a result we define this set by extending *Addresses'* set with one element: *null*. We will use the symbol *adr* with optional indices to denote an element of this set.

- *Objects* is the set of all possible object values. Every object is a pair of:

  - The sequence of names of mixins from which it was build. This sequence is also called the *runtime type* of the object. Notice that in the context of the type system the ordering of mixins is insignificant and we interpret types as sets (see Section 3.9). On the other hand, the result of the execution of a program can be influenced by the reordering of mixins in the runtime types of objects, as described in Section 3.6).
  - The current state of the object, which means the values of all its local fields. This state is a partial function associating field identifiers with addresses.

  One special object value $((\texttt{Boolean}), \emptyset)$ will be denoted later on as *BoolVal*.

- *States* is the set of all possible *states* of the memory. Each state is a partial function, which associates addresses with some object values. Every element of this set represents a global memory state (called sometimes heap). We will use the symbol *st* with optional indices to denote an element of this set.

  Additionally, the rules of our semantics enforce that all states occurring in the during the actual execution of a type correct program are always defined for two special addresses: tt and ff. Moreover, the value of each state at these addresses is *BoolVal*.

- *Environments* is the set, in which each element represents the state of all local identifiers visible in the current scope: the current method or current initialization module. Therefore, each element of the set *Environments* takes one of two below forms:
  - Either an association of local identifier names with addresses in memory. It has, as its domain, names of identifiers defined in the given scope, therefore it is a partial function.
  - Or a pair of the form $(\top, adr)$, where *adr* is an element of *Addresses* set. This means that the execution of the current method is finished, as a result of a `return` statement. Address *adr* is the result of a method evaluated by this `return` statement. This form is never used during the execution of an initialization module.

  We will use the symbol *env* with optional indices to denote an element of this set.
- *ParamValues* is the set, in which each element represents the set of values of initialization parameters supplied during the creation of an object, and processed during the object initialization. Therefore, each element of this set is a partial function assigning addresses to identifiers of parameters.

Therefore, formally, the above sets are defined as follows:

$$\begin{array}{rcl}
\mathit{Addresses'} & = & \mathbb{N} \\
\mathit{Addresses} & = & \mathit{Addresses'} \cup \{\mathit{null}\} \\
\mathit{Objects} & = & \mathit{MixNames}^* \times (\mathit{FieldIDs} \rightharpoonup \mathit{Addresses}) \\
\mathit{States} & = & \mathit{Addresses'} \rightharpoonup \mathit{Objects} \\
\mathit{Environments} & = & (\mathit{LocalIdentifiers} \rightharpoonup \mathit{Addresses}) \cup (\{\top\} \times \mathit{Addresses}) \\
\mathit{ParamValues} & = & \mathit{ParamIDs} \rightharpoonup \mathit{Addresses}
\end{array}$$

7.1.2 Representation of the static execution context

To model the general state of execution, apart from defining the state of objects, variables etc (as we did in the above section), we need also to define how to represent *static context of execution.* By static context we mean the information about the part of the static program structure, within which the actually executed instruction or expression is placed. In other words, this information represents a "pointer" pointing to some block of instructions like a method declaration or an initialization module declaration.

In order to describe the static context of execution we will use an element of set *Contexts*, as defined below:

- *Contexts* — the set contains all available static contexts used during the execution of instructions or expressions. Each static context refers to a method or an initialization module which is currently executed. Therefore it takes one of the following forms:
  - a pair: a name of a mixin and an identifier of a method, or

- an element of *ModDecls* set, as defined in Section 6.4, or
- the special symbol ($\top$), denoting that the current execution occurs outside any method or module. This symbol is used during the execution of the main instructions of the program, placed after all the mixin declarations (see Section 3.1).

Therefore this set is formally defined in the following way:

$$\mathit{Contexts} \;=\; (\mathit{MixNames} \times \mathit{MethodIDs}) \,\cup\, \mathit{ModDecls} \,\cup\, \{\top\}$$

From now on, we use the symbol *ctx* with optional indices to denote elements of this set.

7.1.3 Additional Functions

In this section we define some additional functions which will be needed later on in the rules of semantics and the type checking of Magda.

The *FirstEmpty* function is used to find the first (smallest) empty address in the given state. We say that the address is *empty* in state S if no object is associated with this address in state S. This function is used to obtain an address for the newly created object in the object creation expression.

The *EmptyObject* function takes as a parameter a sequence M of mixin names and returns a non-initialized object created from a given sequence of mixins. This object is a pair consisting of:

- a sequence of mixins M, and
- a partial function defined on field identifiers declared in mixins M. This partial function has value *null* for every such a field.

The *EmptyObject* function is used during evaluation of the object creation expression to obtain a non-initialized object.

For every context *ctx*, we use *IdTypes*(*ctx*) to denote a partial function defined for all identifiers declared in *ctx*. For every local identifier *id* declared in *ctx*, *IdTypes*(*ctx*)(*id*) denotes the declared type of identifier *id*. Therefore, $\mathit{IdTypes}(\top) = \emptyset$.

In the two other cases (in which the context *ctx* refers to a method or an initialization module), the partial function *IdTypes*(*ctx*) is defined as follows:

- When applied to the identifier `this`, it returns the name of the mixin in which the method / ini module is declared.
- When applied to a local variable declared within the body of a method / initialization module referred by *ctx*, the function returns its type present in the declaration.
- When applied to a parameter names of the method referred by *ctx*, it returns its declared type (when *ctx* refers to a method).

- When applied to the name of an input parameter of the ini module referred by *ctx*, it returns its declared type (when *ctx* refers to the initialization module).

Therefore, for the above defined functions we have:

$$\begin{array}{rcl}
\mathit{FirstEmpty} & : & \mathit{States} \rightarrow \mathit{Addresses} \\
\mathit{FirstEmpty}(\mathit{mem}) & = & \mathit{min}\{\mathit{adr} \in \mathit{Addresses} \mid \mathit{adr} \notin \mathit{Dom}(\mathit{mem})\} \\
\mathit{EmptyObject} & : & \mathit{MixNames}^* \rightharpoonup \mathit{Objects} \\
\mathit{EmptyObject}(M) & = & (M, \{(\mathit{fl} \mapsto \mathit{null}) \mid \exists_{T, \mathit{Mix} \in M} (\mathit{fl} : T) \in \mathit{Fields}_{\mathit{Mix}}\}) \\
\mathit{IdTypes} & : & \mathit{Contexts} \rightarrow (\mathit{LocalIdentifiers} \rightharpoonup \mathit{MixNames}^*)
\end{array}$$

7.2 Execution rules

In this formalization of the semantics we use three forms of "evaluates to" judgments:

Instruction evaluation judgments. Such judgments specify that, given:

- a local environment (denoting values of local variables), represented by the element of *Environments* set;
- a static context, represented by the element of the set *Contexts*;
- a state of heap, represented by an element of the set *States*;

instruction *instr* evaluates to a new environment *env'* and a new state *st'*. This models changes in values of local variables and in the global heap:

$$\mathit{env}, \mathit{ctx}, \mathit{st} \models \mathit{instr} \Rightarrow^{I} (\mathit{env}', \mathit{st}')$$

Object expression evaluation judgments. Since all values in Magda are addresses of objects in memory, the returned value is an address (an element of the set *Addresses*). On the other side, the evaluation of an expression does not influence local variables, so an expression will evaluate only to a new state and an address:

$$\mathit{env}, \mathit{ctx}, \mathit{st} \models \mathit{exp} \Rightarrow^{ex} (\mathit{st}', \mathit{adr})$$

Object initialization judgments. Such judgments are used to determine how the initialization modules are executed, and how they consume and produce the initialization parameters. Such judgments specify that, given:

- an address *adr* at which the object to be initialized is stored,
- a current state *st* (containing the object to be initialized at *adr*),
- a sequence *mods* of modules to be analyzed,

- a set $\overline{pars}$ of values of parameters remaining to be consumed by modules *mods*,

the initialization process ends in a state st' in which the object is initialized (and which also reflects other possible side-effects of the initialization process).

$$adr, st \models (mods, \overline{pars}) \Rightarrow^{ini} st'$$

The sequence *mods* is an element of $ModDecls^*$ set (see Section 6.4). The set $\overline{pars}$ is an element of the set *ParamValues*. Every element of the set $\overline{pars}$ is a pair build from a parameter identifier and its value.

When we refer to any of the three kinds of judgments we use the symbol $\Rightarrow$. Therefore $\Rightarrow$ represents one of the three symbols: $\Rightarrow^{I}, \Rightarrow^{ex}, \Rightarrow^{ini}$.

Later on we will use the notion of a *partial judgment* to denote a judgment in which the value on the right hand side of $\Rightarrow$ is unknown. A partial judgment has the form ... $\models$ ... $\Rightarrow$?.

7.2.1 The model of the program execution in big-step semantics

The above three kinds of judgments have been designed in order to allow us to express and prove properties of the following form:

The program P terminates its execution in a state st.

More precisely, we will say that the program P consisting of a sequence of mixin declarations and instructions I terminates in state st if and only if the below judgment, called *final judgment*, can be derived using the rules defined in next sections.

$$\emptyset, \top, \{\mathit{tt} \mapsto BoolVal,\ \mathit{ff} \mapsto BoolVal\} \models I \Rightarrow^{I} (\emptyset, st)$$

7.2.2 Naming convention

We use two different names for all the assumptions occurring in the below defined rules. We use the name *premise* for each judgment of the form ... $\models$ ... $\Rightarrow$ .... All the other assumptions are called *side-conditions.*

Moreover, when we use a sequence of assumptions of dynamic length of the form:

$$env^1, ctx^1, st^1 \models I_1 \Rightarrow^{I} (env_1, st_1) \quad ... \quad env^n, ctx^n, st^n \models I_n \Rightarrow^{I} (env_n, st_n)$$

we will often refer to it as one assumption.

7.2.3 The structure of rules: the sequential ordering of premises

Each rule in our semantics has a specific structure expressing the sequential character of the language. Each premise of the rule can be verified (if it is derivable) in the context of all the preceding premises of that rule. More precisely the *sequential ordering of premises* property is formulated as follows.

Property 3 (Sequential ordering of premises) *Within each rule, the state and the environment on the left side of the* $\models$ *symbol in each premise is determined by states and environments from all the premises occurring earlier within the same rule.*

As a result, our semantics is sequential. To illustrate this sequentiality let us consider the below rule (which is a simplified version of the compound instruction rule):

$$\frac{env, ctx, st \models I_1 \Rightarrow^I (env', st') \qquad env', ctx, st' \models I_2 \Rightarrow^I (env'', st'')}{env, ctx, st \models I_1 ; I_2 \Rightarrow^I (env'', st'')}$$

The sequential ordering of premises and the dependency of the states and the environments in this rule reflects the following sequential intuition behind this rule:

> In order to execute an instruction of the form $I_1 ; I_2$ in env, ctx, st we first need to execute I_1 and then, in the obtained env' and st', we need to execute I_2 to obtain the env'' and st'' which are the result of the execution of $I_1 ; I_2$.

This property is very similar to the one defined as *L-attributed grammar* by Ibraheem and Schmidt in [41] and explored deeper by Ager in [5].

Moreover, our semantics is also deterministic, in the sense that there do not exist two different derivation trees for one judgment. This happens because of the following fact. For each form of a judgment there is either only one rule, or there is more than one rule for some kind of judgment, however their form (together with their side-conditions) ensures that those rules are mutually exclusive. For example, consider the three rules for `while` statement in Section 7.2.4. All those share the same premise, which is evaluation of the boolean condition. And after that common premise, they differ in side-conditions which depend on the result of the evaluation of the premise. The first rule is applicable only if the boolean condition evaluates to ff, while two other rules only if the boolean condition evaluates to tt.

7.2.4 Instruction rules

The rules for the execution of instructions are rather straightforward. The only non-trivial case is the `return` statement, which ends the execution of the whole method body. This termination of execution is performed by putting the special value $\top$ in the environment. As a result, all the instructions containing some nested instructions (like compound, `if` and `while` statements), after the execution of any of their component instructions must verify if the component has executed a `return` instruction. Then, depending on this verification, this compound instruction either finishes its execution, or continues with the execution of subsequent component instructions.

Assignment to a local variable. The execution of the variable assignment instruction $\mathit{VarName}$:= exp does not change the global state (apart from the side-effects of the evaluation of exp), only a value of one variable in the environment. Therefore, on the right-hand side of the judgment we have the state st' returned by the evaluation of exp, and an environment env modified with the new value.

$$\frac{\mathit{env}, \mathit{ctx}, st \models \mathit{exp} \Rightarrow^{ex} (st',\ \mathit{adr})}{\mathit{env}, \mathit{ctx}, st \models\ \mathit{VarName}\ := \mathit{exp} \Rightarrow^{I} (\mathit{env}\{\mathit{VarName} \mapsto \mathit{adr}\},\ st')}$$

Assignment to an object field. The execution of the field assignment instruction of the form $\mathit{exp}_1 . \mathit{Mix} . \mathit{fl}$:= exp_2 begins with the evaluation of the expression exp_1, representing the target object, thus obtaining address adr'. Then, the expression exp_2 to be assigned to this field is evaluated in order to obtain address adr''. At the end, the instruction returns the new state in which the object to be modified is replaced with the new version of this object (represented by newObj). The new version is constructed from the old version of the same object with the second element of the pair (containing values of fields) modified.

$$\frac{\begin{array}{c}\mathit{env}, \mathit{ctx}, st \models \mathit{exp}_1 \Rightarrow^{ex} (st',\ \mathit{adr}') \\ \mathit{env}, \mathit{ctx}, st' \models \mathit{exp}_2 \Rightarrow^{ex} (st'',\ \mathit{adr}'') \\ \mathit{newObj} = (\ st''(\mathit{adr}')|_1,\ st''(\mathit{adr}')|_2\{\mathit{Mix}.\mathit{fl} \mapsto \mathit{adr}''\}\ )\end{array}}{\mathit{env}, \mathit{ctx}, st \models \mathit{exp}_1 . \mathit{Mix} . \mathit{fl}\ := \mathit{exp}_2 \Rightarrow^{I} (\mathit{env},\quad st''\{\mathit{adr}' \mapsto \mathit{newObj}\})}$$

Conditional instruction. The execution of a conditional instruction having the form `if` exp_1 `then` I_1 `else` I_2 `end` begins with the evaluation of the condition expression exp_1. Then, depending on the value of exp_1, the first or the second rule is used. Those in turn continue by executing I_1 or I_2 respectively (starting from the state st' returned by the evaluation of exp_1) and return the state st'' modified during the execution of the appropriate instruction.

$$\frac{\begin{array}{c}\mathit{env}, \mathit{ctx}, st \models \mathit{exp}_1 \Rightarrow^{ex} (st',\ \mathit{tt}\,) \\ \mathit{env}, \mathit{ctx}, st' \models I_1 \Rightarrow^{I} (\mathit{env}'', st'')\end{array}}{\mathit{env}, \mathit{ctx}, st \models \texttt{if}\ \ \mathit{exp}_1\ \ \texttt{then}\ \ I_1\ \ \texttt{else}\ \ I_2\ \ \texttt{end} \Rightarrow^{I} (\mathit{env}'',\ st'')}$$

$$\frac{\begin{array}{c}\mathit{env}, \mathit{ctx}, st \models \mathit{exp}_1 \Rightarrow^{ex} (st',\ \mathit{ff}\,) \\ \mathit{env}, \mathit{ctx}, st' \models I_2 \Rightarrow^{I} (\mathit{env}'', st'')\end{array}}{\mathit{env}, \mathit{ctx}, st \models \texttt{if}\ \ \mathit{exp}_1\ \ \texttt{then}\ \ I_1\ \ \texttt{else}\ \ I_2\ \ \texttt{end} \Rightarrow^{I} (\mathit{env}'',\ st'')}$$

The `return` statement. The execution of `return` exp instruction first evaluates the expression exp, and then returns the state modified by the expression together with a special form of the environment. The returned environment contains the special $\top$ element and the address adr' representing the returned value.

Notice that the old environment is completely discarded. It is safe because the execution of a local method is finished, thus local variables will not be referenced anymore.

The form of environment used here enforces skipping of all further instructions within the same method (see below rules for `while` and compound instructions).

$$\frac{env, ctx, st \models exp \Rightarrow^{ex} (st',\ adr')}{env, ctx, st \models \texttt{return}\ \ exp \Rightarrow^{I} ((\top, adr'),\ st')}$$

The compound instruction. The execution of the compound instruction of the form $I_1;I_2$ begins with the execution of the instruction I_1, yielding an environment env'. The further execution depends on the from of env'.

If env' is not of the form $(\top, ...)$ then instruction I_2 is executed (see the first rule).

If env' is of the form $(\top, ...)$ (what means that a `return` statement has been executed) then, according to the second rule, the instruction I_2 is skipped and the same environment env' is returned. In this way, such a special environment propagates upwards in the program tree up to the point of the call to the current method. As a result it prevents all the subsequent instructions occurring in the same method from being executed.

$$\frac{\begin{array}{c}env, ctx, st \models I_1 \Rightarrow^{I} (env', st') \qquad env' \neq (\top, ...)\\ env', ctx, st' \models I_2 \Rightarrow^{I} (env'', st'')\end{array}}{env, ctx, st \models I_1;I_2 \Rightarrow^{I} (env'', st'')}$$

$$\frac{env, ctx, st \models I_1 \Rightarrow^{I} (env', st') \quad env' = (\top, ...)}{env, ctx, st \models I_1;I_2 \Rightarrow^{I} (env', st')}$$

The empty instruction. An empty instruction evaluates to the same state in which it started the execution. Such empty instruction represents an empty body of a loop, an initialization module, or a method.

$$env, ctx, st \models \epsilon \Rightarrow^{I} (env, st)$$

The expression evaluation instruction. An expression can be used as a special form of instruction, the execution of which amounts to the evaluation of the expression. The result is the original environment (since expressions cannot modify local variables) and the new state st'. Notice that the evaluated address adr is not used.

$$\frac{env, ctx, st \models exp \Rightarrow^{ex} (st', adr)}{env, ctx, st \models exp \Rightarrow^{I} (env, st')}$$

The loop statement. The execution of a loop statement of the form `while (`exp_1`)` I_1 `end` is described by the three below rules.

Its execution begins with the evaluation of the condition expression exp_1, as specified in the first premise of each rule. If the value of exp_1 is $f\!f$ then, according to the first rule, the whole `while` statement ends.

If the value of exp_1 is tt then (according to the second premise of the second and the third rule) the instruction I_1 is executed. When the instruction I_1 finishes and the resulting environment is a pair $(\top, x)$ then the whole `while` statement finishes (see the third rule). Otherwise, according to the second rule, the whole `while` statement is executed once again in the new state and the new environment.

$$\frac{env, ctx, st \models exp_1 \Rightarrow^{ex} (st',\ ff\,)}{env, ctx, st \models \texttt{while (}exp_1\texttt{)}\ I_1\ \texttt{end}\ \Rightarrow^{I} (env, st')}$$

$$\frac{\begin{array}{c} env, ctx, st \models exp_1 \Rightarrow^{ex} (st',\ tt\,) \\ env, ctx, st' \models I_1 \Rightarrow^{I} (env'', st'') \qquad env'' \neq (\top, x) \\ env'', ctx, st'' \models \texttt{while (}exp_1\texttt{)}\ I_1\ \texttt{end}\ \Rightarrow^{I} (env''', st''') \end{array}}{env, ctx, st \models \texttt{while (}exp_1\texttt{)}\ I_1\ \texttt{end}\ \Rightarrow^{I} (env''', st''')}$$

$$\frac{\begin{array}{c} env, ctx, st \models exp_1 \Rightarrow^{ex} (st',\ tt\,) \\ env, ctx, st' \models I_1 \Rightarrow^{I} (env'', st'') \qquad env'' = (\top, x) \end{array}}{env, ctx, st \models \texttt{while (}exp_1\texttt{)}\ I_1\ \texttt{end}\ \Rightarrow^{I} (env'', st'')}$$

7.2.5 Expression evaluation rules

Constants evaluation. Three constants existing in Magda evaluate to their corresponding addresses following the below axioms.

$$env, ctx, st \models \texttt{true} \Rightarrow^{ex} (st,\ tt\,)$$

$$env, ctx, st \models \texttt{false} \Rightarrow^{ex} (st,\ ff\,)$$

$$env, ctx, st \models \texttt{null} \Rightarrow^{ex} (st, null)$$

Local identifier evaluation. The evaluation of a local identifier (for example, a variable or a method parameter) returns the value of the given identifier in the present environment *env*.

$$\frac{adr = env(VarName)}{env, ctx, st \models\ VarName \Rightarrow^{ex} (st, adr)}$$

Field value evaluation. The evaluation of an object field expression *exp*. *Mix* .*fl* consists of two steps. First, the expression *exp*, denoting the target object, is evaluated to obtain address *adr*. Then the evaluation of the whole expression returns the state modified by the evaluation of *exp*, and the value picked from the second element of the target object $st'(adr)$.

$$\frac{env, ctx, st \models exp \Rightarrow^{ex} (st',\ adr)}{env, ctx, st \models exp\,.\,Mix\,.fl \Rightarrow^{ex} (st',\ st'(adr)|_2(Mix.fl)\ )}$$

Call to a method. The evaluation of the method call expression is somewhat complicated. According to the below rule, the evaluation process of an expression of the form $exp\,.\,Mix\,.\,mt(exp_1,\ \ldots\ , exp_n)$ proceeds as follows:

1. The expression denoting the target of the call (exp) is evaluated (according to the first assumption).
2. According to the second assumption, the actual parameters exp_1, ..., exp_n are evaluated to values adr_1, ..., adr_n.
3. According to the third assumption, the mixin Mix' containing the last definition of method $Mix.mt$ is evaluated. The mixin Mix' is chosen from the sequence of mixins $st_n(adr_0)|_1$ from which the target object was created.
4. Then, the new environment env' in which the body of the method will be evaluated is defined (according to the fourth, fifth and sixth assumption). The environment env' contains values assigned to three different kinds of local identifiers:
   - names of method parameters (p_1, ..., p_n) are determined basing on the method declaration (using *MetParams* definition), and mapped into the actual values (adr_1, ..., adr_n) of the parameters;
   - names of local variables declared within the method body $local_1$, ..., $local_k$ (determined using $MetLocals^{(Mix,mt)}_{Mix'}$) are all mapped into *null* value (representing the non-initialized variable);
   - `this` variable is mapped into the address adr_0 of the target object of this call.
5. Finally (according to the last assumption), in environment env' and in the context containing the information about the executed method implementation $(Mix', (Mix, mt))$ the body of the given implementation ($MetInstr^{(Mix,mt)}_{Mix'}$) is evaluated to the state st' and the environment $(\top, adr)$.

   Notice that it is safe to assume that the resulting environment has the form $(\top, adr)$, which means that a `return ...` instruction has been executed (see Section 7.4 for more details).

The obtained state st' and address adr are the result of the execution of the whole method call expression.

$$\frac{\begin{array}{c} env, ctx, st \models exp \Rightarrow^{ex} (st_0, adr_0) \\ env, ctx, st_0 \models exp_1 \Rightarrow^{ex} (st_1, adr_1) \quad \ldots \quad env, ctx, st_{n-1} \models exp_n \Rightarrow^{ex} (st_n, adr_n) \\ Mix' = LastMix(st_n(adr_0)|_1, (Mix, mt)) \\ (p_1 : T_1,\ \ldots\ , p_n : T_n) = MetParams^{(Mix,mt)}_{Mix'} \qquad (local_1,\ \ldots\ , local_k) = MetLocals^{(Mix,mt)}_{Mix'} \\ env' = \{p_1 \mapsto adr_1; \ldots; p_n \mapsto adr_n; local_1 \mapsto null; \ldots; local_k \mapsto null; \mathtt{this} \mapsto adr_0\} \\ env', (Mix', (Mix, mt)), st_n \models MetInstr^{(Mix,mt)}_{Mix'} \Rightarrow^{I} ((\top, adr), st') \end{array}}{env, ctx, st \models exp\,.\,Mix\,.\,mt(exp_1,\ \ldots\ , exp_n) \Rightarrow^{ex} (st', adr)}$$

A `super(...)` method call. Semantics of `super(...)` method call is in many respects similar to the one of an ordinary method call and thus the rule is similar. This rule differs only in two respects:

- The way the mixin Mix' containing the called implementation is computed. In a `super(...)` method call the mixin Mix' is computed according to on the current static context, using the function *LastMixBef* instead of *LastMix*. As a result, the implementation to be executed is picked from the declaration of the mixin which precedes the current mixin in the sequence of mixins $st(env(\texttt{this}))|_1$, from which the current object has been created. The current mixin is, in turn, picked from the static context.

