[The Cyan MOP]

The Cyan Language Metaobject Protocol

ANONYMOUS AUTHOR(S)

Compile-time metaprogramming can act on the compilation of a program directed by a metaprogram. The semantics and code generation of the program may be changed. Metaprogramming support by current languages is either not powerful enough (improved macros) or complex (many details have to be learned, semantic gap between the goals and available features, low-level AST or compiler handling). Metaprogramming in language Cyan is made using a Metaobject Protocol (MOP) that divides the compilation process into phases, each phase with a clear responsibility, which makes it easy to map the goals to the available tools. Contrary to other metaprogramming systems, there is an inversion of control in the Cyan MOP. It asks the metaprogram which changes should be made, the metaprogram does not do the changes itself. This all contributes to making the Cyan MOP simple by what it offers, even though supporting all main functionalities of other protocols and many more.

Additional Key Words and Phrases: Object-oriented languages, computational reflection, metaprogramming, compiler, meta-object protocols

1 INTRODUCTION

In object-oriented programming, software libraries supply classes and protocols for handling data of a specific domain. Classes model the abstraction of that domain and allow the representation of its 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. 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 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 language as Ruby [Perrotta 2010] or Self [Ungar and Smith 1987] (a prototype-based language), and low in languages as Java [Gosling et al. 2014].

Runtime metaprogramming (RTMP) may be the correct design tool, but it has some drawbacks. First, it has an obvious performance penalty, although this can be mitigated with a carefully built system as in CLOS [Kiczales et al. 1991]. Second, no new syntax, checked at compile-time, can be introduced. Then RTMP does not support macros à la Lisp [Barski 2010] or embedding of DSL code in the user code (XML, for example). Third, code produced at runtime can have syntactic and semantic errors that will be discovered when it is too late. Fourth, 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.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

Compile-time metaprogramming (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. But compilation happens one time for any number of program runs. 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 look for errors in the program based on specifications given by the programmers. 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) [Papi et al. 2008]. 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).

The language used in this paper is Cyan [author as the paper 2018], which 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++ [Stroustrup 2013] or Smalltalk [Goldberg and Robson 1983]. 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 goal of this article is to present the Cyan Metaobject Protocol. This is made in Section 2. Section 3 compares the metaprogramming of other languages with the Cyan MOP. The last section concludes.

2 THE CYAN METAOBJECT PROTOCOL

A *metaprogram* is a program that can change the base-level program, which is the regular program. The metaprogram can be written in the same language as the base-level program as in CLOS [Kiczales et al. 1991] or Guile [GNU 2018] (a dialect of language Scheme). Or they can employ different languages as in Cyan, in which metaprograms are Java code.

The Cyan compiler is made in language Java which makes it easy to do metaprogramming in this last language. The compiler is composed by a set of classes, some of which are part of the Metaobject Protocol (MOP). The Cyan MOP is the interface between the 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 of the metaprogram should be used at a certain point of the code. We will call "Compiler-MOP" the Java classes and interfaces ("interface" in the Java sense) that are known to the Cyan MOP.

2.1 An example of Annotation

Before going into the details, we will show some examples of how to use the Cyan MOP. Listing 1 declares a prototype `Student` in Cyan of a package `university`. A prototype plays the role of "class" in class-based languages. Mutable instance variables can be declared with keyword "`var`" as `name` and `number` in this example. If keyword "`let`" or nothing is used, the variable is read-only. "`@property`" and "`@init`" are called **annotations**, which are links between the Cyan source code and the metaprogram.

Listing 1. Prototype Student that uses metaobject annotations

```
1 package university
2
3 object Student
4     @property var String name
5     @property var Int number
6     @init(name, number)
7 end
```

There are two "@property" annotations before the declarations of instance variables name and number. There is another annotation inside prototype Student, @init(name, number). The compiler will create, during parsing, a **metaobject** for each annotation found in the source code. Then the compiler will create three metaobjects in this code. They are referenced by the Abstract Syntax Tree of the program.

The first annotation in line 4 will cause the creation of methods "getName" and "setName:". Idem for the annotation of line 5. The annotation @init(name, number) of line 6 will ask for the insertion of an init: method that initializes instance variables name and number. Constructors in Cyan have always the names init or init:.

