Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

The Cyan Language Metaobject Protocol

José de Oliveira Guimarães[a]

a. Universidade Federal de São Carlos at Sorocaba

Abstract

The compilation of a program can be changed by a metaprogram that acts as a compiler plugin. The process of creating such a metaprogram is called compile-time metaprogramming. The interface between the compiler and the metaprogram is described by a metaprogramming system or a Metaobject Protocol (MOP). A metaprogram can change the compilation process in several ways: it can add or remove program code, do additional checks, replace compiler algorithms, and intercept operations such as object creation, message passing, and field access. Powerful metaprogramming systems have several drawbacks. The metaprogram can have low-level interactions with the compiler, expose private source code information to other files, and introduce non-expected dependencies among language entities. The view of the program by a metacode, which is a snippet of the metaprogram, may be different from the view of other metacode. The calling metacode order, by the compiler, may have unexpected consequences. This article presents the MOP of the prototype-based object-oriented language Cyan. Although the Cyan MOP has the main functionality of other metaprogramming systems, it addresses the metaprogramming problems cited previously.

Keywords object-oriented languages, metaprogramming, metaobject, computational reflection, prototype-based languages, compilers

1 Introduction

Metaprogramming is coding of programs, called *metaprograms*, that treat code as data. The program that is treated as data is called *the base program* or simply *program*. A *metaprogram* can generate new code, change existing programs, or do checks in these programs. Metaprogramming offers mechanisms for code reuse that go beyond that offered by traditional software libraries. It can generate families of related code, as in the case of C++ templates [Str13]. It can separate functional and nonfunctional concerns, as in AspectJ [KHH+01], generate code based on specifications, as in ANTLR 4 [Par13]. It can also support new syntax (macros as in Scala [Bur13]), detect program bugs through static analyzers [Spo20] [Err20], implement new type systems using a

José de Oliveira Guimarães. *The Cyan Language Metaobject Protocol*. In Journal of Object Technology, vol. V, no. N, 2011, pages M:1–41. doi:10.5381/jot.201Y.VV.N.aN

pluggable type system [Bra04], run a program in multiple stages [Tah04], with each stage generating and running a new program, change the program at runtime [RC02] [KCJ03], and support embedded Domain Specific Languages [RAM+12] [BIS16].

In this paper, the main focus is *language support* for Compile-Time Metaprogramming (CTMP), which is handling of a program by a metaprogram at compile-time. Therefore, this text does not deal with metaprogramming that use tools like ANTLR 4 [Par13] and SpotBugs [Spo20]. Nor does it deal with metaprogramming carried out at preprocessing-time (changes made in the source code before the program is parsed), runtime, or loading time (changes in the program are made when the binaries or bytecodes of a virtual machine are loaded into computer memory).

To discuss specific characteristics of compile-time metaprogramming supported by programming languages, we need to define some terms concerning metaprogramming. A *program* is the code that implements a desired functionality for a specific application, which is also called the *base program*. A *metacode* is any piece of code that comprises a *metaprogram*. Metacode work as *compiler plugins* that exchange data with the compiler and can change compilation. The compiler calls metacode at specific points of compilation. Metacode can replace the type checker, code generator, parser, and any other algorithm used by the compiler. They can also add, delete, or replace pieces of *program*. In practice, however, languages restrict the action of metacode.

Metacode can be intertwined with *base program* or defined externally. External metacode may be linked to a source code by syntactic elements called *annotations* such as "`@property`". The compiler calls metacode during one or more compilation phases. The *protocol* specifies which part of each metacode should be called during a given compilation phase. For example, a function[1] or method `duringParsing` may be called during the compilation phase *parsing*. The function or method may do checks or add code to the program.

Languages that support CTMP face many problems caused by unforeseen interactions between the metaprogram and the compiler and conflicts among different pieces of metacode. Metaprogram need to access and change low-level compiler data structures, which is a dangerous operation. Changes to the compiler data by metacode are not recorded. Therefore, if there is a compilation error, the compiler will not be able to point out which metacode produced the invalid code. Metacode often access private information not visible to their compilation unit and different metacode may have different views of the program. Changes in the order of annotations or metacode embedded in the source code may change the compilation, making the code fragile. Checks made by one metacode may be invalidated by code added later by other metacode. Metacode associated with one source file may change another source file. As a result, the semantics of a source file may depend on every metacode in the metaprogram. Metacode may generate metacode, which may cause infinite loops: a metacode generates metacode that generates metacode and so on. Finally, there may be a cycle of information dependency among metacode. In its simplest case, a metacode depends on information produced by another one and vice-versa.

Metaprogramming can be a powerful tool, despite these pitfalls. The Cyan language [Gui20a] and its Metaobject Protocol were designed to address totally or partially each of the problems with Compile-time metaprogramming described previously. Cyan is a statically-typed, prototype-based, object-oriented language that supports Java-like interfaces, generic prototypes, optional dynamic typing, anonymous functions, non-nullable types, and an object-oriented exception system. The Cyan language

[1] Function as in language C

allows the definition of prototypes, which are the counterpart of classes of class-based languages as C++ [Str13] or Smalltalk [GR83]. The compile-time Metaobject Protocol (MOP) of Cyan specifies the relationships between the compiler, the metaprogram, and the program. *Metaobjects* from the metaprogram can add code to the program, which includes new prototypes, fields and methods to prototypes, and statements and expressions to methods. Moreover, they can intercept message passing, field access, subprototyping, and method overriding. *Metaobjects* in Cyan have limited power, they cannot delete program code or replace any compiler algorithm as type checker. Additional checks can be added to a program but no existing checks can be bypassed. The code of Cyan *metaobjects* were previously called *metacode* in this text. A *metaobject* is an object that exists at compile-time with methods called by the compiler at one or more compilation phases. An annotation in the source code, as `@property`, is associated with a metaobject.

The contribution of this paper is to show how the Cyan MOP addresses totally or partially each of the problems with Compile-time metaprogramming described previously. The characteristics of the Cyan MOP, described below, are related to the above mentioned problems.

Metaobject methods return source code, as strings, that are added to the program by the compiler. The compiler tracks down which metaobjects have added code to the program. If the generated code contain errors, the compiler will know whom to blame. Code are only added by metaobject methods that return strings. The Abstract Syntax Tree (AST) is never changed directly. An innovative algorithm guarantees that, during an important compilation phase, the order in which metacode appear in the base code is not important and that there are no circular information dependencies among metaobjects associated with the same prototype. The textual order of the annotations in the code is hardly important. After semantic analysis, metaobjects cannot change the code and, therefore, further checks will never be invalidated by other metaobjects. The compiler has a security mechanism that prevents metaobjects from accessing private information of prototypes of other source files. Code added by metaobjects in one compilation phase may contain annotations. However, these annotations will only be activated in the next compilation phase. Since number of phases is finite, the compilation process will eventually end.

The paper organization is as follows. The problems with metaprogramming, sketched in this section, are detailed in section 2. They are the motivation for this work. Section 3 is a brief introduction to the Cyan language. The Metaobject Protocol of Cyan is explained in section 4. Section 5 compares the metaprogramming systems of other languages with the Cyan MOP.

2 Motivation

Before comparing Cyan with other languages, we describe some problems with metaprogramming, mainly with CTMP, which is the main topic of this paper. Only problems relevant to various metaprogramming systems are considered here; specific drawbacks are not listed. Each problem has a name which is placed in boldface. Cyan jargon is used in the descriptions: a *prototype* is a template from which *objects* are created. A prototype belongs to a *package* as in Java. Let us assume that there is a one-to-one correspondence between prototypes and source files. A metacode *associated with* a source file is either inside the source file, as `A` in Figure 1, or it is linked to an annotation that is inside the source file, as `B` in the same figure. A metacode *associated with* a

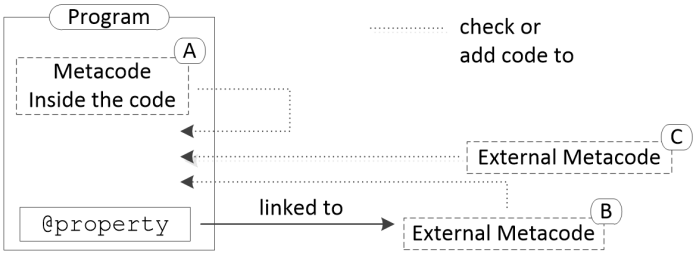

Figure 1 – The program and the metaprogram

prototype (or method) resides inside the prototype or it is linked to an annotation inside the prototype (or method).

MessWithOthers A metacode associated with a source file changes another source file, which is called *obliviousness* [CL03]. The code of a prototype presented in a text editor may not be its real code because other source files can change it. Therefore, code becomes harder to understand.

WhoDependsOnWho The compiler of an object-oriented language typically builds a *prototype dependency graph* representing the relationship between its prototypes. In the *prototype dependency graph*, vertices are prototypes and there is an edge from `R` to `S` if `S` has to be recompiled whenever `R` changes. This is the case when `S` inherits from `R` or declares a variable whose type is `R`.

Metacode have to be taken into account to build the *prototype dependency graph*. Whenever a metacode associated with prototype `S` uses information about prototype `R`, there should be an edge from `R` to `S`. This cannot be done if metacode act in the compiler data structures directly, as when an AST node is passed, as an argument, to a metacode function or method. Handling of the AST node by the metacode is unknown to the compiler and, therefore, it cannot update the *prototype dependency graph* based on it. As an example, suppose a metacode associated with prototype `S` generates a method that returns the number of public methods of prototype `R`. Prototype `S` depends on `R` but this dependency is unknown to the compiler.

KnowsFriendsSecrets A metacode associated with a prototype `S` may generate code or do checks based on private information about prototype `R` as its list of fields, its list of private methods, or even statements of its methods. Use of private information coming from other source files damages modularity because prototype `S` cannot be understood without the knowledge of private information about `R`.

Compiler-Interactions A metacode interacts with the compiler using low-level compiler data structures, like the AST, in several compilation phases. This approach has several drawbacks [SZN15]:

(a) it demands a deep knowledge of the design and implementation of the compiler, which includes details of all the compilation phases and the data structures used. The metacode may require complex AST transformations that should keep compiler invariants (often undocumented);

(b) incorrect AST handling may crash the compiler or make it generate incorrect code;

(c) metacode may bypass compiler checks causing the acceptance of flawed source code. That is, metacode may add code after the compiler does some checks that will never be done in the added code.

Moreover, metacode become tied to the compiler data structures. Changes to these data structures, like the AST classes, invalidate metaprograms.

WhoDidWhat Metacode leave no trace of their activities, as happens when they directly manipulates compiler data structures. Therefore, if a metacode generates an invalid code, detected in later compilation phases, the compiler will issue an error. But it will be unable to point out the metacode that generated the invalid code.

OrderMatters When a prototype has many associated metacode, they can be called in an order that is unclear to the metaprogrammer [PS11] or they may be called in an order that prevents them from producing correct code or doing the intended checks.

An example, cited by Palmer and Smith [PS11], considers a metacode `A` that adds to a class `X` a field for every class in the same source file. The field name is the class name in lower-case (`y` for `Y`). Initially, there is only class `X` in the file but a metacode `B` adds another class `Y`. If `A` is run before `B`, metacode `A` adds only field `x` to class `X`. If it is run after `B`, it adds fields `x` and `y`. But if the semantics of metacode `A` is "adds a field to `X` for every class in the *final* source file, after all code addition made by metacode", then metacode `A` should be the last one to run. But many languages with support to metaprogramming cannot guarantee that.

There are two subproblems of OrderMatters. One is **DifferentViews**: different metacode may have different views of the base program, as happens in the previous example, caused by metacode calling order. When one metacode adds code to the base program, other metacode can view the added code. This is a problem because the calling order may not be clear and also because a change in the metacode textual order in a source file may change the calling order. The developer does not expect that such subtle changes cause drastic code modifications.

Other subproblem of OrderMatters is **InvalidateChecks**. A metacode checks the program that is later changed by another metacode, invalidating the check. For example, metacode `A` issues a compilation error if any prototype field uses underscore in its name. Metacode `B`, run after `A`, introduces a field `color_name`. The check made by `A` is invalidated.

InfiniteMetaLoop Metacode may generate metacode added to the source code, which in turn may generate metacode and so on, creating an infinite loop. As an example, a metacode may generate itself as code, which is the equivalent of a function that just calls itself.

Nontermination

Metacode are called by the compiler. Therefore, if a metacode does not finish its computation, the compiler does not finish it either.

Nondeterminism Metacode are not limited to interact with the source code or the compiler. They can interact with the file system, the network, and other running programs. This means metacode may be nondeterministic. Two different compilations of the program with the same source code may result in two different behaviors: checks may be different and the code added by metacode to the program may differ.