- The way the object being the target of the method call is defined. In the ordinary method call, the expression denoting the target object is a part of the method call expression. In `super()` method call, the target of the call is the value of the variable `this` in the current environment.

$$\frac{\begin{array}{c} env, (Mix, mtID), st_0 \models exp_1 \Rightarrow^{ex} (st_1, adr_1) \;...\; env, (Mix, mtID), st_{n-1} \models exp_n \Rightarrow^{ex} (st_n, adr_n) \\ Mix' = LastMixBef(st_n(env(\texttt{this}))|_1, mtID, Mix) \\ (p_1 : T_1,\ ..., p_n : T_n) = MetParams^{mtID}_{Mix'} \qquad (local_1,\ ..., local_k) = MetLocals^{mtID}_{Mix'} \\ env' = \{p_1 \mapsto adr_1; ...; p_n \mapsto adr_n; local_1 \mapsto null; ...; local_k \mapsto null; \texttt{this} \mapsto env(\texttt{this})\} \\ env', (Mix', mtID), st_n \models MetInstr^{mtID}_{Mix'} \Rightarrow^{I} ((\top, adr), st') \end{array}}{env, (Mix, mtID), st_0 \models \texttt{super}\ (exp_1,\ ...\ , exp_n) \Rightarrow^{ex} (st', adr)}$$

A `new` object creation. The evaluation of a new object creation expression of the form `new` $Mixins[ParID_1{:=}exp_1, ..., ParID_k{:=}exp_k]$ is defined by the below rule, and proceeds as follows:

1. First, the actual initialization parameters exp_1, ..., exp_k, are evaluated to adr_1, ... adr_k. Below we use $\overline{ParID \mapsto adr}$ to denote the set of parameter names with their values evaluated in this step. Notice that from this point the actual ordering of parameters is ignored.

2. Then, the fresh not initialized, object value $objVal$ is computed using the *EmptyObject* function (see Section 7.1.3).

3. Then (according to the third assumption), an address not used in st_k is computed using *FirstEmpty* function (see Section 7.1.3) and denoted as adr'.

4. Finally (according to the last premise), initialization modules are evaluated in the state which contains $objVal$ at address adr'. The process of execution of those modules results in the new state st'' in which the object at address adr' is completely initialized. The sequence of all modules to be evaluated is denoted as $IniModules(mixins)$ (see Section 6.4).

The result of this whole process is the state st'' returned by the evaluation of initialization modules and the address adr' pointing to the newly created object.

$$\frac{\begin{array}{l} env, ctx, st_0 \models exp_1 \Rightarrow^{ex} (st_1, adr_1) \quad ... \quad env, ctx, st_{k-1} \models exp_k \Rightarrow^{ex} (st_k, adr_k) \\ objVal = EmptyObject(mixins) \qquad\qquad adr' = FirstEmpty(st_k) \\ \quad adr',\ st_k\{adr' \mapsto objVal\} \models (IniModules(mixins), \overline{ParID \mapsto adr}) \Rightarrow^{ini} st'' \end{array}}{env, ctx, st_0 \models \texttt{new}\ \ Mixins[ParID_1{:=}exp_1, ..., ParID_k{:=}exp_k] \Rightarrow^{ex} (st'',\ adr')}$$

7.2.6 Object initialization rules

The three below rules specify how the initialization process is being performed. The whole process is driven by the sequence of initialization modules. Each module (starting from the last) in this sequence is checked. If the input parameters of the module are in the set of currently supplied parameters then the module is executed, otherwise it is skipped.

Object initialization end. The first rule (which is an axiom in fact), says that when the sequence of initialization modules is empty, then the whole process ends. Notice that this rule requires also the set of parameters to be empty in order to finish the process. The fact that the set of parameters will be always empty at the end of the process (therefore all parameters will be consumed) is enforced by the type checking rules defined in Section 8.6.

$$adr, st \models ((), ()) \Rightarrow^{ini} st$$

Initialization module skip. The second rule is used in the case when the last module *mod* in the sequence has some input parameters $mod|_1$, and for those parameters there are no values available in the function $\overline{pars}$, as stated by the first side-condition of this rule. In such a case, the initialization process continues for the sequence of modules with the given module removed. In other words, the module is skipped.

$$\frac{\begin{array}{c} mod|_1 \cap dom(\overline{pars}) = \emptyset \neq mod|_1 \\ adr, st \models (\ \overrightarrow{mods}, \quad \overline{pars}) \Rightarrow^{ini} st' \end{array}}{adr, st \models (\ \overrightarrow{mods} \cdot mod, \quad \overline{pars}) \Rightarrow^{ini} st'}$$

Initialization module execution. The below rule describes the case when all the input parameters ($ipar_1$, ..., $ipar_k$) of the last module (*mod*) are contained within $\overline{pars}$. This conditions is expressed by the first and the second assumption of the rule. In such a case we say, that this module is *activated* by the object creation expression which started the initialization process. When a module is activated, then the instructions in the body of the module are executed. We recall that *mod* here is an element of *ModDecls* set (see Section 6.4), thus it is a four-element tuple consisting of: (*i*) input parameters sequence, (*ii*) output parameters sequence (with their types), (*iii*) module body, (*iv*) the mixin name.

The execution of the initialization module proceeds as follows:

1. First (according to the second assumption), input parameters of the module are deducted from the list of not-consumed parameters (denoted as $\overline{pars}$) to obtain the set $\overline{pars'}$.

2. According to the third assumption, identifiers $\overrightarrow{local : T}$, I_1, $\overrightarrow{opID}$, exp and I_2 denote elements of the body of the ini module ($mod|_3$).

3. Then (according to the fourth assumption), the new environment (env) is built from three parts:

   - input parameter names (ip_1, ..., ip_k) mapped into values picked from $\overline{pars}$;
   - local variables ($\overrightarrow{local}$) mapped into $null$ value;
   - `this` variable mapped into the current object (adr).

4. Then, according to the fifth assumption, the instructions I_1 placed with ini module before the `super[...]` call are executed in the new environment env.

5. Then, according to the sixth assumption, the values of output parameters exp_1, ..., exp_l are computed.

6. Then, according to the seventh assumption (and indirectly using the second assumption), a new set of parameter values assigned to their identifiers (denoted as $\overline{pars''}$) is defined in the following way: We begin with the original set of parameters ($\overline{pars}$), then the input parameters of the given module ($Mix.ip_1$, ..., $Mix.ip_k$) are removed, and the output parameters of the given module $opID_1$, ..., $opID_l$, with their values computed in the previous step, are added.

7. Then, according to the eighth assumption, ini modules remaining in the $\overrightarrow{mods}$ sequence are executed with the set of initialization parameters $\overline{pars''}$ defined in the previous point.

8. Finally, according to the last assumption, instructions I_2 (occuring after the `super[...]` call in the body of the module) are executed.

$$\frac{\begin{array}{c} mod|_1 = (Mix.ip_1, ..., Mix.ip_k) \\ \overline{pars'} = \overline{pars} \leftharpoondown^* \{Mix.ip_1 \mapsto adr^I_1; ...; Mix.ip_k \mapsto adr^I_k\} \\ \overrightarrow{local : T}\ \texttt{begin}\ I_1; \texttt{super[}\overrightarrow{opID := exp}\texttt{]}; I_2\ \texttt{end;}\ = mod|_3 \\ env = \{ip_1 \mapsto adr^I_1; ...; ip_n \mapsto adr^I_k; local_1 \mapsto null; ...; local_m \mapsto null; \texttt{this} \mapsto adr\} \\ env, mod, st \models I_1 \Rightarrow^I (env',\ st_0) \\ env', mod, st_0 \models exp_1 \Rightarrow^{ex} (st_1,\ adr^O_1) \quad ... \quad env', mod, st_{l-1} \models exp_l \Rightarrow^{ex} (st_l,\ adr^O_l) \\ \overline{pars"} = \overline{pars'}\{opID_1 \mapsto adr^O_1; ...; opID_l \mapsto adr^O_l\} \\ adr, st_l \models (\ \overrightarrow{mods},\quad \overline{pars"}) \Rightarrow^{ini} st'' \\ env', mod, st'' \models I_2 \Rightarrow^I (env'',\ st''') \end{array}}{adr, st \models (\ \overrightarrow{mods} \cdot mod,\quad \overline{pars}) \Rightarrow^{ini} st'''}$$

7.3 The structure of a big-step derivation

In this section we state a property enjoyed by the semantics of Magda. This property reflects the way the context is manipulated by the rules.

First of all, notice that most of big-step semantics rules have the following shape: The evaluation of a term t (an instruction or an expression) of a given form is defined by means of the evaluation of subterms of the term t. Moreover, each evaluation of a subterm is performed in the same context in which term t is evaluated. More precisely, within each rule used to derive the judgment $...ctx... \models t \Rightarrow ...$, in each premise of form $...ctx'... \models t' \Rightarrow ...$, if t' is a subterm of t then $ctx' = ctx$.

The only rules in which the premises contain a term which is not a subterm of the term present in conclusions are: method call, `super()` call, and the execution of an initialization module. It is easy to see that, within those three rules, each premise containing a term which is not a subterm uses a different context than the one which is used in the conclusion of the rule. Moreover, those are the only rules which "generate" new contexts as well as new terms. Therefore, we have the following property:

Property 4 (Structure of derivation) *For each program P and the derivation tree of its final judgment (see Section 7.2.1) of the form*

$$\emptyset, \top, \{\, tt \mapsto BoolVal,\ ff \mapsto BoolVal\} \models I \Rightarrow^I (\emptyset, st)$$

and for each occurrence of a judgment $...ctx... \models t \Rightarrow ...$ (*) in the derivation tree, such that t is an instruction or expression, we have the following situation:

The judgment (*) occurs in a subtree, which starts from a node having the form $...ctx... \models t' \Rightarrow^I ...$, such that t' is:

- *either the body of the method in which t occurs,*
- *or the body of the initialization module in which t occurs,*

- *or the sequence of main instructions of the program in which t occurs.*

Less formally: Evaluation of every instruction and expression occurs in the subtree which was started by the evaluation of the body of a method, an ini module, or the block of main program instructions which this instruction/expression is part of. Moreover this expression is evaluated in the context of that specific method, ini module or main instructions.

7.4 The method body evaluation

In this section we state a property of the semantics which informally says, that if the evaluation of the method body finishes, then it finishes with an environment containing a value.

Property 5 (Method body evaluation) *For each environment env, context ctx, and state st, each mixin name Mix and method identifier $mtID$, if we have:*

$$env, ctx, st \models MetInstr^{mtID}_{Mix} \Rightarrow^{I} (env', st')$$

for some (env', st'), then env' is a pair of the form $(\top, adr)$ for some address adr.

In order to prove this property, it is enough to see how a derivation tree for $MetInstr^{mtID}_{Mix}$ looks like. First recall that $MetInstr^{mtID}_{Mix}$ (see Section 6.3) is a compound instruction of the form I`;` `return null`. Recall also that the semantics of the compound instruction has two cases. In the first case, when the execution of I returns an environment of the form $(\top, adr)$, the same environment is the result of the whole body. In the second case, when the result of execution of I is an "ordinary" environment, then the result of the whole method body is equal to the result of the execution of `return null`, which is $(\top, null)$.

Chapter 8

Type checking rules

In this section we specify rules which are used to verify the type safety of a program. In other words, those rules ensure that the execution of a program will not get stuck because of a message-not-understood error, or attempts to access a non-existent field or initialization parameter. Without the type checking, such errors could occur when the runtime type of the target object is not a subtype of the static type of the expression.

Recall, that we assume a fixed program P (see Section 6), so we implicitly refer to its elements where needed.

8.1 Type correctness judgments

The type checking rules in Magda are used to derive judgments of the following kinds.

A program is type correct. This judgment states that the program consisting of a series of mixin declarations $M_1, ..., M_n$ and the instruction I is type correct.

$$\vdash_P M_1; M_2; ...M_n; I \ : \ OK$$

A declaration of a mixin is type correct.

$$\vdash_D M_i \ : \ OK$$

A declaration of a method is type correct. This judgment states that a given method declaration *MetDecl* is type correct within the declaration of the mixin *Mix*.

$$\mathit{Mix} \vdash_{met} \mathit{MetDecl} \ : \ OK$$

A declaration of an ini module is type correct. This judgment states that an ini module is correct within the declaration of mixin named *Mix* and in the context of ini modules *PrecModules*, which have been declared textually above within the mixin *Mix*.

The list of ini modules *PrecModules* is needed during type checking, because their input parameters can be referenced by the given module in its output parameter declarations.

$$Mix, PrecModules \vdash_{\overline{ini}} inimoduleDeclaration \ : \ OK$$

An instruction is type correct. This judgment states that an instruction is type correct within the context *ctx* of the method or ini module in which it occurs.

$$ctx \vdash_{\overline{I}} I \ : \ OK$$

In judgments of that kind, the context *ctx* is used to denote the static information about the block of code within which the instruction I is placed and executed. Every value of *ctx* is an element of the set *Contexts* as defined in Section 7.1.2. We remind that, according to the definition in Section 7.1.2, it carries information about one of the below things:

- for instructions placed within a body of a method declaration: the identifier of that method;
- for instructions placed within an ini module: the sequences of input and output parameters of that module, together with its body and the name of the mixin in which the given module was declared;
- for the main program instructions placed after all mixin declarations: an empty context denoted as $\top$.

Moreover, notice that, by knowing the context, we have the information about the types of local variables and other identifiers, which are defined by the function *IdTypes*(*ctx*) (see Section 7.1.3).

An expression is type correct and has a type T. The below judgment states that within the given local context *ctx* of a method or a module, an expression *exp* is type correct and has type T.

$$ctx \vdash_{\overline{ex}} exp : T$$

Every type used in such judgments, as well as any other type in Magda, is a set of mixin names — as stated before in Section 3.9 and Section 4.6. Every context used in the judgment of the above kind is an element of the set *Contexts* as defined in Section 7.1.2.

All the above judgments are used in derivations of judgments of the form:

$$\vdash_{\overline{P}} M_1; M_2; ...M_n; I \ : \ OK$$

A judgment of this form means that the whole program is type correct, which guarantees the type safety of the execution of the program, as defined in Section 10.2 and in Section 11.1.

Similarly as with the big-step semantics rules, we use two different names for all the assumptions occurring in the type checking rules. We use the name *premise* for each judgment of the form ... $\vdash_*$ ..., where $\vdash_*$ is one of the symbols $\vdash_{\overline{I}}, \vdash_{\overline{D}}, \vdash_{\overline{ex}}, \vdash_{\overline{met}}, \vdash_{\overline{ini}}$. All the other assumptions are called *side-conditions*.

8.2 Additional functions

Additionally, within the premises of the type checking rules, we use the following additional functions:

- Function *Rmodules*, which takes as an argument a set of mixin names, and returns a set of ini modules `required`, which are declared within those mixins. The complete definition of this function can be found in Section 8.2.2.
- Function *activated*, which takes two parameters: a sequence of mixin names and a set of initialization parameter identifiers. This function returns a set of ini modules declared within the given mixins, which are activated by the given initialization parameters. The complete definition of this function can be found in Section 8.2.1.
- Function *InputPars*, which takes two parameters: a set of mixin names and a sequence of ini modules. This function returns a set of initialization parameters (with their types) declared in the supplied mixins and ini modules.

  Each element of the set $\mathit{InputPars}(\overline{mixins}, \overline{mods})$ consists of a parameter identifier, and its type. This set consists of input parameters declared in: (i) modules present in mixins $\overline{mixins}$; (ii) modules present in $\overline{mods}$.

According to the above, we have:

$$\begin{aligned}
\mathit{Rmodules} &: 2^{\mathit{MixNames}} \longrightarrow 2^{\mathit{ModDecls}} \\
\mathit{activated} &: (\mathit{MixNames}^* \times 2^{\mathit{paramIDs}}) \longrightarrow 2^{\mathit{ModDecls}} \\
\mathit{InputPars} &: (\mathit{MixNames}^* \times \mathit{ModDecls}) \longrightarrow 2^{\mathit{ParamIDs} \times \mathit{MixNames}^*}
\end{aligned}$$

8.2.1 The function *activated*

The function *activated* takes as arguments a sequence of mixin names and a set of initialization parameter identifiers. This function returns a sequence of ini module declarations chosen from the declarations of mixins supplied as the first parameter. The function chooses modules which are activated by the given set of parameters (see Section 3.8 and Section 4.3.3). This function is defined below using two additional functions: *IniModules* and *activated*$'$. The function *IniModules* (as defined in Section 6.4) takes as a parameter a sequence of mixins, and returns the set of all initialization modules declared within the given modules.

$$activated(\overrightarrow{Mix}, \overline{p}) = activated'(IniModules(\overrightarrow{Mix}), \overline{p})$$

The function *activated'*, as defined below, takes as its parameters: (*i*) a sequence of ini modules and (*ii*) a set of parameters. This function returns a sequence of ini modules, which is a subsequence of the one passed as the first argument.

$$activated'(\epsilon, \emptyset) = \emptyset \qquad \frac{aIM|_1 \cap \overline{p} = \emptyset \neq aIM|_1}{activated'(\overrightarrow{IM} \cdot aIM,\ \overline{p}) = activated'(\overrightarrow{IM},\ \overline{p})}$$

$$\frac{aIM|_2 = \overrightarrow{r : T} \qquad aIM|_1 \cap \overline{p} = aIM|_1}{activated'(\overrightarrow{IM} \cdot aIM,\ \overline{p}) = activated'(\overrightarrow{IM},\ (\overline{p} - aIM|_1) \uplus \overrightarrow{r}) \cup \{aIM\}}$$

The set of parameters $\overline{p}$ triggers the lookup that searches the sequence $\overrightarrow{IM}$ for the last ini module aIM whose input parameters are included within $\overline{p}$. It also assumes that either all parameters of some module are included or none. Once such a module is found, the lookup proceeds recursively by looking for the ini modules that are activated by the set of parameter calculated as $\overline{p}$ minus the input parameters of aIM, plus the output parameters of aIM.

Notice also that, this function performs two correctness checks (otherwise it is not defined): First, it checks that all parameters, starting from $\overline{p}$, are consumed by some ini modules declared in the given mixins. Secondly, it checks that we do not have a case where only part of the parameters of some module is supplied.

The side-condition $aIM|_1 \cap \overline{p} = \emptyset \neq aIM|_1$ ensures that in no situation two of the above rules can be applied, therefore *activated'* is indeed a function.

8.2.2 The function *Rmodules*

The function *Rmodules*, when applied to a set of mixins names $\overline{M}$, returns a set of declarations of ini modules. The returned set contains all the ini modules occurring in the declarations of mixins $baseExt(\overline{M})$, which are marked with `required` keyword. Each ini module in the returned set is an element of *ModDecls* set — see the definition in Section 6.4.

We remind that the set $baseExt(\overline{M})$ (see Section 6.5) contains all mixins in $\overline{M}$ as well as their direct and indirect base mixins.

8.3 Type checking of a program and mixin declarations

In this and in the following sections we present the typing rules. Whenever the premises of a rule are described by itemized sentences, such description is presented in the order of appearance of the premises within that rule.

Type checking of a program. According to the below rule, a program is type correct if all mixin declarations occurring in it are type correct, as well as its main instructions.

$$\frac{\vdash_{D} M_1 \;:\; OK \quad ... \quad \vdash_{D} M_n \;:\; OK \quad ... \quad \top \vdash_{I} I \;:\; OK}{\vdash_{P} M_1; M_2; ...M_n; I \;:\; OK}$$

Type checking of a mixin declaration. According to the below rule, the declaration of a mixin is type correct if the following conditions are met:

- The base mixin expression *BaseExp* contains only names of mixins declared in the program.
- All method declarations met_1 ... met_n in the mixin declaration are type correct.
- All ini module declarations mod_1 ... mod_k occurring in the mixin declaration are type correct with respect to the modules textually occurring above them.
- The type of each field declaration consists of valid mixin names.
- All field names used in field declarations are distinct.

$$\frac{\begin{array}{c} BaseExp \subseteq MixNames \\ Mix \vdash_{met} met_1 \;:\; OK \qquad ... \qquad Mix \vdash_{met} met_n \;:\; OK \\ Mix, \emptyset \vdash_{ini} mod_1 \;:\; OK \qquad ... \qquad Mix, \{mod_1, ..., mod_{K-1}\} \vdash_{ini} mod_k \;:\; OK \\ T_1 \cup ... \cup T_l \;\subseteq\; MixNames \qquad \forall_{i,j}((i \neq j) \Rightarrow (fl_i \neq fl_j)) \end{array}}{\vdash_{D} \texttt{mixin}\ Mix\ \texttt{of}\ BaseExp\ \texttt{=}\ met_1\texttt{;}...\texttt{;}met_n\texttt{;}mod_1\texttt{;}...\texttt{;}mod_k\texttt{;}fl_1\texttt{:}T_1\texttt{;}...\texttt{;}fl_l\texttt{:}T_l\texttt{;}\ \texttt{end} \;:\; OK}$$

8.4 Type checking of mixin members

8.4.1 Type checking of method declarations

The declaration of a method marked as `new`. According to the below rule, a declaration of method *mt* marked as `new` is type correct if the following conditions are met:

- The return type, the parameter types (T_1, ..., T_n), and the types of variables (S_1,..., S_n) contain names of mixins which are declared in the program.
- In the context of the declared method, instructions placed in the body of method are type correct.
- The names of method parameters are different from the special name `this`.
- The names of local variables in the body of the method are different from the special name `this`.
- The names of method parameters are distinct from the names of local variables.

$$\frac{\begin{array}{c} T^1 \ \cup \ T_1 \cup ... \cup T_n \ \cup \ S_1 \cup ... \cup S_k \subseteq \mathit{MixNames} \\ (\mathit{Mix}_c, (\mathit{Mix}_c, \mathit{mt})) \vdash_I I \ : \ \mathit{OK} \\ \texttt{this} \notin \{p_1, ..., p_n\} \qquad \texttt{this} \notin \{v_1, ..., v_k\} \qquad \{v_1, ..., v_k\} \cap \{p_1, ..., p_n\} = \emptyset \end{array}}{\mathit{Mix}_c \vdash_{\mathit{met}} \texttt{new} \ T^1 \ \mathit{mt} \ \texttt{(} p_1 \texttt{:} T_1 \texttt{;} ... \texttt{;} p_n \texttt{:} T_n \texttt{)} \ \texttt{var} \ v_1 \texttt{:} S_1 ..., v_k \texttt{:} S_k \ \texttt{begin} \ I \ \texttt{end} \ : \mathit{OK}}$$

The declaration of a method marked as `override` or `implement`. According to the below rule, a declaration of method *Mix.mt* marked as `override` or `implement`, occurring within a declaration of a mixin Mix_c, is type correct if following conditions are met:

- The mixin name *Mix* is contained within the base mixin expression of the mixin Mix_c.
- The method *mt* was introduced in the declaration of the mixin *Mix*.
- The return type, parameter types, and types of variables contain only names of mixins which are declared in the program.
- The return type of the current method declaration is equal to the one present in the introduction of the implemented method.
- The list of declared method parameters is equal to the list of parameter declarations present in the introduction of the implemented method.
- In the context of the declared method, instructions placed in the body of the method are type correct.
- The names of method parameters are different from the special name `this`.
- The names of local variables in the body of the method are different from the special name `this`.
- The names of method parameters are distinct from the names of local variables.

$$\frac{\begin{array}{c} \mathit{specifier} \in \{\texttt{override}, \texttt{implement}\} \\ \mathit{Mix} \in \mathit{base}(\mathit{Mix}_c) \qquad \mathit{mt} \in \mathit{IntrMethods}_{\mathit{Mix}} \\ T^1 \ \cup \ T_1 \cup ... \cup T_n \ \cup \ S_1 \cup ... \cup S_k \subseteq \mathit{MixNames} \\ T^1 = \mathit{RetType}_{\mathit{Mix}}^{(\mathit{Mix},\mathit{mt})} \qquad (p_1 : T_1, ..., p_n : T_n) = \mathit{MetParams}_{\mathit{Mix}}^{(\mathit{Mix},\mathit{mt})} \\ (\mathit{Mix}_c, (\mathit{Mix}, \mathit{mt})) \vdash_I I \ : \ \mathit{OK} \\ \texttt{this} \notin \{p_1, ..., p_n\} \qquad \texttt{this} \notin \{v_1, ..., v_k\} \qquad \{v_1, ..., v_k\} \cap \{p_1, ..., p_n\} = \emptyset \end{array}}{\mathit{Mix}_c \vdash_{\mathit{met}} \mathit{specifier} \ T^1 \ \mathit{Mix} \, . \, \mathit{mt} \texttt{(} p_1 \texttt{:} T_1 \texttt{;} \ ... \texttt{;} p_n \texttt{:} T_n \texttt{)} \ \texttt{var} \ v_1 \texttt{:} S_1 ... v_k \texttt{:} S_k \ \texttt{begin} \ I \ \texttt{end} : \mathit{OK}}$$

8.4.2 Type checking of an initialization module

According to the below rule, the declaration of an initialization module present in the mixin Mix_c, occurring below the declarations of modules $mods$, is type correct if the following conditions are fulfilled:

- Each output parameter ($Mix_1.op_1$, ..., $Mix_m.op_m$) of the initialization module refers to an existing parameter declaration occurring in some other module. The other module must be declared above in the same mixin, or in one of the base mixins. All those parameters are denoted as $InputPars(base(Mix_c), mods)$ (see Section 8.2). Moreover, these output parameters are declared with some types T^1, ..., T^m.