Now we will describe the relationships between annotations, metaobjects, metaobject classes, the Cyan compiler, and packages. Each annotation in the source code causes the creation, during parsing, of one AST object that represents it.For each of such objects, there is exactly one metaobject, which is an object of a Java **metaobject class**. A single metaobject class can be used to create many metaobjects.

There are rules to be followed to produce and use a metaobject class. It will be necessary to explain low-level details of classes and the compiler. Those details will be important in assessing the good and the bad about the Cyan MOP. First, a metaobject class should be compiled as part of the Cyan compiler, one of its Java classes. The Java compiler will compile the Cyan compiler producing a ".class" file for each ".java" file. A ".class" file contains the bytecodes of the metaobject class. The .class file of a metaobject class should be added to a special directory of a Cyan package (anyone will do). After that, the metaobject class can be removed from the Cyan compiler. When the package is imported by a Cyan source code, the metaobject annotation is made available.

The second step is the compilation of a Cyan program using a Cyan compiler that is unaware of the metaobject class. When a source code imports the package cited previously, the compiler will load, from the special package directory, the ".class" of the metaobject class. Then Java classes are dynamically loaded from the Cyan compiler. For each annotation like "@property" in the code, the compiler will create an object of this metaobject class. The third step is the calling of some methods of the metaobject, described below.

Class CyanMetaobjectProperty is the Java class associated with annotation "property". This class has an inherited method getName that returns the annotation name, the string "property", which is also called **the metaobject name**. File "CyanMetaobjectProperty.class" is in a special directory of package cyan.lang, a package imported by every Cyan source code. Because this package is always imported, the code of Listing 1 can use annotation property without explicitly importing any package. The same is true for annotation init.

Class CyanMetaobjectProperty implements interface IActionProgramUnit_ati that declares method ati_codeToAddToPrototypes. This interface is used by the Cyan compiler after phase "type interfaces" (TI). After phase TI, the Cyan compiler scans all metaobjects of the program being

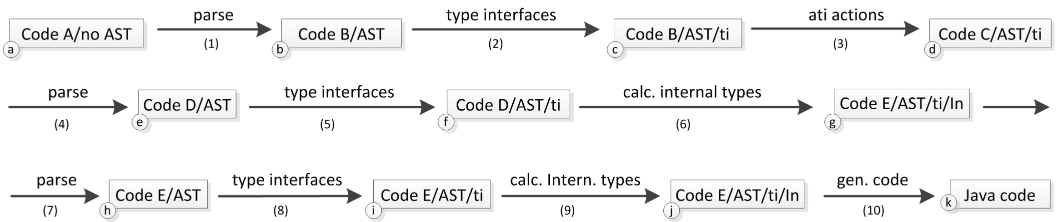

Fig. 1. The Compiler phases

compiled — there is one for each source code annotation. For each metaobject, the Cyan compiler asks if it implements interface `IActionProgramUnit_ati` (ati is "After TI"). If it does, method `ati_codeToAddToPrototypes` defined in this interface is called to generate code. This code is then inserted in the prototype as a string, changing the source code, an array of chars, in memory (the file with the original source code is not altered). After the code insertion, the program is compiled again to bring to life all the changes introduced by metaobjects.

Annotation `property` is associated to the metaobject class `CyanMetaobjectProperty` that implements interface `IActionProgramUnit_ati`. Then for each variable annotated with `property`, the compiler calls method `ati_codeToAddToPrototypes` of the metaobject associated to the annotation. The class of the metaobject is `CyanMetaobjectProperty`. This method produces code for methods get and set of the instance variable. If the variable is read-only, only the get method is generated. The code produced is in string format, more specifically, it is inside a `StringBuffer` object.

The Cyan compiler generates Java source code as the result of compilation. Each prototype is translated to a Java class. Java code can use the translated Cyan code. Then a variable can have a translated prototype as its type and a Java class can inherit from a (translated) prototype.

Java packages, classes, and interfaces can be imported by Cyan source code. Variables whose types are Java classes and interfaces can be used inside Cyan code using, of course, the Cyan syntax. The main restriction is that a prototype cannot inherit from a Java class or implement a Java interface.[1]

Metaprogramming in Cyan is made in Java but Cyan could be used if the compiler underwent some small changes. It would be only necessary to translate to Cyan prototypes the Java classes and interfaces of the compiler-MOP. The Cyan compiler would use the Java classes that result from the compilation of these Cyan prototypes. However, that was not done because there are no tools for Cyan. No adequate IDE support, no debug tools. It would be more difficult to code metaobject prototypes.