NoGeneratedCodeGuarantees

Metacode may generate defective code if they are given full freedom relating to what to generate.

NoContracts

A metacode may demand specific features from the base code it is attached to and vice-versa [LE16]. For example, the metacode may demand the base prototype `T` to declare a method for comparing two `T` objects. And the base code may demand that the metacode adds to the prototype a method `sort` (that should be built with the comparison method). Ideally, there should be a *contract DSL*[2] to specify the agreements between the metacode and the base code. The contract could be enforced by the compiler. If there is no contract DSL, a metacode can check the demands it places on the base code itself. However, these demands would be more precisely described using a DSL code that is easily examined by the developer. Without a contract, the demands that the base code places on metacode are not verified.

Thus, the causes of compilation errors are more difficult to discover. Errors may appear only in the final version of the source code which is a combination of base code with the code added by metacode. To discover the errors, the developer has to examine the source file and scrutinize code generated by metacode, which exposes their implementation details.

CircularDependency To explain this problem, we use two graphs whose vertices are metacode. A *code generation* graph is a tree and there is a directed edge from `A` to `B` if metacode `A` generates code containing embedded metacode `B` or an embedded annotation linked to metacode `B`. Therefore, if there is an edge from `A` to `B`, the compiler runs metacode `A` that generates code containing `B` or a link to `B` and, then, the compiler runs metacode `B`. In a *dependency graph*, there is a directed edge from `A` to `B` if *base-program* information produced or changed by metacode `A` is used by metacode `B`. This information is *a characteristic* of the base program such as the number of prototype fields, the superprototype, or the presence or absence of a given method. The dependency graph may have cycles and this results in the problem CircularDependency.

The compiler has to choose a metacode in a cycle to be the first to be run. Suppose the first one is metacode `B`. Since this vertex is in a cycle, there will be an ingoing edge from another vertex, say `A`, to `B`. By the definition of *dependency graph*, `A` produces or changes information used by `B`. The problem is that the compiler first runs `B` generating code or doing checks based on information that will be later changed when the compiler runs metacode `A`. Therefore, metacode `B` does its job based on an outdated information. When there is a cycle, the CircularDependency problem cannot be solved by choosing any vertex to start from.

Figure 2 shows an example of a *code generation* tree, using dashed edges, and of a *dependency graph*, using solid edges. A simple cycle[3] has only two vertices as the cycle that is composed of `A` and `F` in the figure. Therefore, metacode `A` depends on information produced by `F` and vice-versa. The metacode that runs first will produce bad code because it does not have the information produced by the other. Note that the `F` metacode produces code with no embedded metacode, as there is no dashed arrow

[2]Domain Specific Language

[3]A vertex that has an edge to itself is a cycle. There will only be a problem when the metacode is not correctly implemented because, otherwise the metacode would consider the consequences of the code it produces on itself. Thus, we consider that the simplest cycle having this problem has two vertices.

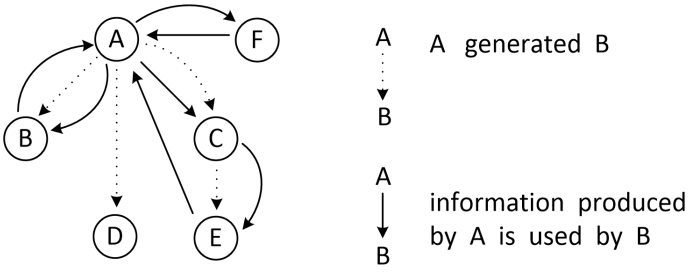

Figure 2 – Two graphs: one showing metacode generation and the other showing which metacode uses information produced or changed by others

coming out. The CircularDependency problem is an extended version of OrderMatters in which there is no adequate order of annotations.

Other Problems

There are also some other problems with metaprogramming that this paper will not further discuss. One of them is the unintended capture of identifiers by the code generated by metacode [KFFD86]. This happens when the generated code uses identifiers, already in use, in the environment where the code is inserted and their semantics are equaled by accident. This problem is solved by renaming identifiers. When the metacode is a Lisp-like macro, the solution is called *macro hygiene*. The second problem occurs when there are no direct links between annotations in the base program and external metacode. Thus, the developer may not know which metacode is associated with her or his base code. The third problem happens when metaprogramming is very powerful. In this case, metacode can profoundly alter the base program in unforeseen ways, making it difficult to understand. For example, metacode may remove methods and statements of the program.

3 The Cyan Language

Cyan is a statically-typed prototype-based object-oriented language. A *prototype* is a template from which other objects may be created, which is the same role played by *classes* in Java [GJS+14], C++ [Str13], C♯ [Csh20], and Smalltalk [GR83]. The difference is that, in Cyan, the prototype itself is an object like any other if it declares a constructor without parameters. The compiler translates Cyan to non-legible Java code. Thus, many language constructs are directly translated into Java, as inheritance, method overriding, message passing, assignment, and prototype declaration (each prototype is translated to a Java class). The two languages interoperate: Cyan code can import Java packages and classes and vice-versa.

3.1 Types and Prototypes

Listing 1 shows the declaration of prototype `Student` of package `university`. A *package* in Cyan is a named set of prototypes and has no important conceptual differences from Java packages. A *compilation unit* is a single source file composed of *import declarations*, with the imported packages for this file, and a single prototype. The file name is composed of the prototype name and extension "`cyan`". Fields that can change their values are declared with keyword `var`, as `number` in the example. The type

Listing 1 – Prototype `Student`

```
package university

object Student
    let String name
    var Int number

    func init: String name, Int number {
        self.name = name;
        self.number = number;
    }

    func getName -> String = name;

    func getNumber -> Int { return number }
    func setNumber: Int number { self.number = number }

end
```

precedes the field name. Read-only fields are declared with `let`, as `name`. If neither `var` nor `let` is used, `let` is assumed. Fields are always private to their prototypes. Inheritance is done with the *Cyan keyword* `extends`. A prototype that does not explicitly inherit from any other prototype inherits from `Any`, the superprototype of every other prototype but `Nil`. There is no sub or superprototype of `Nil`.

3.2 Methods and Message Passings

Cyan employs a syntax for method declaration and message passing in some way similar to Smalltalk. Methods are declared using the *Cyan keyword* `func`. A *unary method* is a parameterless method like `getName` and `getNumber` from the example. Its name does not end with "`:`". The return type is given after "`->`". If missing, the return type is considered to be `Nil` and it is optional to return a value in the method. The method body is a list of statements between `{` and `}` or an expression after "`=`". See methods `getNumber` and `getName`. A method is public unless one of the following Cyan keywords are used before `func`: `private`, `package`, or `protected`.

A *non-unary method*, called a *keyword method*, has one or more *method keywords* or just *keywords*. Each *keyword* is composed of an identifier ending with "`:`" followed by zero or more parameter declarations. Method `setNumber:` of prototype `Student` is a *keyword method* with one *keyword*, `setNumber:`. There may be more than one *keyword*, each one with zero or more parameters. A *unary message passing* is composed of a *receiver* and an identifier that should be the name of a *unary method*:

```
aStudent getName
```

`aStudent` is the *message receiver*. A *keyword message passing* is composed of a *receiver* and one or more *message keywords* or just *keywords* with their parameters:

```
var Box t = Box new; // creates an object
   // t is the message receiver
t get println; // the same as  (t get) println
t add: "xyZ#8Z"  at: 5, 7  doc: "Password for NotSecretAnymore";
```

Both *method keywords* and *message keywords* are called *keywords*. To avoid confusion, *Cyan keyword* is used for reserved words of the language.

`self` is a pseudo-variable that refers, inside a method, to the object that received the message that caused the method execution; the same concept as Smalltalk's `self` and `this` of C++/Java/C♯. *Constructors* are methods with names `init` (no parameters) or `init:` (with parameters). They are used to create objects of a prototype. The compiler adds to the prototype a `new:` method for each `init:` method, with the same parameters. And a `new` method for a `init` method. The return type of the `new` or `new:` method is the prototype in which it is defined. Thus, for `Student` the compiler creates method

```
func new: String name, Int number -> Student { ... }
```

A `Student` object is created as

```
Student new: "Newton", 1
```

or just `Student("Newton", 1)`.

3.3 Interfaces

There are Java-like *interfaces* declared with the Cyan keyword `interface` instead of "`object`". The method bodies should be omitted and a prototype may *implement* an interface using keyword `implements`. It then should define all methods inherited from the interface. By convention, interface names in Cyan start with the letter `I` like `IMachine`. Interfaces are prototypes whose method bodies are not declared explicitly. The compiler supplies the method bodies, which are statements that throw exceptions.

3.4 Generic Prototypes

Package `cyan.lang`, automatically imported by every *compilation unit*, declares generic prototypes `Array` and `Tuple`:

```
var Array<Int> primes = [ 2, 3, 5 ];
var Tuple<String, Int> nameAge = [. "Newton", 85 .];
```

`Array<Int>` is a *generic prototype instantiation*: the compiler creates a new prototype for each different set of arguments to the generic prototype. This example also shows a literal array and a literal tuple. A generic prototype with a varying number of generic parameters has just one parameter followed by `+`, as shown in Listing 2. This is the real code of prototype `Tuple`. Parameter `T` cannot be used inside the prototype using the Cyan syntax. But the metaobject associated with annotation `createTuple` generates code using the real arguments.

3.5 Other Features

Unlike dynamically-typed prototype-based languages such as Self [US87], there is no runtime structural reflection in Cyan. Thus, methods and fields cannot be added to

Listing 2 – The generic prototype with varying number of parameters `Tuple`

```
package cyan.lang
@createTuple
object Tuple<T+>
end
```

Listing 3 – Prototype `Person` that uses metaobject annotations

```
1  package human
2
3  @init(name)
4  object Person
5      @property var String name
6      func test {
7          let Array<String> list = @compilationInfo("field list");
8          list println;
9      }
10 end
```

a prototype at runtime, inheritance cannot be changed, and so on. Cyan is similar to statically-typed class-based languages and it is also similar to the statically-typed prototype-based language Omega [Bla94]. The preferred way of creating an object in Cyan is using method `new`, not method `clone`. There are other features of the Cyan language that are not presented in this section: anonymous functions, the exception handling system (made only with message passing), safe object initialization (fields are initialized before being used, except in a few circumstances), and a generalization of anonymous functions called *context objects*.

4 The Cyan Metaobject Protocol

The Cyan Metaobject Protocol (MOP) describes the *interactions* between the Cyan code being compiled, the compiler, the MOP library, the metaprogram, and annotations in the Cyan code that tells the compiler which metacode should be called during the compilation. The metaprogram in Cyan is composed of Java classes, Cyan prototypes, or a combination of both. The compiler is implemented in Java making it convenient to use Java classes as the metaprogram. But since the compiler translates each Cyan prototype into a Java class, Cyan can also be used as the metaprogramming language.

The following subsection shows a complete example using the Cyan Metaobject Protocol. It stablishes the terminology and explains how the MOP works. Subsection 4.2 shows all Cyan interfaces that can be implemented by metaobjects to implement their desired functionality. Subsection 4.3 explains how the Cyan MOP addresses the problems of section 2. Some shortcomings of MOP are presented in subsection 4.4.

4.1 A Complete Example Explained

An *annotation* or *metaobject annotation* is the syntax element that links the program to a metacode. Listing 3 shows a prototype `Person` that uses three annotations: `property`, `init`, and `compilationInfo`, each one preceded by "`@`". Annotation `compilationInfo` takes a string as parameter and `init` takes as parameter an identifier that is, for practical purposes, also a string. `property` is attached to the declaration of *field* `name` and `init` is attached to prototype `Person`. `init` creates a constructor with field `name`, `property` creates `get` and `set` methods for `name`, and `compilationInfo` generates a literal array with the prototype fields.