- The names of input parameters differ from all the names of input parameters declared in other modules occurring in the same mixin.

- The types of input parameter (T_1, ..., T_n) and the types of variables (S_1, ..., S_n) contain only names of mixins which are declared in the program.

- In the context of the current initialization module, the instructions placed in the body are type correct.

- The names of input parameters of the initialization module are different from the special name `this`.

- The names of local variables in the body of the initialization module are different from the special name `this`.

- The names of input parameters are distinct from the names of local variables.

$$\frac{\begin{array}{c} specifier \in \{\texttt{required}, \texttt{optional}\} \\ \{Mix_1.op_1 : T^1, ..., Mix_m.op_m : T^m\} \subseteq InputPars(base(Mix_c), mods) \\ \forall_{i\in\{1..n\},T} Mix_c.ip_i \notin InputPars(\emptyset, mods) \\ T_1 \cup ... \cup T_n \ \cup \ S_1 \cup ... \cup S_k \subseteq MixNames \\ (\{Mix_c.ip_1, ...\}, \{Mix_1.op_1 : T^1, ...\}, \texttt{var} \ \texttt{...} \ I \ \texttt{end}, Mix) \vdash_{I} I \ : \ OK \\ \texttt{this} \notin \{v_1, ..., v_k\} \qquad \texttt{this} \notin \{ip_1, ..., ip_n\} \qquad \{v_1, ..., v_k\} \cap \{ip_1, ..., ip_n\} = \emptyset \end{array}}{Mix_c, mods \vdash_{ini} \left(\begin{array}{l} specifier \ Mix_c\texttt{(}ip_1\texttt{:}T_1\texttt{;} \ ...ip_n\texttt{:}T_n\texttt{)} \ \texttt{initializes} \\ \texttt{(}Mix_1.op_1...Mix_m.op_m\texttt{)} \ \texttt{var} \ v_1\texttt{:}S_1...v_k\texttt{:}S_k \ \texttt{begin} \ I \ \texttt{end} \end{array} \right) : OK}$$

8.5 Type checking of instructions

In this section we present the rules which are used to verify if instructions are type correct.

Type checking of a compound instruction. A compound instruction $I_1;I_2$ is type correct in a given context *ctx* if both I_1 and I_2 are type correct in context *ctx*.

$$\frac{ctx \vdash_I I_1 \ : \ OK \qquad ctx \vdash_I I_2 \ : \ OK}{ctx \vdash_I I_1;I_2 \ : \ OK}$$

Type checking of an empty instruction. An empty instruction is always type correct.

$$ctx \vdash_I \epsilon \ : \ OK$$

Type checking of a variable assignment. An assignment instruction of the form *VarName*`:=`*exp* is type correct if the variable *VarName* is distinct from `this` and the assigned expression *exp* has the type *IdTypes*(*ctx*)(*VarName*). We recall that *IdTypes*(*ctx*)(*VarName*) (see Section 7.1.3) denotes the type occurring in the declaration of variable *VarName* in the context *ctx*.

$$\frac{VarName \neq \texttt{this} \qquad ctx \vdash_{ex} exp : IdTypes(ctx)(VarName)}{ctx \vdash_I VarName \ \texttt{:=} \ exp \ : \ OK}$$

Type checking of a field assignment instruction. The field assignment instruction $exp_1\,.\,Mix\,.\,fl$`:=`exp_2 is type correct if the following conditions are met:

- The expression exp_1 (denoting the target object) has the type $\{Mix\}$.
- The declaration of the mixin *Mix* contains the field declaration $fl : T$ for some T.
- The expression exp_2 (denoting the value to be assigned) has the type T.

$$\frac{ctx \vdash_{ex} exp_1 : \{Mix\} \qquad fl : T \in Fields_{Mix} \qquad ctx \vdash_{ex} exp_2 : T}{ctx \vdash_I exp_1\,.\,Mix\,.\,fl \ \texttt{:=} \ exp_2 \ : \ OK}$$

Type checking of an expression instruction. The instruction being an expression is type correct, if in the current context the expression has some type.

$$\frac{ctx \vdash_{ex} exp : T}{ctx \vdash_I exp \ : \ OK}$$

Type checking of a conditional instruction. A conditional instruction of the form `if` exp_1 `then` I_1 `else` I_2 `end` is type correct in context *ctx* if the following conditions are met:

- The expression exp_1 (denoting the logical condition) has the type `{Boolean}`.
- Instructions I_1 and I_2 are both type correct.

$$\frac{ctx \vdash_{ex} exp_1 : \{\texttt{Boolean}\} \qquad ctx \vdash_{I} I_1 \ : \ OK \qquad ctx \vdash_{I} I_2 \ : \ OK}{ctx \vdash_{I} \texttt{if} \ exp_1 \ \texttt{then} \ I_1 \ \texttt{else} \ I_2 \ \texttt{end} \ : \ OK}$$

Type checking of a loop instruction. A loop instruction `while (`exp_1`)` I_1 `end` is type correct if the condition exp_1 has type `{Boolean}`, and the instruction I_1 is type correct .

$$\frac{ctx \vdash_{ex} exp_1 : \{\texttt{Boolean}\} \qquad ctx \vdash_{I} I_1 \ : \ OK}{ctx \vdash_{I} \texttt{while (}exp_1\texttt{)} \ I_1 \ \texttt{end} \ : \ OK}$$

Type checking of a `return` instruction. The `return` exp instruction placed in the context of a method is type correct if the expression exp has the return type of the method specified by the context.

$$\frac{(Mix, mtID) \vdash_{ex} exp : RetType^{mtID}_{Mix}}{(Mix, mtID) \vdash_{I} \texttt{return} \ exp \ : \ OK}$$

Type checking of `super[...]` instruction placed in an ini module. The `super[...]` instruction placed in the context of an ini module is type correct if:

- The names $par_1, ..., par_n$ of the parameters used in the expression match the declarations of output parameters present in the context.
- The actual parameters $exp_1 ... exp_n$ have the same types as the output parameters specified by the context.

$$\frac{\begin{array}{c} ctx|_2 = (par_1 : T_1, ..., par_n : T_n) \\ ctx \vdash_{ex} exp_1 : T_1 \quad ... \quad ctx \vdash_{ex} exp_n : T_n \end{array}}{ctx \vdash_{I} \texttt{super [}par_1\texttt{:=}exp_1, ..., par_n\texttt{:=}exp_n\texttt{]} \ : \ OK}$$

8.6 Type checking of object expressions

The type of an identifier.

$$ctx \vdash_{ex} VarName : IdTypes(ctx)(VarName)$$

Types of constants.

$$ctx \vdash_{ex} \texttt{true} : \{\texttt{Boolean}\}$$

$$ctx \vdash_{ex} \texttt{false} : \{\texttt{Boolean}\}$$

The value `null` is a proper value for each type, therefore the type of `null` must be a subtype of all types. Thus, it is the set of all names of mixins declared in the program.

$$ctx \vdash_{ex} \texttt{null} : MixNames$$

Type of a field value dereference. A field dereference expression of the form $exp\,.\,Mix\,.\,fl$ has the type T if:

- The expression exp has some type T_2, which contains the mixin name Mix.
- The declaration of the mixin Mix contains the field declaration $fl : T$.

$$\frac{ctx \vdash_{ex} exp : T_2 \qquad Mix \in T_2 \qquad (fl : T) \in Fields_{Mix}}{ctx \vdash_{ex} exp\,.\,Mix\,.\,fl : T}$$

Type of a method call expression. According to the below rule, a method call expression of the form $exp\,.\,Mix\,.\,mt(exp_1, ..., exp_n)$ has type $RetType_{Mix}^{(Mix,mt)}$ if the following conditions are met:

- The expression exp denoting the target object has type $\{Mix\}$.
- The declaration of the method $Mix\,.\,mt$ contains the declaration of parameters with types T^1, ..., T^n.
- Each of the actual method parameters $exp_1, ..., exp_n$, has the corresponding type in $T^1, ..., T^n$.

$$\frac{\begin{array}{c} ctx \vdash_{ex} exp : \{Mix\} \\ (p_1 : T^1,\ ...,\ p_n : T^n) = MetParams_{Mix}^{(Mix,mt)} \\ ctx \vdash_{ex} exp_1 : T^1 \qquad ... \qquad ctx \vdash_{ex} exp_n : T^n \end{array}}{ctx \vdash_{ex} exp\,.\,Mix\,.\,mt(exp_1, ..., exp_n) : RetType_{Mix}^{(Mix,mt)}}$$

Type of a `super` call expression. The `super(...)` call expression placed in context $(Mix, mtID)$ has type $RetType_{Mix}^{mtID}$ if and only if method $mtID$ declared in mixin Mix is marked with the keyword `override`, and the actual parameters of `super(...)` call have the same types as formal parameters in the method declaration.

$$\frac{\begin{array}{c} MetSpec_{Mix}^{mtID} = \texttt{override} \qquad (p_1 : T_1, ..., p_n : T_n) = MetParams_{Mix}^{mtID} \\ (Mix, mtID) \vdash_{ex} exp_1 : T_1 \qquad ... \qquad (Mix, mtID) \vdash_{ex} exp_n : T_n \end{array}}{(Mix, mtID) \vdash_{ex} \texttt{super}\ (exp_1, ..., exp_n) : RetType_{Mix}^{mtID}}$$

Type of a `new` object creation expression. An expression creating an object from mixins $Mix_1, ..., Mix_n$ is type correct if and only if the following conditions are met:

- The sequence $Mix_1, ..., Mix_n$ is consistent (see Section 6.7).
- For each $k = 1...n$, all base mixins mixin Mix_k are included in $\{Mix_1, ..., Mix_{k-1}\}$.
- All the ini modules declared in mixins as `required` are activated by the supplied set of input parameters.

- The actual initialization parameters $exp^1, ..., exp^i$ have types $T^1, ..., T^i$ matching the declarations of the corresponding formal parameters.

- Each initialization parameter $Mix^k.par^k$ (for $k \in 1...i$) used in the expression is present in the declaration of some initialization module occuring in the declaration of a mixin named Mix^k, and the declared type of this parameter is T^k.

$$\frac{\begin{array}{c} (Mix_1, ..., Mix_n) \text{ is consistent} \\ \forall_{k=1...n} \quad base(Mix_k) \subseteq \{Mix_1, ..., Mix_{k-1}\} \\ Rmodules(\{Mix_1, ..., Mix_n\}) \subseteq activated(\{Mix_1, ..., Mix_n\}, \{Mix^1\texttt{.}par^1, ..., Mix^i\texttt{.}par^i\}) \\ ctx \vdash_{ex} exp^1 : T^1 \qquad ... \qquad ctx \vdash_{ex} exp^i : T^i \\ \forall_{k=1...i} \quad ...\ Mix^k\ \texttt{(}\ ...\ par^k : T^k\ ...\ \texttt{)}\ \texttt{begin}\ ...\ \texttt{end} \in IModules_{Mix^k} \end{array}}{ctx \vdash_{ex} \texttt{new}\ Mix_1\texttt{,}\ ...\texttt{,}\ Mix_n\ \texttt{[}Mix^1.par^1\texttt{:=}\ exp^1, ..., Mix^i.par^i\texttt{:=}\ exp^i\texttt{]} : \{Mix_1, ..., Mix_n\}}$$

8.6.1 Subtyping

The below subtyping rule states that an expression E has type T_2, if it has type T_1, and type T_1 is a subtype of T_2,

$$\frac{ctx \vdash_{ex} E : T_1 \qquad T_1 \preceq T_2}{ctx \vdash_{ex} E : T_2}$$

where the subtyping relation is defined as follows (see Section 6.5 for the definition of *baseExt* function):

$$\frac{T_1 = T_2}{T_1 \preceq T_2} \qquad \frac{T_2 \subseteq T_1}{T_1 \preceq T_2} \qquad \frac{T_2 = baseExt(T_1)}{T_1 \preceq T_2}$$

8.7 The structure of a type checking derivation

In this section we present two properties of the type system of Magda.

8.7.1 The type checking system is syntax driven

Notice that, each rule used to derive the judgment of the form *the term t has some type (or "is OK")*, is *syntax driven*, which means that:

- each premise has the form *the term t′ has some type*, where t' is a subterm of t;

- all premises contain disjoint subterms of t. This means that a term used in some premise is never a subterm of the term occurring in another premise;

- each subterm of t occurs in exactly one premise.

Thanks to such construction of rules, we have the following property.

Property 6 (Syntax driven type checking) *For each program P such that $\vdash_P P : OK$, and each instruction I in P, the derivation tree of $\vdash_P P : OK$ contains exactly one judgment of the form $ctx \vdash_I I : OK$ for some ctx. Similarly, for each expression exp in the program, the derivation tree of $\vdash_P P : OK$ contains exactly one judgment of the form $ctx \vdash_{ex} exp : T$ for some ctx.*

8.7.2 The type checking context

From now on, we use the notion of *type checking context* of an expression exp or an instruction I to denote the context ctx in the judgment of the from $ctx \vdash_I I : OK$ or $ctx \vdash_I exp : T$ occurring within the derivation of $\vdash_P P : OK$. Since all type checking rules are syntax driven, we know that there is exactly one such judgment for each subterm of the program.

Additionally, notice that each rule used to derive a judgment of the form $ctx \vdash_I I : OK$, or $ctx \vdash_{ex} exp : T$ uses in all its premises the same context ctx. As a result, we know that each such judgment for instruction I or expression exp contains the same context as the one present in the judgment stating that the method or the initialization module containing I/exp is type correct. Therefore, the following property holds:

Property 7 (Type checking context of instruction/expression) *For every judgment j of the form $ctx \vdash_{ex} exp : T$ or $ctx \vdash_I I : OK$ in the derivation tree of the judgment $\vdash_P P : OK$, we have:*

- *either judgment j is contained within the subtree rooted at a judgment of the form $\vdash_{met} mt : OK$ or $\vdash_{ini} mod : OK$, and:*
  - *exp or I occurs within the body of mt or mod, and*
  - *ctx occurs in the premise of the judgment $\vdash_{met} mt : OK$ or $\vdash_{ini} mod : OK$ (notice that rule for each of those judgments contains only one premise);*
- *or exp / I occurs within a main instruction of P and $ctx = \top$.*

Less formally: the type checking context of an instruction and an expression is always a context of the surrounding method or the surrounding initialization module. Notice also that, the semantics of Magda enjoys a similar property, as stated in Section 7.3.

8.8 The notion of the biggest type

Notice that, in the presence of subtyping, many expressions in Magda can be assigned with multiple types, which means that for a given exp and ctx, the judgment $ctx \vdash_{ex} exp : T$ holds for many different T, and each such judgment has also many different derivation trees. Therefore, to define the subject reduction property (and some others) we introduce the notion of biggest type.

We say that T is the *biggest type* of expression *exp* in the context *ctx* when it is the biggest (wrt. the inclusion) element of the set:

$$\{T' \mid ctx \vdash_{ex} exp : T'\}.$$

The biggest element here is chosen with respect to inclusion, since each type in Magda is a set of mixin names (see Section 3.9 and Section 4.6). Notice that, in Magda, the notion of biggest type coincides with the notion of principal type (see for example: Joe Wells [64]). Informally: the biggest type is the "maximal type information" we can have about the given expression in the given context. Notice that the biggest type of an expression is a subtype of every type of this expression.

To understand how the biggest type of a specific expression is formed, let us analyze how do all the derivations of a judgment $ctx \vdash_{ex} exp : T'$ look like. Notice that if we remove from the derivation all the occurrences of the subtyping rule, then all the remaining rules are deterministically chosen by the syntactic form of the expression. Moreover, in each rule used to derive the judgment $ctx \vdash_{ex} exp : T'$, type T' does not depend on types of subterms of *exp*. Thus the subtyping rules inside the tree are only used to verify whether the subterms have matching types and do not influence the resulting type of the whole expression. Therefore, all derivations of all types of an expression *exp* have the same shape and differ only in the usage of the subtyping rules for *exp*.

As a result it is easy to see that the type of the expression only depends on the subtyping rules applied "after the last structural rule", which means on top of all other rules.

Therefore, having any derivation of any type of an expression, we obtain a derivation of the biggest type (and the biggest type itself) in the following way: we remove all the occurrences of the subtyping rule used "after the last structural rule", and add one subtyping rule, which replaces the type T generated by the "structural rule" with *baseExt*(T) (for definition of *baseExt* see Section 6.5). As a result, it is easy to see that if the set of all types of an expression has at least one element, then it also has the biggest one.

Furthermore, as a result of the above construction of the biggest type, it is also easy to see that the following property holds.

Property 8 (The biggest types of expressions) *The biggest types of expressions have the following forms:*

- *The biggest type of a variable var is equal to baseExt*(T_1)*, where* T_1 *is the type present in the declaration of the variable var.*
- *The biggest type of a field dereference expression e . Mix . fl is baseExt*(T_1)*, where* T_1 *is the type present in the declaration of field fl in mixin Mix.*
- *The biggest type of a method call expression* $e\,.\,Mix\,.\,mt\,(\overrightarrow{e}\,)$ *is baseExt*(T_1)*, where* T_1 *is the result type of method Mix.mt — denoted as* $RetType_{mt}^{(Mix,mt)}$ *(see Section 6.3) .*
- *The biggest type of an object creation expression* `new` $T_1[...]$ *is* T_1.

Notice that in case of object creation expression the biggest type is T_1, because the type checking rule for the new object creation expressions (see Section 8.6) ensures that $T_1 = baseExt(T_1)$.

Chapter 9

Computation steps

9.1 Traditional approach to type safety

The semantics of Magda defined in Section 7 is a big-step semantics [43] (or natural semantics as it is sometimes called). This is a very useful and natural tool for expressing the behavior of recursive programs.

A big-step semantics can be naturally used to prove statements of the form: "A program P1 terminates correctly and finishes in state S1" (see for example [32]). However, since all the derivation trees built using such semantics represent only terminating programs, such semantics cannot be used to formulate and analyze the non-termination of programs (as the small-step semantics can [59]). Additionally, it is difficult to state that the execution of program P1 gets stuck in a point X, or to prove that the execution of the program will not get stuck. As a result, it is more difficult to express the subject reduction and the type soundness of a language. This problem can be solved partially by adding special values representing the notion of getting stuck (like in [32]) and by adding rules which propagate such special values from the evaluation of some subterms to the whole terms. As a result of such modifications the fact of getting stuck is modeled by the reduction of a program to such a special value. However, this approach requires many additional artificial rules, which often means doubling the number of rules in semantics. Moreover, it cannot be easily verified if such special values and additional rules cover all the cases of the actual "getting stuck" of the program execution.

These problems have already been pointed out in literature. Moreover, there also exist some solutions in [51, 41, 5]. The first two of those solutions use the coinductive interpretation of the big-step style semantics in order to formalize the notion of non-termination of a program. However, this solution also requires additional rules and uses a less intuitive interpretation of semantics.

9.2 Our approach to type safety

To solve the mentioned problems we have decided to develop a new approach to modeling type safety in the big-step semantics. This approach allows us to formalize the notion of intermediate steps of the program execution, and then to formulate statements concerning non-termination, getting stuck etc. Our approach, presented below, allows one to define the notion of intermediate steps of the computation using only the above defined big-step semantics.

Our solution is in many respects similar to the one presented by Ager in [5]. We present the similarities and differences in Section 9.10.

Our solution (as well as Ager's one) exploits the sequentiality of the big-step semantics rules (see Section 7.2.3) to discover the sequence of intermediate steps of the execution inside the derivation tree for the program as well as to model infinite execution.

9.3 The evaluation by the derivation-search

If we take a closer look at the shape of derivation trees built using our semantics, and the shape of the rules (and sequentiality as well as determinism, as introduced in Section 7.2.3) it is easy to see that there exists a greedy derivation-search partial algorithm for the Magda language. By derivation-search algorithm we understand the algorithm which:

> Given a partial judgment (see Section 7.2) of the form $env, ctx, st \models I \Rightarrow^I ?$, where ? is unknown, the algorithm checks if there exist env', ctx', such that $env, ctx, st \models I \Rightarrow^I env', ctx'$ is derivable. Apart from checking if such env', ctx' exist, the algorithm computes their values, together with the derivation tree for such a judgment.

The fact that the algorithm is partial means that if the program terminates (i.e., there exists a derivation tree for a judgment of the form $... \models I \Rightarrow^I (\emptyset, st))$, then the algorithm will find the derivation tree and the value of st. Otherwise the algorithm will loop.

Furthermore the execution of this algorithm (and the building of the derivation tree) computes all the intermediate states and environments and as a result it mimics the execution process of the program. Hence, if the execution of an analyzed program is infinite, the execution of the derivation-search algorithm is also infinite. Therefore this algorithm allows us to build a sequence of program configurations which represent all the intermediate states of the execution of a program.

As a result, the analysis of sequences of program configurations (called later *traces* — see Section 9.6) will allow us to state and prove properties concerning program non-termination and to analyze the process of getting stuck. We will be also able to formulate statements of the form: "a given program gets stuck in the given configuration".

In next sections we describe the details of the algorithm (see Section 9.4), then (using this algorithm) we formalize the notion of a sequence of program configurations (see Section 9.6). Next in Chapter 10, using the notion of a sequence of program configurations, we state and prove the subject reduction theorem saying that all the configurations in a trace of a type correct program are type safe. Finally in Chapter 11 we state and prove the type soundness of Magda by stating that a type correct program will not get stuck in any case except null pointer dereference (see Section 9.8).

9.4 The greedy derivation-search algorithm

The question one answers using big-step operational semantics is the following one: does the program P terminate and if it does then in what state? In other words, the semantics is used to build a derivation for a partial judgment of the form: ... $\models$ *instructions* $\Rightarrow^I (\emptyset, ?)$ and to find the value of ? in that derivation.

For our derivation system, there exists a partial deterministic greedy algorithm which finds the answer to the above question. This algorithm, given a program which terminates, builds a derivation tree for such a judgment by recursively constructing subtrees and then combining them into a final tree. The algorithm is implemented as a recursive procedure. The procedure takes as a parameter a partial judgment of the form $X \models Y \Rightarrow ?$, where $\Rightarrow$ is one of the following symbols: $\{\Rightarrow^I, \Rightarrow^{ex}, \Rightarrow^{ini}\}$ and X, Y are fixed values. We use the notion of a partial judgment here, since the result of evaluation is unknown (marked with ? above). If there exists Z such that $X \models Y \Rightarrow\ Z$ has a derivation tree, then the procedure finds this Z, together with the derivation tree for that judgment.