To fully understand how metaobjects work, it is necessary to explain the compiler phases and how each of them interacts with metaobjects. This is made in the next Subsection.

2.2 The Compiler Phases

The Cyan compiler has ten compilation phases, which are shown in Figure 1. The whole code of a program is parsed three times and there are one partial and two full semantic analyses. This is necessary because each time code is inserted in the program it is compiled again. The compiler is not targeted for efficiency. The upmost rectangle of the left represents the source code (**Code A**) of the program which is composed of one source code, a file, for each Cyan prototype. The compilation

[1]This may change in a near future.

starts in this rectangle and proceeds in the direction of the arrows. The "**no AST**" means that there is no AST at the start of the compilation. The rectangles are labeled using letters inside circles.

Each arrow is labeled by a number and represents a compiler phase. The arrow between (a) and (b) represents the syntactic analysis which produces the AST of (b) and a possible modified source code, indicated by "`Code B`" in rectangle (b), different from "`Code A`" in (a). The code can be modified by metaobjects that act in phase (1). We use "Code B" in both (b) and (c) because phase "type interfaces" does not change the source code, although it does change the AST by assigning types to some entities. Let us explain that.

During parsing, phases 1, 4, and 7, the compiler builds the AST of the program. This tree has objects for every program element. For example, the compiler creates an AST object for instance variable `number` of object `Student` of Listing 1. This AST object refers to a *String* `"Int"` that is the type of the variable in string format. After parsing there is a compiler phase called "type interfaces" (phases 2, 5, and 8), which initializes an instance variable called "`type`" to the appropriate AST object that represents the type. Then the compiler initializes the `type` variable of the AST object for `number` to the AST object that represents prototype `Int`. In phase "type interfaces", only the types external to method bodies are set. Then types are associated with instance variables, method signatures, inherited prototypes, etc. But expressions and variables inside methods do not have types. Note that "interfaces" in "type interfaces" refer to things visible from outside, it is not related to Java interfaces. In phase TI, the compiler also does the semantic analysis related to the types set. For example, it checks whether the type of an instance variable really exists. In rectangle (c) of the Figure, it is used AST/ti to indicate that the AST has been partially typed.

In the text that follows, we will use "class associated to annotation" to mean "metaobject class whose metaobject is associated to the annotation". That is, the Java class of the metalevel whose method `getName` returns the annotation name. For example, "`CyanMetaobjectProperty`" class for annotation "`property`". A metaobject class should inherit from class `CyanMetaobjectWithAt` of the compiler-MOP. It should also implement one or more Java interfaces, some of which are cited in Table 1. This table does not show interfaces of phases 8-9 which are only for checks.

The column **phase #** shows the compilation phase in which each **Java Interface** is used. That is, the compilation phase in which the methods declared in the interface are called by the compiler. A metaobject may add code to the program. If it does and where the code is added is in column "**add code?**". An annotation may be attached to a declaration such as a prototype or an instance variable (as `property`). Or it may not be attached to anything, as metaobject `init`. Column "`attached to`" lists, for each interface, which kind of declaration an annotation associated to a class that implements the interface may be attached. Note that if a metaobject class implements an interface, the annotations associated to it can only be attached to declarations cited in the table. The word "none" in column "`attached to`" means that the annotation need not be attached to anything. This kind of annotation can only be used inside a prototype, outside any method, or inside a method, as a statement or expression.

A metaobject class may implement any number of interfaces of the table, with a few restrictions. Interfaces `A` and `B` cannot be implemented by a class if they should be attached to mutually exclusive declarations. For example, `IActionMessageSend_dsa` and `IActionVariableDeclaration_dsa`. Methods of interfaces `IAction_dpa` and `IAction_dsa` generate code that is added just after the declaration. If one of such methods generate code, the annotation cannot be attached to a declaration. Finally, interfaces `IParseWithCyanCompiler_dpa` and `IParseWithoutCyanCompiler_dpa` are mutually exclusive.

