The Cyan Language Metaobject Protocol

José de Oliveira Guimarães[a,*]

[a] *Computer Science Department at Sorocaba, Federal University of São Carlos, Brazil*

Abstract

Compile-time metaprogramming changes the compilation of a program using a metaprogram. It can be made using a Compile-time Metaobject Protocol (MOP) or without one. The metaprogram can add code to the program, do further checks, and, using a MOP, intercept several language operations such as object creation, message passing, and field access. Current Metaobject Protocols and languages with metaprogramming features have several drawbacks. Metacodes, which are parts of the metaprogram, can change non-local program code, making it difficult to understand and breaking encapsulation because private information is accessed. Error messages are not clear because there is no recording of activities of metacodes. In many systems, the compiler Abstract Syntax Tree is used, a too low-level structure to be handled by regular users. Different metacodes may have different views of the program and they can conflict with each other when adding code to the program. This article presents the MOP of the prototype-based object-oriented language Cyan. It does not have the great majority of the problems of other metaprogramming systems, which are either solved completely or diminished to acceptable levels. The design is unlike any other in which the compiler is in control of the metaprogram execution. These characteristics make the Cyan MOP easy to use although it has all the main functionalities of other protocols and many more.

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

1. Introduction

In object-oriented programming, software libraries supply classes and protocols for handling data of a specific domain. Classes model the abstractions of that domain and allow the representation of their entities. Message passings allow data to be supplied to the libraries and data to be transformed.

However, in languages that do not support compile or run-time metaprogramming, only domain data is transformed. The compiler can type check the use of the libraries but these cannot help the programmer create code or do further checkings in the library use beyond those supplied by the language type system.

*Corresponding author
Email address: josedeoliveiraguimaraes@gmail.com (José de Oliveira Guimarães)

Without metaprogramming, the library cannot make available new syntax, like macros, or support for DSL (Domain Specific Languages), in the user code; it cannot demand additional checkings, not even on its own use; it cannot create new classes based on the user code; and libraries cannot add code to the code that uses them. A Software library is limited to modeling and handling data of its domain.

Software libraries limitations can be overcome, at least partially, with runtime or compile-time metaprogramming. Code of a language that fully supports **runtime** metaprogramming can, at runtime, examine itself, call a method whose name is not known at compile-time, create classes, add methods and fields (instance variables) to classes and/or objects, intercept message passing, and, in general, change any aspect or the runtime behavior of the code. Support to metaprogramming at runtime varies from very high in languages as Ruby [1] or Self [2] (a prototype-based language), and low in languages as Java [3].

Runtime metaprogramming (RTMP) may be the correct design tool, but it has some drawbacks. It has an obvious performance penalty, although this can be mitigated with a carefully built system as in CLOS [4]. No new syntax, checked at compile-time, can be introduced. Then RTMP does not support macros à la Lisp [5] or embedding of DSL code in the user code (XML, for example) that is checked at compile-time. The code produced at runtime can have syntactic and semantic errors that will be discovered when it is too late. RTMP is not manifestly able to check, at compile-time, the user code. A library cannot check its usage or the usage of other code at compile-time.

Compile-time metaprogramming (CTMP) is the handling of a program by a metaprogram at compile-time. The metaprogram can: (a) do checks not demanded by the language used by the program, (b) change the program language semantics, and (c) change the program code (during the compilation only, not permanently). Compared to RTMP, CTMP offers another set of offsets. No transformation of the code at runtime is possible. There is no performance penalty at runtime, although the compilation time is greater. CTMP may introduce new syntax as Lisp macros. Some languages allow embedded DSL code whose syntax and semantics is checked at compile-time. Code introduced by the metaprogram in the user's code has to be compiled and therefore any errors are discovered at compile-time. Finally, a metaprogram may discover errors in the program that would not be pointed out by the compiler. For example, a metaprogram may check that whenever method `==` or `equals` is overridden, method `hashCode` is too. Errors can also be found by a pluggable type system (PTS) [6]. Besides the regular type checks, the PTS may look for additional errors. As an example, it may assure that only non-null values are assigned to a variable whose type is annotated to be non-null. Of course, many languages support a combination of Runtime and Compile-time metaprogramming. For example, LISP support macros (CTMP) and the generation and execution of code at runtime (RTMP).

Compile-time metaprogramming is implemented either with a Metaobject Protocol (MOP), as in CLOS [4], or without a MOP, as in Groovy [7]. The functionalities of both ways are not equivalent. Metaobject protocols have the power of: (a) creating new code, (b) doing additional checkings, and (c) intercepting

message passings, the creation of objects, class field access, subclassing, and method overriding. Then when one of these operations is made, a method of the metaprogram is called. Metaprogramming without a MOP is able to create new code and do additional checks. Its strong point is simplicity. They are much easier to implement and learn than a complete MOP, although without all of the power of one.

Compile-time metaprogramming has many problems, described next, both with or without a MOP. Assume that a program is composed of the regular code and a metaprogram. The last one may be spread inside the program or it may be separated from it. In any case, functions and methods of the metaprogram, named *metacode*, are called by the compiler either when it finds annotations or the metacode itself in the program. An annotation is a syntactic element as "`@createCode`" used in the program that is linked to a metacode that is elsewhere.

A source code (text of a single file) with a metaprogram may change the code of another file. The documentation of that code does not reflect what it does because it has been changed by metacode of other source file and not even the metaprogrammer may be aware of that. A metacode activated inside a source code may know the list of classes/prototypes of the current package or private information of other classes/prototypes. The metacode may generate code or do checks based on that information. This destroys separate compilation because any changes in the package or in internal details of classes/prototypes demand recompilation of all files of the program.

A metacode has access to the Abstract Syntax Tree (AST) of the program. Then it can alter it, including introducing syntax and semantic errors and bypassing the regular semantic checks of the language. Two metacodes may change the same AST object, causing a difficult-to-discover error — no record is kept on who changed who in the AST. Besides that, changes in the source code of the AST classes, such as the removal of a class or method, may invalidate metaprograms.

Two metacodes may have different views of the same source file or even of the program. This happens because one of them, or the annotation associated with it, is put before the other in the source code. A metacode may add code to the program that will not be seen by the other metacode. The order that two or more metacodes are called may not be clear to the metaprogrammer. If the metacode adds code to the program, a wrong order of call means a wrong order of inserted code and wrong final code. It may be difficult to choose the order call of metacodes.

Metacode may create metacode. This can cause an infinite loop and it may introduce cycles in the dependence between metacodes. That is, code produced by a metacode A includes metacode B and it may depend on code produced by metacode B. And vice-versa. Checks made by a metacode may be invalidated by code added by other metacodes later on .

The goal of this article is to present the Metaobject Protocol of the Cyan language [8]. This 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. This language allows the definition of prototypes, which are the counterpart of classes of class-based languages as C++ [9] or Smalltalk [10]. Cyan has a compile-time Metaobject Protocol which 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. Besides that, they can intercept message passing, field access, subprototyping, method overriding, etc.

The Cyan MOP addresses all of the problems with Compile-time metaprogramming described previously. It solves the absolute majority of them. Algorithms and techniques diminish the remaining problems to acceptable levels.

Cyan employs a Smalltalk-like syntax for method declaration and message send, although with important differences. A unary method `get` that returns an object of type `T` is declared as

```
func get -> T { /* code */ }     or     func get -> T = expr;
```

A non-unary method has one or more *method keywords* or just *keywords* ending with ":", each one having zero or more parameter declarations:

```
func add: String password    at: Int line, Int column
     doc: String docStr {  /* code */ }
```

A message passing is composed of a receiver and one or more keywords with their parameters:

```
var Box t = Box new; // creates an object
t get println; // (t get) println
t add: "xyZ#8wZZ"  at: 5, 7   doc: "Password for NotSecretAnymore";
```

This article is organized as follows. Section 2 describes the Metaobject Protocol of Cyan. Some important metaobjects and the use of the MOP in the Cyan libraries are presented in Section 3. Section 4 compares the metaprogramming of other languages with the Cyan MOP. The last section concludes.

2. The Cyan Metaobject Protocol

The Cyan Metaobject Protocol (MOP) is the interface between the language compiler, the regular program, and the metaprogram. It describes the *interactions* between the Cyan code being compiled, the compiler, the metaprogram, and annotations in the Cyan code that tells the compiler which classes/prototypes of the metaprogram should be used at a certain point of the code. The metaprogram in Cyan is composed of Java classes, Cyan prototypes, or a mixture of both. The compiler is made in Java and therefore it is convenient to use Java classes as the metaprogram. But since the compiler translates each Cyan prototype into a Java class, it is as easy to use Cyan as the metaprogramming language. The two languages interoperate reasonable well: Cyan code can import Java packages and classes and vice-versa.

Listing 1: Prototype `Person` that uses metaobject annotations

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

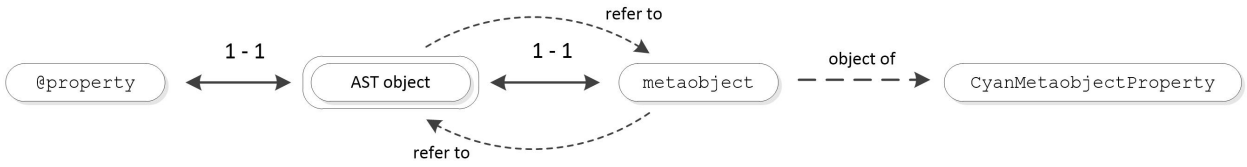

Figure 1: Relation between metaobjects, annotations, and metaobject classes

The following Subsection shows an example that uses the Cyan Metaobject Protocol. The main terms of the protocol also defined.