Basic values (`3`, `3.14`, `'A'`), literal arrays, literal tuples, literal maps, and any

Listing 4 – Prototype `Student`

```
// this is a comment
// the delimiters for 'doc' are {*  and  *}
// the delimiters for 'replaceCallBy' are {:<  and  >:}

@doc{* returns the double of the argument *}
@replaceCallBy(once){:<  2*n  >:}
func twice: Int n -> Int = n + n;
```

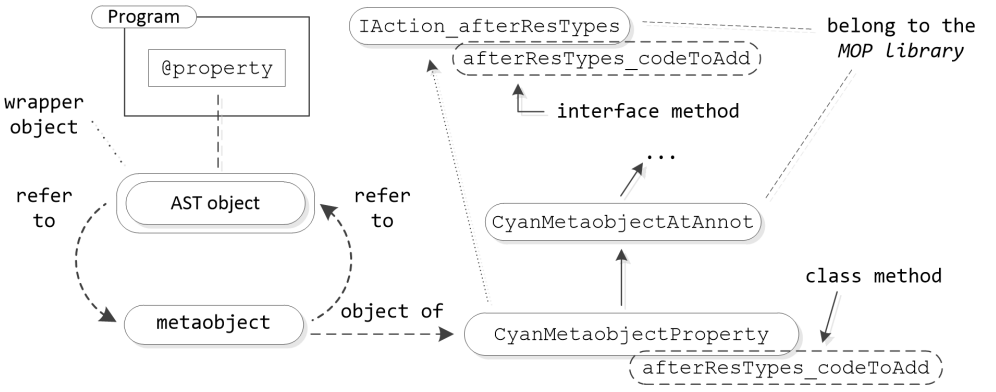

Figure 3 – Relation between metaobjects, annotations, metaobject classes, MOP interfaces, and compilation phases

combination of these can be parameters to annotations. An annotation may be followed by a text given between two delimiters, as shown in Listing 4. This text will be called *attached text* or *attached DSL*[4] *code*. Its semantics is defined by the metaobject, it works like a further string parameter to the annotation. In this example, the *attached DSL code* of `doc` and `replaceCallBy` are documentation in English and an expression, respectively. There are many possible variations of delimiters, described by Guimarães [Gui20b], so that the DSL code and the delimiters do not clash. As a rule, the right delimiter should mirror the left one. Annotations `replaceCallBy` and `doc` are attached to method "`twice:`". `replaceCallBy` takes one parameter.

A *metaobject* is an object of a Cyan prototype or a Java class and the Cyan compiler is able to work with these two languages. A Cyan *package* contains one or more prototypes that are kept in a directory with the same name as the package. A special subdirectory of the package may contain the compiled version of a Java metaobject class or a Cyan metaobject prototype — a "`.class`" file. Whenever the package is imported, the annotations associated with the package's metaobject classes or prototypes can be used in the source file. Therefore, Cyan packages hold two kinds of *resources*: prototypes and metaobjects. Every metaobject class or prototype has a `getName` method that returns the annotation name. This links the annotation in the source code, found during parsing, to the metaobject class or prototype. All the metaobjects used in Listing 3 could have been implemented by a regular developer. However, they belong to the package `cyan.lang` which is imported automatically by

[4]Domain Specific Language

every Cyan source file.

When parsing source code, the Cyan compiler creates, for each annotation, three objects: an object of the AST representing the annotation, a wrapped object of the AST object, and a *metaobject*. The former is used only by the compiler. Metaobjects have only access to wrapped, read-only, and restricted versions of AST objects. The wrappers are also called "AST objects". A *metaobject* is an object of a Cyan prototype or Java class that inherits from prototype or class `CyanMetaobjectAtAnnot`.[5]

A metaobject refers to its associated wrapper AST object and vice-versa. Since the wrapper object represents an annotation in the source code, metaobject methods, through the AST object, have access to information such as annotation parameters, attached DSL code, and the annotation environment (AST objects of the compilation unit, prototype, and method to which the annotation belongs).

Figure 3 shows the relationships, after parsing, among all of the elements related to annotation `property` of Listing 3. The rectangle with round border labelled "AST object" on the left is the *compiler* AST object of the annotation. It is wrapped by another rectangle representing the wrapped AST object used by the MOP. This figure shows that the AST object and the metaobject refer to each other. `CyanMetaobjectProperty` is the Java class of metaobject `property`. It inherits from `CyanMetaobjectAtAnnot` and implements interface `IAction_afterResTypes`, overriding the interface method `afterResTypes_codeToAdd`. This is the method that creates the get and set methods for field `name`. `CyanMetaobjectAtAnnot` extends class `CyanMetaobject`, not shown in the figure. Appendix A shows the complete code, in Cyan, of the prototype of a metaobject called `myproperty`. This is a simplified version of `property`.

We use "*metaobject* `property`" when no confusion arises. If there are two `property` annotations in a code, "*metaobject* `property`" becomes ambiguous because it may refer to metaobjects associated with both annotations. In Listing 3, there is only one annotation for each metaobject and therefore there is no confusion.

MOP library is a name used for two packages: one in Cyan and the other in Java. The *MOP library* in Cyan (Java) contains prototypes (classes) imported by the compiler and used for building metaobject prototypes (classes).

As shown in Figure 3, `CyanMetaobjectAtAnnot` and `IAction_afterResTypes` belong to the Cyan *MOP library*. There are also classes in the Java *MOP library* with the same names. If a metaobject is implemented in Java, there is a Java class for it and therefore the Java package representing the *MOP library* is used. The same concept applies to Cyan. The Cyan *MOP library* mirrors Java *MOP library*. The compiler is capable of interacting with the two libraries.

The Cyan compiler goes through six compilation phases for each source file. The phases are shown inside the rectangle with dashed lines on the left of Figure 4. The flow of control is from top to bottom. Phase **parsing** does the syntactical analysis and builds the Abstract Syntax Tree (AST) of the source file. Some AST objects are associated with a type and have a `type` field, initially set to `null`. For example, some AST classes that declare a `type` field are classes that represent method parameters, prototype fields, implemented interfaces, superprototype, message passings, and expressions.

There are two kinds of AST objects associated with types: those inside method bodies, representing expressions, and those outside method bodies. The `type` field of the latter AST objects is set in phase **resTypes** (resolving types). Thus, field `name`

[5]There is a Cyan prototype and a Java class with this same name. Suffix "AtAnnot" means "annotation that starts with an @". Some annotations do not use this syntax, but they are not described in this paper.

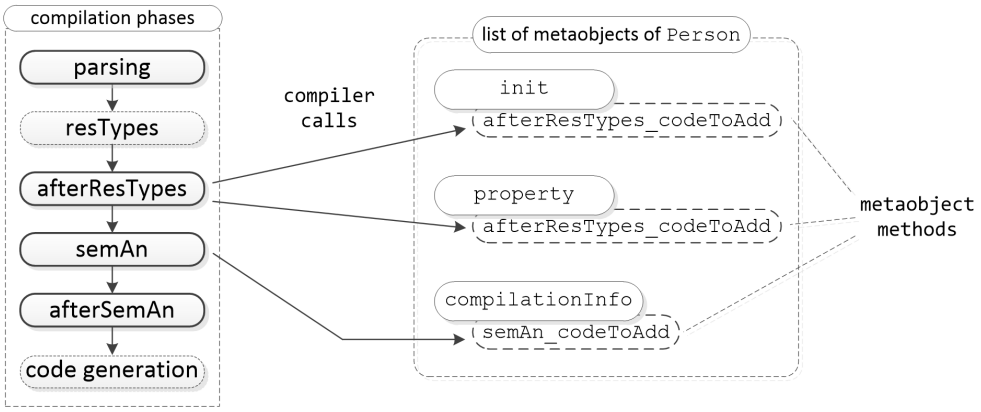

Figure 4 – The compilation phases and their links to methods of metaobjects at compile-time

of `Person` of Listing 3 is represented by an AST object whose field `type` is `null` at the beginning of phase resTypes. During this phase, the compiler sets the `type` field to the AST object representing the prototype `String`. Phase resTypes, therefore, is part of the *semantic analysis* of the source code. The compiler goes through phase resTypes on a source file only after parsing all source files referenced in this file or loading the jar file with the referenced prototypes.

Phase **afterResTypes** means *after resolving types*, which is used only by the Cyan MOP. Some methods of metaobjects are called in this phase. For example, the method of metaobject `property` that adds the get and set methods to the prototype is called in phase afterResTypes. AST objects that represent expressions in the Cyan code have a `type` field set in phase **semAn** (semantic analysis), which is the remainder of the *semantic analysis*. In this phase, the compiler also does further checks required by the language. Phase **afterSemAn** means *after semantic analysis*, which is used only by the MOP. Some metaobject methods are called in this phase. Currently, no metaobject method is called in the last compilation phase, code generation.

The non-dashed rectangles of Figure 4 represent the phases of parsing, afterResTypes, semAn, and afterSemAn associated with interfaces of the *MOP library*. A phase is associated with multiple interfaces but each interface is associated with just one phase. The interface name ends with the phase it is associated with as `IAction_semAn`.

During the parsing of prototype `Person` of Listing 3, the compiler creates a metaobject for each annotation. Then, in each of the phases of parsing, afterResTypes, semAn, and afterSemAn, for all metaobjects, the compiler calls all of the metaobject methods declared in interfaces of that phase. In the `Person` example, the dashed rectangle on the right of Figure 4 shows a list of metaobjects created for this prototype. There are three metaobjects represented by rectangles with the annotation name (`init`, `property`, and `compilationInfo`). In the compilation phase afterResTypes, shown on the left, the compiler calls methods `afterResTypes_codeToAdd` of metaobjects `init` and `property`. Method `afterResTypes_codeToAdd` is declared in interface `IAction_afterResTypes`. In the same way, the compiler calls method `semAn_codeToAdd` in phase semAn.

There is a point that has been left out: at which point during each compilation phase does the compiler call each method? It depends on the interface the method is

declared. Some interfaces are associated with *triggers*. For example, methods of

```
IActionMethodMissing_semAn
```

are called whenever the compiler is not able to find a method that matches a message passing. The "missing method" error triggers calling of the interface methods. The compiler calls methods of interface `IAction_afterResTypes` in phase afterResTypes regardless of any trigger. We may also consider that the trigger is simply the processing of the prototype in this phase.

Metaobjects always generate code as strings. The code is added to a copy of the prototype source code in memory — the original file is not changed. In Listing 3, metaobject `property` generates methods "`getName`" and "`setName:`". Metaobject `init` generates code for a method "`init:`", a *constructor*, setting field `name`. The compiler inserts the code generated by the two metaobjects in the `Person` prototype that goes through the parsing and resTypes phases again. Phase afterResTypes is skipped, and the compilation proceeds to phase semAn. Then, metaobject `compilationInfo` generates a string whose contents is

```
[ "name" ]
```

This is a literal string array with one element: the name of the `Person` field. The compiler inserts the code produced just after the annotation.

After code is inserted in phase semAn, the whole source code is compiled again. But phase afterResTypes is skipped and the metaobjects that act in phase semAn are not used. A planned optimization of the compiler is to compile just the inserted code in phases afterResTypes and semAn. Note that the compiler does not allow an infinite loop of metaobjects producing annotations that produce annotations, and so on. The number of compilation phases is fixed even when metaobjects generate code. For example, if `property` generated a `@property` annotation, it would not be used because the compiler skips phase afterResTypes in the recompilation. And this annotation only acts in this phase.

4.2 The Interfaces of the MOP Library

This subsection explains how metaobjects direct the compilation of Cyan code. That is, how the compiler chooses methods of metaobject classes/prototypes to be called at specific phases of the compilation. Although metaobjects can be implemented in either Java or Cyan, this subsection assumes they are implemented in Cyan. Therefore, the *MOP library* used is composed of *prototypes*, including `CyanMetaobjectAtAnnot`.

In the following text, we will use *current prototype* for the prototype inside which the annotation resides. Therefore, for all of the metaobjects associated with the annotations of Listing 3, the *current prototype* is `Person`. The *current compilation unit* is the compilation unit of the *current prototype*.

Base methods are methods of the *current prototype*, which will be called just *methods*. Methods of the metaobject prototypes or methods of the interfaces of the *MOP library* will be called *metamethods* or just *methods* if no confusion arises.

The design of a metaobject prototype starts by defining its goals. Then, the programmer chooses one or more interfaces to satisfy these goals. If the metaobject should add fields and methods to the *current prototype*, it should implement interface `IAction_afterResTypes`. If it should intercept access to a prototype field, the metaobject prototype should implement interface `IActionFieldAccess_semAn`. And so on. This is very important: the functionality of a metaobject prototype is driven by the interfaces it implements. In some other languages, a metacode decides what to

do during its execution, which is at compile-time of the base program. This is more prone to errors since what to do depends on runtime decisions.

The metaobject prototype should override the interfaces' methods. Every metaobject prototype inherits a method that returns the annotation AST object, the right arrow labeled "refer to" in Figure 3. This object holds information on the annotation parameters, attached DSL code, and the declaration it is attached to. The compiler calls every metaobject method passing as one of the arguments an object with information on the environment of the annotations, including the AST objects of current prototype and current method. The available information varies according to the compilation phase in which the metaobject method was declared. During parsing, metaobjects do not know method statements, other prototypes, or any information on code that comes textually after the annotation. In phase resTypes, metaobjects have access to AST objects that describe everything outside method statements. In phases semAn and afterSemAn, metaobjects have access to all information on the current prototype.