9.4.1 The base logic of the algorithm

The algorithm works in the following way. It applies the recursive procedure to the final goal (which is a partial judgment saying that the whole program evaluates to some unknown environment and state). The recursive procedure in turn implements the following logic: For each supplied partial judgment, the procedure starts from picking the applicable rule (or set of rules), which can be used to derive that kind of judgment. Notice that for most kinds of judgments there exist only one rule which can be used to derive such a judgment, which makes this choice obvious. Then, if one rule is available, the procedure traverses all the premises of that rule (which are ordered sequentially, as mentioned in Section 7.2.3). For each premise, a partial judgment is calculated and for each such partial judgment the procedure recursively calls itself in order to find the value of ? occurring in the partial judgment and to build the derivation tree for that premise. Recall that we use the word "premise" for each assumption of the form ... $\models$ ... $\Rightarrow$ ..., as opposed to the "side-condition" (see Section 7.2.2).

The partial judgments for the premises of the rule are build in the following way: Let us assume that the actual parameter of a procedure is a partial judgment of the form $S \models \ldots \Rightarrow ?$. Recall from Section 7.2.3 that in each rule the first premise of the form $\ldots S_1 \ldots \models t \Rightarrow (\ldots, S_2, \ldots)$ is the one in which $S_1 = S$. Moreover, each premise following the first one, on the left side of the $\models$ symbol, uses a state which is determined by the state used on the right side of the previous premise. Moreover, the resulting state of the conclusion judgment of the rule is determined by the resulting state of the last premise.

Once we have the premises ordered we recursively execute the procedure for those premises. If the recursive execution of the procedure returns with the derivation tree and the finishing state of a given premise, then we use this state to execute the procedure for the next premise. If the procedure finds the derivation tree and the resulting state for the last premise of the given target, then it returns the whole tree for the target rule, with the resulting state of the last premise used as the resulting state of the conclusion of the rule. The algorithm terminates thanks to rules which have no premises, even though those can contain some conditions. Those rules are in fact axioms and therefore serve the role of leafs in the derivation trees.

9.4.2 Multiple rules for one judgment

The only case left which needs additional treatment is the one in which the procedure is executed for a judgment that can be derived using more than one rule. There exist four different forms of judgments derivable using more than one rule: initialization process, compound instruction, `if` and `while` instructions. The procedure deals with that four cases in the following way (depending on the form of the supplied judgment):

- Judgments of the form $(...,...) \models (\ \overrightarrow{mods},\quad \overline{pars}) \Rightarrow^{ini}$? can be derived using three different rules. However, by knowing only the actual values of $\overrightarrow{mods}$ and $\overline{pars}$, the procedure automatically chooses the correct rule, using the following logic:

  When $\overrightarrow{mods}$ is an empty sequence, then the first rule is applied. When $\overrightarrow{mods}$ contains at least one module then only the second and the third one can be applied. In the second case, we analyze the value of $\overline{pars}$ and compare it to the last element of $\overrightarrow{mods}$ (which is the initialization module being analyzed). In case when all the input parameters of the last module have their values in $\overline{pars}$, then the third rule is used, which is ensured by two first side-conditions of that rule. Otherwise, when no input parameter of the last module is in $\overline{pars}$ (when the module has at least one input parameter), then the second rule is used, which is ensured by the first condition of that rule.

  Moreover, the above mentioned side-conditions of the second and the third rule are mutually exclusive, therefore the procedure can deterministically choose which rule to use.

- Judgments of the form $(...,...,...) \models I_1\ ;\ I_2 \Rightarrow^{I}?$ can be derived using two rules. However, when we order the premises of both of those rules, then we can see that they begin with an identical premise, concerning the execution of I_1 instruction. Therefore, the procedure is executed recursively for that first premise, postponing the actual choice of the rule. Then, when this recursive call for I_1 returns with the value of the global state and the value of local environment, the procedure chooses the appropriate rule using the value of the returned environment. This choice can be performed deterministically using the mutually exclusive conditions occurring within those rules (checking whether $env' = (\top, x)$ or not).

- In the case of judgments $(...,...,...) \models$ `if` (exp) `then` $... \Rightarrow^{I}?$ there are also two rules and the procedure deals with them similarly as with a compound instruction. Both of the applicable rules begin with the same premise, concerning the condition expression exp. Therefore, the procedure executes itself recursively for the first premise. Once this recursive call finishes, depending on the returned value (whether it is tt or ff) and the mutually exclusive conditions, the procedure chooses the appropriate rule.

- In the case of judgment $(...,...,...) \models$ `while (`exp`)` I_1 `end` $\Rightarrow^I ?$, there are three rules, however we can deal with them as in the previous cases. Using twice the technique of delaying the choice of the rule until one premise is evaluated, the procedure can limit the set of applicable rules until only one rule is left.

  First of all, notice that all three rules applicable begin with the same premise, concerning the evaluation of the logical expression exp. Therefore, the procedure begins with the recursive call evaluating the expression exp. When the recursive call finishes and this expression evaluated to ff, then the first rule is applied. If the expression evaluated to tt, then we still have the second and third rule to choose from. However, those two rules also share the second premise. Therefore the procedure further delays the choice between those two rules and executes itself recursively for the second (common) premise to evaluate the instruction. Then, when the recursive call finishes and returns the value of the state and the environment, the procedure chooses the appropriate rule using the condition referring to the value of the environment (whether $env'' = (\top, x)$ or not).

Hence, we have shown that, for each judgment, the procedure can deterministically choose one rule to be applied, and calculates the values of state and environment used on the left side of all the premises in the rules (thanks to the sequential ordering of premises — see Section 7.2.3).

9.4.3 A pseudo-code definition of the derivation-search procedure

In this section we present a more detailed definition of the search procedure using pseudo-code. For simplicity, in this definition we have removed all the instructions which build the derivation tree itself. This procedure just checks whether such derivation exists, and if a derivation for some partial judgment exists, then what the result of the evaluation of that judgment is. In other words, it calculates the value of ? in the partial judgment $.... \models t \Rightarrow ?$. All the judgments manipulated within this definition are in fact partial judgments, therefore we have decided to skip the ? symbol.

Additionally, notice that the form of a partial judgment determines whether it describes an evaluation process of an expression, an instruction or an object initialization (depending on whether $\Rightarrow$ is equal to $\Rightarrow^{ex}$, $\Rightarrow^{I}$, $\Rightarrow^{ini}$). Therefore, to obtain a more intuitive and readable presentation, we have split the derivation procedure into three procedures — `SearchE`, `SearchI`, `SearchInit`, responsible for expressions, instructions and object initialization respectively. Moreover, each occurrence of the recursive call to the derivation-search procedure has been replaced with a call to one of the mentioned procedures. The procedure to be called at each point depends on the form of the argument of such recursive call. Finally, the execution of the whole program is defined by the execution of the following procedure: `SearchI(`$\emptyset, \top, \{\mathit{tt} \mapsto \mathit{BoolVal}, \mathit{ff} \mapsto \mathit{BoolVal}\} \models I \Rightarrow^I$`)`

A full definition of the derivation-search procedure written in pseudo-code is given on Figures 9.1, 9.2, 9.3. This code contains keywords which are parts of the derivation-search procedure in pseudo-code, and also keywords and operators which are parts of the terms written in Magda (which serve as values in the program in pseudo-code, and as patterns in pattern-matching). Therefore, to distinguish those two worlds we use underline to denote keywords in the pseudo-code (like `procedure`, `case`) and normal text for the code in Magda. In most of the pseudo-code we exploit the pattern matching technique. We use it in the `case` statements, as well as in the assignments `:=`. As usual (like in SML[54]) we assume that if the value does not match any pattern, then the program fails with an error.

The implementation of procedures `SearchE`, `SearchI` and `SearchInit` follows the pattern described in Section 9.4.1 and in Section 9.4.2. This means that each of those procedures works in the following way:

- First, the pattern matching by the syntactical form of the partial judgment is performed (via the main `case` switch of each procedure).
- Then, for each form of judgment, the premises of the rule responsible for the derivation of the given judgment are processed from left to right. As we mentioned before, this left to right processing is possible thanks to the sequential ordering of premises — see Section 7.2.3.
- In most cases (when there is only one rule for a judgment) the sequence of the instructions within the given branch of `case` statement contains one instruction per each premise in a rule responsible for the given form of judgment.
- In case of the four kinds of judgments described in Section 9.4.2 we apply the mentioned treatment. This means that the procedures first evaluate the premises which can be evaluated without the choice of the rule, and then the procedures choose the appropriate rule and continue with the evaluation of further premises.

9.4.4 Soundness and completeness of the algorithm

The above defined algorithm is sound and complete in the following sense:

Property 9 (Soundness and completeness of the algorithm) *For each program P the following are equivalent:*

- *The program P terminates in a state S, which means that there exists a derivation tree for the final judgment $\emptyset, \top, \{\, tt \mapsto BoolVal,\ ff \mapsto BoolVal\} \models I \Rightarrow^{I} (\emptyset, S)$.*
- *The derivation-search procedure* `SearchI`$(\emptyset, \top, \{\, tt \mapsto BoolVal,\ ff \mapsto BoolVal\} \models I \Rightarrow^{I}\ )$ *finishes and returns the pair $(\emptyset, S)$.*

This property can be easily verified, because the derivation tree can be transformed into the structure of the recursive calls of the derivation-search procedure and vice versa.

Similarly, for each program P which has an infinite execution (in the sense of the intuitive meaning described in Section 3), this procedure will also execute forever.

```
procedure searchI(j)
 case j of
  $env, ctx, st \models VarName\texttt{:=}exp \Rightarrow^{I}$:
      $(st', adr)$ := SearchE($env, ctx, st \models exp \Rightarrow^{ex}$);
      res := $(env\{VarName \mapsto adr\},\ st')$;
  $env, ctx, st \models exp_1 . Mix . fl\texttt{:=}exp_2 \Rightarrow^{I}$:
      $(st', adr')$ := SearchE($env, ctx, st \models exp_1 \Rightarrow^{ex}$);
      $(st'', adr'')$ := SearchE($env, ctx, st' \models exp_2 \Rightarrow^{ex}$);
      $newObj$ := $(\ st''(adr')|_1,\ st''(adr')|_2\{Mix.fl \mapsto adr''\})$;
      res := $(env, st''\{adr' \mapsto newObj\})$;
  $env, ctx, st \models$ if $exp_1$ then $I_1$ else $I_2$ end $\Rightarrow^{I}$:
      $(st', v)$ := SearchE($env, ctx, st \models exp_1 \Rightarrow^{ex}$);
      case $v$ of $tt$ :$(env'', st'')$ := SearchI($env, ctx, st' \models I_1 \Rightarrow^{I}$);
                 $ff$ : $(env'', st'')$ := SearchI($env, ctx, st' \models I_2 \Rightarrow^{I}$);
      endcase;
      res := $(env'', st'')$;
  $env, ctx, st \models$ return $exp \Rightarrow^{I}$:
      $(st',\ adr')$ := SearchE($env, ctx, st \models exp \Rightarrow^{ex}$);
      res := $((\top, adr'),\ st')$;
  $env, ctx, st \models I_1 ; I_2 \Rightarrow^{I}$:
      $(env', st')$ := SearchI($env, ctx, st \models I_1 \Rightarrow^{I}$);
      case $env'$ of $(\top, x)$:res := $(env', st')$;
                     else: res := SearchI($env', ctx, st' \models I_2 \Rightarrow^{I}$);
      endcase;
  $env, ctx, st \models \epsilon \Rightarrow^{I}$:
      res := $(env, st)$;
  $env, ctx, st \models exp \Rightarrow^{I}$:
      $(st', adr)$ := SearchE($env, ctx, st \models exp \Rightarrow^{ex}$);
      res := $(env, st')$;
  $env, ctx, st \models$ while ($exp_1$) $I_1$ end $\Rightarrow^{I}$:
      $(st', v)$ := SearchE($env, ctx, st \models exp_1 \Rightarrow^{ex}$);
      case $v$ of $ff$ :res := $(env, st')$;
                 $tt$ : $(env'', st'')$ := SearchI($env, ctx, st' \models I_1 \Rightarrow^{I}$);
                      case $env''$ of $(\top, x)$:res := $(env'', st'')$;
                                   else: res := SearchI($env'', ctx, st'' \models$ while ($exp_1$) $I_1$ end $\Rightarrow^{I}$);
                      endcase;
      endcase;
 endcase;
 return res;
endprocedure
```

Figure 9.1: The derivation-search procedure for instructions

```
procedure searchE(j)
 case j of
  $env, ctx, st \models$ true $\Rightarrow^{ex}$:       res := $(st,\ tt)$;
  $env, ctx, st \models$ false $\Rightarrow^{ex}$:       res := $(st,\ ff)$;
  $env, ctx, st \models$ null $\Rightarrow^{ex}$:       res := $(st, null)$;
  $env, ctx, st \models VarName \Rightarrow^{ex}$:       res := $(st, env(VarName))$;
  $env, ctx, st \models exp\,.\,Mix\,.\,fl \Rightarrow^{ex}$:
    $(st', adr)$ := SearchE($env, ctx, st \models exp \Rightarrow^{ex}$);
    res := $(st',\quad st'(adr)|_2(Mix.fl))$;
  $env, ctx, st \models exp\,.\,Mix\,.\,mt(exp_1,\ ...,\ exp_n) \Rightarrow^{ex}$:
    $(st_0, adr_0)$ := SearchE($env, ctx, st \models exp \Rightarrow^{ex}$);
    for i := 1 to n do
        $(st_i, adr_i)$ := SearchE($env, ctx, st_{i-1} \models exp_i \Rightarrow^{ex}$);
    $Mix'$ := $LastMix(\ st_n(adr_0)|_1, (Mix, mt)\ )$;
    $(p_1 : T_1,\ ...,\ p_n : T_n)$ := $MetParams^{(Mix,mt)}_{Mix'}$;
    $(local_1,\ ...,\ local_k)$ := $MetLocals^{(Mix,mt)}_{Mix'}$;
    $env'$ := $\{p_1 \mapsto adr_1; ...; p_n \mapsto adr_n; local_1 \mapsto null; ...; local_k \mapsto null; \texttt{this} \mapsto adr_0\}$;
    $((\top, adr), st')$ := SearchI($env', (Mix', (Mix, mt)), st_n \models MetInstr^{(Mix,mt)}_{Mix'} \Rightarrow^{I}$);
    res := $(st', adr)$;
  $env, (Mix, mtID), st \models$ super $(exp_1,\ ...\ , exp_n) \Rightarrow^{ex}$:
    for i := 1 to n do
        $(st_i, adr_i)$ := SearchE($env, (Mix, mtID), st_{i-1} \models exp_i \Rightarrow^{ex}$);
    $Mix'$ := $LastMixBef(st(env(\texttt{this}))|_1, mtID, Mix)$;
    $(p_1 : T_1,\ ...\ , p_n : T_n)$ := $MetParams^{mtID}_{Mix'}$;
    $(local_1,\ ...\ , local_k)$ := $MetLocals^{mtID}_{Mix'}$;
    $env'$ := $\{p_1 \mapsto adr_1; ...; p_n \mapsto adr_n; local_1 \mapsto null; ...; local_k \mapsto null; \texttt{this} \mapsto env(\texttt{this})\}$;
    $((\top, adr), st')$ := SearchI($env', (Mix', mtID), st_n \models MetInstr^{(Mix,mt)}_{Mix'} \Rightarrow^{I}$);
    res := $(st', adr)$;
  $env, ctx, st_0 \models$ new $Mixins[ParID_1{:=}exp_1, ..., ParID_k{:=}exp_k] \Rightarrow^{ex}$:
    for i := 1 to k do $(st_i, adr_i)$:= SearchE($env, ctx, st_{i-1} \models exp_i \Rightarrow^{ex}$);
    $objVal$ := $EmptyObject(mixins)$;
    $adr'$ := $FirstEmpty(st_k)$;
    $st''$ := SearchInit($adr',\ st_k\{adr' \mapsto objVal\} \models (IniModules(mixins), \overline{ParID \mapsto adr}) \Rightarrow^{ini}$);
    res := $(st'',\ adr')$;
 endcase;
 return res;
endprocedure
```

Figure 9.2: The derivation-search procedure for expressions

```
procedure searchInit(j)
 case j of
  $adr, st \models ((), ()) \Rightarrow^{ini}$: res := $st$;
  $adr, st \models (\overrightarrow{mods} \cdot mod,\quad \overline{pars}) \Rightarrow^{ini}$:
    if $mod|_1 \subseteq dom(\overline{pars})$ then
      $(Mix.ip_1, ..., Mix.ip_k)$ := $mod|_1$;
      for i := 1 to k do
          $adr_i^I$ := $pars(Mix.ip_i)$;
      $\overline{pars'} = \overline{pars} \lessdot^* \{Mix.ip_1 \mapsto adr_1^I; ...; Mix.ip_k \mapsto adr_k^I\}$;
      $\overrightarrow{local : T}$ begin $I_1$; super[$\overrightarrow{opID := exp}$] $I_2$ end;  := $mod|_3$;
      $env$ := $\{ip_1 \mapsto adr_1^I; ...; ip_n \mapsto adr_k^I; local_1 \mapsto null; ...; local_m \mapsto null;$ this $\mapsto adr\}$;
      $(env', st_0)$ := SearchI($env, mod, st \models I_1 \Rightarrow^I$);
      for i := 1 to k do
          $(st_i, adr_i^O)$:= SearchE($env', mod, st_{i-1} \models exp_i \Rightarrow^{ex}$);
      $\overline{pars''} = \overline{pars'}\{opID_1 \mapsto adr_1^O; ...; opID_l \mapsto adr_l^O\}$;
      $st''$ := SearchInit($adr, st_l \models (\overrightarrow{mods},\quad \overline{pars''}) \Rightarrow^{ini}$);
      $(env'', st''')$ := SearchI($env', mod, st'' \models I_2 \Rightarrow^I$);
      res := $st'''$;
    elseif
      res:= SearchInit($adr, st \models (\overrightarrow{mods},\quad \overline{pars}) \Rightarrow^{ini}$);
    endif;
 endcase;
 return res;
endprocedure
```

Figure 9.3: The derivation-search procedure for initialization

9.4.5 The model of the program execution

The above defined greedy derivation-search algorithm, models the process of the execution of a given program P, together with its intermediate steps. This process consists of subsequent recursive calls to the above defined procedure. Therefore we will say that the instruction I_1 *executes* the instruction I_2, if during the activation of the procedure for a partial judgment of the form $... \models I_1 \Rightarrow^I ?$ the procedure was activated for a partial judgment $... \models I_2 \Rightarrow^I ?$. Similarly we will say that *during the execution of program P, instruction I was executed in state st*, if during the execution of this algorithm the procedure was activated for the partial judgment $..., st \models I \Rightarrow^I ?$.

The detailed definition of the execution process based on this algorithm is present in Section 9.6.

9.5 Context consistency

In this section we state a property enjoyed by the semantics and the type system of Magda. This property reflects the correspondence between: (i) the context in which an expression or an instruction is evaluated and (ii) the context in which the type checking of an instruction is performed. This property informally says: An instruction or an expression in a type checked program is executed always in the type checking context of that expression or instruction (see Section 8.7.2). Formally, this property is stated as follows.

Property 10 (Context consistency) *Let P be a type correct program. Let $...ctx... \models t \Rightarrow$, where $\Rightarrow \in \{\Rightarrow^I, \Rightarrow^{ex}\}$, be a partial judgment occuring as a parameter of an activation of the derivation-search procedure occurring during the execution of derivation-search (see Section 9.4) for the program P.*

Then the context ctx is a type checking context of the given expression or instruction t (see definition in Section 8.7.2).

To show this property, we first recall the *structure of derivation* property presented in Section 7.3. This property translated into terms of the derivation-search procedure can be formulated as follows: When the derivation-search procedure is executed for a partial judgment containing an instruction or expression, it is called (directly or indirectly) by the execution of that procedure for the body of a method/module or main instruction in which this expression/instruction occurs and this partial judgment contains the same context in which the whole body was evaluated.

Then consider Property 7 (see Section 8.7.2), which says that the type checking context of an instruction and expression is a type checking context of the method/module body in which it occurs. Notice also that the context used to type check the body of the method (see Section 8.4.1) is the context used to execute the body of that method. Similarly, the context used to type check the body of an ini module (see Section 8.4.2) is the context used to execute the body of this module.

All the three above facts conclude the proof of the context consistency property.

9.6 Execution trace

Using the above definition of the derivation-search algorithm we define the notion of the *execution trace*, or shortly *trace*. The trace is a sequence (finite or infinite) of *program configurations*, which represents the execution of the program. Below, in Section 9.6.1 we define what exactly is a program configuration, and in Section 9.6.2 we define how a program trace is constructed for a given program.

9.6.1 Program configuration

Each *program configuration* represents a state of the program execution. Each configuration contains two kinds of information. First of all, it contains the information about the current dynamic state, which means the global memory state and the states of local variables of the currently executed method/ini module, or the values of parameters to be consumed during the object initialization. Secondly, it contains information representing the current point of execution in the static structure of the program. In other words, for each local dynamic information in the configuration, we also have the static information denoting the part of the program to which this information refers. Therefore, each configuration has one of the following forms:

- a tuple of an environment, a static context, and a state.
- a tuple of an environment, a static context, a state, an address, and an expression.
- a tuple of an address, a state, a sequence of ini modules, and a sequence of initialization parameters.
- a state.

For every configuration C, we write $state(C)$ to denote the *state* element of every such a configuration. Notice that each form of configuration contains exactly one *state* element, therefore $state(C)$ is defined for every configuration C.

We will say that configuration C_1 is a *subconfiguration* of configuration C_2 when the configuration C_1 has the form (env, ctx, st) and C_2 has the form (env, ctx, st, adr, exp).

9.6.2 The construction of the execution trace

For a given program P, the *execution trace of program P* is a sequence of program configurations built according to the below described algorithm.

We extend the derivation-search algorithm in the following way: At the beginning of the whole derivation-search algorithm (not at the beginning of the recursive procedure) we declare a variable `tr`, initially equal to an empty list of program configuration. Furthermore, each derivation-search procedure is extended by adding two sequences of instructions in the pseudo-code: one which is to be executed at the very beginning and the second to be executed at the end of the execution of each such procedure.

Each of the two new pieces of pseudo-code is responsible for an addition of one configuration to the tail of a sequence being a value of the variable `tr` (using the `tr.add(...)` in the pseudo-code). One configuration (called *opening configuration*) is added to the value of `tr` at the beginning of the execution of that call. This configuration contains the information about the state of the program before the rule is applied and therefore is constructed from the information present on the left side of the partial judgment supplied to the procedure call. The second configuration (called *closing configuration*) is added to the value of `tr` at the end of the call, i.e., at the moment when the procedure finds the value of ? in $X \models Y \Rightarrow ?$. Such a closing configuration contains the information about the state, environment etc, after that rule was executed. Therefore the closing configuration is built from the evaluated value of ? and from some components of X. We use in the closing configuration these components of X, which have not been modified by the evaluation of the judgment, thus are not part of ?. For example, the expression evaluation judgment $X \models exp \Rightarrow^{ex} ?$ does not contain a new environment on the right side of $\Rightarrow^{ex}$, therefore the closing configuration for this expression uses the environment occurring in X. Notice that this environment also occurs in the opening configuration added by this call.

As a result, in the generated sequence, between opening and closing configurations added by some activation of the recursive, procedure there are configurations generated by the nested activations of procedures originating from the given activation. An example of such sequence and its construction method is shown on Figure 9.6.

The actual forms of opening and closing configurations depend on the judgment which was the argument of the derivation-search procedure. Figure 9.4 shows which form of program configuration is used as an opening and closing configuration for each form of a partial judgment.

The extended version of the derivation-search algorithm is shown on Figure 9.5.