2.1. An Example of Annotations

An *annotation* or *metaobject annotation* is the syntax element that links the program with the metaprogram, more specificaly, to a part of it. Listing 1 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` an identifier that is, for practical purposes, also a string. `property` is attached to the declaration of *field* `name` (an instance variable in Smalltalk jargon).

When parsing source code, the Cyan compiler creates, for each annotation, three objects: an object of the AST private to the compiler, a wrapper object of the AST object, and a *metaobject*. The wrapper object is a read-only version of the compiler AST object that represents the annotation. A *metaobject* is an object of a Cyan prototype/Java class that inherits from prototype/class `CyanMetaobjectWithAt`.[1] There is a pool of metaobject classes and prototypes, taken from packages imported by the current source file, from which the compiler can choose to create the metaobject. The compiler chooses the class or prototype based on the annotation name, which should be returned by method `getName` of the metaobject.

[1]There are both a prototype in Cyan and a Java class with this same name.

Figure 1 shows the relationships among all of these elements. This Figure also shows that the AST object, representing the annotation, and the metaobject refer to each other. `CyanMetaobjectProperty` is the Java class of the metaobject associated with annotation `property`. A metaobject prototype/class is used to create any number of metaobjects. This Figure exhibits two one-to-one relationships: between metaobjects and wrapper objects and between wrapper objects and their annotations.

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

The compiler, at specific compilation phases, call metaobject methods to generate code and do checks. Metaobjects always generate code as strings. The code is added to a copy in memory of the prototype source code — the original file is not changed. In Listing 1, metaobject `property` generates methods "`getName`" and "`setName:`". Metaobject `init` generates code for a method "`init:`" that sets field `name`. Object constructors in Cyan have names `init` or `init:` (with parameters). The compiler inserts the code generated by these two metaobjects just before keyword "`end`". It could be in any place a field or method can appear. The compiler parses and does a partial semantic analysis of this new source code. Then metaobjects of annotations that are inside methods are analyzed. Metaobject `compilationInfo` generates

```
[ "name" ]
```

which is a literal array with one string containing the only `Person` field. The code produced by `compilationInfo` is inserted just after the annotation.

The resulting prototype `Person` is equivalent to the one shown in Listing 2. It is not exactly equal because some auxiliary annotations introduced by the compiler are not shown. Note that the annotations are not removed from the code, they are just marked as used by a suffix like "`#afti`" or "`#dsa`".

The current version of the Cyan compiler has to compile the whole source file after fields and methods are added to a prototype (for e.g., by `property` and `init`) and also after code is added inside methods, as by `compilationInfo`. This is an implementation detail that does not interfere with the semantics of the Metaobject Protocol.

Metaobject classes and prototypes belong to packages and these have to be imported before annotations are used. All metaobjects used in Listing 1 belong to package `cyan.lang` that is automatically imported by every Cyan source code. There are special rules on how to add metaobject classes (in Java) and prototypes (Cyan) to Cyan packages. They are put in a directory `--meta` of the package. Whenever the package is imported, the metaobjects become available. The details are explained by Guimarães [11].

Listing 2: Prototype `Person` that uses metaobject annotations

```
1  package human
2
3  object Person
4      @property#afti var String name
5      @init#afti(name)
6      func test {
7          let Array<String> list = @compilationInfo#dsa("field list")
8                [ "name" ];
9          list println;
10     }
11     func getName -> String = name;
12     func setName: String name { self.name = name }
13     func init: String name {
14         self.name = name
15     }
16 end
```

2.2. The Compilation Phases

A compile-time Metaobject Protocol links the compiler, the program, and the metaprogram. It can only be understood by studying the interface of the compiler visible to metaprogrammers. The most important compiler characteristics, to the MOP, are the compilation phases, shown in Figure 2. The first phase is parsing (dpa, During PArsing). It is represented by a rounded rectangle in the upper left corner of the Figure. In it, the source code is compared to the Cyan grammar and the Abstract Syntax Tree (AST) of the code is created. The AST, marked with label (a) below the thick arrow, is input to the next compilation phase, *type interfaces* (**ti**). Here "*interface*" means a set of language constructs of a prototype that are outside method bodies and that are associated with types. For example, method return value, method parameters, fields (instance variables), the specification of the superprototype, and implemented interfaces. Note that fields belong to this "*interface*" even though they are always private. After the compiler builds the AST, it only knows the type names, as strings. The AST object that represents the type of each of these features has a field `type` that is set to `null`. In phase ti, the compiler has parsed all the prototypes needed to type the current prototype and it is able to assign field `type` of every AST object of the prototype that represents a construct outside method bodies. Then, field `name` of `Person` of Listing 1 is represented by an AST object whose field `type` is `null` at the beginning of phase ti. During this phase, the compiler sets the

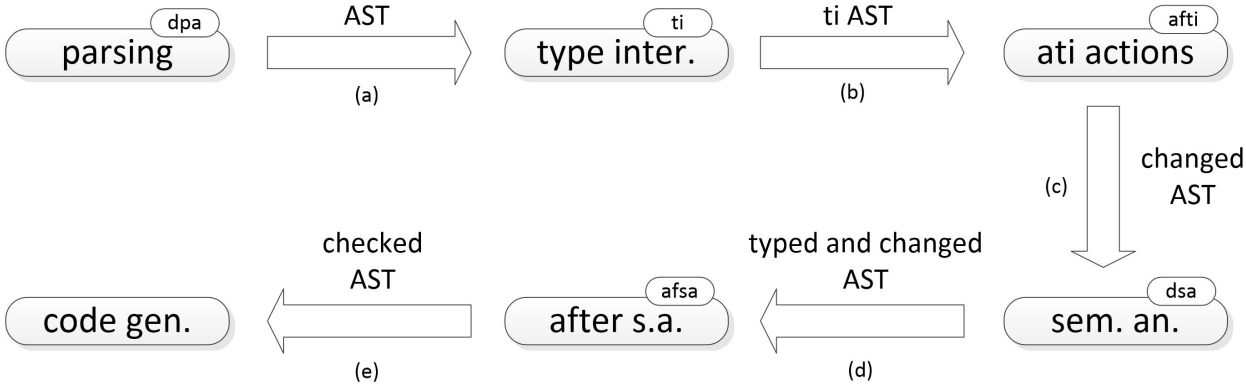

Figure 2: The phases of Cyan compilation

`type` field to the AST object that represents prototype `String`. Of course, before typing the interface of `Person`, the compiler knows the AST object representing `String`.

At the end of phase ti, the compiler has set every field `type` of AST objects of constructs outside method bodies. This changed AST, named *ti AST*, is passed as input to the next phase, *afti actions* (afti, AFter Typing Interfaces). In this phase, metaobjects can add fields and methods to the prototype in which they are. This is what metaobjects `property` and `init` do. The compiler could compile only the added code or the whole source code (a file). Currently, the whole file is compiled, although there are plans to compile only the added code. Anyway, a new AST is built, (c), and passed as input to the next phase, semantic analysis (dsa, During Semantic Analysis). This is, in fact, the second part of the semantic analysis, the first being *ti*. In this phase, a type is assigned to all expressions inside method bodies. Then an AST object that represents a prototype is assigned to field `type` of every AST object of every expression inside method bodies. For example, field `type` of the AST object of a local variable whose type is `Int` is assigned to a reference to the AST object that represents prototype `Int`.

In phase dsa, metaobjects can add code after the associated annotations. This is what `compilationInfo` of Listing 1 does. Again, the compiler can either parse and type just the added code, the method in which the annotation is, or the whole source file. Currently, the compiler compiles the whole file again. Because of suffixes `#afti` and `#dsa` put after the annotations, the compiler does not activate the metaobjects in this new compilation.

The typed AST, possibly changed by metaobjects, is passed as input to the next compilation phase, *after semantic analysis* (afsa, AFter Semantic Analysis). The AST cannot be changed anymore. This phase is only for checks, which can also be made in phases dpa, afti, and dsa. However, they should be made ideally in phase afsa because the AST cannot be changed anymore — changes can invalidate some checks. The last phase is code generation to Java.

2.3. Details of a Metaobject

This Subsection discusses some metaobject variations such as if the annotation is an expression, if it takes parameters, if it is attached to a declaration, and so on. A metaobject class (in Java) or prototype (in Cyan) should inherit from class or prototype `CyanMetaobjectWithAt`. In the call to the super constructor, three parameters should be passed: the metaobject name, the number of parameters, and the kind of declaration that the annotations can be attached to. The default for the number of parameters is zero. It is possible to specify zero or more, one or more, two or more, etc. An annotation can be attached to a prototype field, method, method signature (of a Cyan interface[2]), prototype, local variable declaration, package, or the program. The default is none, as `init` of Listing 1.

An annotation can be an expression, as `compilationInfo`. The metaobject class/prototype has to override some inherited methods that should return the package and prototype of the expression type, as strings.

An annotation may be followed by a text given between two delimiters. This text will be called *attached text* or *attached DSL* (Domain Specific Language). There are many possible variations of delimiters. In this paper, we will use `{*` and `*}` (the left symbol sequence always mirrors the right one). In the example of Listing 3, annotation `doc` takes a text, between `{*` and `*}`, that documents the method.

Annotation `rpn` takes an expression in Reverse Polish Notation as attached DSL. The metaobject evaluates the expression at compile-time. Annotations of metaobject `inline` should be attached to methods. They replace a call to the method by the expression between the delimiters, after replacing the parameters in the expression by the real arguments to the message passing.

2.4. The Interfaces of the MOP

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.