The following subsections describe the interfaces of the *MOP library* that can be implemented by metaobject prototypes. Some interfaces described by Guimarães [Gui20b] are missing because they are either marked as deprecated or are irrelevant to the paper conclusions.

4.2.1 Interfaces of Phase parsing

The interfaces of this phase are used for parsing the *attached DSL code* of an annotation, for generating code after the annotation, and for passing information, like documentation, from the annotations to declarations.

4.2.2 Interfaces of Phase afterResTypes

Interface `IAction_afterResTypes` declares four methods. One is used for adding statements at the beginning of methods of the *current prototype*. Another method is used for renaming methods.[6] The two remaining methods of interface `IAction_afterResTypes`, `afterResTypes_codeToAdd:` and `runUntilFixedPoint`, work together and are explained after some definitions.

A *method signature* is the method declaration without its body, but including keyword `func`. Parameter names are optional. The *signature of a field* is composed of its declaration, preceded by `var` or `let`, without the optional expression assigned to it. Method `afterResTypes_codeToAdd:` returns a tuple composed of two strings: the code of *base fields and methods* (to be added to the *current prototype*) and the *signatures* of these *base fields and methods* (separated by "`;`").

```
func afterResTypes_codeToAdd: ICompiler_afterResTypes compiler,
        Array<
           Tuple<WrAnnotation,
                 Array<ISlotSignature>>> infoList
        -> Tuple<String, String>
```

The first parameter, `compiler`, is a restricted version of the compiler object. It has methods to return the *current prototype*, *the current compilation unit*, methods and fields of the current prototype, and so on. The second parameter, `infoList`, is an

[6] Whenever a method is renamed, the metaobjects should add another *base method* with a name equal to the old *base method* name. This is to prevent the difficult-to-understand compilation error "method was not found" when the developer clearly view the method in the source code editor.

array of tuples, each one composed of an annotation and an array of *base method and field signatures*.[7] The Cyan compiler may call method `afterResTypes_codeToAdd:` of each metaobject multiple times. The following example shows that a single call may be sufficient.

Let's assume a metaobject `addFieldInfo` adds to the current prototype a field whose name is the first annotation parameter, initialized with the number of prototype fields. In a naive implementation, if there are two annotations `addFieldInfo` attached to a prototype with no declared fields, each metaobject will add to the current prototype a field initialized with `1` instead of the correct value `2`. This is because one metaobject does not view the code added by the other one. To work correctly, the metaobject class needs to be changed. Before showing how to do that, we need to understand the algorithm FixMeta used by the compiler in phase afterResTypes.

This algorithm works in *rounds* of method calls using all the metaobjects sharing the same current prototype. Calls to method `afterResTypes_codeToAdd:` of all prototype metaobjects form a *round*. In the first *round*, the second argument for each method call is an empty array — no signatures have been created yet. In all other rounds, the second argument is an array of all *base method and field* signatures generated in the previous round. Therefore, each metaobject can adjust its own code generation because it knows the base method and field signatures generated by other metaobjects in the previous round. The *rounds* end either when all metaobjects produce the same code as in the previous round or when the number of rounds is greater than five.[8] In the last case, the compiler issues an error because the metaobjects were not able to reach an agreement in `5` rounds. If a metaobject defines a method `runUntilFixedPoint` that returns `false`, the compiler only calls its method `afterResTypes_codeToAdd:` in the first round. This is the correct implementation if the code generation does not depend on the code produced by other metaobjects.

Let us return to the `addFieldInfo` example. To work correctly, the metaobject class should define a method `runUntilFixedPoint` that returns `true` and a method `afterResTypes_codeToAdd:` that takes into consideration the code generated by other metaobjects. When these changes are carried out, the metaobjects generate incorrect code in the first round (the fields are initialized with `1`, as before). In the second round, each metaobject knows the code generated by the others. This information is given by the second method argument, `infoList`. Therefore, each metaobject generates correct code (fields initialized with `2`). In the third round, there is no change in the code generated and FixMeta ends.

4.2.3 Interfaces of Phase semAn

The interfaces of phase semAn are:

`IAction_semAn`, for generating code after the annotation
`IActionMessageSend_semAn`, for intercepting message passings
`IActionMethodMissing_semAn`, for intercepting the error "method not found"
`IActionFieldAccess_semAn`, for intercepting get and set of fields
`IActionFieldMissing_semAn`, for intercepting the error "field not found"

The sole method of interface `IAction_semAn` returns a string containing code to be added after the annotation. If only used for checkings, the method should return an empty string. Some annotations produce expressions, like `compilationInfo` of Listing 3. Interface `IActionMessageSend_semAn` is used for intercepting message

[7] A *slot* is either a method or a field, represented by interface `ISlotSignature`.
[8] This number can be changed by a compiler option.

passings. The associated annotations should be attached to base methods. This interface is useful for intercepting message passings when the compiler finds an adequate base method. The metaobject associated with the annotation, which is attached to a base method, may check the message arguments, at compile-time, and replace the message passing by another expression.

For every message passing, the compiler collects the base methods that match it considering the compile-time type of the *message passing receiver*. If this type is `T`, the compiler initially collects the base methods in `T`. Then, the compiler collects the metaobjects associated with these *base methods* implementing interface `IActionMessageSend_semAn`. The *metamethods* of these metaobjects are called. There are three possibilities based on the number of metaobject methods returning a non-empty code string:

1. more than one. The compiler issues an error because there is an ambiguity here;
2. exactly one. The returned code replaces the message passing. This replacement is visible in the next compilation phase, afterSemAn;
3. none. The same search for a base method is done in the superprototypes and superinterfaces of `T`, in this order.

The items above lead us to the conclusion that the order of the annotations of a prototype is not important for interface `IActionMessageSend_semAn`. The metaobject class of `replaceCallBy`, shown in Listing 4, implements this interface.

The Cyan MOP offers a mechanism for introducing *virtual* methods in prototypes; that is, methods that do not exist but whose existence is simulated by metaobjects. Whenever a method for a message passing is not found, a metaobject can replace the message passing by an expression. As an example, a metaobject could simulate the existence of a large number of `get` methods that return the values of virtual fields. The field values could be created on-demand or retrieved from a database.

When the compiler analyzes a message passing in phase semAn, first of all it looks for the type of the receiver expression. Let us suppose it is `T`. If there is no adequate method for the message passing, the compiler collects, into a list, all metaobjects that implement interface `IActionMethodMissing_semAn` and whose annotations reside in prototype `T`. This interface declares two metamethods; one for unary and the other for keyword messages. Then, the compiler proceeds as it does in the case of method

`IActionMessageSend_semAn`

except that implemented interfaces are not taken into account.

Metaobjects whose prototypes implement interface `IActionFieldAccess_semAn` intercept field access. A method of this interface is called when the value of the field is retrieved and another one is called when the field is set. Each metamethod returns code that replaces the operation of getting and setting a field. The annotation should be attached to the prototype field whose access is intercepted. If more than one metaobject tries to replace a field access, the compiler issues an error.

Prototypes may have *virtual* fields that are used as regular fields. This is achieved by metaobjects whose prototypes implement interface `IActionFieldMissing_semAn`. Annotations of these metaobjects should be attached to prototypes. If multiple metaobjects are entitled to handle a *missing field* event, only one of them should return a non-empty code string. Otherwise, the compiler issues an error message.

In phase semAn, the Cyan compiler resolve types in a method's body in the textual order of statement declarations. The metaobject associated with an annotation has

access to the types resolved in lines that come before the annotation. This information can be used for checking or code generation.

There is a metamethod that belongs to the MOP but is not associated with an interface: `replaceStatementByCode`. This metamethod is declared in the superprototype of all metaobject prototypes, `CyanMetaobjectAtAnnot`. Therefore, it can be called inside any metaobject metamethod. As the name implies, `replaceStatementByCode` replaces a statement, which includes expressions, by a code. The statement and the code are given as an AST object and a string, respectively. If called in any compilation phase other than semAn, this metamethod issues an error.

We claim that a metaobject `mo` cannot have a reference to a statement that is inside a prototype `Q` different from its current prototype `P`. A variable of `mo` may refer to the AST object representing a base method of `Q` but metamethod `getStatementList` of this AST object checks if `Q` is the current prototype of the caller, `mo` in this case. Since `P` and not `Q` is the current prototype of `mo`, `getStatementList` throws an exception, proving the claim. Hence, `replaceStatementByCode` can only be used for replacing statements of the *current prototype*.

Metaobjects have access to Abstract Syntax Tree (AST) objects from parameters passed to overridden interface metamethods and from the associated annotations. By calling methods of AST objects, metaobjects have access to information on the current prototype, method, etc. AST objects can be visited using the *Visitor* Design Pattern [GHJV95]. This pattern and metamethod `replaceStatementByCode` are used in the demonstrative metaobject `shout` that belongs to the Cyan libraries. This metaobject visits the AST objects of the current method and replaces all literal strings by the equivalent ones in upper case.

4.2.4 Interfaces of Phase afterSemAn

Checks made in any phase other than afterSemAn can be invalidated by metaobjects that change the code. Therefore, all checks should be made in phase afterSemAn because, from this phase onwards, code cannot be changed. There are four interfaces in phase afterSemAn:

`ICheckSubprototype_afterSemAn`, for checking subprototypes
`ICheckOverride_afterSemAn`, for checking overridden methods
`ICheckDeclaration_afterSemAn`, for checking declarations
`ICheckMessageSend_afterSemAn`, for checking message passings

The sole method of interface `ICheckSubprototype_afterSemAn` is called when the current prototype is inherited or implemented. The AST object of the subprototype is passed as parameter. Through which, the metaobject can do checks. For example, the metaobject may require that an interface be implemented only by prototypes that also inherit from another class, as done by a feature of language Hack [Hac20]. A metaobject annotation whose prototype implements interface `ICheckOverride_afterSemAn` can only be attached to a base method. Whenever the base method is overridden, even in a sub-subprototype, the compiler calls the metaobject metamethod that overrides the only metamethod of this interface. The compiler passes, as an argument to this metamethod, the AST object of the subprototype base method. Annotations associated with metaobjects whose prototypes implement interface `ICheckDeclaration_afterSemAn` should be attached to a *declaration*, which is a prototype, method, field, or local variable. Interface `ICheckMessageSend_afterSemAn` is used for message passing checkings. The corresponding annotations should be at-

tached to the base methods they intend to check. For each message passing, the compiler calls all metaobject methods that match the message, including those whose current prototype is in superprototypes.

4.3 The Cyan MOP and the Problems with Metaprogramming

This subsection shows how the Cyan MOP deals with the metaprogramming problems described in section 2. The problem name is in **boldface** and a short description is in *italics*.

MessWithOthers *A metacode in a file changes another source file.*

A metaobject whose prototype implements interfaces `IActionMessageSend_semAn` or `IActionMethodMissing_semAn` causes non-local changes. That is, a metaobject whose annotation is in prototype `P` may replace a message passing that is in prototype `Q`. The message passing is replaced to obey the semantics of the associated `P` method or the virtual method. The replacement of the message passing in `Q` by a metaobject of `P` is *expected*. The problem with MessWithOthers are the *unexpected* changes that the developer cannot discover. Therefore, we consider that these two interfaces do not cause this problem.

A metaobject that does not implement the interfaces cited in the previous paragraph can only replace or add code to its *current prototype*. This is assured by several mechanisms:

(a) the AST is read-only. Therefore, even if a metaobject has a reference to the AST object representing a prototype that is in another source file, the metaobject cannot change this object;

(b) there is no method in any interface to add code to an external prototype;

(c) method `replaceStatementByCode`, described in Subsection 4.2.3, asks the compiler to replace a statement by a code given as a string. The statement is an AST object. This method can only change the *current prototype* because there is no way of a metaobject, whose current prototype is `P`, has a reference to an AST object representing a statement that is in another prototype `Q`. This happens because some methods of the AST classes have security checks that prevent access to private parts of other prototypes. An exception is thrown if the access is illegal.

WhoDependsOnWho *Metacode are not taken into account when the compiler builds the dependency graph among source files.*

The Cyan MOP addresses this problem because the first statement of all relevant methods of the *MOP library* is a call to a method that builds the dependency graph. That is, if a metaobject calls a method `mm` of the MOP library and this action introduces a dependency among source files, the first statement of `mm` will add this dependency in a compiler *dependency table*. Classes of the AST are included in the MOP library.

