The Magda Language: Ten Years After

Fundamenta Informaticae XXX (2018) 1001–1024
DOI 10.3233/FI-2016-0000
IOS Press

The Magda language: ten years after

Viviana Bono
Dipartimento di Informatica
University of Torino
Corso Svizzera 185, 10149 Torino, Italy
bono@di.unito.it

Abstract. We discuss Magda ten years after its design. Magda is a mixin-oriented programming language and its goal is to improve code modularity and, as a consequence, code reuse. The aim of this paper is to survey Magda and position it in today's programming language scenarios.

Keywords: Object-oriented programming language, mixin, modularity, code reuse.

1. Introducing Magda

From the early nineties up to the new millennium there was a flourishing of studies on object-oriented languages, that:

- clarified aspects of the theory behind objects, notably the milestone *A Theory of Objects* by Abadi and Cardelli [1], the *lambda-calculus of objects* by Fisher, Honsell, Mitchell [15], and the work by Bruce [11];
- helped main-stream languages advance, for example Java generics (formalized elegantly via Featherweight Java in [23]) made their way into Java 5;
- studied alternatives to single and multiple inheritance, the former too rigid to be an effective means of code reuse, the latter perceived as too complicated and too dangerous [12]. There is a large body of work from the seminal work of Bracha and Cook [10], to mixin-based inheritance [29, 10, 9, 3, 16, 7, 4] and trait-based inheritance [31, 13].

Address for correspondence: bono@di.unito.it

Magda stems from this tradition and it is a general-purpose language oriented to code reuse with three well-defined features:

- Magda is *mixin*-based, but with a twist with respect to other mixin-based proposals.
- Magda offers *modular* constructors, as an alternative to traditional ones, which are monolithic.
- Magda imposes *hygienicity* on identifiers of fields and methods, in order to avoid accidental name clashes.

Magda was studied within Jarek Kuśmierek's PhD thesis [25] and presented at the conference ECOOP 2012 [8]. In the latter, the main focus was on the modular initialization protocol. A thorough description of the problems Magda help solving is present in the previous publications. A detailed description of the language syntax can be found in [25]. A proof-of-concept implementation of Magda, the examples present in the paper, and a first proposal for an IDE can be found at [27]. The aim of this paper is to survey Magda and position it in today's programming language scenarios.

This paper is organised as follows: Section 2, Section 3 and Section 4 describe by examples, respectively, the three main features of the language mentioned above. Section 5 hints briefly at the Magda type system. Section 6 discuss the future potential of Magda.

Note. The examples of Section 2.2, Section 3, and Section 5 are taken from [8].

2. Magda is mixin-based

Magda is a *mixin-oriented* language, as opposite to a *class-based* language. A class is intended as a "closed" collection of fields and methods, describing the state of its object instances and the operations on the state, possibly inherited from ancestor classes. A mixin differs from a class as it is inherently "open", since it can assume operations that can be supplied by other templates (i.e., other mixins or classes). In what does a mixin differs from, say, an abstract class? One strong point of mixins is that they can be *composed*, to obtain a tamed form of multiple inheritance, which is linearised and thus does not suffer from the *diamond problem* [10], that is one of the biggest drawbacks of multiple inheritance. It happens when a class inherits from the same parent class via more than one path, possibly producing conflicts among components. Method conflicts are relatively easy to solve, for example by means of overriding, instead state conflicts are trickier, since it is not clear if the state should be inherited more than once or only once, and, for the latter, along which path. Mixin composition enforces a linear, hence unique, inheritance path, therefore avoiding the diamond problem. In most mixin-based proposals [9, 17, 7, 4, 37], mixins and classes live together: a mixin is a class parametrised on a superclass. The same mixin can be applied to different superclasses, originating families of object templates. In Magda, however, there is only one construct: the mixin. A mixin can extend more than one base mixin at once, therefore Magda offers a form of multiple inheritance. Each object in Magda is created from a non-empty sequence of mixins, that are composed together. A sequence of mixins plays a role similar to the one of a class in other languages.

2.1. The Decorator pattern in Magda

The Decorator pattern can be used to extend the functionality of a certain object at runtime (by *decorating* it), independently of other instances of the same class [18]. This pattern is designed so that multiple decorators can be stacked on top of each other, each time adding a new functionality to the overridden method(s). We depict an example here, where a string is first framed within two rows of stars, then duplicated, then these two effects are combined in two different orders, as shown in Figure 1. The reader will forgive us for the lack of graphics in Magda, however this example mirrors the paradigmatic Gang-of-Four example of a window decorated with a border and a scroll bar, in any order [18].

```
Label:
--------------------
| This is a label |
--------------------

FramedLabel:
**************************
--------------------------
| This is a FRAMED label |
--------------------------
**************************

DoubledLabel:
----------------------------
| This is a DOUBLED label |
----------------------------
----------------------------
| This is a DOUBLED label |
----------------------------

multiInherit1(Framed + Doubled):
****************************************
----------------------------------------
| This is a FRAMED and so DOUBLED label |
----------------------------------------
****************************************
****************************************
----------------------------------------
| This is a FRAMED and so DOUBLED label |
----------------------------------------
****************************************

multiInherit2(Doubled + Framed):
****************************************
----------------------------------------
| This is a DOUBLED and so FRAMED label |
----------------------------------------
----------------------------------------
| This is a DOUBLED and so FRAMED label |
----------------------------------------
****************************************

FramedDoubledLabel(Framed + Doubled):
****************************************
----------------------------------------
| This is a FRAMED and so DOUBLED label |
----------------------------------------
****************************************
****************************************
----------------------------------------
| This is a FRAMED and so DOUBLED label |
----------------------------------------
****************************************

DoubledFramedLabel(Doubled + Framed):
****************************************
----------------------------------------
| This is a DOUBLED and so FRAMED label |
----------------------------------------
----------------------------------------
| This is a DOUBLED and so FRAMED label |
----------------------------------------
****************************************
```