A metaobject class/prototype should inherit from `CyanMetaobjectWithAt`. This class/prototype has a method to return the AST object that represents the annotation and this AST object has a method to return the metaobject (see the dashed arrows "refer to" of Figure 1). The annotation AST object has methods to return the attached declaration (if any, as in `property`), the list of annotation parameters (`init` and `compilationInfo` take one parameter each in Listing 1), and the attached DSL code as a char array. The method that returns the attached declaration returns an AST object. Through it, one has access to all of its information. For example, metaobject `property` of Listing 1 knows the name of the field, which is `name`, and its type, `String`. Using this information, the metaobject is able to generate the `get` and `set` methods shown in Listing 2.

[2]Similar to a Java interface, declared with keyword `interface` instead of `object`.

Listing 3: Annotations with attached text

```
object Test
    @doc{*
        This method tests metaobjects 'inline' and
        'rpn'.
    *}
    func test {
        let ten = twice: 5;
        assert ten == 10;
        var Int twelve = @rpn{* 15 1 2 + - *};
        assert twelve == 12;
    }

    @inline{* 2*n *}
    func twice: Int n -> Int = n + n;
end
```

A metaobject class/prototype should implement one or more Java/Cyan interfaces in order to do a useful job. "*Interface*" here means the language construct that allows only the declaration of method signatures (or methods with a default body in Java, but without fields). Both Java and Cyan use keyword `interface` for that. Each interface is associated with a compilation phase. The interface methods overridden in the metaobject class/prototype are called in that phase only. As an example, the metaobject class `CyanMetaobjectProperty` of metaobject `property` implements interface `IAction_afti` that declares a method `afti_codeToAdd`. The compiler calls this method of metaobject `property` in phase afti. It returns code, as a string, that is added to the prototype in which the annotation `property` is. In the following Subsubsections, we describe the main interfaces of the Cyan MOP. Note that for each Java interface there is a Cyan interface with the same name. The Java interfaces are in package `meta` of the compiler. The Cyan interfaces are in package `cyan.reflect`.

An interface whose name ends with `dpa` (afti, dsa, afsa) acts in that compilation phase only. That is, its methods are called only in that phase. However, a metaobject class/prototype can implement several interfaces. There are logical restrictions on the combinations of interfaces. For example, an interface for intercepting message passings cannot be implemented by a metaobject class/prototype that also implements an interface that demands the annotation be attached to a local variable.

2.4.1. IParseWithCyanCompiler_dpa

Metaobjects that admit an attached DSL may use this interface to help to parse the DSL code. It has only one method with a parameter that is a restricted version of the Cyan compiler. The method parameter type, `ICompiler_dpa`, has methods for parsing Cyan statements, types, expressions, identifiers, and method signatures.

```
    // Cyan
interface IParseWithCyanCompiler_dpa extends IParse_dpa
    func dpa_parse: ICompiler_dpa compiler_dpa
end
```

The types of all parameters of methods of Java/Cyan interfaces are Java classes or interfaces. This is because the compiler is made in Java. Then `ICompiler_dpa` is a Java interface even though the above code is in Cyan. This interface declares methods for parsing the attached DSL. For example, method

```
WrExpr expr()
```

parses an expression and returns the AST object of it. The classes of all AST objects used by the MOP are wrapper classes. The metaprogrammer has never access to the original compiler classes through the MOP.

The class of metaobject `inline` used in the previous subsection implements the interface of this Subsubsection. It uses method `expr` of `ICompiler_dpa` to parse the expression of the DSL code attached to the annotation. The compiler takes care of initializing the lexer with the DSL code. Any error messages use the correct line number in the original source file.

If a metaobject does not use a Cyan-like attached DSL, the metaprogrammer can use low-level methods of interface `ICompiler_dpa` for lexical analysis. There are methods `getSymbol`, to return the current lexical symbol, and `next`, to advance the lexical analyzer to the next input symbol.

2.4.2. IParseWithoutCyanCompiler_dpa

Metaobject `doc` used in Listing 3 takes a text that is not Cyan code. It does not need the Cyan compiler to be processed. In cases like this one, the metaobject should implement the following interface

```
interface IParseWithoutCyanCompiler_dpa extends IParse_dpa
    func dpa_parse: ICompilerAction_dpa compilerAction, String code
end
```

The interface method takes two parameters: a restricted version of the compiler and the text attached to the annotation.

Class `ICompilerAction_dpa` has methods that issue compilation errors and return the current method, current prototype, the compilation phase, and so on. The information available in phase dpa is very limited because the AST has not yet been built. `ICompilerAction_dpa` inherits from interface `IAbstractCyanCompiler`

that declares methods for writing and reading files of special directories, to be explained later. There is also a method that returns an identifier that is guaranteed to be different from every other identifier in scope. All interfaces that represent a restricted view of the compiler, like `ICompilerAction_dpa`, inherit directly or indirectly from `IAbstractCyanCompiler`.

2.4.3. IAction_afti

Interface `IAction_afti` declares four methods:

```
afti_beforeMethodCodeList:
afti_renameMethod:
afti_codeToAdd:
runUntilFixedPoint
```

`afti_beforeMethodCodeList:` is used to add code before methods of the current prototype. It should return a list of tuples, each one composed of "method name" and "code to be added to that method". Method `afti_renameMethod:` should return a list of tuples, each one with the old name and the new name.

Method `afti_codeToAdd:` may return `null` if it is used for checks only. Or it may return a tuple with the code of fields and methods (to be added to the current prototype) and signatures of these fields and methods (separated by ";"). The signature of a method is the method without its body, parameter names are optional. Then the signatures of the Cyan methods given in the introduction are

```
func get -> T
func add: String at: Int, Int doc: String
```

The signature of a field is its declaration without the optional expression assigned to it: "`var String name`". The first parameter to method `afti_codeToAdd:` is a restricted version of the compiler. It has methods to return the current prototype, the current compilation unit (source file), methods and fields of the current prototype, and so on.

```
func  afti_codeToAdd: ICompiler_afti compiler,
        Array<Tuple<Dyn, Array<Dyn>>> infoList
    -> Tuple<String, String>
```

The second parameter, `infoList`, is only used if method `runUntilFixedPoint` of the metaobject, declared in `IAction_afti`, returns `true`. This method should return false if the generation of code by `afti_codeToAdd:` does not depend on the code generated by other metaobjects of the same prototype. For example, method `runUntilFixedPoint` of the class/prototype of metaobject `property` returns `false`. The code the metaobject generates, get and set methods, depend only on the attached field. The `Dyn` type in Cyan is the dynamic type, for gradual typing [12]. It is supertype of every other type and the compiler does not check the validity of any message passings when the receiver type is `Dyn`.

A metaobject may depend on the code generated by other metaobjects. For example, metaobject `addFieldInfo` takes two parameters. If the second is `counter`, it adds to the current prototype a field whose name is the first parameter, initialized with the number of prototype fields. The metaobject class implements interface `IAction_afti` and defines a method `afti_codeToAdd` (in Java). In prototype `TestField`, there are two annotations `addFieldInfo`.

```
@addFieldInfo(fieldNum, counter)
@addFieldInfo(numOfFields, counter)
object TestField
    var Int one = 1;
    func sumAll -> Int = one + fieldNum + numOfFields;
end
```

The associated metaobjects should create two fields initialized with 3:

```
    let Int fieldNum = 3;
    let Int numOfFields = 3;
```

However, that is not what happens in the first time the compiler calls method `afti_codeToAdd` of each of the metaobjects passing `null` as the second parameter, `infoList`. Both methods return `2` because they view the original prototype, without fields and methods added by other metaobjects.

Method `runUntilFixedPoint` of the class of `addFieldInfo` returns `true`. Because of this, the compiler calls all methods `afti_codeToAdd:` again. In this time, parameter `infoList` is an array of tuples, each one composed by the AST object of an annotation and a list of field and method signatures. The metaobject associated with the annotation, first tuple element, generated the fields and methods described in the second tuple element. In the example, `infoList` refers to an array with two tuple objects, one for each metaobject. The second element of each tuple is an array with just one element, the description of field `fieldNum` or of `numOfFields`.

In this second round of calls to all `afti_codeToAdd` methods, the metaobjects can adjust their generated code. The number of fields is that of the original prototype, one, plus all the fields described in the list `infoList`. The resulting number is `3`. There is a third round of calls to all `afti_codeToAdd` methods. Now each method returns exactly the same code as in the second round. Because the generated code is equal, there is no fourth round of calls. The compiler makes a new round of calls even if only one meta-object returns a value different from that of the previous round.

Whenever method `runUntilFixedPoint` of a metaobject returns `true`, there is a round of calls to method `afti_codeToAdd` till a fixed point is reached. That is, the calls end when every method returns the same generated code as in the previous round. If method `runUntilFixedPoint` of a metaobject returns `false`,

it does not participate in the rounds but its information is added to `infoList`. The algorithm that calls methods `afti_codeToAdd` till a fixed point is reached will be called **FixMeta**.

The Cyan compiler checks whether the elements of the tuple returned by `afti_codeToAdd:` match. That is, if the fields and methods of the first tuple element are in the second tuple element and vice-versa.

2.4.4. IAction_dsa

Method `dsa_codeToAdd:` of interface `IAction_dsa` returns code, added after the annotation, or `null` if it is used only for checks.

```
interface IAction_dsa
    func dsa_codeToAdd: ICompiler_dsa compiler_dsa -> String
end
```

The parameter type, `ICompiler_dsa`, has methods for returning the current prototype, current method, fields and method of the current prototype, and so on. It inherits from interface `IAbstractCyanCompiler`.

2.4.5. IActionVariableDeclaration_dsa

The metaobject class/prototype of annotations attached to local variable declarations should implement interface `IActionVariableDeclaration_dsa`. It sole method `dsa_codeToAddAfter` adds code after the variable declaration. A local variable can be initialized in its declaration with an expression. Method `dsa_codeToAddAfter` has access to the AST object of the expression and it can check it.