Judgment	**opening configuration**	**closing configuration**
$env, ctx, st \models instr \Rightarrow^{I} (env', st')$	(env, ctx, st)	(env', ctx, st')
$env, ctx, st \models exp \Rightarrow^{ex} (st', adr)$	(env, ctx, st)	$(env, ctx, st', adr, exp)$
$adr, st \models (mods, pars)exp \Rightarrow^{ini} st'$	$(adr, st, mods, pars)$	st'

Figure 9.4: Program configurations

We will often use the notion of *opening configuration of the term t* or *closing configuration of the term t* to denote the opening or closing configuration added to the trace by the recursive call which had as its argument a partial judgment containing t. We use this notion in cases when it is clear from the context what is the exact form of the judgment. Therefore, in all those cases it is clear what the form of a given opening or closing configuration is (see Figure 9.4).

9.6.3 Properties of a program trace

Finally, depending on the behavior of the derivation-search algorithm, we have one of the below situations:

- When the algorithm finishes its execution (either successfully, or because of some error), then the value of variable tr at the end of its execution is the trace of the program.
- When the algorithm has an infinite execution, then the program trace is an infinite sequence understood as the lowest upper bound of the growing sequence tr.

9.6.4 Notion of *enclosing activation*

Suppose an activation X of the recursive procedure called an activation Y. If now activation Y adds configuration C to the trace, then we say that X is an *enclosing activation* (or, interchangeably, the *enclosing call*) of C.

We also say that judgment j is an *enclosing* judgment of configuration C if it the judgment being a parameter of the enclosing call X.

9.7 Successful program termination

Informally, we say that a program *terminated successfully*, when the execution of the main instructions I of the program finished and no error occurred.

Formally there are three definitions of the notion: *a program terminated successfully* (it is easy to see that they are equivalent):

1. The trace contains the closing configuration generated by the first call of the derivation-search procedure, which is the one responsible for the execution of the main instructions of the program. In such a case, this closing configuration is the last one in the trace.
2. A final judgment $\emptyset, \top, \{\mathit{tt} \mapsto \mathit{BoolVal},\ \mathit{ff} \mapsto \mathit{BoolVal}\} \models I \Rightarrow^{I} (\emptyset, st)$ can be derived for some state st.
3. The numbers of closing and opening configurations in the trace are equal.

9.8 Null pointer dereference

Informally, we say that *null pointer dereference* occurred during the program execution if the expression being a target of the method call, a target of the field dereference or used as a boolean condition evaluates to $null$. Such a situation is an error during the execution of the program. In our formulation of program execution trace it means that the derivation-search algorithm reached a configuration in which no rule can be applied, so the procedure fails, because of failing pattern-matching.

Using the above definition of trace, we define the notion in the following way: The *null pointer dereference* occurred during the program execution, if one of the below recursive calls of the derivation-search procedure generates a closing configuration of the form $(..., ..., ..., null, ...)$.

1. The only recursive call for a judgment containing a field value dereference.
2. The first recursive call for a judgment containing an object field assignment.
3. The first recursive call for a judgment containing an `if` instruction.
4. The first recursive call for a judgment containing a `while` instruction.
5. The first recursive call for a judgment containing a method call.

It is easy to see that if the trace contains such a configuration with *null* value then it is one of the last configurations in the trace (it might not be the last one — see the next paragraph). It happens so, because in every of the five above cases, if such a configuration is added by the given recursive call, then the main activation of the derivation-search procedure (of the one of the above five forms) will need to check the value of the state at the given address, which is *null* in this case. And the value of any state is not defined on *null*, so the procedure cannot finish its execution. This means that the given parent activation of the procedure will not be able to add its closing configuration.

In cases of some rules (like object field assignment and method call), some configurations can be added after such closing configuration containing *null* value. Those are the configurations added by the independent recursive calls coming from the same activation of the procedure, before it refers to the address which is *null*. However, as said above, the parent call will never be able to finish and add its closing configuration.

9.9 State preservation property

According to the above definition, each program execution trace enjoys the below formulated *state preservation property.*

Property 11 (State preservation) *For every two configuration C_1 and C_2 in the given trace, such that C_1 occurs before C_2 and for each $adr \in Dom(state(C_1))$ we have:*

$$state(C_1)(adr)|_1 = state(C_2)(adr)|_1$$

In other words, if at some point during the program execution, some object is stored at some address in the memory, then this object is stored at this address forever and it does not change its runtime type.

To prove the state preservation property it is enough to analyze all the rules of our big-step operation semantics. For each rule we have to see how the state occurring on the right side of $\Rightarrow$ is defined. Most of the rules do not modify the state themselves, they only use the state values returned by some recursive call, or supplied on the left side of $\Rightarrow$. The only exceptions which perform modifications of the state (that is: generate new state values distinct from the ones previously used) are the following two rules: (*i*) new object creation rule, and (*ii*) field assignment rule. The new object creation rule uses existing state and modifies its value on an address on which the state was not defined before. As a result, it does not change the value of the state at any address defined before. The field assignment rule does modifications of existing objects, however it does not modify their runtime type, which concludes the proof.

One of the consequences of the above property is the following fact: Every state in every configuration in a program trace is a superset of the set $\{\mathit{tt} \mapsto \mathit{BoolVal},\ \mathit{ff} \mapsto \mathit{BoolVal}\}$.

9.10 Comparison with previous work

Our solution is in many respects similar to that of Ager [5], which presents how to translate any big-step semantics into a stack machine. Our solution does not use a stack machine, however it uses similar concepts.

The main difference is that Ager's general approach generates a non-deterministic machine for each language, containing some form of judgment, for which there exists more than one rule in semantics. On the other hand, even though Magda's semantics contains a few judgments derivable using more than one rule, like `if`, `while` and object initialization, it is still deterministic. This is due to the strategy of delaying the choice of the appropriate rule after some parts of it are evaluated. For example, in the case of `if` we first evaluate the premise which is common to both of those rules, and choose the appropriate rule basing on the result of that evaluation.

Even though sharing of common premises does not occur in each semantics, we believe that most of the semantics having multiple rules for one judgment can be reformulated into a form, when rules for one judgment share common premises. It happens so, because such form reflects the fact that the language described by the semantics is deterministic in its nature (which is the case for the vast majority of languages).

```
tr := list();

procedure searchI(j)
 $env, ctx, st \models instr \Rightarrow^{I}$  := j;
 tr.add( ($env$, $ctx$, $st$));     /* the opening configuration is added to the trace */
 case j of
  ....
 endcase;
 tr.add( (res$|_1$, $ctx$, res$|_2$)); /* the closing configuration is added to the trace */
 return res;
endprocedure

procedure searchE(j)
 $env, ctx, st \models exp \Rightarrow^{ex}$  := j;
 tr.add( ($env$, $ctx$, $st$));
 case j of
  ....
 endcase;
 tr.add( ($env$, $ctx$, $st'$, res$|_1$, res$|_2$) );
 return res;
endprocedure

procedure searchInit(j)
 $adr, st \models (mods, pars)exp \Rightarrow^{ini}$  := j;
 tr.add( ($adr$, $st$, $mods$, $pars$) );
 case j of
  ....
 endcase;
 return res;
 tr.add( res );
endprocedure
```

Figure 9.5: Program trace construction using the derivation-search procedure

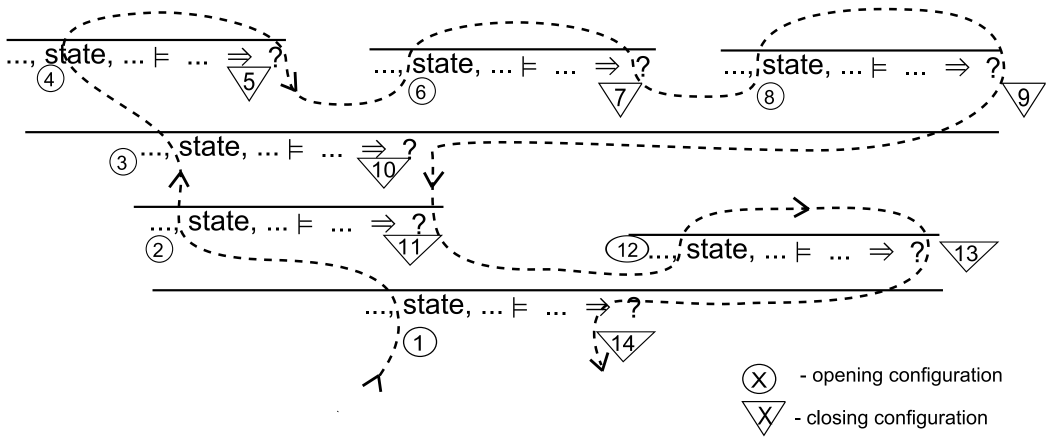

Figure 9.6: An example of a program trace

Chapter 10

Subject reduction

Using the above definitions of the derivation-search algorithm (see Section 9.4) and the execution trace (see Section 9.6) we can formulate and prove the subject reduction property for Magda.

10.1 Preliminary definitions

The subject reduction theorem of the Magda language states that each configuration in a trace of a type correct program is type safe in some sense. Before we formally state this theorem we need to define a few notions, which will be used in its formulation, in order to describe properties of configurations.

10.1.1 Type safety of the state

We say that state *st is type safe*, if for each $obj \in Rng(st)$ two conditions are met:

1. for each $Mix \in obj|_1$ and for each $fl\!:\!T_1 \in Fields_{Mix}$, we have $Mix.fl \in Dom(obj|_2)$;
2. and for each $Mix.fl \in Dom(obj|_2)$ we have either:
   - $obj|_2(Mix.fl) = null$, or
   - $st(obj|_2(Mix.fl)) = (T_2, ...)$, and $(fl\!:\!T_1) \in Fields_{Mix}$, for T_1 such that $T_2 \preceq T_1$.

Informally, the state st is type safe if (i) each object in st has defined values of all the fields declared in mixins from which it has been created, and (ii) the value of each field of each object in st is either $null$, or an address pointing to an object of a type compatible with the declaration of the given field.

10.1.2 Type safety of the environment

We say that an environment *env is type safe with respect to the state st and the context ctx*, if one of the following conditions holds:

- env is a function from identifier names to addresses, its domain is equal to the domain of $IdTypes(ctx)$ and for each $var \in Dom(env)$ we have one of the two cases:
  - $env(var) = null$, or
  - $st(env(var)) = (T_2, ...)$, such that $T_2 \preceq IdTypes(ctx)(var)$;
- env is a pair of the form $(\top, adr)$, such that one of the below conditions holds:
  - $adr = null$, or
  - $st(adr) = (T_2, ...)$, $ctx = (Mix, (Mix', mt))$ and $RetType^{(Mix',mt)}_{Mix} = T_1$, such that $T_2 \preceq T_1$. In that case we say that the *runtime type of* $st(adr)$ *is a subtype* of the current method result type.

10.1.3 Type safety of initialization parameters

For each partial function $pars \in ParamValues$ of assigned parameter values to their identifiers, we say that *pars is type safe with respect to the state st*, if for each $Mix.par \in dom(pars)$ we have one of the following situations:

- $pars(Mix.par) = null$, or
- $st(pars(Mix.par)) = (T_2, ...)$ and within the declaration of the mixin Mix there exists exactly one[1] declaration of the input parameter par, and this declaration has the form $par : T_1$, for some T_1 such that $T_2 \preceq T_1$.

We say that the set of initialization parameters *pars is consistent with the sequence of modules mods* if $activated'(mods, dom(pars))$ is defined.

In other words, the set $pars$ is consistent with $mods$, if all parameters in the set $pars$ will be consumed by modules in $mods$ and at the end of that process no parameters will be left.

10.1.4 Consistency of an address and a sequence of initialization modules

For a sequence of modules $\overrightarrow{mods}$ and state st we say that the address adr is *type consistent with* $\overrightarrow{mods}$ *and* st, if for each $(..., ..., ..., m) \in \overrightarrow{mods}$ we have $m \in st(adr)|_1$.

Informally: an address is consistent with the sequence of modules, if the runtime type of the object pointed by the address contains all the mixins in which those initialization modules have been declared.

[1]Notice that the uniqueness of declaration of an input parameter is guaranteed in type correct programs — see the rule for ini module declaration in Section 8.4.2

10.1.5 Type safety of the configuration

We say that the configuration C is *type safe* if one of the following conditions holds:

- The configuration C has the form (env, ctx, st) and:
  - the environment *env* is type safe with respect to the state *st* and the context *ctx*;
  - the state *st* is type safe.
- The configuration C has the form (env, ctx, st, adr, exp) and:
  - the environment *env* is type safe with respect to the state *st* and the context *ctx*;
  - the state *st* is type safe;
  - the address *adr* is type safe with respect to *exp*, *ctx*, and *st*, which means that: $adr = null$ or $st(adr) = (T_2, ...)$ and $T_2 \preceq T_1$ where T_1 is the biggest type of *exp* in *ctx* (see Section 8.8).
- The configuration C has the form $(adr, st, mods, pars)$ and:
  - the state *st* is type safe;
  - the set of parameters *pars* is type safe with respect to *st*;
  - the set of parameters *pars* is consistent with *mods*;
  - the address *adr* is type consistent with *mods* and *st*;
- The configuration C has the form *st* and:
  - the state *st* is type safe.

It is easy to see that if C_2 is type safe and C_1 is a subconfiguration of C_2 (see Section 9.6.1), then C_1 is also type safe.

10.2 Subject reduction formulation

Finally, the subject reduction property for Magda is formulated as follows:

Property 12 (Subject reduction) *For each type correct program (see Section 8.3), and a trace built by the derivation-search algorithm for that program, each configuration occurring in that trace is type safe.*

In other words, the subject reduction theorem says that during the whole execution process of the program (which is equivalent to the execution of the derivation-search algorithm), the state of the heap and all the values computed are type safe.

10.3 Subject reduction proof

The subject reduction property for Magda is proven by induction with respect to the order of configurations within the execution trace.

For each configuration, we pick the judgment and the rule which determined this configuration.

Notice that each opening configuration is completely determined by the components (state, environment...) of the partial judgment supplied to the call of the recursive procedure, and does not depend on that execution of that procedure. As a result, the proof of the safety of every opening configuration depends on the enclosing activation of the procedure (see Section 9.6.4).

Each closing configuration C, added by the activation a of the procedure, is calculated and determined by a, or partially calculated by some recursive call performed by a, before this configuration is added.

Therefore, the proof of the induction step is organized in the following way. For each judgment and each rule used for the evaluation of that judgment we prove that:

- opening configurations added by all the recursive call performed by this rule are type safe, and
- closing configuration (if any) added by that call is type safe;

under the assumption that all the configurations added to the trace before are type safe.

Additionally, since the opening configuration added by the top level call has no enclosing call (so will not be covered by such case analysis), we cover the proof of this case separately in Section 10.3.1.

10.3.1 No enclosing judgment.

There is one opening configuration in each trace for which there is no enclosing judgment. This configuration is the one added to the trace by the first call to the procedure, which has the whole program as its parameter (see Section 7.2.1). It is always the first configuration in each trace and has the form: $\emptyset, \top, \{\mathit{tt} \mapsto \mathit{BoolVal},\ \mathit{ff} \mapsto \mathit{BoolVal}\}$.

To prove that this configuration is type safe, we have to verify if the environment is type safe, and if the state is type safe as well. The domain of an empty environment is empty, while $\mathit{IdTypes}(\top)$ is also an empty set, therefore those domains coincide. Furthermore, the environment has no elements to be verified so it is type safe with respect to the state and the context. The domain of state st contains two elements: ff , tt, and those two addresses are both assigned with $\mathit{BoolVal}$ object which does not have any fields, therefore state st is also type safe, which concludes the proof of type safety of this opening configuration.

10.3.2 Assignment to a local variable.

Evaluation of a variable assignment of the form var:=exp performs one recursive call (responsible for the evaluation of the expression exp) and the opening configuration of this judgment is the same as the opening configuration of the enclosing judgment. Therefore by induction this configuration is type safe.

This rule generates a closing configuration, which is constructed from a subconfiguration of the closing configuration generated by the evaluation of the expression *exp*, by performing one modification: the environment is modified at position *VarName*. Therefore, since the closing configuration of the recursive call is type safe, we only have to show that assignment of *adr* to *VarName* in the new environment will not spoil the type safety of that configuration.

Since the closing configuration of the recursive call (evaluating *adr* from *exp*) is type safe by induction, we know that the runtime type of $st'(adr)$ is a subtype of the biggest type of *exp* (or $adr = null$). On the other hand, since the assignment instruction is type correct (see the rule for variable assignment in Section 8.5), we know that $exp : IdTypes(ctx)(VarName)$. Therefore we know that the runtime type of $st'(adr)$ is a subtype of $IdTypes(ctx)(VarName)$.

10.3.3 Assignment to an object field.

This rule contains two recursive calls:

- the first one, responsible for the evaluation of the target object, uses an opening configuration which is equal to the opening configuration of the enclosing call. Therefore by induction this configuration is type safe;
- the second one, responsible for the evaluation of the value to be assigned, uses an opening configuration which is a subconfiguration of the closing configuration of the first call. Therefore by induction this configuration is also type safe.

The closing configuration of this judgment is constructed from the subconfiguration of the closing configuration generated by the call evaluating exp_2 with one modification: *st* is modified in a way that the value of one field of one objects is modified and we have to show that new state is type safe. All the remaining fields of that object are non-modified so we only have to show that the new value of modified field fulfills the requirements of the type safety of state.

This field is assigned with adr''. By the type safety of the closing configuration evaluating exp_2, we know that the runtime type of $st''(adr'')$ is a subtype of the biggest type of exp_2 (or *null*). By the type correctness of this assignment (see the rule for field assignment in Section 8.5), we know that $exp_2 : T$, where T is type of the field according to its declaration. As a result, we know that the runtime type of $st''(adr'')$ is a subtype of T (or *null*), thus the new state is type safe.

10.3.4 An `if` statement.

This judgment has two rules. Each of them has two premises/recursive calls:

- the first one is responsible for the evaluation of the boolean condition. This call adds the same opening configuration as the opening configuration of the whole `if` statement, therefore it is type safe by induction.

- the second one in both rules is responsible for the execution of the instructions in `then` or `else` branch. The opening configuration of the second call is a subconfiguration of the closing one of the first recursive call (which is earlier in the trace), so by induction it is type safe also.

Each of the rules uses as its closing configuration the closing configuration of their second recursive call. Therefore by induction such configuration is already type safe.

10.3.5 A `return` statement

The evaluation of `return` statement has one recursive call, which has an opening configuration equal to the opening configuration of the enclosing judgment, so it is type safe by induction.

A closing configuration generated by the rule for `return` statement is constructed from two elements of the closing configuration (denoted as C) of the only recursive call (responsible for evaluation of the expression to be returned). The closing configuration of a `return` statement consists of: (i) the state occurring in C (which is type safe by induction) and (ii) a new environment of the form $(\top, adr)$. Therefore we have to prove that the address adr used in the environment fulfills all the necessary conditions. The address adr occurs in the configuration C. Therefore by the type safety of this configuration we know that either adr is equal to $null$ (which suffices for our configuration to be type safe), or the runtime type of $st'(adr)$ is a subtype of the biggest type of exp. On the other hand, by the fact that this `return` instruction is type correct (see the rule for `return` statement in Section 8.5) we know that $exp : RetType^{ctx|_2}_{ctx|_1}$. As a result we know that the runtime type of $st'(adr)$ is a subtype of $RetType^{ctx|_2}_{ctx|_1}$, which concludes the proof.

10.3.6 A compound instruction

The evaluation of a compound instruction performs up to two recursive calls. The first one has the same opening configuration as the enclosing judgment, which is at an earlier position in the trace, so it is type safe by induction. The second recursive call (if exists) supplies the opening configuration, which is equal to the closing configuration of the first call, which is also present in the trace at an earlier position, so is type safe by induction.

This kind of judgment has two rules. The first one uses the closing configuration of the second recursive call, while the second one use the closing configuration of the first recursive call. However in both cases this is a closing configuration which already has been added to the trace so it is type safe by induction.

10.3.7 An expression evaluation.

This kind of judgment has one rule, which has one recursive call with the same opening configuration as enclosing judgment. As a result this opening configuration is type safe by induction.

This rule, generates a closing configuration which is a subconfiguration of the closing configuration of the recursive call, therefore this one is also type safe by induction.

10.3.8 A `while` statement.

The evaluation of `while` statement performs one, two or three recursive calls, where the execution of each recursive call depends on the result of the previous one. The first recursive call evaluates the boolean expression and uses the same opening configuration as the enclosing judgment, so is type safe by induction. Then, if that expression evaluated to tt then the second recursive call, responsible for the instructions of the loop, is executed. This second recursive call uses as its opening configuration the one which is a subconfiguration of the closing configuration of the first call so is also type safe by induction. Then, if that instruction has not executed `return` (so the environment does not contain the symbol $\top$), then the third recursive call is executed, which repeats the execution of the whole loop. That call, however, also has an opening configuration which is equal to the closing configuration of the second call, so this configuration is type safe by induction.

The `while` judgment has three rules, however all of them share a common property: they return the closing configuration, or the subconfiguration of a closing configuration of the last recursive call. Therefore in each case such generated closing configuration is also type safe.

10.3.9 An empty instruction.

An empty instruction does not perform any recursive calls.

The closing configuration added by the evaluation of such a statement is always a subconfiguration of the opening one, so it is always type safe by induction.

10.3.10 Constant evaluation.

There are three constants: `null`, `true` and `false`.

The evaluation of a constant does not perform any recursive calls, therefore it never plays the role of the enclosing judgment.

The evaluation of each of those constants generates a closing configuration which is built from the opening configuration of the same call (which is type safe by induction), with the addition of an address *adr* and expression *exp* as the fourth and the fifth element of the tuple. Therefore we just have to show that *adr* is type safe with respect to *st*, *ctx* and *exp*. However, it is easy to see that during the evaluation of the first constant, the address *adr* is equal to *null*, which fulfills the conditions required for the configuration to be type safe.

In the second and the third case, $adr \in \{\mathit{tt}, \mathit{ff}\}$, and $st(\mathit{tt}) = st(\mathit{ff}) = \mathit{BoolVal} = ((\texttt{Boolean}), \emptyset)$. Moreover, the type checking axioms for constants are: $\texttt{true} : \{\texttt{Boolean}\}$ and $\texttt{false} : \{\texttt{Boolean}\}$, so the types match.

10.3.11 A local identifier.

The evaluation of a local identifier does not perform any recursive calls,

The closing configuration generated for a local identifier contains the state, and the environment of the opening configuration, therefore those are type safe by induction. The only thing which needs a verification is the returned adr, which is looked up in the environment env. By the type safety of the opening configuration we know that runtime type of $st(env(VarName))$ is a subtype of $IdTypes(ctx)(VarName)$ (or $env(VarName)$ is equal to $null$).

On the other hand, the biggest type of the variable evaluation expression is equal to $baseExt(IdTypes(ctx)(VarName))$, see Property 8. Therefore by the definition of subtyping (wrt. $baseExt$ function, see Section 8.6.1) we know that the runtime type of the value of adr is subtype of the biggest type of $VarName$ expression.

10.3.12 Field dereference.

The evaluation of field dereference performs one recursive call which uses the same opening configuration as the field dereference, thus is type safe by induction.

The closing configuration generated for such judgment contains the state and the environment of the closing configuration of the recursive call, which evaluated the target object expression. Therefore, we only need to show that the address adr is either equal to $null$ or object $st(adr)$ has the runtime type being a subtype of the biggest of the expression. However, we know the address adr is a value of a field of an object. Therefore, by the type safety of the state st, we know that this value of the field is either $null$ (which ends the proof) or is a value with the runtime type being a subtype of the declared type of the field. As a result, the runtime type of the expression is also a subtype of the biggest type of the field dereference (which is equal to the $baseExt$ of the declared type — see Section 8.8).

10.3.13 The ordinary method call.

The evaluation of a method call performs the following recursive calls: one responsible for the evaluation of the target object, a sequence of calls responsible for the evaluation of parameters, and the final call evaluating the body of the method.

The recursive call responsible for the evaluation of the target object uses the opening configuration of the enclosing judgment, while each call responsible for the evaluation of a parameter value uses an opening configuration which is a subconfiguration of the closing configuration of the preceding call. As a result those configurations are type safe by induction.