Metaobjects may add code to the program in several compiler phases. The code is added as a string changing the source code that is internal to the compiler — the original files are not modified. After the code is added, the program has to be compiled again. Then there is a phase 4, parsing,

because phase 3, ati actions, may change the source code, see Figure 1. And there are phases 7-9 because phase 6 may change the source code again. In phase 1, metaobjects may add code after the annotations. But the added code is compiled in sequence, this addition of code does not demand recompilation.

A metaobject can generate code if its class implements one of the interfaces of Table 1 that are used by the compiler to generate code. The code to be injected into the program should be returned, as a string, by one of the interface methods. For example, `IAction_dsa` has a method

```
StringBuffer dsa_codeToAdd(ICompiler_dsa compiler_dsa)
```

that returns the code to be inserted by the metaobject whose class implements this interface. The metaobject may just do checks. In this case, this method should return `null`.

Column "**add code?**" of Table 1 cites several different options for metaobjects to add code to the program. Code may be added just after the annotation as for interfaces `IAction_dpa` and `IAction_dsa`. Second, a metaobject may create new prototypes in the same package if its class implements, for example, `IActionNewPrototypes_dpa`. Interface `IActionProgramUnit_ati` is the only one that allows the addition of methods and instance variables to a prototype. A metaobject of a class that implements `IActionVariableDeclaration_dsa` can add code after a local variable declaration. An annotation associated to a class that implements `IActionMessageSend_dsa` should be attached to a method. Whenever the method may be called in a message passing, a method of this interface is called. If may return null or a code that should replace the message passing. If there are several methods that may be called, the metaobject of the most specific one, found at compile-time, is used. Finally, the annotation may do just checks, no new code is added to the program. After phase 7, metaobjects cannot change the program anymore.

Metaobjects cannot remove code. For example, they cannot remove a method from a prototype. But they can rename a method in phase 3. Parameters to a metaobject annotation can be literals of basic types, strings, literal arrays, literal tuples, and literal hash tables. Any combinations and nestings of these values are allowed. In the bottom (leaves) there should be literal values of basic types.

Annotations may be followed by a text between the characters `{*` and `*}`, which will be called "delimiters".

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

Usually, the text between the delimiters will be a Domain Specific Language (DSL) code, but not always, as in this example. Anyway, the text will be called "DSL of the annotation" or "annotation DSL". Here the metaobject associated to `inline`, attached to method `twice:`, will replace the message passing by the text between the delimiters. Then, a message send

```
M twice: 2 + 3;
```

will be replaced by

```
2*(2 + 3)
```

The metaobject class of this annotation implements interface `IActionMessageSend_dsa`. A method of the metaobject is called whenever a message passing will call method `twice:` of prototype `M`.

The class associated to an annotation that takes a DSL code must implement one of the following interfaces:

`IParseWithCyanCompiler_dpa`   `IParseWithoutCyanCompiler_dpa`

Each interface declares a parse method that takes a restricted compiler as parameter. The parameter

Table 1. Interfaces for metaobject classes for phases 1-7

Java Interface	**phase #**	**add code?**	**attached to**
IAction_dpa	1, 4	yes, after the annotation	none, prototype, method, field, local variable
ICompilerInfo_dpa	1, 4, 7	no	prototype, method, field
IParseWithCyanCompiler_dpa	1, 4, 7	no	none, prototype, method, field, local variable
IParseWithoutCyanCompiler_dpa	1, 4, 7	no	none, prototype, method, field, local variable
IActionNewPrototypes_dpa	1, 4	yes, new prototype	none,prototype, method, field, local variable
IActionProgramUnit_ati	3	yes, methods and fields	none, prototype, method, field, local variable
IActionNewPrototypes_ati	3	yes, new prototype	none, prototype, method, field, local variable
ICheckProgramUnit_bti	b. 5	no	prototype
ICheckDeclaration_ati2	a. 5	no	prototype, method, field, local variable
IAction_dsa	6	yes, after the annotation	none, prototype, method, field, local variable
IActionVariableDeclaration_dsa	6	yes, after the declaration	local variable
IActionMessageSend_dsa	6	yes, expression that replace the message passing	method
ICompileTimeDoesNotUnderstand_dsa	6	yes, expression that replace the message passing	method, prototype
IActionFieldAccess_dsa	6	yes	prototype

to the method of the first interface, **with** Cyan compiler, is a subset of the Cyan compiler. Through it, the metaobject class can call methods to parse a Cyan statement, expression, and so on. The returned Cyan AST object can be kept in the metaobject annotation. The second interface, **without**, defines a method that also takes a compiler as parameter. But this compiler only offers the very basic functionalities such as the text of the DSL code. In both cases, an AST is built from parsing, kept in the metaobject, and used in later compilation phases. To make that possible, the metaobject class must implement an interface used posteriorly. As `IActionMessageSend_dsa`, which is used in phase 6.