KnowsFriendsSecrets *Metacode in one source file know private information of another file.*

Metaobjects whose annotations reside in a prototype have the same program view as this prototype. This ensures that a metaobject whose annotation is inside a prototype does not know the secrets of other prototypes. This is enforced by two techniques:

(a) methods of the AST return more information or less information according to the caller. The amount of information varies to match the *current prototype* view of the program. For example, class `WrProgramUnit` of the AST represents a prototype and declares a method `getMethodDecList` returning the list of methods of the prototype. Suppose a metaobject whose annotation is in prototype `P` sends message `getMethodDecList` to an AST object representing prototype `Q`. This method takes an argument that is a compilation environment. Through it, the method can identify prototype `P`. `getMethodDecList` returns a list of Cyan methods that includes the `Q` public methods and: (a) the `package`[9] methods of `Q` if `P` and `Q` are in the same package; (b) the `protected`[10] methods of `Q` if `P` is a subprototype of `Q`.

(b) methods of the AST throw exceptions if the metaobject is trying to retrieve private information of other prototypes; that is, a metaobject whose annotation is in `P` tries to retrieve private information of `Q`.

Compiler-Interactions *Metacode interact with compiler low-level structures.*
We consider that this problem is addressed in Cyan for several reasons:

(a) metaobjects use *simplified* and high-level wrapped versions of the compiler data structures. The wrapped AST classes mirror the language features they represent. That means they are not highly subject to change. They are modified only when the language change;

(b) the wrapped data structures are read-only. There is no way of crashing the compiler by calling the wrong methods;

(c) metaobjects do not add code by handling the AST (calling its methods or changing fields). Therefore, metaobjects cannot bypass a compiler check by adding code after the compiler does that check.

WhoDidWhat *The compiler does not link an inserted code to the metacode that made the insertion.*

The compiler keeps track of the annotation associated with the metaobject that asked for replacement or addition of code. If there is an error in the source code replaced or added by a metaobject, the compiler can point out the line and the source file of the annotation associated with the metaobject.

OrderMatters *The order in which metacode are called inside a source file changes metacode behavior.*

For each prototype, the Cyan compiler processes the metaobjects in the textual order of their annotations. To explain that, the term *metaobject metaprototype* will refer to the prototype of the metaobject (it is in the metaprogram). In each compilation phase, for each prototype and for each metamethod `mm` of each interface `IN` of the *MOP library* of that phase, the Cyan compiler calls metamethod `mm` of every metaobject of the *current prototype*. The calling order is the *textual order* of the metaobject annotations in the source code of the *current prototype*. Assume that the *metaobject metaprototype* implements interface `IN`. If the order of metamethod calls

[9] A method preceded by the Cyan keyword `package`. It is visible in all package prototypes.
[10] Methods preceded by the Cyan keyword `protected`, visible in all subprototypes.

is not important, the annotation order in the source code is also irrelevant. In the following paragraphs, we will examine all interfaces and their methods to discover if the order of calls is important or not.

The calling order of metaobject methods in phase parsing is not important for two reasons: (a) metaobjects can add code which will be visible by other metaobjects only in the next phase and (b) metaobjects can add information to declarations (such as documentation) but this data cannot be read in phase parsing. A method of interface `IAction_afterResTypes` is used for renaming methods. The order of calls is not important because the compiler issues an error if two metaobjects try to rename the same base method. Another metamethod of this interface adds statements to the beginning of base methods of the current prototype. There may be two or more metaobjects that try to add statements to the same base method. In this case, the textual order is *important*, the statements are added in the textual order of the metaobject annotations. If necessary, a metaobject may demand to be the only one to add statements to a given method.

Algorithm FixMeta of subsection 4.2.2 calls method `afterResTypes_codeToAdd:` in rounds. In each round, every method can view the information produced by all of the calls of the previous round. Thus, the call order is not important.

In phase semAn, metaobjects can only add code, in the *current prototype*, after the annotation. The code added by other metaobjects in this phase will only be visible in the next phase, afterSemAn. Thus, as far as code generation is concerned, the calling order of the methods is not important. Metaobjects can replace a message passing or field access by some other code. The calling order is not important because only one metaobject can replace the message passing or field access. If two or more try to do a replacement, the compiler issues an error. Methods of interfaces of phase afterSemAn do checks in an immutable program. Since they cannot add code, the calling order of their methods is irrelevant.

Subproblem **DifferentViews** only happens in phase semAn. In this phase, a metaobject knows the types of all expressions that come before its annotation in a method body. Hence, if there are two annotations in the same method, the one that comes textually afterwards has more information than the first. The information available to metaobjects, in phase parsing, cannot be changed by them and, therefore, all metaobjects have the same program view. In phase afterResTypes, all methods view the AST built in the previous compilation phase, resTypes. Methods of metaobjects that participate in algorithm FixMeta view also what other methods have produced in each round of the algorithm. Therefore, all of these metaobject share the same program view. Some metaobjects choose not to participate in this algorithm because the code produced by other metaobjects is unimportant for them. In phase afterSemAn, all metaobjets view the AST produced in the previous phase. Therefore, all of them have the same program view.

The subproblem **InvalidateChecks** of OrderMatters happens only if a metaobject does checks in compilation phases different from afterSemAn. This means that the metaobject is poorly designed, which is not a flaw of the Cyan MOP. Checkings should be done in phase afterSemAn when code is in its final form.

InfiniteMetaLoop *Metacode can generate metacode that, in its turn, generate metacode, and so on.*

The Cyan MOP prevents this error by enforcing drastic rules: annotations added to the base program, by metaobjects, in a compilation phase are only active in the

next phase.

Nontermination *Metacode may not finish its computation.*

In Cyan, metaobject methods may not finish their computations. An easy and costly solution to this problem does exist: metacode would be called in a new thread and given a time limit for execution. Each metaobject would supply to the compiler a maximum execution time, subject to a limit given as a compiler option.

Nondeterminism *Metacode is nondeterministic.*

Metaobjects are regular Cyan objects which can interact with external libraries. Therefore they can be nondeterministic.

NoGeneratedCodeGuarantees *Metacode may generate defective code.*

Metaobjects in Cyan can produce defective code.

NoContracts *The contract between the metacode and the base code is explicitly stated.*

This kind of contract is not supported by the Cyan MOP. Note that NoContracts is similar to the problem that motivated the creation of *concepts* [GJS+06] for C++ templates (See subsection B.3). *Concepts* are predicates on generic prototype parameters. They restrict what a parameter can be, like "parameter `T` should have a unary method `init`".

A solution to problem NoContracts would be to add a concept-like DSL to specify: (a) the restriction a metaobject expects from the *current prototype* and (b) the code a metaobject should generate. This DSL will certainly be more complex than concept DSLs because the diversity of code generation and checking of metaobjects is much greater than that of generic prototypes.

CircularDependency *Metacode may depend on information produced or changed by other metacode. This dependency relation may be circular.*

Metaobjects cannot access any information produced by other metaobjects in phase parsing, preventing this problem from occurring. In phase afterResTypes, algorithm FixMeta of subsection 4.2.2 deals with circular dependencies. This algorithm ensures that all metaobjects that participate in the algorithm have the same information on the current prototype. However, FixMeta is not useful for some unusual metaobjects. For example, suppose a metaobject generates a field for each prototype method and another metaobject generates a method for each field. This results in an endless loop and, in Cyan, there is no fix for that.

FixMeta only considers information on the current prototype. There may be a cyclic dependency among metaobjects associated with annotations of different prototypes. For example, an annotation associated with prototype `A` depends on information about prototype `B` and an annotation associated with `B` depends on `A`. This dependency may be unsolvable or it may be possible to extend algorithm FixMeta for its solution. The latter was deemed too complex and was not added to the Cyan MOP. There is no *circular dependency* in phase semAn because changes caused by metaobjects are only visible in the next compilation phase. In phase afterSemAn, all metaobjects view the same code and, therefore, there cannot be any *circular dependency*.

4.4 Shortcomings of the Cyan MOP

A metaobject may generate code containing annotations. But the annotations' associated metaobjects will only be activated in the next compilation phase. Therefore,

neither in phase afterResTypes nor in semAn, a metaobject can use other metaobjects for generating code at this same phase. To exemplify this limitation, let us suppose a metaobject `propertyAll` takes pairs (name, Type) as arguments and generates fields with that name and type and get and set methods for them. This metaobject cannot generate

```
@property var Type name
```

for each pair, in phase afterResTypes, because metaobject `property` would be used only in the next phase, semAn (when it does nothing). However, the generation of get and set methods can be put in a library and imported by both metaobjects. This is how metaobjects can be composed.

Some MOP features are missing in Cyan, such as intercepting compiler error messages and code generation. These features are planned to be added to the language soon. Currently, there is no interface for adding code to a subprototype whenever the *current prototype* is inherited. Although this would be an intrusive feature, it may be added to the Cyan MOP. Metaobjects, in phase semAn, cannot view the code generated by others, which could prevent the building of some metaobjects. However, we analyzed metacode of all the languages cited in this paper and we could not find any metacode that needs this feature.

5 Comparison with Related Work

This section presents some metaprogramming systems and how they are related to Cyan. The first subsection describes mechanisms for code generation, the benefits and drawbacks of each one. Subsections 5.2 and 5.3 compare Cyan with runtime metaprogramming and static analysis tools, respectively. Languages and systems supporting compile-time metaprogramming are presented in subsection 5.4. They are analyzed, in relation to the problems of section 2, in subsection 5.5. The last subsection presents some problems with the Cyan MOP.

5.1 How Code is Generated and Represented

Metaprograms generate code in many representations using several mechanisms [SBF15], described next.

`As text.` Code is generated in string format. Metaobject `mypropery` of Appendix A exemplifies how this works. There is no guarantee that the generated code is error-free. This is the mechanism used by Cyan which will be compared to the following ones.

`Handling of the program Abstract Syntax Tree.` Code is generated by creating AST objects that represent it, if the compiler is implemented in an object-oriented language. The developer has to know a large number of classes (more than one hundred in Cyan). Code generation is difficult because it demands the mapping, by the metaprogrammer, of human legible source code into AST objects. AST handling has the advantage that the metaprogram compiler usually catches all syntactic errors of the generated code. The remainding errors, if any, are caught by the base compiler. If the metacode inserts the generated objects directly into the AST, the base compiler will not be able to point out the metacode that generated the offending code. In Cyan, metaobjects generate code as strings that are inserted in the source code with markings that reveal the origin of the inserted code. If the compiler discovers any

errors in this final code, it points out exactly which annotation is associated with the metaobject that produced the offending code.

`Quoting.` A special language syntax transforms text into AST objects. Therefore, the metaprogram handles text that is converted into AST objects. The quoting mechanism will be presented using examples in Converge [Tra08], a Python-based language. A *quasi-quoted* expression `[| code |]` builds the AST of `code`. Inside `code`, there may appear annotation `${ code2 }` meaning that `code2` is to be inserted into `code`. The AST produced by a *quasi-quote* can be either evaluated at compile-time or used to produce code inserted in the base program.

In Cyan, quasi-quote-like substitution is done with string handling, which is much simpler for two reasons. First, because it uses operations known by every programmer (string handling). Second, there is no confusion between metacode with base code, the base code is wrapped in strings and it is not an AST object. The downside of the Cyan approach is that any checkings are delayed until the code produced by the metaobject is compiled. Code within quasi-quotes is checked at compile-time, although usually only for syntactical errors.

`Macros.` Macros in high-level languages were first introduced in Lisp [Har63]. In this language, a macro is a function called at compile-time[11] to produce code that replaces the macro call. This is the definition of "macro" used in this paper. Currently, there are more sophisticated versions such as that of languages Nemerle [SMO05], Rust [KN18], and Scala [Bur13]. Macros are used for local changes only, a macro call is replaced by code. They are not capable of other types of code modifications and checks allowed by Cyan: add fields and methods to prototypes, intercept several operations, and check the final prototype code.

`Generic classes, functions, and prototypes.` This encompasses C++ class templates [Str13] in which a new class is created for each new instantiation of the class. That is, for each new set of class parameters, a new class is created. This is also what Cyan does with generic prototypes. This mechanism is different from generic classes of languages such as Java in which all generic instantiations share the same binary class code. The C++ template mechanism offers a compile-time Turing-complete functional language for template generation [Vel03]. In Cyan, metaobjects can be used for generating base code for generic prototypes as in prototype `Tuple` of Listing 2. The most powerful of such metaobjects is `insertCode` of Appendix B. This metaobject takes an interpreted Cyan code as an attached DSL text, interprets it, and adds the code it produces to the current prototype.

`Specialized languages.` Domain Specific Languages are used for generating code. AspectJ [KHH+01] [Asp20] is a Java extension for *Aspect-Oriented Programming* (AOP) [KLM+97]. In this paradigm, code for an *aspect* of a program, like error handling and logging, is grouped together and put in just one place instead of being scattered around in the program. In AspectJ, several operations can be intercepted like method calls, field access, and creation of objects. This is specified through an *aspect language*, a DSL, resembling Java. The AspectJ compiler, directed by user-code, can add methods, fields, and constructors to classes and change inheritance and implemented interfaces.