2.4.6. IActionMessageSend_dsa

Message passings are intercepted by metaobjects that implement interface `IActionMessageSend_dsa`. The associated annotations should be attached to methods. This interface is useful to intercept message passings when the compiler finds an adequate method. The metaobject associated with the annotation, which is attached to a method, may check the message arguments, at compile-time, and replace the message passing by another expression.

```
interface IActionMessageSend_dsa
    func dsa_analyzeReplaceKeywordMessage:
            WrExprMessageSendWithKeywordsToExpr messageSendExpr,
            WrEnv env
        -> Tuple<Dyn, Dyn>


    func dsa_analyzeReplaceUnaryMessage:
            WrExprMessageSendUnaryChainToExpr messageSendExpr,
```

```
            WrEnv env
        -> Tuple<Dyn, Dyn>

    func dsa_analyzeReplaceUnaryMessageWithoutSelf:
            WrExprIdentStar messageSendExpr,
            WrEnv env
        -> Tuple<Dyn, Dyn>
end
```

For every message passing, the compiler collects the method signatures that match it. The result is put in an ordered list. The matched method signatures are searched for in breadth-first order in a search starting in the receiver type. The receiver type is a Cyan interface or a prototype that is not an interface. If the receiver type is a non-interface prototype, the search continues in the immediate superprototype and in its implemented interfaces. If the receiver type is an interface, the search continues in the immediate superinterfaces (an interface can inherit from more than one superinterface).

Each method signature belongs either to an interface or to a non-interface prototype. Since Cyan interfaces are regular prototypes, method signatures in interfaces are really regular methods with a body supplied by the compiler. Therefore every method signature is associated with a method.

Then, whenever the compiler finds a message passing, it builds a list of all method signatures that matches it and each of them is associated with a method that may have attached annotations. The compiler selects the metaobjects, associated with the annotations, that implement interface

`IActionMessageSend_dsa`

and calls method

`dsa_analyzeReplaceKeywordMessage:`

if the message passing uses keywords (like "`at:  put:`") or calls method

`dsa_analyzeReplaceUnaryMessage:`

if the message is a unary message with an explicit receiver like "`x print`". Message sends to "self" without this keyword, like "`unary`",[3] are handled by method `dsa_analyzeReplaceUnaryMessageWithoutSelf:`.

The compiler keeps calling the metaobject methods till it finds one that returns a non-`null` value, a tuple. When this happens, it replaces the message passing, in the source file, by the string returned as the first tuple element. The second tuple element is the type of the expression given as a string in the first tuple element. There may be several annotations to the same method whose associated classes/prototypes implement interface `IActionMessageSend_dsa`. The compiler calls the metaobject methods according to the *textual* order of the annotations.

[3] There is a method called `unary` in the current prototype.

The AST object that represents the original message passing is passed as the first parameter to the methods of `IActionMessageSend_dsa`. The second parameter is an environment object. It has methods for getting the current prototype and current method, search for fields, methods, and prototypes, read and write files of special directories, issue error messages, etc.

The class of metaobject `inline` used in Listing 3 implements interface `IActionMessageSend_dsa`. The string that replaces the original message passing takes into consideration the arguments to the message, if any.

2.4.7. IActionMethodMissing_dsa

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. When the compiler analyzes a message passing in phase dsa, it first finds the type of the expression that received the message. Suppose it is `T`. Then it looks for a matching method in T, its superprototype, super-superprototype, and so on. Interfaces are not taken into account. If no method is found, the compiler collects metaobjects that implement interface `IActionMethodMissing_dsa` and put them in a list.

```
interface IActionMethodMissing_dsa
    func dsa_analyzeReplaceMessageWithkeywords: WrExpr receiver,
            WrMessageWithKeywords message, WrEnv env
        -> Tuple<Dyn, Dyn>
    func dsa_analyzeReplaceUnaryMessage: WrExpr receiver,
                WrSymbol unarySymbol,    WrEnv env
        -> Tuple<Dyn, Dyn>
end
```

The list is ordered: first metaobjects whose annotations are attached to `T`, then annotations attached to methods of `T`, and then the process is repeated with the superprototype of `T`.

The compiler calls method `dsa_analyzeReplaceUnaryMessage:` of each of the list metaobjects, in the list order, if the message passing is a unary message. It stops when the first method returns a non-`null` value. The tuple returned has the same meaning as the tuple returned by methods of

`IActionMessageSend_dsa`

That is, a string with an expression and the type of the expression. The mechanism with method `dsa_analyzeReplaceMessageWithkeywords:` is the same.

This interface is used to simulate the existence of methods in a prototype, which is the same as to introduce *virtual* methods in it. For 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. A restriction could be that only fields from a pre-existing list could be used with `get`. Although this metaobject has not been built, it is easy to codify. Metaobject `grammarMethod` described in Subsection 3.2 implements interface `IActionMethodMissing_dsa` to simulate the existence of a method whose keywords are specified using a regular expressions. It makes it easy to create Domain Specific Languages using only regular Cyan message passings.

2.4.8. IActionFieldAccess_dsa

Metaobjects can intercept field access if their classes/prototypes implement the following interface. The annotation should be attached to the field to be intercepted.

```
interface IActionFieldAccess_dsa
    func dsa_replaceGetField: WrExpr fieldToGet, WrEnv env
        -> String
    func dsa_replaceSetField:
            WrExpr fieldToSet,
            WrExpr rightHandSideAssignment,
            WrEnv env
        -> String
end
```

Method `dsa_replaceGetField:` of a metaobject is called whenever:

(a) its annotation is attached to a field `f` of the current prototype;

(b) the metaobject class or prototype implements interface `IActionFieldAccess_dsa`;

(c) `f` is used inside an expression either using just "`f`" or "`self.f`".

There may be more than one annotation attached to a field whose associated metaobject class/prototype implements interface `IActionFieldAccess_dsa`. In this case, the compiler calls all methods `dsa_replaceGetField:` in the textual order of the annotations. If more than one method returns a non-`null` string, the compiler issues an error.

Method `dsa_replaceSetField:` is called whenever an assignment is made to a field. It works much like method `dsa_replaceGetField:`. Parameter `rightHandSideAssignment` is the expression assigned to the field.

2.4.9. IActionFieldMissing_dsa

Prototypes may have *virtual* fields that are used as the regular ones. This is achieved by metaobjects whose classes/prototypes implement interface `IActionFieldMissing_dsa`. Annotations of these metaobjects should be attached to prototypes.

```
interface IActionFieldMissing_dsa
    func dsa_replaceGetMissingField:
            WrExprSelfPeriodIdent fieldToGet, WrEnv env
        -> Tuple<String, String, String>
    func dsa_replaceSetMissingField: WrExprSelfPeriodIdent fieldToSet,
            WrExpr rightHandSideAssignment, WrEnv env
        -> String
end
```

The methods of this interface work much like those of interface `IActionFieldAccess_dsa`. The methods parameters have similar semantics. The return value of method `dsa_replaceGetMissingField:` is a tuple with three elements: the last one is the expression that replaces the reading (get) of the field. The first and second tuple elements are the package and prototype of the virtual field. Method `dsa_replaceSetMissingField:` just return the code that replaces the assignment to the virtual field.

The virtual fields should always be preceded by `self` as in "`self.name`". If only "`name`" is used, the compiler does not know if this is a field access or a call to a unary method.

2.4.10. ICheckSubprototype_afsa

The compilation phase afsa comes after dsa. In it, prototype code cannot be changed. Therefore, this is the ideal phase for checks because they will not be invalidated by metaobjects that change the code.

Inheritance should be planned [13] because of the interrelationships among the methods of a prototype, revealed by message sends to `self`. These relationships are one of the reasons inheritance violates encapsulation [14] — the subprototype designer should know internal details of the superprototype methods.

Interface `ICheckSubprototype_afsa` is used to partially solve this problem. Its method is called whenever a prototype is inherited and, therefore, the method can demand the subprototype has some features, as methods, implemented interfaces, and even method statements.

Suppose an annotation is attached to prototype `P` and the class/prototype of the associated metaobject implements this interface.

```
interface ICheckSubprototype_afsa extends ICheck_afti_afsa
    func afsa_checkSubprototype:
            ICompiler_dsa compiler_dsa,
            WrProgramUnit subPrototype
end
```

Then method `afsa_checkSubprototype:` of the metaobject is called whenever the prototype is inherited, even indirectly.[4] The method parameters are a restricted view of the compiler and the subprototype. The prototype with the annotation, the superprototype, can be got through the annotation AST object:

```
// Java code
WrProgramUnit superProto = (WrProgramUnit ) getAttachedDeclaration();
```

Annotations whose metaobjects implement `ICheckSubprototype_afsa` can also be attached to Cyan interfaces. This can be used, for example, to restrict the prototypes that can implement a given interface. A feature of Language Hack [15] demands that an interface is only implemented by classes that also inherit from another class. This can be done in Cyan with metaobject `restrictImplementation`.

2.4.11. ICheckOverride_afsa

Prototype `Any` in Cyan is the superprototype of every other prototype but `Nil`. It declares methods `==` and `hashCode`. These methods are interrelated: whenever a prototype overrides method `==`, it should override `hashCode` too. Otherwise equal objects may have different hash codes. This interrelationship is checked by a metaobject associated with an annotation attached to method `==` of `Any`, `overrideToo`. The metaobject class implements interface `ICheckOverride_afsa`. Whenever the attached method, `==`, is overridden, method `afsa_checkOverride:` of metaobject `overrideToo` is called by the compiler.

```
interface ICheckOverride_afsa extends ICheck_afti_afsa
    func afsa_checkOverride: ICompiler_dsa compiler, WrMethodDec method
end
```

The method parameters are a restricted view of the compiler and the AST of the overridden method, from the subprototype. The annotated method, from the superprototype, can be got from the AST object of the annotation.

```
// Java code
WrMethodDec superMethod = (WrMethodDec ) getAttachedDeclaration();
```

2.4.12. ICheckDeclaration_afsa

Interface `ICheckDeclaration_afsa` is used for checks in phase afsa. Annotations associated with metaobjects whose classes/prototypes implement this interface should be attached to a *declaration*, which is a prototype, method, field, or local variable.