The DSL code of an annotation can direct the code generation and checkings. For example, metaobject `overrideTest` is used to create test cases. An annotation of it should be attached to a method. Whenever the method is overridden in a subprototype a test case is created based on the DSL code of the annotation. Metaobject `concept` implements *concepts* [Gregor et al. 2006] [Bagge and Haveraaen 2009], which are predicates on types used as real parameters to generic prototypes.

344
345
346
347
348

```
@concept(test){* T has [ func plus: T other -> T ] *}
object GroupWork<T>    // generic prototype
    // elided
end
```

349 The DSL code of this example demands that `T` have a method `plus:`. There are many other
350 restrictions, they are all checked at compile-time by the metaobject.
351 As another example, a metaobject associated with an annotation attached to a prototype could
352 do checkings when the prototype is inherited. Depending on the DSL code of the annotation, the
353 metaobject could check if some methods are correctly overridden. For example, if a certain method is
354 overridden, the subprototype method should call the original method. The call relationships among
355 the subprototype methods should obey a pattern defined by the DSL code of the annotation. The
356 most basic one would be like this: "the method that overrides `draw` should call the superprototype
357 method as its first statement". Inheritance would be less error-prone with a metaobject like this.
358 One can wonder why `IActions_dsa` or `IActionProgramUnit_ati` are necessary since interface
359 `IActions_dpa` can be used to generate code. The difference is in the parameter type of the methods
360 of these interfaces. They are all restricted views of the Cyan compiler. Interface `IActions_dpa`
361 is used during parsing, in which the Cyan compiler does not have too much information on the
362 code. Interface `IActionProgramUnit_ati` is used in phase 3, in which the compiler knows the
363 types of instance variables, the methods of the current prototype, the prototypes of a package, etc.
364 The compiler object passed to methods of `IActions_dsa` has even more information: it knows, for
365 example, the types of local variables. Each method of each interface listed in Table 1 is called by
366 the Cyan compiler in specific points of the compilation. It is the Cyan compiler that creates objects
367 that are restricted compilers. They usually have a reference to the current object that is the Cyan
368 compiler.
369 Interfaces used in phases 3 to 9 declare methods that have access to the declaration the annotation
370 is attached to, the current prototype, the current method, and so on. The declaration is an AST object
371 of the Cyan compiler that supplies a method for visiting its components. This method implements
372 the Visitor Design Pattern. Then a metaobject can gather information on the declaration it is
373 attached to, the current method or prototype, and so on.
374 Metaobjects can signal errors either through the compiler object passed as parameter or sending
375 a message `addError` to `self`. Errors in the code generated by a metaobject are detected only in the
376 next compilation phase. However, the compiler will point exactly the line/column of the annotation
377 that produced the code that caused the compilation error. And the full file with the error, with the
378 code inserted by annotations, is written to the file system.
379 A metaobject class that implements `IAction_dsa` may return a code that is an expression, which
380 should be inserted after the annotation in a method body. This class should override two inherited
381 methods to return the package and prototype that is the type of the code produced by the metaobject.
382 Then metaobjects are typed. For example, metaobject `@lineNumber` has type `cyan.lang.Int`:
383

384
```
    var thisLineNumber = @lineNumber;
```
385

386 The compiler will deduce that `thisLineNumber` has type `Int`.
387 Metaobjects of the same prototype can communicate with each other before phases 3, 6, and 9.
388 Each metaobject can make some information available and retrieve all information supplied by
389 other metaobjects of the same prototype. Using a special compiler option, metaobjects of the same
390 package can also communicate with each other. Of course, this destroys separate compilation of
391 source files.
392

The code generated by Cyan may capture or not identifiers in the scope it is inserted. This is a choice of the programmer. If capture is not intended, the metaobject class should use a method that returns a name that is guaranteed not to conflict with any other program name.