Genoupe [DLW05b] [DLW05a] is a C♯ extension whose generic classes can make use of a language for creating new classes (existing classes cannot be changed). There

[11] At compile-time means "when the code is compiled", which may be at runtime of a previously existing program. That is, a macro may be created at runtime and called using the Lisp function `eval`.

are a `foreach` and `if` statements used for generating code. Although Genoupe offers a high degree of type safety at compile-time, this language does not guarantee that the generated code is well-typed. *Generators* written in SafeGen [HZS05], a metalanguage for Java, produce only well-formed Java code. SafeGen uses a theorem prover fed with first-order logical sentences representing properties of the generated code. If the prover cannot assure that the generated code is well-formed Java code, an error is issued. SafeGen statements `#foreach` and `#when` are used for repetition and decision, much like the equivalent statements of Genoupe. CTR [FCL06] extends C♯ with *transformers* which are constructs combining patterns and generation templates. Whenever a *transformer* matches a code, like a class, the generation template is applied. It can, for example, add a method to the class or create new classes. The well-formedness of the generated code is checked both by CTR and the compiler. In Cyan, just the compiler checks the generated code.

Generic classes in MorphJ [HS11] specify how to build other classes based on the fields and methods of their type parameters. This technique is called *morphing*. The classes instantiated from the same generic class may have different structures. MorphJ generic classes are checked without the knowledge of their real parameters. Hence, malformed code is detected at an early stage. The language offers positive and negative patterns for code generation. *Trait functions* in the model MTJ [RT07] take parameters and are composed of `requires` and `provides` clauses. A *trait function* contained in a class is called when real arguments are supplied. Then, the fields and methods of the `provides` clause are added to the class. The `requires` clause imposes constraints on the real arguments. The calling of a *trait function* works similarly to an annotation in Cyan whose metaobject adds fields and methods to the current prototype. PTFJ [MS12] extends MTJ with patterns borrowed from MorphJ. Miao and Siek [MS14] extend PTFJ introducing pattern-based code generation at the statement level. That is, method statements can be generated based on pattern matching. For example, a statement is generated only if a class has a given method. cJ [HZS07] is a Java extension with predicates on the type parameters of generic classes. A predicate works as a *static if* for code generation. For example, a method is added to the generic class only if the parameter `X` is a subclass of class `Y`. The type-checking of a generic class is modular. It can be made before any instantiations.

Cyan has none of the safety guarantees of Genoupe, SafeGen, CTR, MorphJ, or MTJ. Metaobjects can generate code with not only type errors but also with lexical and syntactical errors. However, the creation of new classes in these languages can be emulated in Cyan using generic prototypes. Metaobjects have access to the parameters of a generic prototype and can use them to generate code as in prototype `Tuple` of Listing 2. Some safety guarantees would result from the use of metaobject `concept` of subsection B.3. This metaobject can be used to check if the arguments to a generic prototype obey predicates, thus preventing future compilation errors.

`Other Mechanisms.` MetaFJig* [SZ13] allows the combination of classes by a set of composition operators to support *active libraries*. A customized version of a class is created by composing other classes and by calling methods that return classes. Since a class may have nested classes, a customized version of a library can be created. MetaFJig* ensures that errors are never caused by already compiled metacode. The MOP of Cyan has the power to generate prototypes at compile-time. Thus, it has the power of creating customized libraries of prototypes. However, there are neither static guarantees nor a DSL to help with this job.

5.2 Runtime Metaprogramming

Iguana/J [RC02] supports dynamic adaptation of behavior of classes and objects through *protocols*. The operations that can be intercepted are object creation and deletion, method call, method dispatch, method execution, and field access. Reflex [TNCC03] is a Java extension that also supports behavioral reflection by modification of classes at loading time. Unlike Cyan, Iguana/J and Reflex support only runtime metaprogramming and they do not support structural changes like the addition of methods to classes.

The Smalltalk MOP is fundamentally different from that of Cyan because it cannot change the program. However, a Smalltalk program can change itself at runtime using methods inherited from fundamental classes such as `Behavior` which are outside of the MOP. In Python 3 [Ram15], metaclasses are used to change classes, including adding code to them. Each class has a single metaclass, a limitation that drastically reduces the complexity of metaprogramming in Python and, at the same time, limits its usefulness. A metaclass does not have access to the AST of its class and it cannot intercept class inheritance and method override in a subclass. There are no compile-time guarantees with respect to metaclasses because classes are created only at runtime.

5.3 Static Analysis Tools

Static analysis tools, such as Spotbugs [Spo20] for Java and PMD [PMD20] for multiple languages, work by traversing the AST of a program. They use rules and patterns to detect performance problems, errors, vulnerabilities, code style, and code quality issues. The functionality of static analysis tools that depend on the AST of a single source file can be implemented by metaobjects in Cyan that act in phase afterSemAn. However, the Cyan MOP does not support any mechanism for metaobjects associated with annotations of different source files to share information. That would be unsafe since the order of compilation of source files is not fixed.

STLLint [GS06] is a static checker for C++ software libraries. It considers the semantics of the library instead of the semantics of the language. STLLint can detect that, in a method call, a wrong parameter will cause a runtime error. An example of error detection, detected by STLLint, is an attempt to dereference a past-the-end iterator. A metaobject whose prototype implements interface `ICheckMessageSend_afterSemAn` intercepts message passings and can check its arguments. The metaobject has access to the AST of the method with the message passing even when the metaobject annotation is in a different source file. Therefore, the Cyan MOP can do some of the checkings of STLLint.

5.4 Compile-Time Metaprogramming

The prime example of a Metaobject Protocol is that of CLOS [KdRB91] [KAR+93] [Pae93] [BGW93] [DG87], an extension of Common Lisp [Ste90] with features for object-oriented programming. The CLOS MOP acts at runtime, allowing the interception of several operations: object creation, allocation of memory, calculus of superclass precedence,[12] method calls, field access, and many more. The MOP of this language

[12]The superclasses have to be ordered because the language supports multiple inheritance.

uses *metaclasses* which are the classes of classes and methods,[13] which are objects too. By using a user-made metaclass for a class we change its expected behavior. For example, a metaclass can introduce a field into a class that keeps how many objects were created. The method that creates instances of the class may increment this field every time it is called.

OpenC++ [Chi95] is a C++ extension in which metaclasses for classes and methods are given the opportunity of changing the AST after parsing. The metaclass of a class `C` may intercept method calls whose receivers have type `C`. The method call may, after the interception, be changed or replaced. The MOP of OpenC++ also allows interception of variable declarations, creation of objects, and reading and writing of fields. OJ [TCIK00] [Tat99] is a Java extension in which a class may be associated with a user-defined metaclass. Methods of the metaclass have the opportunity of changing the AST. For example, a method called `translateDefinition` of a metaclass may add methods to the class. `expandFieldRead` can change reading of a class field. The user-defined metaclass can also define methods for intercepting object creation, array allocation, writing to fields, method calls, and casts to the class.

Languages Xtend [Xte20], Groovy [KGK+07], and Nemerle [Nem18] [Ska05] support *compile-time metaprogramming* without a Cyan-like Metaobject Protocol. We will say that these languages support *metaprogramming features*. They share many similar characteristics, described below, and therefore will be considered together.

(a) Annotations are attached to classes, methods, and other declarations.

(b) An annotation is linked to a *Processor Class* (PC) that can implement interfaces and define methods that change the compilation.

(c) Methods of the PC are invoked during several *compilation moments* which occur before, during, and after the traditional compilation phases; e.g. before, during, and after parsing.

(d) Methods of the *Processor Class* have parameters that represent language elements that can be changed at compile-time. For example, the AST object of the annotated class or method is passed as an argument. Methods of the PC can, using these AST objects, add methods to an annotated class, change inheritance, add statements to an annotated method, change method statements, and so on. Any AST object reachable from the method arguments can be changed. Therefore, a method can be added to a class that is not annotated or directly related to the annotated class. The class may, for example, be just the type of a parameter of a method accessible to the PC method.

(e) A method of the *Processor Class* that overrides an interface method is used in the compilation phase associated with that interface (much like Cyan). However, there is no order among the classes or the annotations of a class. Consequently, the view of a class by methods of a PC is not well-defined.

A *compiler plugin* is composed of metacode called in *hooks* of a language's compiler. These plugins change the compilation process and, therefore, add compile-time metaprogramming features to the language. The difference, in usage, between the terms *compiler plugin* and *metaprogramming* is the emphasis in the implementation aspects of the former and conceptual aspects of the latter. Languages Scala [ST20], Java

[13] CLOS have both *methods* and *generic methods*. To our goals, it is not necessary to distinguish them.

[Ora19], X10 [NS07], Kotlin [Kot20], TypeScript [typ20], and Rust [Rus20] support *compiler plugins*. Java annotation processors [Dar06] are compiler plugins that allow checkings but not code modifications. They are used, for example, for implementing pluggable types [che18]. Project Lombok [Kim10] is a Java annotation processor whose supported annotations can add code to classes because it uses non-supported downcasts. Compiler plugins will not be discussed in depth in this paper because there is a shortage of good documentation about them. However, languages whose compilers accept plugins have all of the main characteristics of languages supporting *compile-time metaprogramming* without a Metaobject Protocol, discussed above.

BSJ [PS11] (Backstage Java) supports metaprogramming without a Cyan-like MOP. Like Xtend, Groovy, and Nemerle, the AST is handled directly. Unlike these languages, BSJ was created to prevent some common problems with metaprogramming. Therefore,

1. the language prohibits non-local changes. A metacode associated with a class can only change the class. A metacode inside a method can only change the method;

2. the compiler detects conflicts between different parts of the metaprogram, like two metacodes trying to add code to the start of a method. Depending on the order of insertion, the results would be different;

3. there is a mechanism to give the order of execution of the metacodes. The compiler creates a dependency graph based on directives `#target` and `#depends` of metacodes. The metacode of a target is executed before its dependents and it can view the changes made to the AST by them. This is complex because a metacode can create metacode itself.

5.5 Metaprogramming Systems and their Problems

This subsection discusses the metaprogramming problems of section 2 that occur in languages with metaprogramming powers similar to Cyan. Occasionally, other languages may be cited too. Languages with more limited capabilities have, due to their lack of power, fewer problems:

(a) if the language generates code using class patterns, it cannot have any of these problems. Usually, a language uses patterns and some other forms of code generation. Therefore, it may have some of these problems;

(b) languages supporting a single metaclass for each class cannot have the WhoDidWhat, OrderMatters, InfiniteMetaLoop, and CircularDependency problems because they only exist if there is more than one metacode acting on the same code;

(c) runtime metaprogramming cannot have the problems associated with compilation like WhoDependsOnWho, Compiler-Interactions, and CircularDependency;

(d) some languages allow the interception of operations like object creation and message passing but not the addition of code. Problems MessWithOthers and OrderMatters cannot occur with them.

The problem name is in **boldface** and a short description of which is in *italics*.

MessWithOthers *A metacode in a file changes another source file.*

Languages OJ, Xtend, Groovy, and Nemerle allow non-local changes by AST handling. CLOS, OpenC++, and BSJ limit the changes to the scope of the metaclass or metacode. In Cyan, a metaobject can change a prototype different from the current prototype only if the change is expected. Hence, the language addresses this problem.

In AspectJ, cross-cutting concerns of a program are coded in one or more *aspect language* source files. Therefore, these files may change several other files. This is the *expected* behavior because, by definition, some program features are grouped into aspect language files. Conversely, annotations of languages supporting metaprogramming are inside regular source files. If they are allowed to change other source files, the developer may not be aware which files will be changed. And which annotations of other files will change a given source file. Unlike AOP, which uses a static *aspect language*, the metacode associated with an annotation decides the source file it will change at runtime (*runtime* for the metaobject, *compile-time* for the base program). The source files changed could even vary from compilation to compilation. We conclude that non-local changes, made by metacode, are justified for AOP but not for metaprogramming with annotations.

WhoDependsOnWho *Metacode are not taken into account when the compiler builds the dependency graph of source files.*

In OpenC++, metacode associated with a class does not have any information about other classes. In all other languages with metaprogramming features, the dependencies caused by metacode are not computed by the compiler. Cyan stores the dependencies in a table and, therefore, addresses this problem.