[4] That is, the method is called even when `R` inherits from `Q`, a subprototype of `P`.

The class of metaobject `immutable` implements this interface. This metaobject checks whether the fields of the prototype are read-only[5] and theirs types are immutable. The basic Cyan types, like `Int` and `String`, are immutable.

2.4.13. ICheckMessageSend_afsa

Message passings can be checked in phase dsa using interface `IActionMessageSend_dsa`. However, this is flawed because metaobjects can introduce new code in this same phase and this new code will not be checked by the method of this interface. The correct procedure for message passing checks is to implement interface `ICheckMessageSend_afsa`. The checks may use the message arguments, receiver, and even the prototype in which the message passing is. In the code that follows, the parameters of all methods but one are elided.

```
interface ICheckMessageSend_afsa extends ICheck_afti_afsa
    func afsa_checkUnaryMessageSend: // parameters ...
    func afsa_checkUnaryMessageSendMostSpecific: // parameters ...
    func afsa_checkKeywordMessageSend: // parameters ...
    func afsa_checkKeywordMessageSendMostSpecific:
            WrExpr receiverExpr,
            WrProgramUnit receiverType,
            String receiverKind,
            WrMessageWithKeywords message,
            WrMethodSignature methodSignature,
            WrProgramUnit mostSpecificReceiver,
            WrEnv env
end
```

Interface `ICheckMessageSend_afsa` is used when, in phase afsa, the compiler analyzes a message passing. It collects all method signatures that match the message passing and put them in a list. This is made as described in Subsection 2.4.6 for interface `IActionMessageSend_dsa`. The compiler removes from the list methods that are not linked[6] to a metaobject whose class implements interface `ICheckMessageSend_afsa`. The compiler builds a second list with the metaobjects associated with the methods. Metaobjects of the most specific method are first in the list. If two metaobjects of the same method implement this interface, the one associated with the annotation that comes textually first appears first in the list. Then the compiler

[5] declared with `let` and not `var`.

[6] That is, there is no annotation attached to a removed method such that its associated metaobject class or prototype implements interface `ICheckMessageSend_afsa`.

calls method `afsa_checkKeywordMessageSend:` of all list elements, in the list order. If the metaobject annotation is in the first method, the most specific, the compiler also calls method

`afsa_checkKeywordMessageSendMostSpecific:`

The parameters of methods of `ICheckMessageSend_afsa` include the message receiver expression, the receiver type[7], the message, the method signature corresponding to the message, the most specific prototype (where the first adequate method for the message was found), and the environment.

2.4.14. ICommunicateInPrototype_afti_dsa_afsa

Metaobjects of the same prototype whose classes or prototypes implement interface

`ICommunicateInPrototype_afti_dsa_afsa`

can communicate before phases afti, dsa, and afsa. When analyzing a prototype and before any of these phases, the compiler collects in a list all metaobjects whose classes or prototypes implement this interface.

```
interface ICommunicateInPrototype_afti_dsa_afsa
    func afti_dsa_afsa_shareInfoPrototype: WrEnv env -> Dyn
    func afti_dsa_afsa_receiveInfoPrototype:
            Set<Tuple<String, Int, Int, Dyn>> annotationInfoSet,
            WrEnv env
end
```

The it calls method `afti_dsa_afsa_shareInfoPrototype:` of each metaobject. The value returned is put as the last element of a tuple whose first three elements are: the metaobject name, the metaobject number considering all annotations of this prototype,[8] the metaobject number considering all annotations with this same name. This tuple is added to a set with information of all metaobjects of the prototype that wanted to communicate with each other.

In a second step, the compiler calls all methods

`afti_dsa_afsa_receiveInfoPrototype:`

passing as parameters the tuple set and an environment.

2.4.15. IActionNewPrototypes_dpa, IActionNewPrototypes_afti, and IActionNewPrototypes_dsa

These interfaces declare one method each. When the interface is implemented by a metaobject class/prototype, the method should return the code of a prototype that is inserted in the current package. The sole method of each interface takes a compiler parameter whose types are `ICompilerAction_dpa`, `ICompiler_afti`, and `ICompiler_dsa`.

[7] A string that may be `"SUPER_R"`, `"SELF_R"`, `"EXPR_R"` and `"PROTOTYPE_R"`.

[8] Metaobjects are numbered in textual order in a prototype, starting with `1`.

2.4.16. IActionFunction

Class `WrEnv` has a method for searching, by name, metaobjects whose classes or prototypes implement interface `IActionFunction`. This interface has an `eval` method that returns an object whose type (package and prototype) is given by the two other interface methods.

```
interface IActionFunction
    func eval: Dyn input -> Dyn
    func getPackageOfType -> String
    func getPrototypeOfType -> String
end
```

The search for a metaobject is made in the packages imported by the current source file. Usually, a metaobject class or prototype that implements `IActionFunction` inherits from `CyanMetaobject` and does not implement any other interface. A metaobject whose class or prototype implements the interface of this Subsubsection is called a *function metaobject* because it is used as a function.

2.5. The Abstract Syntax Tree

The Cyan compiler is made in Java and therefore the AST is implemented in Java. Every AST class declares an `accept` method that calls the `accept` method of its components and then calls method `visit` of parameter `visitor`. Therefore objects of the AST can be visited using the Design Pattern *Visitor* [16].

```
public void accept(WrASTVisitor visitor, WrEnv env)
```

Objects of the AST are got by calling methods of the metaobject itself (message passings to `this` or `self`) and from parameters to the MOP interfaces described in Section 2.4. For example, suppose an annotation is attached to a method. The AST object that describes the method is returned by method `getAttachedDeclaration` of the metaobject. The current prototype is got by calling method `getDeclaringObject` of the AST object that describes the method.

```
WrMethodDec attachedMethod =
    (WrMethodDec ) this.getAttachedDeclaration();
WrProgramUnit currentPrototype =
    attachedMethod.getDeclaringObject();
```

Interface `ICompiler_dsa` declares a method for replacing a statement by any code.

```
boolean replaceStatementByCode( WrStatement stat,
        WrCyanMetaobjectWithAtAnnotation annotation,
        StringBuffer code, WrType codeType)
```

Objects of this interface are passed as parameters to several methods of interfaces of phase dsa. Then metaobjects that act in this phase can ask the compiler to replace statement `stat` (the first method parameter) by `code` that should have type `codeType`. `annotation` is the annotation that is asking for the replacement.

There are several security mechanisms in the AST classes. The Java exception `MetaSecurityException` is thrown if a metaobject whose annotation is in prototype `P` tries to

1. replace a statement that is not in `P`;
2. get the fields of a prototype that is not `P`;
3. retrieve the statements of methods in phase afti;
4. add documentation, examples, and features to a prototype that is not `P`;
5. search for non-visible methods in a prototype that is not `P`.

In general, the exception is thrown whenever the metaobject tries to retrieve information of another prototype that is not visible to the prototype in which its annotation is.

There is a recording mechanism in the AST classes: whenever a metaobject of a prototype `P` asks for information on a prototype `R`, `P` is put in the list of prototypes that depend on `R`. Then if `P` asks for the superprototype of `R` or its method signatures, `P` is added to the list of dependents of `R`. In future versions of the Cyan compiler, whenever `R` changes and needs to be recompiled, `P` will be compiled again too.

3. Metaobjects in Action

Metaobjects can generate new code and do checks in a program, two activities that pervade all software domains. It is therefore not a surprise that the Cyan MOP is 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: create virtual fields in prototypes, design embedded DSLs, assure prototypes are immutable, software testing (including the Cyan compiler), method argument checking, implement pluggable type checkers, log method calls, intercept the creation of objects, debug Cyan code, optimize code (by replacing message passings by the values calculated previously), support object replication in distributed systems [17], configure programs, enforce software requirements,[9] generate boilerplate code, and implement Design Patterns [16].

To show the power of the Cyan MOP, we will present some of the most important metaobjects in the next Subsections.

[9] Anything, including method pre-conditions, restrictions on inheritance and interface implementation, code that an overridden method should have, etc

3.1. Metaobjects in Interpreted Cyan

The prototypes of a Cyan package are put in a single directory with the package name. A subdirectory with name `--meta` contains the `.class` files[10] resulting from the compilation of metaobject classes and prototypes. Whenever the package is imported in a Cyan source file, the metaobjects become in scope.

To streamline this process, one can use several metaobjects that use the Cyan interpreter at compile-time. An example is in Listing 4. The attached DSL of the first annotation declares, in line 4, a method in Cyan that belongs to interface `ICheckSubprototype_afsa`. The parameters, `compiler` and `subPrototype`, are automatically declared. And so is variable `metaobject` that refers to the metaobject associated with the annotation. The method statements are in a dynamically-typed version of Cyan that is interpreted. The method between lines 4 and 16 issues an error if prototype `HumanTest` is inherited by a prototype of another package. The annotation starting at line 20 has a sole Cyan statement that calls the function metaobject `shouldCallSuperMethod` of package `cyan.lang`. It checks whether the overridden method calls the superprototype method as its first statement.