3 COMPARISON TO OTHER METAOBJECT PROTOCOLS

CLOS [Kiczales et al. 1991] [Paepcke 1993] [Kiczales et al. 1993] supports a runtime Metaobject Protocol. A class declaration specifies its name, slots (instance variables), superclasses, etc. Methods are defined outside classes and method dispatch is based on the type of all method arguments, not only the first as in most OO languages. We will say that a method is associated with the classes of its parameters. And that a class is associated with methods that take it as one of the parameter types. Each program element such as class, slot, or method is an object of another class called its metaclass. The metaclass is associated with methods that define the element behavior. For example, the behavior can be the allocation of new objects and their initialization, selection of the method to be called, slot get and set.

Each program element uses a default metaclass. But an element may specify a user-defined metaclass that may change its behavior. Then a class with a non-standard metaclass may allocate its slots in a hashtable. Slot get and set are redefined to use the hashtable. A method with a non-standard metaclass may log its calls or count how many times it was called. Classes and methods can be created at runtime as was expected in a very dynamic language.

Several operations can be intercepted and changed at runtime by creating new metaclasses and defining methods that take the metaclasses as parameter types. The MOP specifies which methods can be user-defined and what they do.

CLOS depends on code optimization for being efficient since it is made at runtime, in contrast with Cyan, that has a compile-time MOP. Metaclasses of CLOS are roughly equivalent to metaobject classes of Cyan, but with important differences. In CLOS each program element has a default metaclass that takes care of the default behavior. There are no default metaobjects in Cyan. A would-to-be *default metaobject* is regular compiler code not specifically delimited. If an annotation is used, methods of its metaobject may be called *after* the default code. The metaobject method does not replace any compiler behavior. It may add code to the program and do *additional* checks. No compiler code stops running because of it. In Cyan several annotations can be attached to the same program element (prototype, method, etc.). In CLOS each element has a sole metaclass, making it more difficult to compose behavior. A Cyan annotation can be an expression inside a method, in CLOS this is not possible. A Cyan metaobject class may implement several Java interfaces of the MOP. Then a bundle of behavior is grouped in a single place and can be got by using a single annotation. A prototype annotation can change or check methods and a method annotation can change or check its prototype. In CLOS, an equivalent result is obtained by defining multiple metaobjects. Each one has a more limited vision of the program. This may be considered a good or bad point depending on the circumstances. In Cyan a metaobject can visit the AST nodes of all AST objects it can grab. For example, a metaobject associated with an annotation attached to a prototype can visit the AST nodes of this prototype. The equivalent of this in CLOS would be to access code as a list at runtime. To our knowledge, this cannot be done. In Cyan, the code is added using strings that may contain both syntactic and semantic errors that will be discovered at compile-time in the place of insertion. In CLOS, code that is inserted is usually compiled first, ruling out at least syntactic errors although there may be type errors at runtime because the language is dynamically typed.

Languages Guile Scheme [GNU 2018], a dialect of language Scheme, and the Scheme-based MOP described by Feigl [Feigl 2011] have the same benefits and drawbacks as CLOS. To a less extent, so does OpenC++ [Chiba 1995], a language that has a compile-time MOP for C++. It uses

default metaobjects for translating the AST after each part of it has been built. The default behavior is to do nothing. A metaobject can change, for example, the access to a data member (instance variable) of a class. In this case, the class should be declared with a metaclass that defines a method `CompileWriteDataMember` that is able to change the AST node for the writing to data members. The new AST node is returned by this method. The AST is built by a function using strings. Then any errors in the code produced are detected inside the metaobjects. In Cyan, the code is produced as strings and errors are detected after the code is inserted in the program. As CLOS, a class can have a single metaclass in OpenC++. And all changes and checks have to be made after parsing and before the more valuable semantic information becomes available. In Cyan, metaobjects can act in three more compilation steps for adding code and checking and in two more steps for checkings only.

Metaprogramming in language Groovy [Koenig et al. 2007] [Groovy 2017] is made through AST transformations that may be global or local. Global transformations are not associated with annotations, they are a visit to every program class. In Cyan, that can be made with annotations attached to the whole program, a topic not discussed in this paper. Local transformations are associated with annotations, each one should be attached to a type (class, interface, etc.), method, field, method, and so on.