The only non-trivial case here is the last recursive call, which evaluates the instructions of the method body in the new environment env'. This is non-trivial since the environment env' is a function constructed from scratch, therefore we have to prove that env' is type safe with respect to st_n. In order to prove that env' (which is a function, not pair of the form $(\top, adr)$) is type safe with respect to st_n and the context $(Mix, (Mix, mt))$ we have to first show that its domain matches the definition of environment type safety, and then, that its values fulfill all the requirements.

The domain of the environment env' is constructed from three parts: $MetParams_{Mix'}^{(Mix,mt)}$, $MetLocals_{Mix'}^{(Mix,mt)}$ and `this`. Therefore, the domain of environment env' is equal to the domain of $IdTypes((Mix', (Mix, mt)))$ (see the definition of *IdTypes* in Section 7.1.3). As a result the domain fulfills the requirements of the type safety.

The values of variables in env' also fulfill the conditions of type safety, because:

- The local variables are assigned the *null* value, therefore are trivially correct,
- Identifier `this` is assigned adr_0 value. The mixin Mix' is defined as a value of *LastMix* function, therefore it is equal to one of the mixins in the runtime type of object $st_n(adr_0)|_1$. Therefore we know that the runtime type of $st_n(adr_0)|_1$ is a subtype of type $IdTypes(Mix', (Mix, mt))(\texttt{this}) = Mix'$.
- Method parameters are assigned with adr_1, ..., adr_n, which are the respective values of the expressions exp_1, ..., exp_n calculated by recursive calls to the procedure. For each such adr_i, we have to prove that it is *null* or it has the type which is a subtype of the T^i (where T^i is a declared type of a method parameter).

  Assuming that $adr_i \neq null$ we know that $st_n(adr_i)=(T_i^A, ...)$. Then, since the closing configuration of the call evaluating adr_i is type safe, we know that T_i^A is a subtype of the biggest type of exp_i. On the other side, thanks to the fact that this method call expression is type correct (see rule for method call in Section 8.6), we know that $exp_i : T^i$. Thus, by the definition of the biggest type (see Section 8.8) we know that the biggest type of exp_i is a subtype of T^i. Therefore, by the transitivity of subtyping we have that T_i^A is a subtype of T^i, which concludes this proof.

The closing configuration generated by the evaluation of a method call has the form $(env,\ ctx,\ st',\ adr,\ exp.Mix.mt(...)\ )$. This configuration is type safe for the following reasons:

- The state st' is type safe, because it is a part of the closing configuration generated by the evaluation of the method instructions, and this configuration is type safe by induction.
- The environment env is type safe with respect to st' and ctx, because env is type safe with respect to st and ctx (which are part of the opening configuration, which is type safe by induction) and by the state preservation property (see Section 9.9).
- The address adr is type safe with respect to exp, ctx and st, for the following reasons: The closing configuration generated by the evaluation of the method's instructions is type safe (by induction), therefore we know that adr is equal to *null* (which concludes this proof) or the runtime type of $st'(adr)$ is a subtype of $RetType_{Mix}^{(Mix,mt)}$. By the definition of subtyping, the runtime type of $st'(adr)$ is also a subtype of $baseExt(RetType_{Mix}^{(Mix,mt)})$, which in turn is a biggest type of $exp.Mix.mt(...)$.

10.3.14 A `super` method call

In this case the justification is the same as above. The only significant difference in this rule is that Mix' is constructed using a different function *LastMixBef*, but this function also returns a mixin, which is an element of the runtime type of $st_0(env(this))$. Therefore, the justification used in the case of method call is also valid here.

The closing configuration generated by this judgment is the same as in the case of an ordinary method call, and the proof is identical.

10.3.15 A new object creation.

For such a judgment there is a sequence of recursive calls responsible for the evaluation of parameter values, and one recursive call responsible for the execution of initialization modules.

The recursive calls responsible for the evaluation of the parameter values are type safe by induction, because: (*i*) the opening configuration of the first recursive call is equal to the opening configuration of the enclosing call, and (*ii*) the opening configuration of each subsequent call is a subconfiguration of the closing configuration of the preceding judgment.

The only non-trivial thing is the last recursive call which evaluates the partial judgment $adr, st \models (\overrightarrow{mods}, \overline{pars}) \Rightarrow^{ini} ?$, because the opening configuration of this recursive call is constructed from scratch. To see that the new state $st_k\{adr' \mapsto objVal\}$ is type safe, notice that it is constructed from st_k (which is type safe by induction) with just $objVal$ added at the position not used before ($adr' = FirstEmpty(st_k)$). Moreover, $objVal$ is an object having all the fields declared in mixins initialized with a *null* value (see definition of *EmptyObject* in Section 7.1.3). Therefore the new state is type safe by definition.

The set of parameters *pars* is type safe with *mods*, because the object creation expression is type correct and the type checking rule for a new object creation (see Section 8.6) ensures the condition of the type safety of the set of parameters (see Section 10.1.3).

The address *adr* is type consistent with the sequence of modules *mods* (see Section 10.1.4), because the sequence *mods* in the object creation rule is constructed from the runtime type of *adr*.

The last thing remaining to do in order to prove that this opening configuration is type safe is to show that *pars* is type safe with respect to st'' (see Section 10.1.3). Consider the parameter $\{par_i \mapsto adr_i\} \in pars$, where adr_i is a result of the evaluation of exp_i in some preceding recursive call. The closing configuration of the evaluation of adr_i is type safe by induction, therefore we know that the $adr_i = null$ (which concludes the proof) or the runtime type of $st''(adr_i)$ is a subtype of the biggest type of exp_i. On the other hand, the fact that the new object creation is type correct ensures that $exp_i : T_i$, where T_i is a declared type of the initialization parameter p_i. Therefore, because the biggest type of exp_i is a subtype of each type of exp_i, we know that the runtime type of $st''(adr_i)$ is a subtype of T_i.

The closing configuration generated by a new object creation judgment has the form $(env, ctx, st'', adr', exp)$. Such a configuration is type safe for the following reasons:

- The state st'' is type safe because it is a part of the closing configuration generated by the last recursive call (responsible for the execution of initialization modules), which in turn is type safe by induction;

- The environment env is type safe with respect to st'' and ctx, because it is type safe with respect to st and ctx (which are components of the opening configuration of the current call) and by the state preservation property;

- The address adr' is not $null$, and we know (by the way $objVal$ is defined and by the state preservation property), that $st''(adr')|_1 = Mixins$. On other hand we know that the biggest type of this object creation expression is equal to $Mixins$, which concludes the proof.

10.3.16 Object initialization.

The object initialization judgment has three rules which can be used to derive it. The first rule does not contain any recursive call, therefore it never plays the role of an enclosing judgment. The second rule has one and the third rule has four kinds of recursive calls, whose opening configurations are type safe for the following reasons:

- The only recursive call of the second initialization rule has a type safe opening configuration, because most of the opening configuration is the same as the enclosing configuration of the enclosing judgment. The only difference is that the last element of the sequence of initialization modules ($mods$) is dropped.

  This element of the configuration is used in two conditions of the configuration type safety. The last condition however (stating that the address is type consistent with $mods$) is still valid since we have removed one element of the state $mods$, so it is even easier now.

  The less obvious thing is to prove that $\overline{pars}$ is still consistent with $mods$. In order to prove that, we have to keep in mind that $mod|_1 \cap dom(\overline{pars}) = \emptyset \neq mod|_1$, which is a side-condition of the second object initialization rule. Furthermore, by induction, we know that $\overline{pars}$ is consistent with $\overrightarrow{modules; mod}$, which means that $activated'(\overrightarrow{modules; mod}, dom(\overline{pars}))$ is defined. When we look at the definition of the function $activated'$, keeping in mind the above side-condition we see that the only way a value of $activated'(\overrightarrow{modules; mod}, dom(\overline{pars}))$ is defined is using the value of $activated'(\overrightarrow{modules}, dom(\overline{pars}))$, which concludes the proof of this case.

- The first recursive call of the third initialization rule is responsible for the execution of I_1. This call is supplied with state st which comes from the opening configuration of the enclosing judgment, therefore is type safe by induction. However, env and ctx are constructed within this rule, so we have to analyze the type safety of env. It is type safe for the following reasons:

  - The domain of env contains local variables, input parameters of the initialization module, and `this` variable, therefore it fulfills the requirements of the environment type safety.
  - The local variables are assigned with $null$ values, so they trivially fulfill the conditions of type safety.
  - The identifier `this` is assigned with adr, however, by induction we know that the opening configuration of the enclosing rule was type safe. As a result we know that adr is consistent with the whole sequence $\overrightarrow{modules; mod}$, therefore in particular the runtime type of adr contains also the mixin in which mod was declared, which in turn is equal to $IdTypes(ctx)(\texttt{this})$.
  - Identifiers ip_1, ..., ip_k are assigned with adr^I_1, ..., adr^I_k. However, by induction we know that $pars$ is type safe with respect to st, therefore we know that the runtime types of $st(adr^I_1)$, ..., $st(adr^I_k)$ are subtypes of $IdTypes(ctx)(ip_1)$, ..., $IdTypes(ctx)(ip_k)$.

- The second assumption, being a set of recursive calls (one per each output parameter of the given module), has the opening configurations of the form (env', mod, st_i) for $i \in \{0, ..., l-1\}$. Each of these configurations is a subconfiguration of the closing configuration of the preceding call, therefore is type safe by induction.

- The third recursive call (responsible for the execution of next modules) has the opening configuration of the following form: $(adr, st_l, \overrightarrow{modules}, \overline{pars''})$. This configuration is type safe for the following reasons:
  - State st_l is type safe because it is a part of the closing configuration of the previous call, which in turn is type safe by induction.
  - The initialization parameters $\overline{pars''}$ are type safe with respect to st_l because:
    1. By induction we know that the opening configuration of the enclosing call is type safe, therefore we know that $\overline{pars}$ is type safe with respect to st.
    2. By 1. and the state preservation property (see Section 9.9) we know that $\overline{pars}$ is type safe with respect to st_l.
    3. By 2. and the definition of the type safety of the set of parameters we know that $pars' = pars \leftthreetimes^* \{...\}$ is also type safe with respect to st_l.
    4. By the type safety of the closing configurations of the calls in the second set, combined with the state preservation property, we know that each adr^O_i is either equal to $null$ or the runtime type of $st_l(adr^O_i)$ is a subtype of the biggest type of exp_i.
    5. By the fact that this `super[...]` instruction is type correct (see Section 8.5), we know that $exp_1, ..., exp_l$ have types used in declarations of $opID_1, ..., opID_l$.
    6. By 4. and 5. and by the transitivity of the subtyping relation we know that the runtime types of $st_l(adr^O_1), ..., st_l(adr^O_l)$ are subtypes of the declared types $opID_1, ..., opID_l$ (or they are $null$ values).

7. Therefore, by 6. and 3. we know that $\overline{pars''}$ obtained from $\overline{pars'}$ by addition of $\{opID_1 \mapsto adr_1^O; ...; opID_l \mapsto adr_l^O\}$ is type safe with respect to st_l

– By induction, $\overline{pars''}$ is consistent with $\overrightarrow{modules}$ because:

1. $\overline{pars}$ is consistent with $\overrightarrow{modules; mod}$, therefore we know that the value of $activated'(\overrightarrow{modules; mod}, dom(pars))$ is defined.
2. the fact that this initialization rule (executing mod) was applied implies that the value of $activated'(\overrightarrow{modules; mod}, dom(pars))$ was derived using the last rule (see Section 8.2.1);
3. therefore, looking at the derivation of $activated'(\overrightarrow{modules; mod}, dom(pars))$ we know that $activated'(\overrightarrow{modules}, dom(pars''))$ is defined.

– The address adr is type consistent with $\overrightarrow{modules}$ and st_l for the following reasons: (i) adr is type consistent with $\overrightarrow{modules; mod}$ and st, because the opening configuration of the considered judgment is type safe (by induction); (ii) therefore, by the state preservation property (see Section 9.9), we know that adr is type consistent with $\overrightarrow{modules; mod}$ and st_l; (iii) therefore adr is also type consistent $\overrightarrow{modules}$ and st_l, because $\overrightarrow{modules}$ is a shorter sequence.

- The fourth recursive call (responsible for the execution of I_2) generates the following opening configuration: (env', mod, st''). By the fact the (env', mod, st_0) is a closing configuration of the first recursive call (thus is type safe by induction) we know that env' is type safe with respect to mod and st_0. By the state preservation we know that env' is type safe with respect to mod and st''. On other hand, we know that st'' is type safe, because st'' is a part of the closing configuration of the third recursive call, therefore is type safe by induction. Therefore configuration (env', mod, st'') is type safe.

The object initialization process generates closing configurations consisting of the state only. The initialization process uses three different rules, and each of them returns the state st which is used in some preceding configurations. The first rule (responsible for the termination of initialization process) generates the closing configuration from the state coming from the opening configuration. Both, the second and the third rules, use the state which is a closing configuration of the last recursive call of that rule. As a result the state is type safe by induction in all those cases.

The above case-analysis of all rules and all configurations generated by them, concludes the subject reduction proof.

Chapter 11

Type soundness

In this section we formulate and prove the type soundness theorem. We state it using the notion of the program execution trace (see Section 9.6) and prove it using the subject reduction property (see Section 10).

11.1 Type soundness formulation

Informally, the type soundness property states that if the program is type correct (see Section 8.3), then during its execution nothing can "go wrong". In other words, every type correct program will not get stuck because of an object which does not support a requested method or does not contain a requested field. The only accepted reason for the program to fail is the null pointer dereference (see Section 9.8).

Formally, the type soundness property is defined in the following way:

Property 13 (Type soundness) *For every type correct program we have one of the following three situations:*

- *The program terminates successfully (see Section 9.7).*
- *The program gets stuck on a null pointer dereference (see Section 9.8).*
- *The program execution diverges, which means an infinite trace (see Section 9.6.3).*

11.2 Type soundness proof

We prove the below property which is equivalent to the type soundness defined above:

Property 14 (Type soundness reformulated) *If the trace of the type correct program is finite, and a null pointer dereference has not occurred during the program execution, then its last configuration is a successful program termination.*

We prove this property by contradiction. This means that we analyze every finite trace and each configuration in such a trace and, for each such configuration, we prove that after it there will always be another configuration (which means that the execution does not get stuck), except for the two cases: of a successful termination and of a null pointer dereference.

In this proof we will say that *configuration has been added unconditionally*, or that some recursive call is performed *unconditionally*, when it is performed: (i) without checking further conditions, and (ii) without performing additional calculations which could fail for some reasons. For example, we say that some recursive call is performed unconditionally when a partial judgment (see Section 7.2), being its parameter, consists of a state, an environment and a context calculated before by other recursive calls or supplied as a parameter.

Notice that, for a given configuration C, in order to prove that C is not the last configuration in the trace, it is enough to show that the next configuration is added unconditionally. Whenever the next configuration after C is not added unconditionally, we prove that C is not the last one in the following way:

- If there is some additional condition which is checked in the semantics and thus in the derivation-search procedure, then we prove that this condition is fulfilled and the execution can proceed and add next configurations to the trace.
- If the state, environment or context being a part of the partial judgment or returned is calculated by some functions, then we show that such state/environment/context is always defined.

For every form of the judgment j and every rule we prove that:

- opening configuration C added by the call for the judgment j is not the last one (so there will be other added afterwards)
- closing configuration C added by every recursive call performed by the call for j is not the last one. It means that when the recursive call adds the closing configuration and returns, there will be some configuration added afterwards. The only case when closing configuration can be the last one is the null pointer dereference.

To prove that the execution cannot get stuck in the given configuration we utilize the type safety of all the configurations in the trace (using Property 12 in Section 10.2).

This way we cover all configurations in the trace, except for the closing one added by the top-level activation of the derivation-search procedure (which is not a recursive call performed by other call). However, such closing configuration is indeed a last one, and it represents a successful program termination (see Section 9.7).

In all the below cases, we will use C to refer to the configuration for which we prove that it is not the last one.

11.2.1 Assignment to a local variable.

The execution of the derivation-search procedure for a variable assignment starts from performing unconditionally the recursive call to evaluate the assigned expression (which in turn adds the subsequent opening configuration), therefore its opening configuration cannot be the last configuration.

Closing configuration added by recursive call When the recursive call responsible for the evaluation of the assigned expression ends, then it always does so with the value of the desired form. Therefore in this case the new environment of the form $(env\{VarName \mapsto adr\})$ is always defined. As a result, there will always be a next configuration in the form of a closing configuration of this assignment.

11.2.2 Assignment to an object field.

The execution of the procedure for a field assignment also starts from performing unconditionally the recursive call to evaluate the target object. Therefore, its opening configuration cannot be the last one.

The call for a target object. When the recursive call responsible for the evaluation of the target object finishes, then the second recursive call is unconditionally executed. Therefore, after C there always be the configuration added by the second recursive call.

The call for assigned value. When the recursive call for assigned value finishes by adding configuration C, the activation of the procedure for the assignment finishes also, so it will also add its own closing configuration to the trace after C.

To see that this closing configuration is always defined, we have to show that the state $st''\{adr' \mapsto newObj\}$ is defined. For this it is enough that $st''(adr')$ is defined. By the subject reduction we know that, the closing configuration added by the first call is type safe (see Section 10.1.5). This in turn, in conjunction with the assumption that no null pointer dereference occurred means that $st'(adr') = (T_2, ...)$, which in conjunction with the state preservation (see Section 9.9) guarantees that $st''(adr') = (T_2, ...)$. Therefore $st''(adr')$ is defined.

11.2.3 An `if` statement.

In case of `if` statement, the execution of the procedure starts unconditionally from the execution of the recursive call to evaluate the boolean expression, so the opening configuration added by `if` statement cannot be the last configuration.

The call for boolean condition. The recursive call for an expression evaluation returns a pair consisting of a state st and an address adr. By the assumption that no null pointer exception occurred, we know that adr is different from $null$. The fact that the closing configuration generated by that call is type safe ensures that the runtime type of $st(adr)$ is a subtype of the biggest type of the expression. Moreover, by the fact that the conditional instruction is type correct (see Section 8.5), we know that this expression has type `{Boolean}`. Therefore we know that the runtime type of $st(adr)$ is subtype of `{Boolean}`.

On the other hand, the fact that the program is type correct ensures that no object of a type containing `Boolean` mixin can be created (see Section 8.6 and Section 6.7). Thus, by the state preservation we know that only *ff* and *tt* addresses point to values of type `{Boolean}`.

Therefore we know that one of the two rules can be applied, so there always is a next configuration which is the opening configuration added by the recursive call for the instruction I_1 or I_2 (depending on the value of *adr*).

The call for instruction. When the recursive call for instruction in the `if` statement adds its closing configuration and finishes, then the whole `if` instruction also finishes unconditionally and adds its own closing configuration.

11.2.4 A `return` statement.

Similarly as in the case of `if` statement, the procedure starts unconditionally from the evaluation of the returned expression, so C cannot be the last configuration.

The call for returned expression. When the recursive call for the expression to be returned adds its closing configuration and finishes, the call for the `return` statement also unconditionally adds its closing configuration and finishes.

11.2.5 Compound instruction.

The execution of a compound instruction starts unconditionally from the execution of the first instruction, so the opening configuration of the compound instruction is never the last one in the trace.

The call for the first instruction. When the recursive call for the first instruction finishes, we can have one of the two cases. In the first case the environment returned by the evaluation of the first instruction is a pair, and then the execution of the whole compound instruction unconditionally finishes, adding to the trace the closing configuration identical to the one of the first instruction.

In the second case (when the environment is not a pair, but a function), the second concatenated instruction is executed, so the configuration which follows is an opening configuration added by the recursive call responsible for the execution of the second instruction.

The call for the second instruction. When the execution of the second instruction finishes by adding its closing configuration, then the whole compound instruction also finishes, adding the identical closing configuration to the trace.

11.2.6 Expression evaluation instruction.

The execution of expression evaluation instruction unconditionally starts from the evaluation of expression, therefore the configuration C cannot be the last configuration in the trace.

The call for the expression. When the expression evaluation instruction finishes, then the whole instruction unconditionally finishes adding its own closing configuration.

11.2.7 A `while` statement.

The execution of `while` loop instruction unconditionally starts from the evaluation of the boolean condition, which in turn starts from adding its opening configuration to the trace, therefore C cannot be the last one.

The call for the boolean condition expression. When the evaluation of the boolean condition finishes, we know that it finished with a *tt* or *ff* value (as in the case of `if` instruction). Then, depending on that value we have one of the two cases. In case of the value *ff*, the whole `while` statement finishes, which means that the closing configuration for the whole `while` statement is added.

In case of the *tt* value, the next step is the execution of the loop body, which unconditionally starts from the addition of its opening configuration.

The call for the body of the loop instruction. When the body of the loop finishes, then we have one of the two cases.

In the first case, when the returned environment is a pair (representing the fact of the execution of `return` statement), the whole `while` statement unconditionally finishes and adds its closing configuration to the trace.

Otherwise, the whole loop is unconditionally executed once again, thus it adds its opening configuration to the trace after C.

The recursive execution of the `while` loop. When the recursive execution of the whole loop adds its closing configuration, then the current execution also finishes and also unconditionally adds its closing configuration.

11.2.8 Constant evaluation.

All three constants unconditionally evaluate to a closing configuration, so the opening configuration for such judgment cannot be the last one.

The evaluation of a constant does not perform any recursive calls.

11.2.9 Local identifier evaluation.

The opening configuration of a variable evaluation is always followed by the closing configuration. To see that the closing configuration is always defined it is enough to show that the returned address $env(VarName)$ is always defined. This, in turn, happens for the following reasons: The fact that the expression is type correct (see Section 8.6) ensures that $VarName$ is in the domain of $IdTypes(ctx)$. The subject reduction ensures the fact that the opening configuration is type safe, which in turn ensures that the domain of env is equal to the domain of $IdTypes(ctx)$ which finally ensures that $env(VarName)$ is defined.

Similarly as with constant evaluation, the evaluation of a local identifier also does not perform any recursive calls.

11.2.10 Field evaluation.

The evaluation of a field evaluation expression unconditionally starts from the evaluation of the target object, which in turns adds its opening configuration, thus there is always a next configuration after C.

Recursive call: the evaluation of the target object. When the evaluation of the target object finishes with a configuration C, then the evaluation of the whole expression finishes also, with a configuration which differs from C in the returned address, having the form $st'(adr)|_2(Mix.fl)$. To see that this address is always defined and thus the closing configuration of the whole expression will always be added, observe that:

1. The address adr is different from $null$ (because of the assumption that no null pointer exception occurred during the recursive call).
2. The type safety of the closing configuration added by the recursive call ensures that the runtime type of $st'(adr)$ is a subtype of the biggest type of the target object expression. Thus we know that $st'(adr)|_2$ is defined.
3. Furthermore, the fact that the expression is type correct ensures us that the target object expression has a type containing Mix. Thus the runtime type of $st'(adr)$ also contains Mix.
4. By the type correctness of the whole expression we know that $fl \in Fields_{Mix}$.
5. Thus, by the type safety of the closing configuration, in particular by the safety of st' we know that $st'(adr)|_2(Mix.fl)$ is defined

11.2.11 Call to a method.

The evaluation of a method call unconditionally starts from the evaluation of the target object, therefore the opening configuration of method call is always followed by the opening configuration of the evaluation of the target object.

The call for evaluation of the target object. When the evaluation of the target of the method call finishes by adding its closing configuration, we can have two different cases depending on whether the method has parameters.

If the method has parameters, then the expression representing the value of the first actual parameter is unconditionally evaluated. Thus its evaluation always adds its opening configuration to the trace after C.