Figure 1. A decorated label (from the execution of the `TestLabel` Magda program)

Within a language without mixins, one obvious solution would be to create a hierarchy of classes, as depicted in Figure 2, but this solution is a bad one, since it introduces redundancies of classes and code duplication (in this example, in the method `PrintLabel()`). The Decorator design pattern (see Figure 3) offers a better solution, based on object composition and dynamic binding. Intuitively, a concrete component object of type `Label` can be wrapped by a decorator object of type `Doubled` or by a decorator object of type `Framed`, which in turn can be wrapped by another decorator object of

type `Framed` or by another decorator object of type `Doubled`. The method `PrintLabel()` in `Label` pretty-prints the content of the label, in `Framed` adds a row of stars before and after forwarding the call to its `component` field, in `Doubled` forwards the call to its `component` field twice.

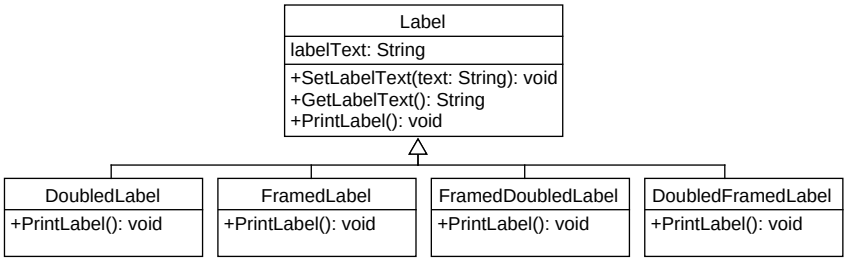

Figure 2. An inheritance-only-based solution

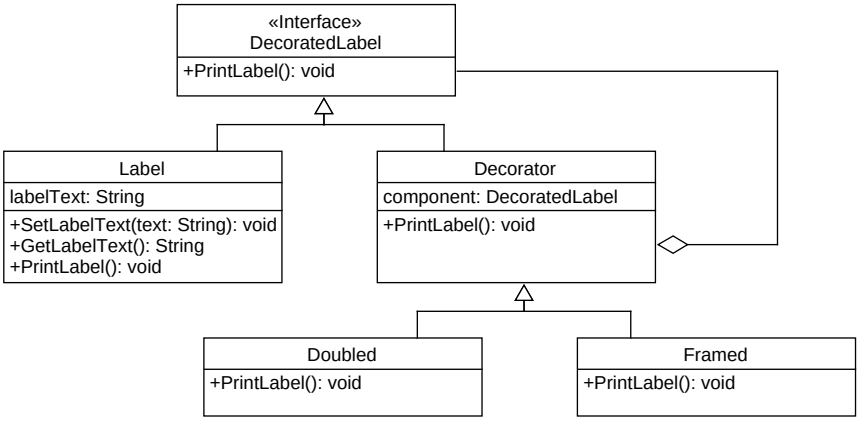

Figure 3. A Decorator-based solution

Implementing the Decorator in Magda is easy and effective thanks to mixin composition. To produce the sought decorations, we write four mixins. The first one extends the base mixin Object, declares a field and three methods. Methods must be prefixed either with `new` or with `override`, indicating, respectively, an introduction or a redefinition. Each access to a field or call to a method is prefixed with the name of the mixin where the identifiers was introduced (with a `new` keyword); this *hygienic* approach to identifiers is to avoid accidental name clashes. The `this` represent the self object, as usual. The mixin `String` is part of the Magda library.

```
mixin Label of Object =
  labelText:String;
```

```
  new void SetLabelText (text:String)
  begin
    this.Label.labelText := text;
  end;

  new String GetLabelText ()
  begin
    return this.Label.labelText;
  end;

  new void PrintLabel ()
    lineFeed:String;
    lenText:Integer;
    i:Integer;
  begin
    lenText := this.Label.labelText.String.length();
    lineFeed := "\r\n";
    i := 0;
    while(i < (lenText+4))
      "-".String.print();
      i := i + 1;
    end;
    lineFeed.String.print();
    "| ".String.print();
    this.Label.labelText.String.print();
    " |".String.print();
    lineFeed.String.print();
    i := 0;
    while(i < (lenText+4))
      "-".String.print();
      i := i + 1;
    end;
    lineFeed.String.print();
  end;

end;
```

The second and third mixins extend the previous mixin and redefine a method.

```
mixin DoubledLabel of Label =

 override void Label.PrintLabel ()
    lineFeed:String;
```

```
    begin
     lineFeed := "\r\n";
     lineFeed.String.print();
     super();
     super();
     lineFeed.String.print();
    end;

end;


mixin FramedLabel of Label =

  override void Label.PrintLabel ()
    lineFeed:String;
    lenText:Integer;
    i:Integer;
   begin
     lenText := this.Label.labelText.String.length();
     lineFeed := "\r\n";
     i := 0;
     while(i < (lenText+4))
       "*".String.print();
       i := i + 1;
     end;
     lineFeed.String.print();
     super();
     i := 0;
     while(i < (lenText+4))
       "*".String.print();
       i := i + 1;
     end;
     lineFeed.String.print();
   end;

end;
```

We now use a `let` assignment to give names to sequences of mixins representing mixin compositions, that will be used later on in the code into `new` expressions to create objects. We also introduce a fourth mixin. It is an "empty" mixin and it has the purposes of giving a name to a compound mixin and therefore of inducing a new type.

```
let FDLabel = Label, FramedLabel, DoubledLabel, FramedDoubledLabel;
```

```
let DFLabel = Label, DoubledLabel, FramedLabel, FramedDoubledLabel;

mixin FramedDoubledLabel of Label, FramedLabel, DoubledLabel =

  // This mixin has an empty body

end;
```

The reader should pay attention to the difference between names for mixin compositions introduced via `let` (such as `FDLabel`), that are mere syntactic sugar to shorten `new` expressions, and mixin definitions (such as `FramedDoubledLabel`), possibly extending other mixins.