An annotation is associated with one or more previously compiled AST transformation classes. Each of such classes is associated with one of six compilation phases, starting with the semantic analysis. To simplify the explanation, we will assume that an annotation is associated to a sole AST transformation class. Whenever an annotation is attached to an element (class, method, etc.), a method `visit` of the AST transformation class is called in the compilation phase it is associated with. Then this `visit` method can change the element the annotation is attached to. For example, an annotation attached to a method can add statements to this method. The class of the method is accessible too, then the annotation can add methods to this class. Groovy offers several features to help buiding AST objects. One of them is the `macro` method that produces an AST object much like quotes and splices of other languages [Dotty 2018] [Sheard and Jones 2002] [Tratt 2005] [Nemerle 2017].

Let us now compare Groovy with Cyan. The former language compiles every source code one time, as usual. The latter currently does almost three full compilations, making the compilation time much greater. But the several compiler phases of Cyan allows a clear division on what can be made in each phase. Methods and fields can only be inserted in phase 3 before their use is checked in phase 6. The code inside methods can only be inserted in phase 6. This code may use methods and fields introduced in phase 3. Checkings in the final source code can be made in phases 8-9. In Groovy there is no such division, which is good for compiler performance but a little bit more difficult to reason about. The landscape of Java interfaces of the MOP gives a clear view of what interfaces a metaobject class should implement. Based on the goals of the metaobject, it is not difficult to choose the interfaces to implement.

There are Groovy facilities for helping building AST objects from code. But that does not eliminate AST handling totally. The insertion of code must be made using an AST object and with an AST object to be inserted. Errors are detected, mostly, when the AST is built, in the AST transformation class. Or in later compilation phases. In Cyan, code generation can be made without any AST handling, which is much more intuitive and easy. The downside is that errors are only detected after the code is inserted in the program. Besides that, code generation is always by returning a value of methods of metaobjects. These are methods implementing the Java interface methods of the MOP. This mechanism is safer than to allow, in Groovy, the AST transformation classes insert AST objects in the AST. One of the design objectives of the Cyan MOP was to reduce AST handling to a minimum, which was achieved.

In Groovy, AST objects can be changed at will. Then methods of a class can be removed, for example. Cyan allows one to replace an expression or statement of a method by another. But not more than that. Then Groovy is much more flexible, allowing unlimited AST handling. Cyan only allow the addition of code in more controlled and limited ways.

Some methods of the Java interfaces of the MOP take an abstract Cyan compiler as an argument. Then it is easy to get the current prototype or search for a local variable using this argument. The compiler object passed as parameter depends on the compilation step: there is less information during parsing and more during the semantic analysis. In Groovy the same information can be got but through the AST.

Cyan has a syntax for DSL attached to an annotation, as `@inline`. There is support in Cyan for compiling DSL code that is Cyan-like, a compiler is supplied to the metaobject. The AST of the parsed DSL code can be stored as a field in the AST object of the annotation and used in a later compilation phase, a feature that is extensively used.

In Groovy, DSLs have to be given inside a literal String argument to the annotation, there is no special syntax for them (a minor issue). Groovy libraries can be used to convert strings into AST objects but, to our knowledge, there is no *standard* way of sharing these objects between compilation phases. Different compilation phases are treated by different AST transformation classes.

Annotations in Cyan can be considered as expressions, as `@lineNumber`, and used whenever an expression is expected. In this case, they should implement interface `IAction_dsa`. That makes it easy to have DSL code inside Cyan code. One could have an annotation `@python` whose attached DSL is Python code, for example. The annotation results in an object can have a `run` method that executed the Python code.[2] In Groovy, this is not possible, annotations cannot be expressions.

An AST transformation class should be in the classpath[3] in order to be found by the Groovy compiler. Cyan associates annotations with packages, which is simpler and more modular. To use an annotation, just import its package.

In Cyan, metaobjects attached to prototypes and methods can intercept the following operations: inheritance, field access (either if the field exists or if it does not), method overriding, message passing (either if the method exists and if it does not). For example, annotation `overrideTest`, when attached to a method, produces a test code based on its attached DSL whenever the method is overridden. In Groovy these operations cannot be intercepted although the interception can be simulated with a global AST transformation. That demands to visit all AST objects representing classes of a program and detecting the operations. In fact, method "`macro`" cited was implemented in this way.