The statements of annotation `onOverride` and the method of line 4 can be put in a file of special directories `--data` of packages. The file statements are run by calling method `runFile`:

```
runFile: "main.runThisFile(param)";
```

There are a dozen metaobjects like those shown in Listing 4. Each implements some interfaces of Subsection 2.4, most of them implement the interfaces of phases `afti` and `dsa`.

3.2. Metaobject `grammarMethod`

This metaobject simulates the existence of a method whose keywords are given through a regular expression specified in an annotation attached to a method. That creates all *virtual* methods that match the regular expression; that is, 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.

```
object Schedule
    @grammarMethod{*
        (name: String  (at: String)? (with: String)* )+
```

[10] Cyan prototypes are compiled to Java classes. The Java compiler produces at least a `.class` file for every `.java` file, possibly more due to anonymous classes.

Listing 4: Prototype `HumanTest` that uses interpreted Cyan in annotations

```
package human

@onSubprototype_afti_dsa_afsa{*
    func afsa_checkSubprototype {
        let WrEnv env = compiler getEnv;
        let WrProgramUnit attachedProgramUnit =
            metaobject getAttachedDeclaration;
        let String packName = attachedProgramUnit getPackageName: env;
        let String subPackName = subPrototype getPackageName: env;
        if subPackName != packName {
            metaobject addError:
                "Prototype " ++ attachedProgramUnit getFullName ++
                " can only be inherited in its own package. It is " ++
                "being inherited by " ++ subPrototype getFullName;
        }
    }
*}
open
object HumanTest
    @onOverride{*
        call: "shouldCallSuperMethod";
    *}
    func test {
    }
end
```

```
    *}
    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. Then the following 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 in an array of tuples, in this example. There are rules to discover the type of the annotated method parameter, which depends on the regular expression. The compiler will tell the correct type if a wrong one is given. So, give any type to the parameter and await the compilation error with the correct one.

In the attached DSL to a `grammarMethod` annotation, a list of functional metaobjects may be given after the regular expression. Functional metaobjects were presented in Subsubsection 2.4.16. Method `eval` of each functional metaobject is called passing as parameter a tuple consisting of the receiver expression and the message, two AST objects that describe completely the original message passing.

Prototype `Out` of package `cyan.lang` has a virtual C-like method `printf:` which takes a format string followed by parameters to be printed. If the first parameter is a literal string, a functional metaobject checks if the parameters match the string. If not, a compile-time error is issued.

3.3. Metaobject `concept`

Generic prototypes in Cyan employ syntax similar to class templates in C++ or generic classes in Java.

```
object GroupList<T> ... end
```

A generic prototype is *instantiated* when real arguments are supplied to it:

```
var GroupList<GroupElem> groupList;
```

`GroupElem` is a prototype. For sake of simplicity, assume that only prototypes can be arguments to a generic prototype instantiation.

Inside `GroupList`, parameter `T` can be used in any place a type is expected: as the type of variables, parameters, fields, return value type of methods, and inside expressions.[11] The compiler parses `GroupList<T>` but it only does the semantic analysis after an instantiation, when `T` is replaced by a real prototype. Then semantic errors are possible after an instantiation like `GroupList<GroupElem>`. A variable whose type in `GroupList<T>` is `T` has type `GroupElem` after the instantiation. If the variable is the receiver of a message passing for which there is no method in `GroupElem`, there will be a compilation error. The error would be in a code that was not made by the prototype user and it will be generally difficult to understand.

Concepts were devised to help the compiler issue clearer error messages in the instantiation of a template class in C++. Stroustrup [18] proposed this feature for the language C++, although it has not been accepted yet.[12] *Concepts* are predicates 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, a prototype implements another interface, that a parameter is an interface or a non-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 every 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 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 Section 2.4, 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.

3.4. 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

[11] Prototypes are expressions because they are objects in Cyan, a prototype-based language.

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

all basic prototypes such as `Int`, `Char`, and `String`. Metaobjects are used extensively in this package and because 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 use 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 code varies with 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. Then, 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 `--test` of the package. A method of `Any` simulates, at compile-time, that every method of a prototype is an anonymous function — there is, currently, no language feature for that.

4. Comparison with Related Work

The prime example of a Metaobject Protocol is that of CLOS [4] [19] [20] [21] [22], an extension of Common Lisp [23] with features for object-oriented programming. The CLOS MOP acts at runtime, allowing the intercepting of several operations: object creation, allocation of memory, calculus of superclass precedence,[13] method calls, field access, and many more. The MOP of this language uses *metaclasses* which are the classes of classes and methods,[14] 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++ [24] is a C++ extension in which metaclasses for classes and methods are given the opportunity of changing the AST after parsing. A metaclass for 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 the interception of variable declarations, creation of objects, and read and write to fields.

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

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

OJ [25] [26] 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 the read of a class field. The user-defined metaclass can also define methods for intercepting creation of objects, array allocation, write to fields, method calls, and casts to the class.

AspectJ [27] [28] is a Java extension for *Aspect-Oriented Programming* (AOP) [29]. In this paradigm, code for an *aspect* of a program, like error handling and logging, is grouped and put in just one place instead of being scattered in the program. In AspectJ, several operations can be intercepted like method calls, field access, and creation of objects. The AspectJ compiler, directed by user-code, can add methods, fields, and constructors to classes and change inheritance and implemented interfaces.

Languages Xtend [30], Groovy [7], and Nemerle [31] [32] support *compile-time metaprogramming* without a Metaobject Protocol. We will say that these languages support *metaprogramming features*. They share many similar characteristics and therefore will be considered together. The common characteristics of these languages are:

(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 in several phases of the compilation, like before parsing, after parsing, before typing members (similar to afti of the Cyan compiler), after semantic analysis, during code generation, etc;

(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 then, using these AST objects, add methods to an annotated class, change inheritance, add statements to an annotated method, change method statements, and so on. They can more: any AST object reachable from the method arguments can be changed. Then a method can be added to a class that is not annotated or directly related to the annotated class. The class may be, for example, 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. Then the view of a class by methods of a PC is not well-defined.

BSJ [33] (Backstage Java) supports *metaprogramming features* but not a MOP. Annotations and pieces of metacode between `[:` and `:]` compose the metaprogram. 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 its own source file, 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 at 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 dependence graph based on directives `#target` and `#depends` of metacodes. The metacode of a target is executed before that of its dependents. This is complex because a metacode can create itself metacode.

In order to compare Cyan with other languages, we describe some problems with Metaobject Protocols. To each one is given a name in boldface.

MessOthers A metaobject associated with an annotation in a source file changes another source file. That makes it difficult to reason about a prototype because we do not know its code by looking at the source file in the IDE or text editor. It is not enough to read the documentation of the metaobjects it uses because other source files can change it. The problem can happen even inside a prototype because annotation of a method could change another method; add statements to it, for example. Non-local changes like those described make it hard to understand the code.

In Cyan, a metaobject can only change its own prototype. However, metaobjects attached to methods or inside a method can add fields and methods to the prototype. And even add statements to the start of other methods. This is not a big problem in practice because the addition of code does not usually prevent other methods from working. 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 AspectJ, a source file can change code in other files; for example, the metacode of a file can add fields and methods to a class in another file. Code may be added using a pattern language with the consequence that it is even difficult to foresee which parts of the program are affected.

WhoDependsOnWho The compiler of an Object-Oriented Language typically builds a *dependence graph* among its source files. In a prototype-based language, suppose there is a one-to-one correspondence between source files and prototypes. In the *dependence graph*, vertices are prototypes and there is an edge from `A` to `B` if `B` has to be recompiled whenever `A` changes. This is the case if `B` inherits from `A` or declares a variable whose type is `A`.

Metaobjects have to be taken into account in order to build the *dependence graph*. Whenever a metaobject of prototype `B` uses information on prototype `A`, there should be an edge from `A` to `B`.

The AST classes used by the Cyan MOP mirror that of the compiler. Not only the AST is read-only but it also records who asks for information on who. Then the compiler is able to build the *dependence graph* of the prototypes.[15] In OpenC++ and CLOS, the metaclasses do not have information on other classes, unless

[15]Currently this graph is built but not used because the compiler compiles a whole package at once.

some non-standard mechanisms are used. In all other languages, the dependencies caused by metacode is not computed by the compiler.

KnowsPackage A metaobject should not know any information on the package of any prototype but its name and some features assigned to it, as the documentation. If it knows the list of package prototypes, for example, it could generate code based on this information. The addition or removal of prototypes of the package would cause the recompilation of the prototype in which the metaobject is. Even the addition of a single method in a prototype should cause the recompilation of another prototype.

In Cyan, metaobjects have limited information on the prototype package. It only knows general information as the package name, its documentation, associated examples, etc. It does not know the list of its prototypes. This same is true in OpenC++. Languages with *metaprogramming features* knows all information available in the AST, which generally include package information.

KnowsFriendsSecrets A metaobject in a prototype `B` may generate code or do checks based on private information of prototype `A` as its list of fields, its list of private methods, or even statements of its methods. Any changes in `A` should cause the recompilation of `B`.

Cyan limits the view of a prototype `R` by metaobjects of another prototype `S`. Some methods of the AST return only the visible data, as only the public methods, and some thrown an exception if a metaobject of prototype `S` is asking private data of `R`. We are unaware of any other language that limits the visibility of the AST by metaobjects or metacode.

MyselfChangedAST A metaobject may change the AST directly, a low-level approach that demands a deep knowledge of the compiler. Not only AST changes are not recorded but innocent AST handlings may crash the compiler. Metaobject classes or prototypes become tied to the compiler AST classes, alterations in the last ones invalidate metaprograms.

The AST supplied to metaobjects in Cyan is read-only, code is added by other means. The AST classes only have methods demanded by the language features, they are not tied to a particular compiler implementation. The MOP AST objects are really wrappers to the real AST objects of the compiler. Language OJ, languages with metaprogramming features, and BSJ permit changes in the AST.

WhoDidWhat The MOP may not record which metacode did what with the program. That is, changes in the program by the metaprogram are not recorded. Then, if there is a compilation error, the compiler does not know who is to blame. If the final program is not correct, the user does not know who changed what, she or he does not even know the final code after all transformations caused by the MOP. The lack of recording activities of the metaprogram may be caused by the direct handling of the AST, by the language features themselves (as in AspectJ), or, in a Runtime MOP, by assignments and method calls.[16]

In Cyan, metaobjects *ask* the compiler to change the code, they do not change it directly by AST

[16]The superprototype could be changed, for example, by the assignment "`self.parent = other`".

handling. All modifications are recorded, which means compiler error messages point out exactly who produced the code with errors. For example, if two metaobjects, in phase afti, added two fields with the same name to the prototype, the compiler will point out the error telling exactly which annotations are associated with the metaobjects. To our knowledge, no other Metaobject Protocol or language with metaprogramming features is able to do that.

DifferentViews Metaobjects may have different views of the prototype in which they are. Code added by one metaobject may not be seen by another one that, because of this, generates wrong code.

An example in Cyan was described in Subsubsection 2.4.3: two metaobjects generates two fields initialized with the number of prototype fields. Since they do not know each other, the fields are initialized with a wrong value, one less than the correct number. This is solved by using algorithm **FixMeta**. Method `afti_codeToAdd` of a metaobject knows the code produced by methods with this name of other metaobjects and therefore it knows the final set of fields and methods of the prototype. However, this is not true for the other methods called in phase afti.

The *dependence graph* cited in problem **WhoDependsOnWho** may have cycles. In this case, the compiler has to do phase ti (Figure 2) of all cycle prototypes before proceeding to the full semantic analysis. A metaobject of prototype `R` of the cycle may add fields and methods in phase afti based on information of another prototype `S` of the cycle (the number of its methods, for example). In this same phase, a metaobject of `S` may add a method to it, invalidating prototype `R`. The ideal solution to this problem is to extend the use of algorithm **FixMeta** to all prototypes of a cycle in a *dependence graph*. We regarded that unnecessary. There is only the recommendation that a metaobject of a prototype should not take any actions (generate code, do checks) based on the non-existence of methods of another prototype. If a method does exist in another prototype, it will never be removed. But if it does not exist, a metaobject can add it in phase afti.

In phase dsa, metaobjects do not know the statements and expressions inserted by other metaobjects inside methods. This is not commonly a problem in our experience. It is rare that a metaobject generates statements or expressions, which is all they can generate in phase dsa, that depends on other method statements. Algorithm **FixMeta** could be adapted for phase dsa. However, we did not deem that necessary. In phase afsa, all metaobjects have the same code view because no changes are allowed anymore.

Of the languages compared with Cyan in this Section, OpenC++ and BSJ offer the same view for all metaobjects.

OrderMatters Metaobject methods can be called in an order that is not clear to the metaprogrammer. They can insert code that is not in the correct order. For example, two metaobjects may insert code at the beginning of the same method. Which metaobject acts first will determine which code is inserted first in the method. The MOP may consider this a conflict, issuing an error. Or it may demand the metaprogram specifies an execution order, as in BSJ. In AspectJ, a keyword may declare the execution order of the metacode.

In Cyan, this kind of problem only happens in phase afti with the `IAction_afti` method that inserts statements at the beginning of methods of the current prototype. The insertion order is the textual order of the metaobjects. This is not an ideal solution because it limits what is possible to do with this feature of the MOP.

InfiniteDependence Metaobjects may produce code with annotations. Using annotations as vertices and an edge from annotation `P` to `Q` if `P` generates `Q`, the resulting graph is a tree. Another graph may be built using an edge from `P` to `Q` if `P` depends on the information produced by `Q`. This is a *dependence graph* of metaobject information. This graph may have cycles, as in the `addFieldInfo` example of Subsubsection 2.4.3 — each metaobject depends on the other. A metaobject may even depend on a metaobject whose annotation it produced.

Cyan deals with infinite dependences in three ways. First, only fields and methods can be added in phase afti and the metaobjects do not have access to information on method statements. Therefore, metaobject methods of this phase cannot depend on code produced in a later phase, dsa, and cannot depend not even on method statements. Second, algorithm **FixMeta**, described in Subsubsection 2.4.3, eliminates most infinite dependencies among metaobjects caused by code generated by methods `afti_codeToAdd` in phase afti.

Third, metaobject methods can only add code *inside* methods in phase dsa, that is, statements and expressions. Then there cannot be dependencies from metaobjects that add fields and methods to prototypes. Then there could be dependencies among metaobjects in phase dsa that should be dealt with using two mechanisms: metaobject communication (what code I am inserting) and checks in phase afsa. In our experience, dependencies in phase dsa are not common. Depending on user experiences with the Cyan MOP, it may be necessary to prohibit metaobjects from accessing information that is outside the method in the dsa phase.

In OpenC++ and BSJ, metacodes have the same view of the code. In OpenC++, this is caused by the limitations of metaprogramming, one metaclass per class. In BSJ, the same view was built on purpose to prevent conflicts.

InvalidateChecks Checks should always be done after changes, by metaobjects, are not allowed anymore. If the compiler allows changes in any compilation phase, a check made by a metaobject may be invalidated by a modification by another one.

Checks should be made in Cyan in phase afsa because no alterations are allowed anymore. Then the metaprogrammer is sure the checks will not be invalidated by the addition of new code. Since there is a single metaclass for a class in OpenC++, there is no danger that a check is invalidated by another metacode; there is no other. In BSJ, the dependence graph is specified by labels in the metacode. Then, it is enough to make metacode that does checks to depend on every other metacode. It will be executed after all of them.

TooPowerful A sufficiently powerful Metaobject Protocol permits deep changes in the program by the metaprogram. Generally, radical code changes are not good. They make the code unreadable. Among the

modifications we consider radical we cite: remove methods and fields, alter who is the superprototype of a prototype, remove implemented interfaces, remove method statements, add or remove method parameters, change method return type, and rename prototypes, fields, or methods.

None of the radical changes cited above are allowed in Cyan. The goal of the Cyan MOP is to provide functionalities without disrupting the semantics too much. Code can be added, never removed, and user choices, as which is the superprototype of a prototype, are respected. A Metaobject Protocol with great power may be a design choice, as in the case of CLOS [4], created to simulate several Lisp dialects.

Now we compare some characteristics of Cyan with other languages. In this language, there may be more than one annotation per prototype or method, unlike languages as CLOS, OpenC++, and OJ, in which there is one metaclass per class or per method.

Metaobjects in Cyan can do additional checks, they do not replace existing ones as is possible in CLOS. This makes the MOP simpler and the code clearer because there is one only semantics, that of Cyan itself.

Metaobjects ask the compiler to add code, rename a method, or replace a statement (including expressions). This is unlike all Metaobject Protocols and languages with metaprogramming features. Metaobject design in Cyan is to choose which interfaces of Section 2.4 should be implemented, by the metaobject class or prototype, to achieve its goals. The code to be added is returned by methods. Then the main decisions are made when designing a metaobject class or prototype, not at runtime inside the metacode. One can discover, in broad terms, what a metaobject does by looking at the interfaces it implements, without examining its code. No other language with metaprogramming has such characteristics.

Code is added as strings and the MOP records who did what. Some languages use quasi-quotes, with code between delimiters, like