What follows is a program written in Magda, that exploits the mixins previously defined. We alternate code and explanations, with the goals of showing how to implement the Magda version of the Decorator and of introducing some syntax and characteristics of the language, as a byproduct.

In the first part, there are inclusions of the Magda library (containing some basic utilities) and of the mixins displayed above.

```
include "Magdalib/library.magda";
include "Mixinlib/mixinLabel.magda";
include "Mixinlib/mixinFramedLabel.magda";
include "Mixinlib/mixinDoubledLabel.magda";
include "Mixinlib/mixinFramedDoubledLabel.magda";
```

In Magda, a possible way of writing a program is to encapsulate a "main" method into a mixin.

```
// Main
mixin MainClass of Object =

    new Object mainProgram()

    mainLabel:Label;
    mainFramedLabel:FramedLabel;
    mainDoubledLabel:DoubledLabel;
    multiInherit1:Label;
    multiInherit2:Label;
    mainFramedDoubledLabel:FramedDoubledLabel;
    mainDoubledFramedLabel:FramedDoubledLabel;
```

The mixin `MainClass` extends the base mixin Object and defines a new method called `mainProgram()` with no arguments and `Object` as the return type (used here as the Java `void` type). In the method there are variable declarations: the types are mixin names.

Then five objects are created[1]. We will describe (modular) constructors and object creation later on. The last two objects mix frame and duplication in two different orders and realize the decorations.

[1] In this version of Magda, it is necessary to list all the super-mixins of the mixin(s) from which the object is created, and the supermixins come first in the list. This feature is obviously inconvenient and will be overcome in possible future versions of the language.

In general, the order among unrelated mixins (i.e., mixins that are not in the same hierarchy, even if they might share ancestors), such as `FrameLabel` and `DoubleLabel`, matters and can originate different effects (as in our case). The order, in particular, imposes an order over method redefinitions, when present.

```
begin

mainLabel := new Label[];
mainFramedLabel := new Label, FramedLabel[];
mainDoubledLabel := new Label, DoubledLabel[];
multiInherit1 := new Label, FramedLabel, DoubledLabel[];
multiInherit2 := new Label, DoubledLabel, FramedLabel[];
```

The following assignments show other ways of composing the same mixins.

```
// mainFramedDoubledLabel :=
//        new Label, FramedLabel, DoubledLabel, FramedDoubledLabel[];
// mainDoubledFramedLabel :=
//        new Label, DoubledLabel, FramedLabel, FramedDoubledLabel[];

// Thanks to the 'let', a more compact syntax:
mainFramedDoubledLabel := new FDLabel[];
mainDoubledFramedLabel := new DFLabel[];
```

What follows, it is a portion of code where methods are called on the previously created objects.

```
"Label:\r\n".String.print();
mainLabel.Label.SetLabelText("This is a label");
mainLabel.Label.PrintLabel();
"\r\n".String.print();

"FramedLabel:\r\n".String.print();
mainFramedLabel.Label.SetLabelText("This is a FRAMED label");
mainFramedLabel.Label.PrintLabel();
"\r\n".String.print();

[...]

"multiInherit1(Frame + Double):\r\n".String.print();
multiInherit1.Label.SetLabelText("This is a FRAMED and DOUBLED label");
multiInherit1.Label.PrintLabel();
"\r\n".String.print();

[...]
```

```
  "FramedDoubledLabel(Double + Frame):\r\n".String.print();
  mainDoubledFramedLabel.Label.SetLabelText("This is a DOUBLED and FRAMED label");
  mainDoubledFramedLabel.Label.PrintLabel();
  "\r\n".String.print();
  end;

end;
```

The statement `mainDoubledFramedLabel.Label.PrintLabel()` is a method call. The method `PrintLabel()` (introduced in the mixin `Label`) is called on the object `mainDoubledFramedLabel`. In the statement that follows, an object from mixin `MainClass` is created, in order to invoke the `mainProgram(): (new MainClass[]).MainClass.mainProgram()`.

2.2. Virtual and abstract methods

A method redefinition has a different syntax from method introduction: a redefinition begins with an `override` keyword and contains the name of the mixin in which the redefined method was introduced. The body of a method redefinition can use a `super(...)` expression to call a previous implementation of the method. When a `super(...)` expression is evaluated in a method *mt* of a mixin M_c belonging to a mixin sequence $\overrightarrow{M}$, the previous implementation of *mt* is the one present in the last mixin in $\overrightarrow{M}$ preceding M_c, which contains a definition or a redefinition of the method *mt*. As a result, the actual method body called by `super(...)` within a method redefinition depends on the mixins (and their order) which have been used to create the object in the corresponding `new` expression. Indeed, we saw this in the example of the Decorator in Section 2.1. Because we choose explicitly which method from which mixin to redefine when declaring a method redefinition, there are no problems when two base mixins of a given mixin contain method definitions with the same name. Moreover, this explicitness also ensures that accidental name clashes are avoided when a new method is added to a mixin extended by other mixins.

A mixin declaration can also contain a declaration of a new method identifier without the body, marked with the keyword `abstract`. The body must be supplied by another mixin, by marking it with the keyword `override`. The following example shows some features of the abstract methods.