In case of no method parameters, the body of the method is executed, however in a newly constructed environment and in a new context. To see that it is always executed, we have to show that the following partial judgment (responsible for the execution of that body) is always defined:

$$env', (Mix', (Mix, mt)), st_n \models MetInstr^{(Mix,mt)}_{Mix'} \Rightarrow^I ?$$

To do this we have to show (similarly as we did before when dealing with the opening configuration of the `super` method call) that the following four expressions are defined:

- $\mathit{Mix}' = \mathit{LastMix}(st_n(adr_0)|_1, (\mathit{Mix}, mt))$.
- $\mathit{MetParams}^{(\mathit{Mix},mt)}_{\mathit{Mix}'}$
- $\mathit{MetLocals}^{(\mathit{Mix},mt)}_{\mathit{Mix}'}$
- $\mathit{MetInstr}^{mtID}_{\mathit{Mix}'}$

Since we assumed the program is type correct, it follows that all object creation instructions are type correct. Therefore all the objects are created from consistent mixin sequences (see Section 6.7) which, in conjunction with the state preservation property (see Section 9.9), ensures that the runtime type of $st_n(adr_0)$ is also a consistent sequence, knowing that adr_0 is not null. Finally, thanks to the type safety of the last closing configuration and type correctness of this method call expression, we know that the runtime type of $st_n(adr_0)$ contains Mix, thus by the consistency we know that Mix' is defined.

The next two points are just a consequence of the fact that the function *LastMix* always returns a mixin which contains a declaration of the supplied method.

The call for evaluation of a parameter value. When the evaluation process of a method parameter finishes, then we have two cases, depending on whether the parameter was the last one or not.

If this is not the last parameter, then the next parameter is unconditionally evaluated and adds its opening configuration after C.

In case of the last parameter, the next configuration is the opening configuration added by the evaluation of the method body in a new environment. This evaluation of the method body will always start by adding its opening configuration, because the partial judgment responsible for that call is always defined. The justification for that fact is identical as in the previous point.

The call for evaluation of the method body. When the process of the execution of the method body finishes, the whole execution of the method call also finishes, assuming that those instructions evaluate to a value of the form $((\top, adr), st')$. However, this is a direct consequence of Property 5 (see Section 7.4). Therefore the next configuration will always be the closing one added by the method call expression.

11.2.12 A `super` method call.

The evaluation of a super method call unconditionally starts from the evaluation of the first actual parameter expression (if there is such), so the opening configuration of that call is the next configuration.

In case of no parameters the first recursive call will be the one responsible for the evaluation of the redefined method body. Then to prove that the recursive call can be performed, and thus C is not the last configuration in the trace, we have to show the following:

1. Mixin $\mathit{Mix}' = \mathit{LastMixBef}(st_n(env(\texttt{this}))|_1, \mathit{mtID}, \mathit{Mix})$ is defined.
2. Initialization parameters $\mathit{MetParams}^{\mathit{mtID}}_{\mathit{Mix}'}$ and $\mathit{MetLocals}^{\mathit{mtID}}_{\mathit{Mix}'}$ are defined, so $\overrightarrow{par}$ and $local$ are defined.
3. Instructions $\mathit{MetInstr}^{\mathit{mtID}}_{\mathit{Mix}'}$ are defined.

To see that (1) holds, observe that in a type correct program each object is created from a consistent mixin sequence (see Section 6.7), thus thanks to the state preservation property (see Section 9.9) we know that the runtime type of the current object ($env(\texttt{this})$) is also a consistent mixin sequence, thus Mix' is defined.

Then, (2) and (3) are just a consequence of the fact that *LastMixBef* function always returns a mixin which contains a declaration of the supplied method.

Recursive call for evaluation of a parameter value. When the evaluation of a parameter finishes, then we have one of the two cases.

If the evaluated parameter is not the last one, the evaluation of next parameter is unconditionally executed, adding new opening configuration to the trace after C.

Otherwise, the body of the method is executed. In this case, we need to show that the new environment and the body are defined. This proof is similar to the proof for the opening configuration of the super call without parameters (see Section 11.2.12).

The call for evaluation of a method body. When the execution of the method body finishes, then the whole execution of the super call also finishes, assuming that the body evaluated to a tuple of the form $((\top, adr), st')$, which is however ensured by the Property 5 (see Section 7.4).

11.2.13 A new object creation.

The first step of the evaluation of a new object creation expression depends on whether there are some initialization parameters used in this expression. If there are such parameters, then the situation is trivial, since then we unconditionally evaluate the first parameter. So there is for sure a next configuration after C.

In case of no initialization parameters, the next configuration to be added to the trace is the one added by the recursive call responsible for the execution of the initialization modules. However, knowing that the object creation expression is type correct, we know that $base(mixins)$ is defined, thus all the names used in this sequence are in fact introduced by the existing mixin declarations. Therefore $\mathit{IniModules}(mixins)$ is defined as well as $\mathit{EmptyObject}(mixins)$. As a result, the recursive call can always be performed and there is always the configuration added by this recursive call, thus the configuration C is not the last one.

Recursive call for evaluation of an initialization parameter. When the evaluation of an initialization parameter finishes, then we can have one of two cases.

If the evaluated parameter was not the last one, then the next parameters is evaluated unconditionally and its evaluation adds an opening configuration after C.

If the evaluated parameter was the last one, then the next configuration in the trace will be the opening configuration added by the execution of the sequence of initialization modules. However, to see that evaluation of those modules can be successfully started we have to show that the partial judgment

$$adr',\ st_k\{adr' \mapsto objVal\} \models (\mathit{IniModules}(mixins), \overline{ParID \mapsto adr}) \Rightarrow^{ini} ?$$

representing those modules is always defined. This holds because:

- Address $adr' = \mathit{FirstEmpty}(st_k)$ is always defined since we assume an infinite set of addresses, and the domain of each state is finite.
- Object $objVal = \mathit{EmptyObject}(mixins)$ is always defined (see Section 7.1.3), since the type correctness of the object creation expressions ensures that *mixins* is a consistent sequence, therefore *mixins* is a sequence of names of mixins declared in the program.
- Set of ini modules $\mathit{IniModules}(mixins)$ is always defined since the type correctness of the new object creation expression ensures the fact that $\mathit{activated}(mixins, ...)$ is defined. The definition of *activated* requires in turn the $\mathit{IniModules}(mixins)$ to be defined.

Recursive call for execution of initialization modules. When the execution of the sequence of initialization modules finishes, then the execution of the whole object creation expression unconditionally finishes, adding its closing configuration to the trace.

11.2.14 An object initialization.

In this case we first check if the list of modules is empty. If this list is empty, then the set of parameters also needs to be empty, which is ensured by the fact that the set of parameters is consistent with the empty sequence of modules (as a part of the type safety of the opening configuration). Therefore, when the list of modules and the set of parameters are empty, there is no recursive call, and the closing configuration is always added by the call which added the opening configuration C.

In the case when the list is non-empty and additionally, input parameters of the last module in the list coincide with the supplied parameters, then the module is executed and the next configuration in the trace is the opening configuration of the recursive call responsible for the execution of the sequence of instructions I_1, occurring within the ini module.

In the last case, when the list is not empty, and the input parameters of the last module are not in the supplied set of parameters then we unconditionally execute the procedure for the list with the last module removed. This, in turn, means that this recursive call always adds next opening configuration to the trace.

Recursive call: execution of the rest of ini modules during module skip. When the rule for the module skip is used, the remaining initialization modules are executed. When the execution of that list of modules finishes by adding its closing configuration, then the execution of the whole list finishes unconditionally, adding its closing configuration to the trace.

Recursive call: execution of the first part of the module. When the execution of the first part of the module denoted as I_1 (containing instructions which occur before the `super[...]` instruction) finishes, then we have one of the two following cases.

In the first case, when the currently executed module has some output parameters declared, the first parameter is evaluated unconditionally. Therefore the evaluation of the first parameter adds its opening configuration to the trace.

In the second case, when the current module does not have any output parameters, the remaining modules are executed unconditionally in the previously calculated state, environment and context, thus adding a new opening configuration.

Recursive call: evaluation of the initialization parameter in `super[...]`. When the initialization parameter finishes, then similarly as in the above case of "the execution of the first part of the module" we have two cases, and those cases are dealt in the same way. Either there exists a next parameter to be evaluated and its evaluation adds an opening configuration, or the sequence of the remaining modules is executed which also adds a new opening configuration.

Recursive call: execution of the tail of modules after module execution. When the rest of the sequence of modules finishes its execution, then second part of the ini module denoted as I_2 (containing instructions which occur after the `super[...]` instruction) is unconditionally executed and adds an opening configuration to the trace.

Recursive call: execution of the second part of the module. When the second part of the ini module finishes its execution, then the execution of the whole module (with all next modules) finishes unconditionally and adds its closing configuration.

Chapter 12

Implementation

The Magda language, together with the formal specification described in this thesis, has also a working proof-of-concept implementation [47]. This implementation, apart from all the above mentioned features, contains also additional ones like:

- generics in the Java style (see [24, 23]),
- support for Java code snippets, required for accessing external libraries and performing system calls,
- more built-in types.

However, we will not discuss those features here. The compiler works by performing the following steps:

- It performs the lexical analysis of Magda code, with a parser built using the JavaCC framework [1]. The result of this analysis is a semantic tree of the whole program.
- It performs all the static type checks on the generated tree. The type checks are performed according to the description in Chapter 8.
- It generates Java code, from the semantic tree of the Magda program.
- It executes the Java compiler to compile the generated Java code.

However, notice that Magda's model of inheritance as well as the model of methods redefinition and referencing is significantly different from the one of Java. Therefore, the Java code generated from our code is not a one-to-one translation. In particular, a method in Magda is not translated into a method in Java (it is instead translated into a class). Below we outline the key features of the translation we perform:

- Each method, ini module, field, mixin, and also each object in Magda is represented by a separate Java object.

- For each of those Magda constructs, there is a one corresponding base class in Java: `CMagdaMethod` (with one method, `Execute`), `CMagdaMixin`, `CMagdaObject`, etc.

- Each object in Magda is represented by an object of class `CMagdaObject`, which contains an array of objects representing all methods from all mixins from which this object has been created. Analogously, each object contains also an array of objects representing all fields.

- Each declaration of a method is translated into a declaration of a subclass of the class `CMagdaMethod`. This class has one method implemented, called `Execute`. This method contains a translation of the instructions of the Magda method.

  The method `Execute`, in order to be general enough, takes two arguments. The first one is the `CMagdaObject` representing the target of the method call, i.e., `this`. The second one is an array of `CMagdaObject`-s representing all the arguments of the actual Magda method. The result type is also of type `CMagdaObject`. This way, the generated Java code does not use types used in the source Magda program. Therefore, the Java compiler does not verify, if the `CMagdaObject` supplied to a method contains methods or fields which will be used in the method.

  However, since our compiler first performs all type checks on the Magda code, and the type system of Magda is sound (see Section 11), we know that all such type errors are caught at this stage.

- Each call of a method on a given object performs a search in this array according to the semantics of *LastMix* function (see Section 6.6).

Despite the fact that the code is not generated in the most efficient way, this compiler demonstrates that the presented language design gives rise to a real language. Additionally, it allows users to experiment with the syntax and the semantics of the language and to verify how the language works in practice (this compiler comes together with a set of simple examples).

Chapter 13

Related work

In this chapter we present other existing solutions designed to solve the problems we are concerned with (or at least problems similar to ours). This chapter is divided into five sections. Section 13.1 discusses different approaches to modularization of initialization process. Section 13.2 presents different approaches to solve the problem of clashes of non-hygienic method identifiers. Section 13.3 compares our solution with other approaches extending the reusability of components. Section 13.4 discusses how encapsulation works in Magda. Section 13.5 presents a few language constructs, which are present in other languages but not in Magda, and discusses how those can be represented in Magda.

13.1 Modularization of constructors

There are at least eight techniques used by designers to solve some of the problems concerning object initialization:

1. by avoiding explicit initialization protocols,
2. by declaring a constructor with one parameter of type "container" containing all the initialization parameters (such as, for example, of type `Vector` or of type `Dictionary`),
3. by using the *container classes* design pattern,
4. by using default parameter values,
5. by referencing parameters by name,
6. by using the constructor propagation mechanism of Java Layers [26],
7. by using object factories [29],
8. and by using solution proposed by Eisenecker et al. [35] for mixin-based programming in C++.

The first three solutions are, in fact, programming techniques which can be used in most object-oriented languages, while the last five are actual language features (implemented in existing languages or languages extensions). In the following section, we will discuss those solutions in more detail.

13.1.1 Avoiding the initialization protocol

A class written using the approach of avoiding explicit initialization protocols contains one parameterless constructor (or none), while the real initialization process is implemented in a list of ordinary methods. Those methods must be called on objects explicitly after their creation. Here is an example. Instead of declaring the following class (as in Figure 2.1):

```
class ColorPoint {
  ColorPoint(int x, int y, int R, int G, int B)  {...}
  ColorPoint(int x, int y, int C, int M, int YC, int K)  {...}
}
...
x= new ColorPoint (100, 100, 255, 0, 255);
```

one might declare a class with a parameterless constructor, and a set of methods responsible for initialization:

```
class ColorPoint
{ ColorPoint();
  void setPosition(int x, int y)
  void setColRGB(int R, int G, int B)
  void setColCMYK(int C, int M, int YC, int K)
}
...
x = new ColorPoint();
x.setPosition(100, 100);
x.setColorRGB(255, 0, 255);
```

In a program written in such a manner, a class may contain many small methods, each of them responsible for different options of a different layer of the initialization, so that a sufficient level of modularity is achieved. However, a programmer using this class does not have any form of verification whether the object is properly initialized. The programmer may create an object from the class and make any of the following errors (without being warned by the compiler): (*i*) forget to call some of the methods responsible for the initialization; (*ii*) call too many of them; (*iii*) call them in an incorrect order.

Existing formal specification languages, like JML [49] and the Design by Contract in Eiffel [52], can allow some verification of this protocol. However, such approaches require to specify additional assertions, which is, in general, a difficult and time-consuming task. Additionally, due to the general nature of assertions and to a great freedom of possible

```
class ColorPoint
{ ColorPoint (Dictionary d)
  { if ((d.get("R")<>null) && ...)
      {... /* process RGB data */ ...}
    else if ((d.get("C")<>null) && ...)
      {... /* process CMYK data */...}
    else
      throw new Exception ("Wrong parameters!!");
  }
}
```

Figure 13.1: A Constructor with container-like parameter

combinations, those assertions can be checked during the execution of the program, but they cannot be verified statically. A static verification is sometimes possible with the support of theorem provers, but usually such tools require some manual support from the programmer.

13.1.2 Container parameters

Constructors may have one parameter of a "container" type, such as `Vector` or `Dictionary`. The container structure will contain values of all the initialization parameters, for instance indexed by their names. Then, a constructor may perform a dynamic verification inside, like in the example visible on Figure 13.1.
This approach has the following disadvantages:

- The class contains one constructor which must perform a lot of checks for many cases of the initialization process.
- It does not allow any static checking of the set of chosen parameters. As a result, when a programmer chooses an inappropriate subset of parameters, or even a non-existing parameter, then he/she will not be warned during the compilation.

13.1.3 Container classes

The idea behind the *container classes* design pattern is to use a separate class for passing the set of parameters used to initialize a given general property (as the position or the color of `ColorPoint` on Figure 2.1). Each of those classes must have a set of constructors (with their respective parameters) equal to the set of the possible options of initialization for the given property. The `ColorPoint` example (see Figure 2.1) written using this design pattern would look as follows:

```
class Color
{ Color (float r, float g, float b);
  Color (float c, float m, float y, float k);
}
class Position {...}
class ColorPoint
{ ColorPoint (Color c, Position p)
}
```

The use of such approach allows one to avoid the problems of: (*i*) exponential growth of the number of constructors; (*ii*) unnecessary code duplication. However, this approach has the following drawbacks:

- When the first version of some class has only one option of initialization of some property, then it looks like there is not need to use the container class. However, if we do not predict that some property can have multiple options in the future (by packing it into a container class) then future conservative class modifications which add some new options will not be able to use this design pattern.

- It only works in the cases when the set of options of initialization for a whole class is a cartesian product of sets of options for some base properties. It cannot be used in more complicated cases, like those when: (*i*) we add an option which, by supplying one value, initializes distinct properties (using distinct container classes); (*ii*) we add a new option for initialization of some subset of fields, which are packaged into one container class;

- It makes object creation expressions more complicated and less efficient: One first has to create the container objects, and then pass them to the actual constructor of the class, from which an object is to be created.

13.1.4 Optional parameters

There is another mechanism which, in principle, was not designed to solve those problems, but can be used to solve one of them: methods and constructors with *default* parameter values (which is present, for example, in Delphi [3] and C++ [63]). Thanks to this mechanism, it is possible to declare fewer constructors, being able to treat some parameters of existing ones as optional. Then, when one of the parameters is not supplied in the given call, its default value is used. This mechanism does not help, though, when it is necessary to have a mutually exclusive choice among different parameters (of different types), because one cannot limit which combinations of optional parameters are allowed.

This solution works well used together with referencing parameters by their names (see section below).

13.1.5 Referencing by name

Another feature which can be found in some languages (but not in any of the main-stream ones like Java and C♯), which may help, is referencing parameters of methods and constructors by their names (that is, not necessarily passing the actual parameter values in

```
class Point
{ propagate Class1(String s)  {I_1;}
  propagate Class1(int i)     {I_2;}
}
class Class2<T> extends T
{ propagate Class2(double j)  {I_3;}
  propagate Class2(boolean k) {I_4;}
}
```

Figure 13.2: Java Layers example

the order in which they are declared). Such approach, present for example, in Flavors [56], Objective-C [44], and Ocaml [50], solves two problems:

- it discards some of ambiguities (caused by constructor overloading), because parameters with the same (or compatible) type can have different names;
- it allows a wider use of default parameter values because normally we can use default values only for a sequence of parameters being a suffix of the sequence of all parameters, while in this approach it is possible to use defaults for any subset of the set of available parameters.

However, this feature only solves problems of optional parameters and discards some ambiguities, but does not prevent an exponential number of constructors and code duplication in the case of multiple options of initialization of orthogonal object properties.

13.1.6 Constructor propagation in Java Layers

The Java Layers language [26] has a feature called "constructor propagation", which can be illustrated by the example[1] present on Figure 13.2.

With such declarations the class `Class2<Class1>` will have all the combinations of the "propagated constructors" of both combined classes. In this case it will in fact mean to have the same list of constructors as the class visible on Figure 13.3.

This approach solves the problem of the exponential number of constructors if the sets of options of parameters are in different classes. However, this can only do a cartesian product of sets of constructors from different classes. One of the things that cannot be done in the Java Layers approach is the addition of new options of initialization of some properties defined in the original class. Therefore, if we want to write a subclass of a class `C` with the purpose of adding another option for initializing the existing set of properties declared in `C`, then we cannot do this using Java Layers.

[1]The example is a modified version of an example taken from the web page of the Java Layers `http://www.cs.utexas.edu/~richcar/cardoneDefense.ppt`. Also the syntax is slightly modified to look more Java-like.

```
class Class3
{ Class3 (String s, double j) {I_1; I_3;}
  Class3 (int i    , double j) {I_2; I_3;}
  Class3 (String s, boolean k) {I_1; I_4;}
  Class3 (int i    , boolean k) {I_2; I_4;}
}
```

Figure 13.3: Java Layers class translated into Java

```
mixin HSBColorPoint of ColorPoint =
 optional HSBColorPoint(h:float, s:float, b:float)
   initializes (ColorPoint.r, ColorPoint.g, ColorPoint.b)
 begin
   ...
 end;
end;
```

Figure 13.4: Magda code not representable in Java Layers approach

To better understand this, let us consider the example on Figure 13.4 written in Magda. This is an extension of classes shown in Figure 2.1, however rewritten in Magda.

Such code, in the Java Layers approach, would require copying all combinations of the propagated constructors.

13.1.7 Object factories

A recent work concerning initialization protocols is [29]. It shows how object factories can be integrated in a language like Java in such a way that they use the same syntax as normal object creation. Using this approach, it is possible to override the constructors of a class, and even to write an object creation expression of the form `new` I`(...)`, where I is an interface, thus giving more flexibility in the management of the initialization code. As an effect, in some situations this reduces the amount of code which needs to be written. For example, when a designer needs to add one initialization option to an existing class, he can just extend the list of the constructors of that class.

However, in this approach, each allowed set of initialization parameters must correspond to one constructor, therefore, in general, it does not avoid the problem of the exponential growth.

The more important benefits of that approach are: (*i*) the separation of the initialization from the class itself, so that a client instantiating an interface can even not know the implementing class; (*ii*) the possibility of modifying an initialization protocol in a way in which, for instance, the object returned by `new` expression is an already existing one (not a newly created one).

13.1.8 Mixin-based programming in C++

The work [35] is a study on constructors for a mixin-based programming style in C++ (where mixins are implemented by using templates). This paper describes some of the problems we have also pointed out, and proposes a solution for the problem of the non-composability of mixins (see Section 2.1.8). Nevertheless, the proposal solves only the problem of non-composability and it requires automatic generation of additional C++ code (which in fact can be exponential in the size of all mixin code). It also requires that the programmer declares additional type parameters in each mixins, which must be passed to constructors of its ancestor classes. This may cause significant overhead when programming large libraries of mixins.

13.2 Identifier clashes and parallel development safety

Thanks to its hygienic identifier approach, Magda solves all the problems referring to identifier clashes presented in Section 2.2. Some of those problems have also already been spotted by other researchers. As an effect, in many practical languages and theoretical calculi, some mechanisms have been implemented to solve those problems at least partially.

This section below contains description of other solutions to this problem.

13.2.1 Delphi

In the Delphi language [3], which is currently the most popular implementation of Object Pascal, there is a direct distinction between the method implementation which introduces a new method and the one that redefines a method (via the keyword `override`). Nevertheless, all the references to methods are by name, therefore ambiguities at this point can still occur.

13.2.2 C♯

The designers of C♯ [40] have already seen the problems of possible accidental conflicts between different versions of the libraries. Therefore, in C♯ some features were implemented to address some of the problems we are concerned with.

First of all, C♯ allows one to distinguish an overriding method implementation from an introducing method implementation, via the use of the keywords `new` and `override`. However, similarly to Delphi, it allows the programmer to have more than one introduction, therefore a method implementation marked with `override` can override a method introduction different from the one intended.

Additionally, C♯ syntax distinguishes between: (*i*) `virtual` (dynamically dispatched) methods and (*ii*) statically dispatched methods, and as default behavior it chooses the statically dispatched ones. The approach chosen by the C♯ designers is that most of the methods cannot be overridden, therefore for methods not intended to be overridden at all, accidental overriding cannot occur. However, for `virtual` methods we can still have a problem: when introducing a method for the second time in a subclass, the implementation intended to `override` the first one now redefines the second one. Also, a method call expression can still suffer from ambiguous binding.

Finally, in a class implementing a method introduced in an interface, a programmer may declare explicitly from which interface this method comes. This is useful when a method of the same name is declared in two different interfaces. The syntax is, in some respects, similar to ours:

```
interface I { void mt(); };
interface J { void mt(); };
class C : I, J
{ void I.mt()   {...}
  void J.mt()   {...}
}
```

However, this is only possible for methods introduced in interfaces and only with respect to the implementation of the method (not with respect to the method call), therefore it solves only some of the problems. Additionally, it has also some awkward behavior: the method `mt()` cannot be executed on objects of type `C` without casting on the interface.

13.2.3 Java 1.5

In Java 1.5 there has been added an optional annotation `@Override`, which instructs the compiler to verify if the superclass contains a declaration of the same method. And if not, then the compiler raises an error. However, if the method is not annotated as `@Override` and it happens to be an overriding implementation (as a result of accidental name clash), then the compiler will not raise an error, which is because the compiler needs to be backward compatible with the old code.

Additionally, there still cannot be two different methods of the same name, so when accidental override occurs, the only choice is to rename the method.

13.2.4 Eiffel

The Eiffel language [52] features some of the above described mechanisms.

First of all, a distinction in the syntax between method introduction and override also exists here, via the usage of the keyword `redefines`. However, while Delphi and C♯ allow one to have a few distinct introductions of a method with the same name, Eiffel raises an error when a new introduction of a method with an old name is found.