KnowsFriendsSecrets *Metacode in one source file is aware of private information of another file.*

We are unaware of any language, other than Cyan, that: (a) supplies AST objects to metacode and (b) uses security checks for preventing a prototype from accessing private information of another prototype. The latter needs an explanation. Private source code information is only represented using AST objects. A metaobject `mo` whose current prototype is `P` may have a reference to an AST object `obj` of another prototype `Q`. If `mo` calls a method of `obj` for retrieving private information about `Q`, this method will throw an exception. Every AST method that return private information verifies if the caller has permission to call it.

Compiler-Interactions *Metacode interact with compiler low-level structures.*

Compiler plugins and languages with *metaprogramming features* strongly depend on internal compiler details and, therefore, they have all of the Compiler-Interactions problems. The OJ MOP permits direct changes in the AST although it supplies a simplified version of the AST classes to the metacode. Cyan addresses this problem since its metaobjects view read-only and simplified compiler data structures.

WhoDidWhat *The compiler does not link an inserted code to the metacode that performed the insertion.*

Cyan keeps track of which metaobjects change the base code. Converge [Tra08] tracks down who produced which code to issue precise error messages. It goes beyond

Cyan in two aspects: (a) every bytecode[14] knows its origin, which can be used in runtime error messages, and (b) an AST node can be associated with more than one location (an error may be associated with more than one source). To our knowledge, no other language solves problem WhoDidWhat.

OrderMatters *The order in which metacode are called inside a source file changes metacode behavior.*

This problem occurs in all languages that allow direct handling of the AST because a metacode views the changes made by metacode executed before. If the metacode call order is changed, the view changes too. Usually, there is no way of specifying that, after a certain compilation phase, the AST is read-only. Therefore, checkings made by a metacode may be invalidated by code added by other metacode. In AspectJ, a keyword may declare the execution order of the metacode. In BSJ, metacode may declare its dependencies. A metacode with clause `#depends label` is only executed after a metacode with a `#target label` clause. The latter metacode can view changes made by the previous one. Cyan addresses this problem except in two cases: (a) code addition at the start of base methods (the addition is made in annotation order) and (b) in phase semAn, metaobjects whose annotations come later view a more detailed AST of the statements that come before (types are resolved).

InfiniteMetaLoop *Metacode can generate metacode that, in its turn, generate metacode, and so on.*

Any sufficient powerful metaprogramming system, such as CLOS and OpenC++, faces this problem. If metacode can generate metacode to be processed in the same compilation phase, then infinite loops may arise. Cyan addresses this problem because: (a) annotations inside code that was generated by metaobjects are only active in the next compilation phase and (b) algorithm FixMeta of subsection 4.2.2 always finishes its execution.

Nontermination *Metacode may not finish its computation.*

SafeGen, MorphJ, and Meta-traits ensure termination of code generation [SZ13]. In general, the termination of code generation is guaranteed only if the generated code is composed of code patterns or the metacode is limited to a few kinds of statements. Cyan does not address this problem.

Nondeterminism *Metacode is nondeterministic.*

Every metaprogramming system that allows the use of external code is nondeterministic because this code can, for example, access files. Therefore, only very limited systems, such as C++ templates, are deterministic. Genoupe [DLW05b] uses memoization to evaluate expressions at compile-time in a class generator. Thus, two identical expressions always return the same value, even if they return a random number. However, this does not prevent nondeterminism because a class generator may call code that returns a different value in each compilation, even with the same arguments to the class.

NoGeneratedCodeGuarantees *Metacode may generate defective code.*

Only a few languages offer a high degree of safety for code that is generated at compile-time: Genoupe, SafeGen, CTR, and MorphJ. They generate code based in patterns. DynJava [OMY01] is a Java extension that supports quasi-quotes with

[14]Source code is translated into bytecodes of the Converge VM.

information on the context in which they can be used. The context includes name of base class, local variables, fields and methods, and so on. These typed quasi-quotes and rules of the language assure that code produced at runtime is type-safe. Cyan offers no guarantee in relation to the generated code.

NoContracts *The contract between the metacode and the base code is explicitly stated.*

SafeGen arguments to metacode may be restricted by predicates. For example, a metacode can accept only non-abstract classes as arguments. The pattern in a *transformer* of CTR limits the classes it can match. Therefore, the pattern works as a contract between meta and base code. The `requires` clause of a trait function of model MTJ imposes constraints on real arguments and the `provides` clause supplies the code that will be added to a class. MTJ has the best solution to the NoContracts problem. Metaobject `concept` of subsection B.3 can specify a contract between metacode and the base code. Since its use is optional, Cyan does not enforce this contract.

CircularDependency *Metacode may depend on information produced or changed by other metacode. This dependency relation may be circular.*

Circular dependency occurs in compiler plugins and language with metaprogramming features. In BSJ, the execution order of metacode may be specified. This does not solve the problem because there may be no correct order of execution — remember example with metaobject `addFieldInfo` in subsubsection 4.2.2. Cyan addresses this problem in all but one case: metaobjects associated with several prototypes in phase afterResTypes. A solution to this case would demand the extension of algorithm FixMeta of subsection 4.2.2 to several prototypes, a complex solution for a not-too-common problem.

6 Conclusion

The Cyan MOP combines a full MOP, like that of CLOS, with metaprogramming features of recent languages such as Groovy and BSJ. It addresses completely or partially the metaprogramming problems MessWithOthers, WhoDependsOnWho, KnowsFriendsSecrets, Compiler-Interactions, WhoDidWhat, OrderMatters, InfiniteMetaLoop, and CircularDependency. The Cyan MOP fails in Nontermination, Nondeterminism, NoGeneratedCodeGuarantees, and NoContracts. These problems are not addressed by any metaprogramming system that use an irrestrict metalanguage. The more freedom for generating code, the more difficult it is to solve these problems.

The design of a metaobject class or prototype in Cyan starts by choosing the interfaces it should implement. The interfaces are chosen to match the goals of the metaobject. Therefore, the metaprogrammer, guided by these goals, make the most important decisions *before* starting to code. In each compilation phase, metaobject methods *ask* the compiler to add code. As a result, the metaprogram acts passively in relation to the compiler, which is in control of the execution flow of the metaprogram. This architecture makes it relatively easy to build metacode compared with other metaprogramming systems with the same power. In other systems, the decisions are taken at metacode runtime with the help of the original compiler data structures.

The Cyan MOP supports six kinds of metaobject annotations. Only the most important of them were described in this paper. Other kinds of annotation are: (a) literal numbers ended by an identifier (like `101bin` or `0AH2_Hex`), (b) literal strings starting with an identifier (like `xml"<s>XML code</s>"`), (c) macros (each start with an identifier after which any syntax is allowed), (d) annotations to types that

implement pluggable types [Bra04] [che18] [PAC+08] (like `String@regex("a*[A-Z]")` or `Char@letter`), and (e) Codegs (code + eggs), visual metaobjects that demand a plugin to an IDE[15] (an annotation `@color(red)` allows one to choose a color using a menu, during editing time). The metaobject classes or prototypes of all kinds of metaobject can implement most interfaces of subsection 4.2. For example, the metaobject of a number annotation, like `101bin`, could add fields and methods to the current prototype (but it does not). It certainly generates number `5` as code in phase semAn.

There are several future works for the Cyan MOP. One is to allow metaobjects to change the original source files, whenever necessary. Other future work is to support variable ownership like in language Rust [KN18]. The Cyan compiler is available for download at `cyan-lang.org`. In this site, one can find the language manual, a complete description of the Cyan MOP, and a list of approximately one hundred metaobjects with examples.

References

[Asp20] The aspectj language, 2020. URL: `https://www.eclipse.org/aspectj/doc/next/progguide/language.html`.

[BGW93] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. Object-oriented programming. chapter CLOS in Context: The Shape of the Design Space, pages 29–61. MIT Press, Cambridge, MA, USA, 1993.

[BIS16] Aggelos Biboudis, Pablo Inostroza, and Tijs van der Storm. Recaf: Java dialects as libraries. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, page 2–13, New York, NY, USA, 2016. Association for Computing Machinery.

[Bla94] G. Blaschek. *Object-oriented programming with prototypes*. Monographs in Theoretical Computer Science - An Eatcs Series. Springer-Verlag, 1994.

[Bra04] Gilad Bracha. Pluggable type systems. In *In OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[Bur13] Eugene Burmako. Scala macros: Let our powers combine! on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2489837.2489840`.

[che18] The checker framework manual: Custom pluggable types for java, 7 2018. URL: `https://checkerframework.org/manual/checker-framework-manual.pdf`.

[Chi95] Shigeru Chiba. A metaobject protocol for c++. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 285–299, New York, NY, USA, 1995. ACM. `doi:10.1145/217838.217868`.

[CL03] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report 329, Computer Science Technical Reports - Iowa State University, 2003.

[15] Integrated Development Environment

[Csh20] C# language specification, September 2020. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction.

[Dar06] Joe Darcy. Java specification request 269: Pluggable annotation processing api, 2006. URL: http://jcp.org/en/jsr/detail?id=269.

[DG87] Linda G. DeMichiel and Richard P. Gabriel. The common lisp object system: An overview. In *European Conference on Object-oriented Programming on ECOOP '87*, pages 151–170, Berlin, Heidelberg, 1987. Springer-Verlag.

[DLW05a] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Generative programming for c#. *ACM SIGPLAN Notices*, 40(8):29–33, 8 2005. doi:10.1145/1089851.1089857.

[DLW05b] Dirk Draheim, Christof Lutteroth, and Gerald Weber. A type system for reflective program generators. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, GPCE'05, page 327–341, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11561347_22.

[Err20] Error prone, feb 2020. URL: https://github.com/google/error-prone.

[FCL06] Manuel Fähndrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, page 275–284, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1173706.1173748.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJS+06] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in c++. *SIGPLAN Not.*, 41(10):291–310, October 2006. doi:10.1145/1167515.1167499.

[GJS+14] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[GS06] Douglas Gregor and Sibylle Schupp. Stllint: Lifting static checking from languages to libraries. *Softw. Pract. Exper.*, 36(3):225–254, March 2006.

[Gui20a] José de Oliveira Guimarães. The cyan language. 2020. URL: http://cyan-lang.org/articles/.

[Gui20b] José de Oliveira Guimarães. The cyan language metaobject protocol. 2020. URL: http://cyan-lang.org/articles/.

[Hac20] Hack. The hack programming language, May 2020. URL: http://hacklang.org/.

[Har63] Timothy P. Hart. Macro definitions for lisp. Technical report, Cambridge, MA, USA, 1963.

[HS11] Shan Shan Huang and Yannis Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst.*, 33(2), February 2011. doi:10.1145/1890028.1890029.

[HZS05] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with safegen. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, GPCE'05, page 309–326, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11561347_21.

[HZS07] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Cj: Enhancing java with safe type conditions. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, AOSD '07, page 185–198, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1218563.1218584.

[KAR+93] Gregor Kiczales, J.Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow. chapter Metaobject protocols: Why we want them and what else they can do, pages 101–118. MIT Press, Cambridge, MA, USA, 1993.

[KCJ03] Sam Kamin, Lars Clausen, and Ava Jarvis. Jumbo: Run-time code generation for java and its applications. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, page 48–56, USA, 2003. IEEE Computer Society.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM. doi:10.1145/319838.319859.

[KGK+07] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[Kim10] Michael Kimberlin. Reducing boilerplate code with project lombok, 2010. URL: http://jnb.ociweb.com/jnb/jnbJan2010.html.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, second edition, 2018. URL: `https://doc.rust-lang.org/book/2018-edition/index.html`.

[Kot20] Compiler plugins (kotlin), April 2020. URL: `https://kotlinlang.org/docs/reference/compiler-plugins.html`.

[LE16] Florian Lorenzen and Sebastian Erdweg. Sound type-dependent syntactic language extension. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 204–216, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2837614.2837644`.

[MS12] Weiyu Miao and Jeremy Siek. Pattern-based traits. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 1729–1736, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2245276.2232057`.

[MS14] Weiyu Miao and Jeremy Siek. Compile-time reflection and metaprogramming for java. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, page 27–37, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2543728.2543739`.

[Nem18] Nemerle. The nemerle programming language, September 2018. URL: `http://nemerle.org`.

[NS07] Nathaniel Nystrom and Vijay Saraswat. An annotation and compiler plugin system for x10. Technical report, Technical Report RC24198, IBM TJ Watson Research Center, 2007.

[OMY01] Yutaka Oiwa, Hidehiko Masuhara, and Akinori Yonezawa. Dynjava: Type safe dynamic code generation in java. In *In JSSST Workshop on Programming and Programming Languages, PPL2001, March 2001*, 2001.

[Ora19] Oracle. Interface plugin, October 2019. URL: `https://docs.oracle.com/en/java/javase/11/docs/api/jdk.compiler/com/sun/source/util/Plugin.html`.