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

mixin M2 of M1 =
  override 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;

mixin M4 of M1 =
  new String Met1()
  begin
    return super().String.add(" with redefinition from M4");
  end;
end;


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

(new M1, M3, M2 []);
// does not compile, M3.Met1 not suited as first definition of M1.Met1
// (contains call to super())

(new M1 []);
// does not compile, implementation of M1.Met1 missing

(new M4 []);
// does not compile, M4.Met1 is a new method, it cannot reference a super()
```

3. On modular constructors

We developed for Magda a *Modular Initialization Protocol* approach (*MIP* for short) based on small, composable pieces called *initialization modules* (*ini modules* from now on).

An ini module accords with the following syntax:

$\{$ `required` | `optional` $\}$ `Mixin` $(\mathtt{p}^1\colon \mathtt{T}^1;\ldots;\mathtt{p}^n\colon \mathtt{T}^n)$ `initializes` $(\mathtt{p}_1;\ldots;\mathtt{p}_k)$
`begin`
$\mathtt{I}_1$;
`super`$[\mathtt{p}_1{:=}\mathtt{e}_1,\ldots,\mathtt{p}_n{:=}\mathtt{e}_n]$;
$\mathtt{I}_2$;
`end`

First of all, the signature of an ini module declares whether its usage is *required* or *optional*, meaning whether its execution is mandatory or not during the initialization of an object. Then, in the

style of Java constructors, which are named with the name of the declaring class, we have the name `Mixin` of the mixin in which this ini module is declared. Finally, each ini module comes with two list of parameters: the parameters in the first list $(\mathtt{p}^1\colon \mathtt{T}^1; ...; \mathtt{p}^n\colon \mathtt{T}^n)$ are called *input parameters*, and they must be supplied to execute the ini module; the parameters in the second list $(\mathtt{p}_1; ...; \mathtt{p}_k)$ are called *output parameters* and they make reference by name to input parameters declared in other modules, whose values will be computed by the current ini module and supplied, via the `super` construct, to such other modules. We will show how Magda's ini modules are activated by a running example in the next section.

3.1. An example

The program shown in Figure 4 exploits two mixins containing ini module declarations. The first mixin (`Point2D`) contains the declaration of ini module `mod1`, requiring parameters `x` and `y` and not computing any output parameter. That is why the `super[]` instruction in `mod1` does not contain any parameter assignment. The second mixin (`Point3D`) contains the declarations of two ini modules. The first one (`mod2`) expects one input parameter `z`, which is used to initialize the field `fz` of the object. The second one (`mod3`) has one input parameter `other` and three output parameters. Those output parameters refer to the input parameter of the module `mod2` declared in the same mixin, and to the parameters of the module `mod1` declared in the base mixin. Its body contains a `super[...]` instruction, which computes the values of the three output parameters.

Finally, the `MainClass.MainMatter` method contains two object creation expressions. The first expression supplies the values of three initialization parameters: `z` of mixin `Point3D`, and `x` and `y` of mixin `Point2D`. When this instruction is executed, an object of type `Point3D` is created and then the initialization protocol proceeds as follows:

1. the first parameter `Point3D.z` is supplied to module `mod2`, and that module is the first one to be executed;
2. when its execution reaches `super[]`, the search begins for the next module to be executed, which is `mod1`, because parameters `x` and `y` are yet to be consumed;
3. the last instruction of the module `mod1`, the `super[]` call, does nothing, since all parameters have been consumed.

The second object creation expression supplies the value of one initialization parameter, `Point3D.other`. When this instruction is executed, an object of type `Point3D` is created:

1. the optional module `mod3` is executed, and it computes the three parameters `x`,`y`,`z`;
2. the third parameter is passed to the module `mod2`;
3. when the execution of `mod2` reaches the `super[...]` call, the remaining two parameters are supplied to module `mod1`;
4. the last instruction of the module `mod1`, the `super[]` call, does nothing, since all parameters have been consumed.

```
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 4. Object initialization in Magda

As a result, all the ini modules declared in the program are executed during the creation of this object, in a bottom-up order: `mod3`, `mod2`, `mod1`.

3.2. The MIP constructs

The `new` construct Every object creation expression has the following syntax:

$$\texttt{new MixinExpression}[p_1{:=}e_1,...,p_n{:=}e_n]$$

where `MixinExpression` is an ordered sequence of mixins, followed by a (possibly empty) list of initialization parameters, together with their values. When such an expression is evaluated, the list of ini modules within the mixins beloging to `MixinExpression` is traversed. Each module for which there are input parameters supplied is executed and, when the execution of a given module reaches a `super[...]` instruction, the traversal procedure is resumed. A module is then executed when its input parameters either are supplied directly in the object creation expression or are supplied indirectly by another module (via its output parameters), which was executed before. In [8] a semi-formal semantics of this mechanism is described.

The `super` construct The `super[...]` instruction in an ini module is different from the `super(...)` expression within overriding method bodies, because the parameters supplied in `super[...]` do not have to be the parameters which will be passed to the ini module activated by this `super[...]` instruction. Those parameters can be in fact passed to other modules which will be executed later on. To understand this better, consider again the example in Figure 4. The `super[...]` instruction in the module `mod3` supplies the values of three initialization parameters and calls the module `mod2`. However, only one of these parameters (`z`) is consumed by `mod2`, while the remaining parameters (`x`, `y`) will be supplied to another ini module (`mod1`). Moreover, `super[ ]` instruction within `mod2` does not contain any parameters, however it calls the module `mod1`, that takes two parameters which have been computed by `mod3`.

3.3. How ini modules solve the problems of traditional constructors

In the following, we discuss how Magda's ini modules solve some problems of traditional constructors.

Optional parameters. We can introduce an optional ini module for each set of optional attributes as (input) parameters without increasing exponentially the number of ini modules with respect to the number of the attributes, since each ini module takes care only of its input parameters. Also, there is no problem with parameter lists of the same length and compatible types, thanks to named parameters.

Unnecessary constructor dependencies. Any new ini module introduced in a mixin is independent from the ini modules already present. We see this in the following example (continuation of the example from Figure 4):

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

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

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

The initialization module in mixin `ColorPoint` is independent from the ones in mixin `Point2D`, in particular it does not refer to any of `Point2D` ini modules' parameters. An ini module contains references to input parameters of other ini modules only when it computes their actual values (if this is the case, they are present as output parameters of the ini module); for instance, the ini module from Figure 4 with input parameter `other:Point3D` refers to parameters `x`, `y`, and `z` as its output parameters.