Additionally, Eiffel allows one to supply a distinct implementations for different methods with the same name inherited from different abstract classes (which play the role of interfaces in Eiffel). This can be achieved via the `rename` operation on the methods coming from different ancestor classes (notice also that Eiffel supports multiple inheritance) and the subsequent redefinition of each of the renamed methods.

```
mixin M \{void mt() \{...\} \};
mixin N \{void mt() \{...\} \};
class A = M(N(Object));
...
M a = new A();    // type M indicates a view
N b = new A();    // N indicates another view
a.mt();           // mt() from M is called
b.mt();           // mt() from N is called
```

Figure 13.5: The concept of view in MixedJava

13.2.5 C++

In C++, in particular in the presence of templates, the problem of non-hygienic identifier binding was pointed out by Smaragdakis and Batory in [61]. The solution proposed to solve ambiguities during method calls was to use the prefixing of the method name with the class name all the time (which is a feature of C++: `<class>::<method>(...)`). However, the hygienic programming is not enforced by the language, and, additionally, problems with ambiguities concerning the override are not addressed by this solution.

13.2.6 Traits

Schärli et al., in their work on traits [60], have also tackled the problem of accidental override. In order to solve this problem, they decided to: (*i*) not accept trait composition when accidental clashes between two traits used to build a class occur; and (*ii*) allow manual renaming of methods coming from traits. Thanks to that, in a language with traits, program will never suffer from the accidental override of methods from different traits, since this problem will be spotted during the trait composition.

However, the traits approach requires manual modifications of different parts of the code, whenever a method is added to a trait which causes conflict in other parts of code. Additionally, resolving of such conflict might break some dependencies as presented in Section 2.4.5. Moreover, a method implementation coming from a trait can still override accidentally one present in a superclass.

13.2.7 MixedJava

In the MixedJava language [36], the problem of having multiple implementations of methods with same name (but coming from separate mixins) is dealt with the concept of *view* of an object. If an object contains two methods with the same name, the one to be called at a given point is the ones declared in the mixin, which is the static type (called also view) of the target of the given method call. The example of such behavior is present on Figure 13.5.

However, in a context where both methods are visible we still have a problem, as the chosen method might not be the one we expect.

13.2.8 MixGen

In the work on first class genericity for Java by Eric Allen et al. [6] a mixin is implemented by a generic class using its parameter as its ancestor. They introduced the notion of "hygienic mixin" to describe the semantics introduced by Flatt et al. and adapted it successfully to the world of generics. In contrast to MixedJava, MixGen has a compiler generating Java VM compatible bytecode, which uses the fully fledged name of the class in which a method is introduced to prefix the method name itself.

However, this prefixing is not visible in the source code because it is done implicitly during each compilation and class-loading, therefore the binding of methods in some class may change accidentally after modifications in other classes.

13.2.9 Fragile base-class problem formulation

The study on the "fragile base-class problem" by L. Mikhajlov and E. Sekerinski [53] shows many different problems which can occur in unknown descendant classes, following the modification of an ancestor class. However, those problems are "semantical clashes" (concerning accidental incompatibility of behavior of modified methods), while in this thesis we tackle "syntactical clashes" (concerning accidental compatibility of declarations of added methods).

13.2.10 Summary of the comparison

In our opinion, none of the solutions presented above solve the problem of identifier clashes as completely as our solution does. What our methodology of hygienic identifiers offers can be summarized in the following sentence. Implementation of new functionality performed by addition of new method identifiers and new fields in an existing class (or in an existing mixin) will never change the behavior of existing code (except, of course, for code using reflection mechanisms for finding methods by their names, which avoid static verification). This property might be seen as a special case of the general *Flexibility Theorem*, formalized in [53]. It can be also rephrased as: "the code is safe for the past and for the future". We mean by that, that the code does not break any existing code, neither will be accidentally broken by any further safe-looking changes in other parts of the program on which it depends.

On a first thought somebody might argue that our approach makes the code less readable. However, since more and more components from different vendors of different versions are used to build software nowadays, it is becoming important to develop mechanisms which decrease the chances of inter-component incompatibility problems. Therefore we believe that such modification is not a big cost for the guarantees of safety we provide. Additionally, the problem of a little longer source code can be also solve with the help of development tools designed specifically to work with Magda. Such tools could hide the annotations with mixin names in normal program browsing (and show them only on demand). Additionally, when programer writes a short form of identifier reference, like `obj.m1()`, they could also automatically expand (using default visibility rules) to the expanded one, like `obj.M_1.m1()`.

13.3 Code reuse mechanisms

The reusability of software components has been in the scope of research for many years already. Many researchers found single inheritance mechanisms not satisfactory and developed numerous solutions to enable more extensive reuse. We have discussed shortcomings of those in Section 2.4, however in this section we briefly summarize differences between most of the widely known existing solutions and the approach we used in Magda.

13.3.1 Multiple inheritance

The most popular implementation of multiple inheritance paradigm is present in C++ [63]. Other versions of that paradigm are present also in Dylan [31], Python [13] and Loglan [45]. However, those solutions have complicated semantics and are subject to specific conflicts as mentioned before in Section 2.4.3. Additionally, all of them also suffer from the conflicts of identifiers presented in Section 2.2.

One of the features present in C++ multiple inheritance is private inheritance. Apart from removing from the type the information about all the methods and state of its private superclass, this feature also influences multiple inheritance. When a class A inherits from two classes B_1 and B_2, where each of those privately inherit from a class C, then class A will in fact contain two instances of class C which are not visible via the public interface. One of those instances will be visible by the B_1 part of A, while the other one through the B_2 part of A.

Such feature is not directly available in Magda. This could be however simulated by placing in mixins B_1 and B_2 a field of type C and use it in the same way. This way any object built using mixins B_1 and B_2 will contain two fields of type C. Moreover, since in C++ this inheritance relation was not externally visible, it is a functionally equivalent solution. Additionally, as often stated (see the GOF book [37]), in many such cases it is more suitable to use composition than inheritance.

13.3.2 Mixins

Mixins are a well-known solution to the reuse and modularization problem which is simpler than multiple inheritance. Mixins have been first informally introduced in the Flavors language [56], and then applied by Bracha more widely in JIGSAW [20] and formalized by Bracha and Cook in [22]. Later on, mixins had many more formal models as the one presented by Ancona et al. in [11] and in work on MixedJava by Flat et al. in [36]. Those have also been implemented as part of JAM language [10], as well as MixGen [6], and more recently in Scala [58] and NewSpeak [21].

However, mixins have not yet achieved very wide acceptance. It is believed that one of the main problems is the "fragile class hierarchies" problem, as described in Section 2.4.4. The concept of mixin is also a base building block in Magda language, however thanks to our hygienic approach, mixins do not suffer from the above mentioned problem. And our understanding is this is a first mixin-based solution which is completely free from that problem.

Another difference between Magda and most of the other mixin-based solutions is that in the latter ones the mixin is a construct which is used to transform one class into another. And classes as well as interfaces still play significant role in such languages. For example, the requirement for the class which is a parametric superclass of a mixin in MixGen language [6] is specified in the following way: this class needs to implement an interface or to be a subclass of some given class. A similar solution is also present in MixedJava [36]. On the other hand, in Magda the mixin is the only entity used to create objects from, to reuse code, and define nominal types at the same time. As a result, conditions placed on the "parametric superclass" in Magda (called base mixin expression) are also specified using a sequence of mixin names. Thus, Magda has a simpler and easier to understand semantics.

13.3.3 Traits

The trait construct is a solution which was developed as a successor of the mixin-based approach which allows the programmer to work around problems of accidental name clashes. Initially it was first introduced in a untyped setting as an extension of Smalltalk (which has a working implementation) [33, 60]. However, later on it was studied also in a typed (thus often restricted) setting by Smith et al. as an extension of Java [62] and by Oscar Nierstrasz et al. [57] as well as Bono et al. in a calculus called FRJ [16]. Finally it was implemented in the statically type checked language Fortress [9, 7, 8]. However, the typed setting of Fortress enforced many limitations, in particular, method renaming is not allowed in Fortress. The advantage of the composition mechanism available in trait-based solutions is that they warn the programmer when name clashes occur and allow her/him to solve them by modifying the way traits are composed in the class definitions (which is not available in most mixin solutions). This modification is performed using operators like method hiding and aliasing.

However, it is important to notice that traits do not inherently protect the user from the problems caused by name clashes. The change in one part of the code might require modification in other parts of the code (possibly written by another programmer). Additionally, renaming of one method might make some trait incompatible with other collaborating components, effectively breaking some hidden dependencies. All those problems have been described in detail in Section 2.4.5.

Finally, we think that traits grown into a complicated solution, having many different operators to built new classes from while renaming, aliasing, hiding, freezing and unfreezing existing methods. With respect to that, we believe that our solution might appear more attractive to programmers because it seems to have a simpler semantics with only one modularization unit and only one operator to combine such units into a new sequence. The fact that all the code is in mixins, makes the reuse also simpler, while in case of trait-based solutions, there are different rules to reuse code present in traits and different for reusing of code present in classes as glue-code etc.

13.4 Encapsulation

An additional characteristic which makes our approach different from most of other approaches to component reuse (as multiple, as well as mixin and trait based inheritance) is the approach to encapsulation. In this section by encapsulation we understand the property of hiding from the user of a class the implementation details such as mixins/traits which have been used to create the class, or its superclasses.

In Magda, the set of mixins from which the object has been created is always externally visible, since all the references to methods and fields need to be prefixed with the name of the mixin from which the method comes.

On the other hand, most of the existing solutions try to hide this knowledge from the user, so this can be seen as an advantage over the Magda approach. However, it is important to notice that most of the solutions fail to completely hide this knowledge from the user. This means that it is not completely transparent for the user, because the user which is not aware of such "internal structure" of the class, might, from time to time, run into different kinds of problems. Below we shortly describe in what cases such internal structure is visible to the clients, thus the encapsulation is violated.

First of all, in C++, when one declares a class inheriting from two other classes, one should know if there are some common superclasses of those two classes. This is needed in order to properly choose at that point one of the semantics of inheritance (private, public or virtual one).

In the case of a mixin-based solutions like MixGen [6], or MixedJava [36], a class can have many distinct implementations of one method. Then to call a method declared in a specific superclass or mixin, one has to cast the type of an object to that specific type. If he or she is not aware of that fact, then assigning the same object to another variable might suddenly change the semantics of the call of some method, because the method is of some more specific or more general type.

In the case of freezable traits [34], to properly unfreeze a method in some trait or class one needs to know in which supertrait this method has been declared. Additionally, the user needs to be aware of which methods are called in which other methods (as described in Section 2.4.5), therefore the encapsulation is also violated even at the level of methods.

Therefore, in all those cases the user needs to be aware of the internal structure of the class to use it properly.

Thus, since in general we believe that the solutions with clear rules are the best and easiest to use, we decided to keep the structure of object (presenting a list of mixins from which it has been created) visible all the time.

13.5 Constructs not present in Magda

Magda is a language with only one base notion which is the mixin and one operator, which concatenates sequences of mixins to create an object from. However, the mixins construct is general enough to simulate many other notions. The mixin construct at the same time plays the role of a type, an interface and a supplier of implementation and a unit of reuse.

As a result, the interface concept can be represented in Magda by a mixin which has all methods marked as "`abstract`". Then a class implementing such an interface is represented by a mixin which refers to this "interface-mixin" in its base mixin expression, and contains the declaration of methods which implement methods introduced in the given "interface-mixin". Similarly, an abstract class can be represented in Magda in a similar way as an interface, namely by a mixin in which some subset of methods is marked as "`abstract`".

Method renaming and hiding as present in Eiffel [52] and traits [60, 33] are also not available in Magda. However, it is important to notice that those mechanisms have been developed mainly to work around problems with the identifier clashes. On the other hand, in Magda there is no chance of name clash of two different introductions of the same identifier. Therefore there is no significant need to use such mechanisms as renaming and hiding in Magda, thus we decided to skip it for simplicity. One might imagine other scenarios, where method renaming is used to change the interface of the class, and those cannot be simulated in Magda. However, we believe that such scenarios are rare and we have not found significant references to them in the literature.

In Magda there is also no distinction between units of reuse (traits) and generators of instances (classes) as in the traits approach. This distinction is used mainly to allow the user to control manually the process of combination of many traits. The main reason to introduce this approach was to allow the user to manually resolve name clashes, and, as mentioned above, this problem does not exist in Magda.

Chapter 14

Conclusions

14.1 Summary

In this thesis we tried to solve a few problems, described in Section 2, which restrict freedom and safety of modularization and composition in OO languages. After studying many existing options (as described in Section 13) we have introduced three new features in our language (see Section 1.3), namely: modular initialization protocol, hygienic identifiers and purely mixin-based design. As a result we were able to design the Magda language, which helps the programmers to write highly modular, customizable code, which will not break accidentally. The principles of the language were first informally introduced the big-step semantics have been introduced (see Section 7) as well as the type system (see Section 8). Then, the type soundness has been proven (see Section 10). As an additional result, the thesis contains a new way of proving type soundness of languages defined by a big-step semantics.

The two key and unique properties, which are guaranteed by the design of the Magda language are the following:

- An addition of a new identifier will never break the existing code (see Section 3.11.1).
- Any two independent mixins, which are not explicitly exclusive (see Section 3.11.1) can be safely combined together (see Section 13.2.10 for more details).

The only price of all those features is, as we believe, a little less concise syntax caused by longer identifier references.

However, considering that contemporary software has a longer and longer life cycles, and the costs of maintenance of existing software systems are becoming larger and larger (also when compared to the costs of development of new systems), we believe that safety and modularity of languages will play a more and more crucial role in the future.

Thus, we believe that the approach presented in this thesis forms a useful contribution to the design of OO languages.

14.2 Future work

14.2.1 Implicit Genericity

A recent result of our research is the mechanism called *implicit genericity*. This mechanism allows the programmer to pick an existing library, and reuse it at some point with some specific modifications, while retaining other existing uses of it in the same application in their original form. The modification enabled by this approach is a replacement of one class with another compatible one. The replacement of one class with another means that all object which would originally be created from one class will instead be created from the new one. Such a replacement should be possible on any code, without anticipation of this modification by its author. Notice that such a tool allows one to redefine almost any part of code of some existing library implemented by means of any method. It happens so because one can always replace the class in which the method was declared with another one, which has this specific method overridden. We have first presented this idea by adding it to Java, thus obtaining the language called ImpliJava [46].

One of the critical aspects of the static verification of type safety of such replacements is the verification whether some class can be safely used to replace another one. To ensure that a class can be used to replace another one, the compiler has to make sure that it supports all the constructors of the other class (more details in [46]). Unfortunately in Java this often requires a significant amount of work to be done in order to make some class compatible with another (like copying of all the signatures of constructors etc.). However, in Magda it seems much easier, or not requiring any work at all. In Magda, assuming that a mixin does not add any required initialization modules, such a mixin can be safely added to any other sequence of mixins, and will not cause any object creation expressions to fail because of incompatible initialization.

This means that Magda is a language which should work seamlessly with the implicit genericity solution. Thus, we plan to analyze such an extension of Magda in the near future.

14.2.2 First-class genericity

Typical genericity in the style of Java Generics is called second-class genericity, because type arguments can be used only in type expressions. In such an approach, type arguments cannot be used in any object expressions (as for example object creation expressions), which is in turn possible in first-class genericity. One of the reasons for that, is the fact that generics has been added to Java as a backward compatible solution, which means that the code written using generics can be compiled into a bytecode which runs on an older virtual machine, thanks to the so called erasure mechanism. However this is not the case in C♯. Another, more fundamental, problem with generics in Java as well as in C♯ is that the upper bound of some type parameter, which enforces the actual parameter value to be a subtype of some boundary type, does not guarantee anything about the list of constructors present in such actual parameter class. Therefore, one cannot use the value of such parameter to create a new object from.

Then, to overcome this problem, the designers of MixGen [6] (which is an extension of Java with the first class genericity) decided to specify bounds for parameters of generic classes by writing (i) a supertype, together with (ii) a list of signatures of constructors which need to be supported by the class being the actual parameter.

When we consider adding such a first class genericity to Magda it seems obvious, that such additional requirements referring to the initialization protocol would not be needed. It happens so, because when a sequence of mixins M_1 is a subtype of sequence M_2, then it just means that M_1 is a larger set of mixins (see Section 3.9 and Section 8.6). Thus, the sequence M_1 being a subtype of M_2 guarantees that the set of initialization parameters supported by M_1 will also be a superset of such a set in M_2. The only thing that might limit the usage of M_1 in contexts where M_2 was used would be an addition of a required module with a non-empty set of input parameters. Then, any object creation from M_1 requires supplying those additional parameters.

Therefore, it seems that the addition of first-class genericity to Magda should be more natural than in Java and would create a useful and expressive mechanism. We believe that this is a worthy direction of further research.

Bibliography

[1] *JavaCC*: A Java Compiler Compiler. Available at `https://javacc.dev.java.net/`.

[2] *Visual Basic .NET Language Reference*. Microsoft Press, 2002.

[3] *Delphi Language Guide*. Borland Software Corporation, 2004.

[4] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[5] M. S. Ager. From natural semantics to abstract machines. In *Logic Based Program Synthesis and Transformation*, volume 3573 of *LNCS*, pages 245–261. Springer-Verlag, 2005.

[6] E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. In *Proc. OOPSLA'03*, pages 96–114, 2003.

[7] E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, and G. L. S. Jr. Project fortress: A multicore language for multicore processors. *Linux Magazine*, September:38–43, 2007.

[8] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification. Technical report, Sun Microsystems, 2008.

[9] E. Allen, J. J. Hallett, V. Luchangco, S. Ryu, and G. L. Steele, Jr. Modular multiple dispatch with multiple inheritance. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1117–1121. ACM, 2007.

[10] D. Ancona, G. Lagorio, and E. Zucca. Jam — a smooth extension of Java with mixins. In *Proc. ECOOP'00*, volume 1850 of *LNCS*, pages 145–178. Springer-Verlag, 2000.

[11] D. Ancona and E. Zucca. An algebra of mixin modules. In *Proc. Workshop on Algebraic Development Techniques'97*, volume 1376 of *LNCS*, pages 92–106. Springer-Verlag, 1997.

[12] D. W. Barron, editor. *Pascal: The Language and its Implementation.* John Wiley, 1981.

[13] D. Beazley and G. V. Rossum. *Python. Essential Reference.* New Riders Publishing, 1999.

[14] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008.

[15] L. Bettini, V. Bono, and S. Likavec. A Core Calculus of Higher-Order Mixins and Classes. In *Proc. TYPES '03 (Selected Papers)*, volume 3085 of *LNCS*. Springer-Verlag, 2004.

[16] V. Bono, F. Damiani, and E. Giachino. On traits and types in a java-like setting. In *Proc. Fifth IFIP International Conference on Theoretical Computer Science - TCS'08*, IFIP International Federation for Information Processing, pages 367–382. Springer-Verlag, 2008.

[17] V. Bono and J. D. M. Kuśmierek. FJMIP: A calculus for a modular object initialization. In *Proc. FCT 2007*, volume 4639 of *LNCS*, pages 100–112. Springer-Verlag, 2007.

[18] V. Bono and J. D. M. Kuśmierek. Modularizing constructors. *Journal of Object Technology*, vol. 6 no. 9. Special Issue: TOOLS EUROPE 2007:297–317, 2007.

[19] V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. ECOOP'99*, volume 1628 of *LNCS*, pages 43–66. Springer-Verlag, 1999.

[20] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, The University of Utah, 1992.

[21] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, and E. Mirand. The Newspeak Programming Platform. Technical report, Cadence Design Systems, 2008.

[22] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA'90*, pages 303–311. ACM Press, 1990.

[23] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. *SIGPLAN Notices*, 33(10):183–200, 1998.

[24] G. Bracha, M. Odersky, D. Stuotamire, and P. Wadler. *GJ Specification.* May 1998.

[25] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[26] R. J. Cardone. *Language and Compiler Support for Mixin Programming.* PhD thesis, The University of Texas at Austin, 2002.

[27] W. J. Chun. Python 2.2. Q&A with Guido van Rossum, creator of Python. *Linux J.*, 2002(98):4, 2002.

[28] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, (5(2)):56–68, 1940.

[29] T. Cohen and J. Gil. Better construction with factories. *Journal of Object Technology*, 6(6):109–129, 2007.

[30] S. Cook. OOPSLA'87 panel P2 varietes on inheritance. In *Proc. OOPSLA'87 Addendum to Proceedings*, pages 35–40. ACM Press, 1987.

[31] I. D. Craig. *Programming in Dylan.* Springer-Verlag, 1996.

[32] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: Fickle$_{II}$. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(2):153–191, 2002.

[33] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 2:331–388, 2006.

[34] S. Ducasse, R. Wuyts, A. Bergel, and O. Nierstrasz. User-changeable visibility: resolving unanticipated name clashes in traits. In *Proc. OOPSLA'07*, pages 171–190. ACM, 2007.

[35] U. W. Eisenecker, F. Blinn, and K. Czarnecki. A solution to the constructor-problem of mixin-based programming in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, 2000.

[36] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. POPL '98*, pages 171–183. ACM, 1998.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995.

[38] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Longman, 1983.

[39] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification.* Addison-Wesley, Sun Microsystems, 2005.

[40] A. Hejlsberg, P. Golde, and S. Wiltamuth. *C♯ Language Specification.* Addison-Wesley, 2003.

[41] H. Ibraheem and D. A. Schmidt. Adapting big-step semantics to small-step style: Coinductive interpretations and "higher-order" derivations. In *Second Workshop on Higher-Order Techniques in Operational Semantics (HOOTS2).* Elsevier, 1997.

[42] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions in Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[43] G. Kahn. Natural semantics. In *Proc. STACS'87, 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[44] S. Kochan. *Programming in Objective-C*. Sams, 2004.

[45] A. Kreczmar, A. Salwicki, and M. Warpechowski. *LOGLAN '88—report on the programming language*. Springer-Verlag, 1990.

[46] J. D. M. Kuśmierek. Implicit first class genericity. In *Proc. Software Composition 2009*, volume 5634 of *LNCS*, pages 107–124, 2009.

[47] J. D. M. Kuśmierek. MTJ: Magda Language Compiler. Available at `http://duch.mimuw.edu.pl/~jdk/`, 2009.

[48] J. D. M. Kuśmierek and V. Bono. Hygienic methods, Introducing HygJava. *Journal of Object Technology*, vol. 6 no. 9. Special Issue: TOOLS EUROPE'07:209–229, 2007.

[49] G. Leavens and Y. Cheon. *Design by Contract with JML*. 2003.

[50] X. Leroy. *The Objective Caml System Release 3.09*. Institut National de Recherche en Informatique et en Automatique, 2005.

[51] X. Leroy and H. Grall. Coinductive big-step operational semantics. *CoRR*, abs/0808.0586, 2008.

[52] B. Meyer. An Eiffel Tutorial. Technical Report TR-EI-66/TU, ISE, 2001.

[53] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proc. ECOOP'98*, volume 1445 of *LNCS*, pages 355–382. Springer-Verlag, 1998.

[54] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[55] G. Monteferrante. *javamip2java*: A preprocessor for the JavaMIP language. Available at `http://www.di.unito.it/~bono/papers/javamip/`, 2006.

[56] D. A. Moon. Object-oriented programming with flavors. In *Proc. OOPSLA'86*, pages 1–8. ACM Press, 1986.

[57] O. Nierstrasz, S. Ducasse, S. Reichhart, and N. Schärli. Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006, Institut für Informatik, University of Bern, Switzerland, 2005.

[58] M. Odersky, P. Altherr, V. Creme, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The Scala Language Specification, version 1.0. Technical report, Programming Methods Laboratory, EPFL, 2006.

[59] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, University of Aarhus, 1981.

[60] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behaviour. In *Proc. ECOOP'03*, volume 2743 of *LNCS*, pages 248–274. Springer-Verlag, 2003.

[61] Y. Smaragdakis and D. S. Batory. Mixin-based programming in C++. In *Proc. GCSE'00*, volume 2177 of *LNCS*, pages 163–177. Springer-Verlag, 2001.

[62] C. Smith and S. Drossopoulou. Chai: Traits for Java-like Languages. In *Proc. ECOOP'05*, volume 3586 of *LNCS*, pages 453–478. Springer-Verlag, 2005.

[63] B. Stroustrup. *The C++ programming language.* AT&T, 1997. Third edition.

[64] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.