[PAC+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM. `doi:10.1145/1390630.1390656`.

[Pae93] Andreas Paepcke. In Andreas Paepcke, editor, *Object-oriented Programming*, chapter User-level Language Crafting: Introducing the CLOS Metaobject Protocol, pages 65–99. MIT Press, Cambridge, MA, USA, 1993.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[PMD20] Pmd source code analyzer project, April 2020. URL: `https://pmd.github.io`.

[PS11] Zachary Palmer and Scott F. Smith. Backstage java: Making a difference in metaprogramming. *SIGPLAN Not.*, 46(10):939–958, October 2011. doi:10.1145/2076021.2048137.

[RAM+12] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. *Higher Order Symbol. Comput.*, 25(1):165–207, March 2012. doi:10.1007/s10990-013-9096-9.

[Ram15] L. Ramalho. *Fluent Python: Clear, Concise, and Effective Programming.* O'Reilly Media, 2015.

[RC02] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 205–230, Berlin, Heidelberg, 2002. Springer-Verlag.

[RT07] John Reppy and Aaron Turon. Metaprogramming with traits. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, page 373–398, Berlin, Heidelberg, 2007. Springer-Verlag.

[Rus20] *Compiler Plugins (Rust).* apr 2020. URL: https://doc.rust-lang.org/1.5.0/book/compiler-plugins.html.

[SBF15] Yannis Smaragdakis, Aggelos Biboudis, and George Fourtounis. Structured program generation techniques. In Jácome Cunha, João Paulo Fernandes, Ralf Lämmel, João Saraiva, and Vadim Zaytsev, editors, *Grand Timely Topics in Software Engineering - International Summer School GTTSE 2015, Braga, Portugal, August 23-29, 2015, Tutorial Lectures*, volume 10223 of *Lecture Notes in Computer Science*, pages 154–178. Springer, 2015. doi:10.1007/978-3-319-60074-1.7.

[Ska05] Kamil Skalski. Syntax-extending and type-reflecting macros in an object-oriented language. Master's thesis, University of Wroclaw, Poland, 2005. Nemerle.

[SMO05] Kamil Skalski, Michal Moskal, and Pawel Olszta. Meta-programming in nemerle, 2005. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.101.8265.

[Spo20] Spotbugs manual, February 2020. URL: https://spotbugs.readthedocs.io/en/stable/.

[ST20] Lex Spoon and Seth Tisue. Scala compiler plugins, jan 2020. URL: https://docs.scala-lang.org/overviews/plugins/index.html.

[Ste90] Guy L. Steele. *Common LISP: The Language (2nd Ed.).* Digital Press, USA, 1990.

[Str13] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Professional, 4th edition, 2013.

[SZ13] Marco Servetto and Elena Zucca. A meta-circular language for active libraries. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, page 117–126, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2426890.2426913.

[SZN15] Amanj Sherwany, Nosheen Zaza, and Nathaniel Nystrom. A refactoring library for scala compiler extensions. In Björn Franke, editor, *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9031 of *Lecture Notes in Computer Science*, pages 31–48. Springer, 2015. doi:10.1007/978-3-662-46663-6.2.

[Tah04] Walid Taha. *A Gentle Introduction to Multi-stage Programming*, pages 30–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-25935-0_3.

[Tat99] Michiaki Tatsubori. An Extension Mechanism for the Java Language. Master's thesis, University of Tsukuba, Japan, 1999.

[TCIK00] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. Openjava: A class-based macro system for java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering, Papers from OORaSE 1999*, pages 117–133, London, UK, UK, 2000. Springer-Verlag.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, page 27–46, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/949305.949309.

[Tra08] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):31:1–31:40, October 2008. doi:10.1145/1391956.1391958.

[typ20] Using the compiler api (typescript), apr 2020. URL: https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, December 1987. doi:10.1145/38807.38828.

[Vel03] Todd L. Veldhuizen. C++ templates are turing complete. Technical report, 2003.

[Xte20] Xtend. Xtend — modernized java, September 2020. URL: https://www.eclipse.org/xtend/.

About the author

José de Oliveira Guimarães is a Professor at the Computer Science Department of the Federal University of São Carlos at Sorocaba, SP, Brazil. Contact him at josedeoliveiraguimaraes@gmail.com or visit www.cyan-lang.org.

Acknowledgments This project was financed, under Process number 2014/01817-3, by FAPESP, the research support agency of the State of São Paulo, Brazil. We thank Gustavo M. D. Vieira and the anonymous reviewers for their helpful comments.

A Metaobject myproperty Implemented in Cyan

```
package main
import meta, import java.lang, import java.util

object CyanMetaobjectMyProperty
  extends    cyan.reflect.CyanMetaobjectAtAnnot
  implements cyan.reflect.IAction_afterResTypes

  // Cyan do not support enum types yet. Strings are used instead
  func init {
    super init: "myproperty", "ZeroParameters", [ "field" ]
  }

  override
  func afterResTypes_codeToAdd:
        ICompiler_afterResTypes compiler,
        Array<Tuple< WrAnnotation,
                  Array<ISlotSignature>>> infoList
        -> Tuple<String, String> {

      // cast a Java value of class IDeclaration to
      // the Java class WrFieldDec
    var WrFieldDec field = JavaCast<WrFieldDec>
        asReceiver: getAnnotation getDeclaration;
    var String fieldName = field getName;
    var nameUpper = (fieldName[0] toUpperCase) ++
            (fieldName substring: 1);
    var String ivTypeName = field getType getFullName;
    var String methodGet = " func get$nameUpper -> $ivTypeName ";
    var String methodSet = " func set$nameUpper: $ivTypeName other ";
    var methodsSignature = "$methodGet;\n $methodSet; ";
    var methodsCode = "$methodGet = $fieldName;\n" ++
        "$methodSet { self.$fieldName = other; }\n";

    return [. methodsCode, methodsSignature .];
  }

  override
  func runUntilFixedPoint -> java.lang.Boolean = false;

    // methods that override methods of interface
    IAction_afterResTypes
    // go here. These methods do nothing
end
```

B Metaobjects in Action

Cyan metaobjects are used in several areas with an enormous diversity of objectives. More than one hundred metaobject classes and prototypes were created for a variety of goals. To show the power of the Cyan MOP, we will present some of the most important metaobjects in the next subsections.

B.1 Metaobjects in Interpreted Cyan

A metaobject prototype, after successfully compiled, should be put in a special directory of any package. To use the metaobject, a *compilation unit* imports this package. To streamline this process, package `cyan.lang` supplies some metaobjects that accept attached DSL code in *interpreted Cyan*. For example, annotation `onOverride` takes an attached DSL code that is run whenever the associated method is overridden in a subprototype.

```
@onOverride{*
    if  (method getStatementList: env) getStatementList size < 10
{
        metaobject addError: (method getFirstSymbol: env),
            "method test should have at least 10 statements"
    }
*}
func test {  }
```

In this case, the interpreted Cyan code demands that the overridden method has at least ten statements. Each metaobject of `cyan.lang` accepting interpreted Cyan code, as an attached code, has pre-defined variables like `method`, `env`, and `metaobject` in this example. There are variables for each parameter of the metamethod used (as `method`) and for the current metaobject (`metaobject`) and compilation environment (`env`).

Package `cyan.lang` has more complex metaobjects whose attached interpreted Cyan code can do multiple tasks like adding fields and methods to the current prototype, communicating with other metaobjects, creating new prototypes, checking during phase afterSemAn, and so on. The interpreted Cyan code can be put in files and loaded by metaobjects, thus reusing them. As one last example, Listing 5 shows an annotation that inserts nine methods in the current prototype whose names vary from `power_2` to `power_10`. The `insert:` method accepts two arguments: the signature of a method and its full definition.

B.2 Metaobject `grammarMethod`

This metaobject simulates the existence of a method whose keywords are given by a *regular expression* specified in an annotation attached to a method. The metaobject creates all *virtual* methods that match the regular expression; i.e. methods whose keywords match those of the regular expression. Calls to these methods are redirected to the annotated method. In the next example, annotation `grammarMethod` is attached to method `meet` of `Schedule`. Its attached DSL specifies a keyword pattern using a regular expression. Symbols `?`, `*`, and `+` mean that the preceding expression is optional, can be repeated zero or more times, and can be repeated one or more times, respectively.

Listing 5 – Annotation `insertCode` adds methods to the current prototype

```
1   @insertCode{*
2       // adds to the prototype methods like
3       //     func power_num: Int n -> Int = n*n ... *n;
4       //  "= $s;" is equal to "= " ++ s ++ ";"
5       for num in 2..10 {
6           var sig = "func power_$num: Int n -> Int ";
7           var s = "n";
8           //  ++ is concatenation of strings
9           for p in 2..num {
10              s = s ++ "*n"
11          }
12          insert: sig,
13                  sig ++ "= $s;"
14      }
15  *}
```

```
object Schedule
    @grammarMethod{*
        (name: String  (at: String)? (with: String)* )+
    *}
    func meet: Array<Tuple<String,
                  Union<some, String, none, Any>,
                  Array<String>>> p {
         // elided
    }
end
```

Message passings to expressions of type `Schedule` that do not match any methods are matched against the regular expression. If there is a match, method `meet:` is called passing the arguments packed as a single parameter. The following example is a single message passing intercepted by metaobject `grammarMethod`, which replaces it by an expression that packs the arguments and calls method `meet:`. Since the language is prototype-based, prototypes are objects that can receive messages.

```
Schedule name: "Kandinsky" at: "Garden" with: "Matisse"
         name: "Frida" with: "Picasso" with: "Mondrian"
         name: "Leonardo";
```

The arguments are packed into an array of tuples, in this example. There are rules for building the type of the annotated method parameter, which depends on the regular expression.

B.3 Metaobject concept

Concepts were devised to help the compiler issue clearer error messages in the instantiation of a template class in C++. Gregor et al. [GJS+06] proposed this feature for the language C++, although it has not been accepted yet.[16] *Concepts* are predicates

[16] Concepts may be added to the upcoming language version.

on template/generic parameters. They are implemented in Cyan using metaobject `concept`, without any help from the language itself. The DSL code attached to the annotation specifies the restrictions that the generic parameters should obey. In the example that follows, `T` is required to define three methods: `unit`, `*`, and `inverse`, with the given signatures.

```
@concept{*
    T has [ func unit -> T    func * T -> T    func inverse -> T ]
*}
object GroupList<T> ...  end
```

The DSL of the code attached to the `concept` annotation has statements for requiring that a prototype inherits another one, a prototype implements an interface, a parameter is an interface, a prototype declares a set of methods (used in the above example), a prototype belongs to a set of prototypes, and the negation of each of these statements. There are two statements that are not restrictions on parameter types: one loads a statement list from a file and executes them and the other one creates test files. Both use special package directories managed by the Cyan MOP. The environment object and the restricted compiler object, passed as parameters to interface methods described in subsection 4.2, have methods to read and write to files of these special directories. Each Cyan package can have the directories `--data` (for DSL code like those of metaobject `concept`), `--test` (for tests), and others not described in this paper.

B.4 Metaobject in the Cyan Libraries

Package `cyan.lang` is imported by every Cyan source file and defines prototype `Any`, the top-level prototype, generic prototypes for tuples, unions, and anonymous functions, the `Array<T>` prototype, and all basic prototypes such as `Int`, `Char`, and `String`. Metaobjects are used extensively in this package. Since there is a large interaction between it and the Cyan language, we can assure that not only package `cyan.lang` but also the Cyan language would be very different without the Cyan MOP. A small list of metaobject used by this package follows.

Metaobjects check that methods `eq:` and `neq:`, for testing object references, are only defined in `Any` and basic types. Metaobjects create fields and methods for instantiations of the generic prototypes `Function` and `Tuple`, with any number of parameters. The generated code by the metaobject varies depending on the number of parameters and methods, such as `==`, are added to the code of an instantiation of `Tuple` based on the tuple elements. Method `sort` is inserted in an instantiation `Array<P>` of `Array` if `P` defines a method `<=>`. Prototypes of basic types inject code into their `Array` instantiations. As a result, there is a method `sum` that returns the sum of all elements of an object of `Array<Int>`. Method `isA:` tests if the receiver object is an instance of the parameter. A metaobject tests whether the argument is really a prototype. Metaobjects of annotations attached to method `==` of `Any` check whether the argument is compatible with the receiver. For example, it is a compile-time error to compare an `Int` with a `Char` because the result will always be false. Another metaobject demands that, if `==` is overridden in a subprototype, `hashCode` has to be overridden too. And yet another metaobject generates code for testing the overridden method. This code is put in a special directory of the package.