Multiple initialization options and code duplication. Each option corresponds to a certain ini module, and each ini module deals with one option without need for code duplication. An example is the one above that mixes colors with coordinates: this is a typical case that would cause code duplication when using classical constructors.

Fragile overloaded constructors. In Magda, the choice of the appropriate ini modules is done using the names of the parameters and no form of overloading is present. As a result, there cannot be ambiguities.

Problems with traditional mixins. Traditional mixins are basically classes parametrized over a superclass, that is, in addition to mixin composition (similar to the one of Magda's) there is also *mixin application*, which applies a mixin to a class to obtain a fully-fledged subclass (the class argument playing the role of the parent). Therefore, they suffer from the same problems of class-based languages with respect to initialization. We refer to Jam [4] (a Java extension with mixins) for a discussion on the matter. In particular, in Jam the authors decided to circumvent the problem by choosing to disallow the declarations of constructors in mixins, and forcing programmers to write constructors manually in all classes resulting from a mixin application. We thus believe that the adoption of ini modules could improve code modularization also within traditional mixins like the ones of Jam.

4. On hygienicity

As hinted before, Magda enforces a *hygienic* way of declaring and referencing identifiers of fields and methods. This approach modifies the way declarations of new methods, overriding of existing methods, field access and method calls are specified. In fact, in Magda the method and field names are qualified by the name of the mixin in which they have been introduced. The objective of this

hygienic approach is to avoid accidental name clashes. These clashes can cause different kinds of problems, from non-compilation to unexpected behavior during the program execution (e.g., due to accidental overridings). However, this approach, while improving inter-component compatibility, is surely the weakest point of the language seen from the usability perspective of a programmer. In fact, prefixing each method name with the mixin name may result in a code which is verbose and difficult to read and to maintain. Therefore, we thought of designing and implementing an IDE tool to deal with non-ambiguous identifiers in a smoother manner.

The IDE has been implemented by taking advantage of the pre-built language implementation infrastructure offered by the framework Xtext [14] of Eclipse, obtaining an Eclipse plugin. This framework is successfully exploited for implementing programming languages, via compilers and interpreters. Xtext is a framework for the development of programming languages as well as other domain-specific languages (DSLs), which eases this task by providing a high-level setting that generates most of the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse. The plugins generated by Xtext already implement most of the recurrent artifacts for a language IDE, and they can be easily customised by "injecting" (relying on Google-Guice) the specific language mechanisms implementations. Therefore the code that is needed for customizing the IDE is essential: there are some stubs that only need to be filled with language-dependent code to obtain the various IDE functionalities.

First of all, the IDE offers a set of standard IDE-functionalities, as:

- *compiling* and *executing* of Magda programs;
- *syntax coloring*, to give the user syntactic or semantic information through colors;
- *hyperlinking*, which makes the code navigable;
- *outline view*, which shows the user a graphical, simplified structure of code;
- *validation*, and raising of related errors and warnings;
- *quick fixes* associated with errors or warnings, which represents hints to the user to solve them.

However, as said above, the main goal of our Magda IDE is to give the programmer a way to go beyond the intrinsic verbosity of the language, by making to him/her available the possibility of working on two different views of the code:

- a standard view of Magda code, according to the syntax of the language; this view on the code is called *extended*, because in this case all the hygienic identifiers are directly visible and editable by the programmer: all field and method identifiers are prefixed with the explicit name of the mixin in which the field or method was introduced, and the programmer can directly edit them;
- a *shortened* view on Magda code, that hides the mixin names of hygienic identifiers, which are, then, not directly editable by the programmer; the shortened view of the IDE represent a restricted view of the underlying complete syntax of the language; in this way the programmer can work on a syntax which is much similar, from this point of view, to that of the most common programming languages, such as Java and C++.

The shortened view functionality had some related problems which had been addressed in the implementation. In particular, since in this view the programmer does not display directly the field or method prefix (made of mixins names), we had to solve the problem of possible ambiguous identifiers. We say that a field or method identifier is *ambiguous* when this identifier is present in more than one mixin from which the object was created.

A non-ambiguous identifier is completed automatically by the IDE by adding appropriately in the code the only candidate of mixin name, if this exists. Instead, in the presence of ambiguity, the programmer is offered two different ways to solve this ambiguity:

- by using the content-assist functionality, the programmer can choose a specific version of the ambiguous field or method, and therefore solve the problem a priori;
- by writing an ambiguous identifier, the programmer obtains an error (*"Called method or selected field is ambiguous"*) and then she will be able to solve it by using a quick fix which proposes the possible mixin names.

After choosing (a priori or by the quick fix) a mixin name and so a specific version of the method, the programmer can however change it anytime she wants, by using a quick fix associated with an info which signals to the user the presence of other versions of the field or method with that name from other mixins.

The second main feature of our Magda IDE is a support for getting right the mixin sequences in an object creation. In fact, in order to simplify the execution semantics which otherwise would be very complex, the current version of the Magda syntax imposes some constraints at the time of the creation of an object. In particular:

- all the mixins declared in the formal type of the object must appear in the object creation sequence;
- for each mixin, also its base mixins must appear (a base mixin is similar, in essence, to a superclass);
- base mixins must appear before their respective submixins.

Our IDE helps the programmer respecting these constraints by reporting their possible violations via specific errors and offering the user some related quick fixes.