```
<: int x = 0; :>
```

of BSJ. The code between `<:` and `:>` is transformed into an AST object. Cyan use pure strings, which are handled with regular string operations. Unlike quasi-quotes, there are no ambiguities on the code produced using strings. The above code, for example, could be a field or a local variable declaration, two different AST objects.

In addition to some Cyan features cited previously, we are unaware of any metaprogramming system that supports the following Cyan characteristics:

(a) the variety of annotation parameters, that may be literals of basic types, arrays, tuples, maps, and any combination of them all;

(b) DSL code attached to annotations, a convenient way of passing a string as a parameter;

(c) the possibility of using the Cyan compiler for parsing and doing the semantic analysis of DSL code (Subsubsection 2.4.1);

(d) a standard way of writing and reading files to special package and prototype directories used to store

DSL code, temporary data, documentation, and tests;

(e) annotations can be expressions, as `compilationInfo`.

5. Conclusion

Compile-time Metaobject Protocols can be used in many areas. Metaobjects can generate boilerplate code and do checks beyond those made by the language. They make local changes in the code, adapting it to new needs. For example, they can simulate the existence of fields and methods. They can be used to optimize code,[17] logging, documenting code, implement embedded DSL, implement *concepts* (Subsection 3.3),

The Cyan MOP supports seven kinds of metaobject annotations. Only the most important of them was described in this paper. The other annotation kinds are: (a) literal numbers ended by an identifier (like `101bin` or `0AH2_Hex`), (b) literal strings starting with an identifier (like `xml" XML code"`), (c) macros (each start with an identifier after which any syntax is allowed), (d) annotations to types that implement pluggable types [34] [35] [6] (like `String@regex("a*[A-Z]")` or `Char@letter`), (e) Codegs, visual metaobjects that demand a plugin to an IDE[18] (an annotation `@color(red)` allows one to choose a color using a menu, during editing time), and (f) files in package directories `--dsl` (each file is transformed into one or more Cyan prototypes before parsing any other Cyan files).

The metaobject classes or prototypes of all metaobject kinds, but (f), can implement most interfaces of Subsection 2.4. Only implementations prohibited by logic are prohibited by Cyan. Then the metaobject of a number annotation, like `101bin`, could add fields and methods to the current prototype (it does not). It certainly generates number `5` as code in phase dsa.

The Cyan MOP combines a full MOP, like that of CLOS, with metaprogramming features of recent languages as Groovy and BSJ. The design of a metaobject class or prototype in Cyan starts with the choice of the interfaces it should implement. The interfaces are chosen to match the goals of the metaobject. If it should add fields and methods to the prototype, interface `IAction_afti` is implemented. If checks should be done, some interface of phase afsa should be implemented. Then the metaprogrammer, guided by the goals, make the most important decisions *before* starting coding. In each compilation phase, metaobject methods *ask* the compiler to add code. Then the metaprogram acts passively in relation to the compiler, who is in control of the execution flow of the metaprogram. The compiler checks if the metaprogram is only doing legal operations. The compiler also records all changes asked by metaobjects. Because of this, error messages point out exactly who caused any errors in the Cyan code introduced by metaobjects. If they could handle the AST directly, there would be no way of knowing who changed what.

[17] As in metaobject `inline` cited in Subsection 2.3.

[18] Integrated Development Environment

The AST is read-only by metaobjects and a copy of the compiler AST — it reflects the language, not internal compiler details. AST objects check whether a metaobject has permission to access the information it is asking for. If not, an exception is thrown. A metaobject of a prototype `P` that asks for the method list of another prototype `Q` gets exactly the list of methods of `Q` visible in `P`. The metaobjects obey the visibility rules of the language. Metaobjects may introduce dependencies among source files in the same way Cyan code does. Then a prototype `A` depends on `B` if the compiler, when compiling `A`, uses any information of `B`. Dependencies are extended to metaobjects in Cyan. The compiler records when a metaobject of `A` uses information of `B`, adding `B` to the list of prototypes `A` depends on.

A Cyan program is composed by packages specified in a *project file* that contains code of a DSL called *Pyan*. Annotations can be used in Pyan. They can be attached either to the whole program or to packages. Every annotation that can be attached to a prototype can be attached to the program or a package. The result is the same as to attach the annotation to all prototypes of the program or to all prototypes of the package. There are metaobject classes and prototypes that can only be used in the Pyan code. They implement an interface that is not discussed in this paper.

Annotations may take an attached DSL code that can be used to generate code and do checks. Therefore Cyan permits embedded DSL code. This can also be implemented by literal string annotations. For example, XML code and regular expressions can be embedded in strings — errors in the code are discovered at compile-time.

There are several planned future works for the Cyan MOP. One of them is to allow metaobjects to change the original source files. Currently, code they generated is added to a memory copy of the files. Other work is to support variable ownership like language Rust [36]. Some few interfaces, like those of Subsection 2.4, should also be added to the protocol. They would be used to intercept field access in phase afsa and to add implemented interfaces to a prototype. Annotations should be allowed to be attached to statements and expressions.

The web page `cyan-lang.org/articles` has a Cyan program with all the examples of this paper and metaobject annotations for every interface cited here. The language compiler is available for download at `cyan-lang.org`. It consists of around 61,000 of non-blank and non-commented lines of Java code (SLOC-P). The complete description of the Cyan MOP is made by Guimarães [11].

Acknowledgments: this project was financed, under Process number 2014/01817-3, by FAPESP, the research support agency of the State of São Paulo, Brazil.

References

[1] P. Perrotta, Metaprogramming Ruby, 1st Edition, Pragmatic Bookshelf, 2010.

[2] D. Ungar, R. B. Smith, Self: The power of simplicity, SIGPLAN Not. 22 (12) (1987) 227–242. doi:10.1145/38807.38828.
URL http://doi.acm.org/10.1145/38807.38828

[3] J. Gosling, B. Joy, G. L. Steele, G. Bracha, A. Buckley, The Java Language Specification, Java SE 8 Edition, 1st Edition, Addison-Wesley Professional, 2014.

[4] G. Kiczales, J. des Rivières, D. G. Bobrow, The Art of Metaobject Protocol, MIT Press, Cambridge, MA, USA, 1991.

[5] C. Barski, Land of Lisp: Learn to Program in Lisp, One Game at a Time!, 1st Edition, No Starch Press, San Francisco, CA, USA, 2010.

[6] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, M. D. Ernst, Practical pluggable types for java, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, ACM, New York, NY, USA, 2008, pp. 201–212. doi:10.1145/1390630.1390656.
URL http://doi.acm.org/10.1145/1390630.1390656

[7] D. Koenig, A. Glover, P. King, G. Laforge, J. Skeet, Groovy in Action, Manning Publications Co., Greenwich, CT, USA, 2007.

[8] J. de Oliveira Guimarães, The cyan language, http://www.cyan-lang.org (2019).

[9] B. Stroustrup, The C++ Programming Language, 4th Edition, Addison-Wesley Professional, 2013.

[10] A. Goldberg, D. Robson, Smalltalk-80: the language and its implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[11] J. de Oliveira Guimarães, The cyan language metaobject protocol, http://www.cyan-lang.org (2019).

[12] J. Siek, W. Taha, Gradual typing for objects, in: Proceedings of the 21st European Conference on Object-Oriented Programming, ECOOP'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 2–27.
URL http://dl-acm-org.ez31.periodicos.capes.gov.br/citation.cfm?id=2394758.2394762

[13] J. Bloch, Effective Java, 3rd Edition, Addison-Wesley, Boston, MA, 2018.
URL https://www.safaribooksonline.com/library/view/effective-java-third/9780134686097/

[14] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, SIGPLAN Not. 21 (11) (1986) 38–45. doi:10.1145/960112.28702.
URL http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/960112.28702

[15] Hack, The hack programming language (Nov. 2017).
URL http://hacklang.org/

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] F. A. Ugliara, G. M. D. Vieira, J. de Oliveira Guimarães, Transparent replication using metaprogramming in cyan, in: Proceedings of the 21st Brazilian Symposium on Programming Languages, SBLP 2017, ACM, New York, NY, USA, 2017, pp. 9:1–9:8. doi:10.1145/3125374.3125375.
URL http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/3125374.3125375

[18] B. Stroustrup, Concept checking - a more abstract complement to type checking, Tech. Rep. N1510=03-0093, C++ Standards Committee Papers. ISO/IEC JTC1/SC22/WG21 (Oct. 2003).
URL http://www.stroustrup.com/n1510-concept-checking.pdf

[19] G. Kiczales, J. Ashley, L. Rodriguez, A. Vahdat, D. G. Bobrow, MIT Press, Cambridge, MA, USA, 1993, Ch. Metaobject protocols: Why we want them and what else they can do, pp. 101–118.

[20] A. Paepcke, in: A. Paepcke (Ed.), Object-oriented Programming, MIT Press, Cambridge, MA, USA, 1993, Ch. User-level Language Crafting: Introducing the CLOS Metaobject Protocol, pp. 65–99. [link].
URL http://dl.acm.org/citation.cfm?id=166848.166855

[21] D. G. Bobrow, R. P. Gabriel, J. L. White, Object-oriented programming, MIT Press, Cambridge, MA, USA, 1993, Ch.

CLOS in Context: The Shape of the Design Space, pp. 29–61.
URL http://dl.acm.org/citation.cfm?id=166848.166854

[22] L. G. DeMichiel, R. P. Gabriel, The common lisp object system: An overview, in: European Conference on Object-oriented Programming on ECOOP '87, Springer-Verlag, Berlin, Heidelberg, 1987, pp. 151–170.
URL http://dl.acm.org/citation.cfm?id=34985.35000

[23] P. Seibel, Practical Common Lisp, 1st Edition, Apress, Berkely, CA, USA, 2012.

[24] S. Chiba, A metaobject protocol for c++, in: Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '95, ACM, New York, NY, USA, 1995, pp. 285–299. doi:10.1145/217838.217868.
URL http://doi.acm.org/10.1145/217838.217868

[25] M. Tatsubori, S. Chiba, K. Itano, M.-O. 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, Springer-Verlag, London, UK, UK, 2000, pp. 117–133.
URL http://dl.acm.org/citation.cfm?id=646954.713484

[26] M. Tatsubori, An Extension Mechanism for the Java Language, Master's thesis, University of Tsukuba, Japan (1999).

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of aspectj, in: J. L. Knudsen (Ed.), ECOOP, Vol. 2072 of Lecture Notes in Computer Science, Springer, 2001, pp. 327–353.

[28] AspectJ, The aspectj language (2018).
URL https://www.eclipse.org/aspectj/doc/next/progguide/language.html

[29] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Akşit, S. Matsuoka (Eds.), ECOOP'97 — Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 220–242.

[30] Xtend, Xtend — modernized java (Sep. 2018).
URL https://www.eclipse.org/xtend/

[31] Nemerle, The nemerle programming language (Sep. 2018).
URL http://nemerle.org

[32] K. Skalski, Syntax-extending and type-reflecting macros in an object-oriented language, Master's thesis, University of Wroclaw, Poland, nemerle (2005).

[33] Z. Palmer, S. F. Smith, Backstage java: Making a difference in metaprogramming, in: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, ACM, New York, NY, USA, 2011, pp. 939–958. doi:10.1145/2048066.2048137.
URL http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/2048066.2048137

[34] G. Bracha, Pluggable type systems, in: In OOPSLA'04 Workshop on Revival of Dynamic Languages, 2004.

[35] Checker, The checker framework manual: Custom pluggable types for java (7 2018).
URL https://checkerframework.org/manual/checker-framework-manual.pdf

[36] S. Klabnik, C. Nichols, The Rust Programming Language, 2nd Edition, No Starch Press, 2018.
URL https://doc.rust-lang.org/book/2018-edition/index.html