In the implementation of the Magda IDE with Xtext as an Eclipse plugin, we also used two generic features of Eclipse in order to support the main feature of the IDE, that is, the shortened view on the underlying code, which hides to the programmer the mixin names of the hygienic identifiers:

- Eclipse projection. It is an Eclipse feature to support collapsing and expanding of portions of text. In particular, in the text editor of the Eclipse Java development tools (JDT), this feature of the framework has been introduced to implement the code folding functionality, which allows the user to fold individual methods and classes. The basic idea of projection is very simple: without projection, we have a document to hold the text and a corresponding view to provide the UI for the text; by introducing projection, we have a master document, which contains the

entire text, and an additional projection document attached to it. The contents of a projection document are a subset of the contents of the master document; in other words, the content of a projection document consists of portions of the master document; the UI shows the user this projection document, hiding it the other portions of text in the underlying master document. However, the Eclipse standard projection feature allows only to collapse or expand entire rows of text, therefore we had to generalize it to be able to collapse or expand portions of a row (as in the case of a mixin name in a hygienic identifier).

- Eclipse reconciler. It is an Eclipse component which periodically re-parses the source file in order to update the abstract in-memory representation (AST). In fact, as the user types in the editor, the text can become out of sync with the underlying model that is often used for semantic manipulation, such as refactoring. When a reconciler is installed on an editor, a queue is created to record all the changes that occur; when the user makes a typing break, the reconciler then removes items from the queue and updates the abstract model accordingly with the current text in the editor. In our Magda IDE, in the shortened view on the code we had the need to manage automatically the mixin names on updates or deletions by the user in the text editor: this is because the programmer cannot directly edit a mixin name prefix of a hygienic identifier in the shortened view. Therefore, we implemented an extension of the Eclipse reconciler in order to validate the hidden mixin names and remove them automatically from the underlying master document when they are no longer valid.

5. Hints on the Magda type system

Types. Magda is a statically typed language and enjoys a type soundness property guaranteeing absence of message-not-understood errors. This is formally defined and proved in [25]. What distinguishes Magda from other languages, then, is the way the type expressions are formed and the way subtyping is verified.

Every type in a Magda program is an unordered sequence of mixin names. When a variable or a field is declared of type T, it means that its value can be either `null` or an object created from a sequence of mixins which contains at least the mixins present in T (and maybe more).

For example, consider the program in Figure 4. The type of `p1` in method `MainClass.MainMatter` is `Point3D`, while the type of `p2` is `Point2D`. The second declaration means that the variable `p2` can hold either the `null` value or a reference to an object created using a 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`.

We remark that, on the one hand, *in an object creation expression* `new` the base mixins cannot be omitted in the mixin sequence because the order in which mixins are present in the sequence is significant, as shown in the Decorator example of Section 2.1. On the other hand, *in types* the removal and ordering of base mixins are, instead, insignificant. Summarizing, the order of mixins matters in `new` expressions from an operational point of view, but not in types, that is, not from a static semantics perspective.

We say that type T_2 is the *fully expanded form* of type T_1 if T_2 is the type obtained by appending to T_1 all direct and indirect base mixins of T_1.

Now consider the following mixin:

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

The requirements enforced by the type of variable `v2` are stricter than the ones enforced by the 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.

In general, we say that type T_2 is a *subtype* of type T_1 (denoted $T_2 \preceq T_1$) when the fully expanded form of T_1 is a subset of the fully expanded form of T_2. As a consequence of this definition, we define the type of the `null` value as the set of all mixin names used within a program.

Other properties. Our modular approach to initialization is motivated by the fact that we want each mixin to be as composable with other mixins as possible, in order to minimise the amount of the code to be written (or, worse, duplicated). Magda then enjoys an unusual safety property which intuitively means that no accidental conflicts can happen.

We say that an identifier `p` is a *transitive output parameter* of an ini module `m` if: (*i*) either `p` is an output parameter of `m`; (*ii*) or `p` is an output parameter of an ini module `n`, such that at least one input parameter of `n` is a transitive output parameter of `m`.

We say that two mixins `M_1` and `M_2` are *exclusive* if: (*i*) there exists a mixin `M_b` which is a direct or an indirect base mixin of both `M_1` and `M_2`; (*ii*) and there exists an input parameter `ip_b` of an ini module in `M_b`, such that both `M_1` and `M_2` contain a required ini module each, `m_1` and `m_2`, and `ip_b` is a transitive output parameter of `m_1` and `m_2`.

Intuitively, the safety property ensures that if there is no ambiguity in the runtime choice of ini modules inside a set of mixins (i.e., they are not *non-exclusive* two by two), then the mixin elements of this set can be composed in any order.

Magda enjoys another important property: Magda's modularity guarantees that the client code of a library written in Magda will never break as a consequence of any addition of members to the library's mixins (modulo exclusivity). The property is as follows: for any set of well-typed mixins and any well-typed client code which uses it, if it is possible to extend a mixin in the set with a new method, or to add a new optional ini module to it, in such a way that the mixin itself is still well typed, Magda guarantees that: (*i*) all the other mixins in the set and the client code will still be well typed (i.e., they will still compile); (*ii*) the result of the execution of the client code will not change.

This property does not extend to the case of method override, however this is unavoidable, as override may change the semantics of a method. Magda's features are such that this is the only

case in which it happens, as opposed in other languages, where also additions of members can cause unexpected changes.

6. Conclusions and future work

Magda was designed with the following goals in mind: (*i*) organising the code while programming; (*ii*) simplifying code reuse; (*iii*) maximising the ratio between the trouble of learning new concepts and the benefits afterwards.

Often mixins are compared with traits [13]. A trait is a set of methods without state and classes are formed by composing traits. Classes may define state and contain the so-called *glue code*, that are essentially setters and getters. Conflicts on names are dealt with directly by the programmer with operators such as exclusion, aliasing and renaming. We think that avoiding conflicts instead of dealing with them goes more in the direction of being able to reuse third-party code without too much trouble. Moreover, with Magda there are fewer new concepts to learn with respect to traditional class-based languages, which might appeal to someone willing to try a new language only if it provides a gradual learning curve.

Scala's traits [37] (that are indeed mixins, as they can contain state) are a powerful tool for software composition. However, as traditional mixins, they are used along classes and class-based inheritance. We believe that Magda has a semantics which is simpler and easier to understand.

Mixins (and traits) have been designed to tame multiple inheritance and managed to improve reusability in many aspects. However, they were never very successful even though some form of them are present especially in untyped, dynamic languages, such as Ruby and Javascript. There is also a recent study on how to adapt Magda-style ini modules into Smalltalk [30], towards a more effective form of stateful traits [6].

Today mixins are popular mainly for organising CSS code, in order to reuse style descriptors. Languages like Less [36] and Sass [21] are CSS extension languages that feature mixins, pre-processed away into pure CSS, similarly to macros.

The question now is: is there a future for the Magda language?

General-purpose languages are not the main subject of research at present. Groovy [35] is maybe one of the few recent general-purpose languages. It works on the Java platform and one of its most interesting features is that it is optionally typed (optional typing is a form of gradual typing [32]). It also offers a trait construct, close to the traditional mixin construct, however, no form of constructor is present in the Groovy traits. There has been also a bunch of new proposals of languages for mobile devices, like Swift [5] and Kotlin [24]. The difference here with respect to past language proposals is that these ones build on compatibility with mainstream languages, for instance the former bases on Objective-C and the latter on Java. In other words, now reuse is mostly directed towards the exploitation of the libraries and the compatibility features of other, widely used, languages, more than on language constructs.

Still, we believe that there is some space for Magda and its principles:

- Kuśmierek introduced a novel style to formalize a big-step operational semantics within Magda. The new approach arises from the observation that the typical type soundness property formulated via a big-step operational semantics is weak, that is, there are kind of faulty typing rules

that do not violate the typical big-step formulation of soundness. For instance, consider a wrong typing rule for an `if` instruction that omits to verify that the condition expression is of type `bool`. As a result, a program which uses an integer expression as condition will pass the type checking correctly. However, such a program will get stuck when reaching the point of evaluation of this condition, if the evaluation rules for the `if` instruction requires (correctly) the condition to evaluate to `true` or `false`. Nevertheless, the typical type soundness statement "If the program terminates, then it does so with a correct value" still holds even in the presence of the wrong rule. This example hints that the classical notion of type soundness is not strong enough, because properties that concern the intermediate configurations that may emerge in the course of a program evaluation are disregarded. On the other hand, the option of using a small-step operational semantics is not always an option, because it is less intuitive to build and understand. The Magda semantics formulated according to the new style can be found in [25], and this approach was further discussed in [26]. More work is needed to fully formalize which classes of languages can benefit of such a style of big-step operational semantics and for a deeper comparison with co-inductive operational semantics styles [28, 22, 2, 20, 33].

- The Magda static type system ensures strong unicity properties among labels of methods and parameters of ini-constructors (see Section 5). Such unicity properties are not suited if they are to be enforced on totally dynamic settings like CSS extension languages, however we believe that they can be adapted for a sort of soft typing, hinting at gradual typing [32] for such languages. CSS code is in fact very difficult to organise and maintain, even with the Less and Sass mixin addition, and a form of soft typing might help here.

- An innovative use of Magda would be exploiting some of it in a Domain Specific Language for composing ontologies [19]. It is possible to reuse an ontology O1 or part of it in another ontology O2 by importing O1 entirely in O2, or by mentioning a class, a property, an individual of O1 inside O2, for example by using OWL (Web Ontology Language [38]). However, the soundness of these operations is left to the ability of the OWL programmer, that must pay attention to superpositions, semantic mismatches, different choice of modelling present in the two ontologies. A precise notion of ontology composition mechanism based on mixins, with a related type system, could be of help in this setting. A good starting point to get inspiration from is the NeOn methodology framework [34], in particular the scenarios from 3 to 6, concerning reusing, merging and reengineering ontological resources.

- Last but not least, it would be still a good bet to maintain and improve Magda itself, as a good example of language design. A relatively simple addition to Magda could be exceptions. In fact, a natural way of adding checked exceptions would be adding them to ini modules; then, there would be no need to repeat the declarations in the ini modules of the derived mixins, since they work as extensions, not as replacements of the parent ini modules. Moreover, it would be good to introduce a form of ini module override. Finally, it would also be interesting to study a novel form of method selection similar to the one of ini module selections, that is, based on parameters' names.

  Other functionalities may be introduced in the IDE, in particular:

– It would be useful to introduce a support for ini modules, since in the current implementation of the language the textual ordering of ini modules declarations in a mixin can have an impact on object initialization; this is necessary in order to (a) obtain real modularity, and (b) not making too complex the underlying semantics. As a consequence, however, the programmer could be confused by this order constraint, and the IDE could therefore support him/her.
– The IDE could be expanded with other classical functionalities as wizards to create Magda projects or refactoring support, not included in the first version of the prototype.
– The current implementation of the Magda language [27] is a prototype based on Java: a preprocessor translates Magda source code into Java code. This prototype is barely adequate to test the main innovative features of the language. Therefore, it would be interesting to implement a real compiler for Magda, and this could also be done using the Xtext framework.

Acknowledgements. The story of the language called Magda started back in circa 2005, when Pawel contacted me with a manuscript written by one of his master students, Jarek Kuśmierek, describing a proposal for a yet another new object-oriented programming language. However, as Pawel put it, "there was something in it" and this "something" resulted in a collaboration among Jarek, Pawel and myself, ending successfully in Jarek's PhD thesis.

I also want to express my gratitude to the reviewers. Their observations helped me a great deal to revise the paper.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer New York, Inc., Secaucus, NJ, USA, 1996. ISBN 978-0-387-94775-4. doi: 10.1007/978-1-4419-8598-9.

[2] Mads S. Ager. From natural semantics to abstract machines. In *Logic Based Program Synthesis and Transformation*, volume 3573 of *LNCS*, pages 245–261. Springer, 2005. doi: 10.1007/11506676_16.

[3] Davide Ancona and Elena Zucca. An algebra of mixin modules. In *Proc. Workshop on Algebraic Development Techniques '97*, volume 1376 of *LNCS*, pages 92–106. Springer, 1997. doi: 10.1007/3-540-64299-4_28.

[4] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam — a smooth extension of Java with mixins. In *Proc. ECOOP '00*, volume 1850 of *LNCS*, pages 145–178. Springer, 2000. doi: 10.1007/3-540-45102-1_8.

[5] Apple Inc. Swift Website. `https://swift.org/`.

[6] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008. doi: 10.1016/j.cl.2007.05.003.

[7] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. ECOOP '99*, volume 1628 of *LNCS*, pages 43–66. Springer, 1999. doi: 10.1007/3-540-48743-3_3.

[8] Viviana Bono, Jarek Kuśmierek, and Mauro Mulatero. Magda: A new language for modularity. In *Proc. ECOOP 2012 - Object-Oriented Programming - 26th European Conference*, volume 7313 of *LNCS*, pages 560–588, 2012. doi: 10.1007/978-3-642-31057-7_25.

[9] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah, 1992.

[10] Gilad Bracha and William R. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90 Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, volume 25 of *ACM SIGPLAN Notices*, pages 303–311, 1990. doi: 10.1145/97945.97982.

[11] Kim B. Bruce. *Foundations of object-oriented languages - types and semantics*. MIT Press, 2002. ISBN 978-0-262-02523-2.

[12] Steve Cook. OOPSLA'87 panel P2 – varieties on inheritance. In *Proc. OOPSLA'87 Addendum to Proceedings*, pages 35–40. ACM Press, 1987.

[13] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006. doi: 10.1145/1119479.1119483.

[14] Eclipse Industry Working Groups. Xtext Website. `https://www.eclipse.org/Xtext/`.

[15] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda-calculus of objects and method specialization. *Nordic J. of Computing*, 1(1):3–37, 1994. Preliminary version appeared in Proc. *LICS '93*, pp. 26–38.

[16] Matthew Flatt, Shiram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Proc. POPL '98*, pages 171–183. ACM, 1998. doi: 10.1145/268946.268961.

[17] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. In *Formal Syntax and Semantics of Java*, pages 241–269. Springer, 1999. doi: 10.1007/3-540-48737-9_7.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994. ISBN 0201633612.

[19] Anna Goy and Diego Magro. What are ontologies useful for? In M. Khosrow-Pour, editor, *Encyclopedia of Information Science and Technology, Third Edition*, pages 7456–7464. IGI Global, 2015.

[20] Carl A. Gunter and Didier Rémy. A Proof-Theoretic Assesment of Runtime Type Errors. AT&T Bell Laboratories Technical Memo 11261-921230-43TM, 1993.

[21] Hampton Catlin, Natalie Weizenbaum, Chris Eppstein, and numerous contributors. Sass Website. `https://sass-lang.com/`.

[22] Husain Ibraheem and David A. Schmidt. Adapting big-step semantics to small-step style: Coinductive interpretations and "higher-order" derivations. *Electr. Notes Theor. Comput. Sci.*, 10:121, 1997. doi: 10.1016/S1571-0661(05)80692-9.

[23] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi: 10.1145/503502.503505.

[24] Jet Brains. Kotlin Website. `https://kotlinlang.org/`.

[25] Jarek Kuśmierek. *A Mixin Based Object-Oriented Calculus: True Modularity in Object-Oriented Programming*. PhD thesis, Warsaw University, Departement of Informatics, 2010. Available at `http://www.mimuw.edu.pl/~jdk/mixiny.pdf`.

[26] Jarek Kuśmierek and Viviana Bono. Big-step operational semantics revisited. *Fundam. Inform.*, 103(1-4):137–172, 2010. doi: 10.3233/FI-2010-323.

[27] Jarek Kuśmierek, Mauro Mulatero, and Marco Naddeo. The Magda language implementation. `http://sourceforge.net/projects/magdalanguage`.

[28] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *CoRR*, abs/0808.0586, 2008.

[29] David A. Moon. Object-oriented programming with flavors. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, USA, Proceedings.*, pages 1–8. ACM Press, 1986. doi: 10.1145/28697.28698.

[30] Marco Naddeo. *A Mixin Based Object-Oriented Calculus: True Modularity in Object-Oriented Programming*. PhD thesis, University of Torino and University of Lille co-tutelle, 2017. Available at `https://hal.inria.fr/tel-01651738`.

[31] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behaviour. In *Proc. ECOOP '03*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003. doi: 10.1007/978-3-540-45070-2_12.

[32] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *Proc. ECOOP 2007 - Object-Oriented Programming - 21st European Conference*, volume 4609 of *LNCS*, pages 2–27. Springer, 2007. doi: 10.1007/978-3-540-73589-2_2.

[33] Allen Stoughton. An operational semantics framework supporting the incremental construction of derivation trees. *Electr. Notes Theor. Comput. Sci.*, 10:122–133, 1997. doi: 10.1016/S1571-0661(05)80693-0.

[34] Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, and Mariano Fernández-López. The neon methodology framework: A scenario-based methodology for ontology development. *Applied Ontology*, 10(2):107–145, 2015. doi: 10.3233/AO-150145.

[35] The Apache Software Foundation. Groovy Website. `http://groovy-lang.org/`.

[36] The Core Less Team. Less Website. `http://lesscss.org/`.

[37] The Scala Group. Scala Website. `http://www.scala-lang.org/`.

[38] W3C OWL Working Group. OWL Website. `https://www.w3.org/OWL/`.