In Cyan, there is a standard way of communication between metaobjects of the same source file. They just have to implement a Java interface of the MOP. A metaobject can share any data and receive the data shared by the other metaobjects. There is no *standard* way of doing that in Groovy.

Language Nemerle [Nemerle 2017] has most Groovy metaprogramming features with the addition of some others: a Lisp-like sophisticated macro system, pattern matching for AST objects, and a higher-level AST handling. Macros use quotes and splices comparable to Converge [Tratt 2005] and Template Haskell [Sheard and Jones 2002]. Pattern matching can be used with quoted code, which makes metaprogramming easier. And AST handling can be made as simple as in the code

```
m.Body = <[  a = 0; $(m.Body) ]>
```

which inserts an assignment at the beginning of a method `m`. Cyan does not offer any of the Nemerle

[2] This annotation has not been built, although that would not be difficult.

[3] Where the compiler looks for classes.

features cited above. They depend on quoted code, which can be very handy to work with, but it is not as simple as the string handling supported by Cyan.

4 CONCLUSIONS

Metaobjects in Cyan have multiple uses. They can be used for eliminating boilerplate code through code generation at compile-time, for embedding DSL code in Cyan code, and for checking the use of prototypes, methods, message passing, etc.

Metaobjects are associated with regular Cyan packages. When a package is imported, its metaobjects can be used. No special compiler option is necessary. Interfaces direct how a metaobject will be used by the compiler. Then the designer of a metaobject class chooses the interfaces needed for the goals of the metaobjects. This is a design-time decision made before compilation. Therefore, the metaobject class does not need to choose, at compilation time, when to add code, do checkings, etc. The compiler, directed by the interfaces, take the actions. This simplifies the metaobject classes because the flow of execution becomes predictable. In our experience in implementing some one hundred metaobject classes, this is very important.

Metaobjects can act in multiple compiler phases and they can keep, in their fields, data on the previous phases, a feature that is extensively used. A single metaobject can produce code during parsing, in phase 3, and during semantic analysis. It may also do checkings in phases 1, 3, 6, 8 and 9. A metaobject may generate code with annotations whose metaobjects generate code or do checkings in later phases. A metaobject can add code in multiple places in a single step. Cyan metaobjects can communicate with each other in the same source file, even if their metaobject classes are different.

The MOP was built to minimize the use of the AST. Code to be inserted should be given as a Java string and most of the information necessary for metaobjects may be got from the compiler object passed as a parameter to most methods. Only a few of the nearly one hundred metaobjects need to access the AST. However, the AST is available.

Metaobjects of Cyan are used extensively in the prototypes of package `cyan.lang`, automatically imported by every Cyan source file. This package contains the fundamental prototypes of the language: the basic types, `Any`, `Array<T>`, prototypes for the exception system, tuples, and anonymous functions. The code of a tuple prototype such as `Tuple<Int, String>` is produced by a metaobject. Idem for prototypes that are the types of anonymous functions. There are metaobjects for supporting generic prototypes, documentation, testing, tests for the Cyan compiler itself, etc.

Many features of the Cyan MOP were not discussed in this article. There is a standard way of handling files shared among the metaobjects and between compilations. Metaobjects may be used in the project file that describes the packages of a program. And there are other kinds of metaobjects: literal numbers ending with an identifier such as `0101bin` or `FFAB_Hex`, literal strings starting with an identifier such as `r"0*1+"`, pluggable types (like `Int@range(1,10)` and `Char@letter`), macros à La Lisp, and literal objects delimited by a sequence of characters such as "`[# it*2 | 1..100 #]`". All of these metaobject kinds have all or almost all of the power of the metaobjects described here. For example, Cyan macros can add methods to the current prototype, communicate with other, and generate code in phase 6 of compilation.

The Cyan compiler is fully functional and supports all of the features cited in this text. And many more, there is no space to describe further details. The compiler and documentation can be found in `www.cyan-lang.org`. This site has an "articles" page with all the code of this paper.

REFERENCES

Same author as the paper. 2018. The Cyan Language. http://www.cyan-lang.org/

589 Anya Helene Bagge and Magne Haveraaen. 2009. Axiom-Based Transformations: Optimisation and Testing, In Proceedings
590 of the Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008), Jurgen J. Vinju and Adrian
591 Johnstone (Eds.). *Electronic Notes in Theoretical Computer Science* 238, 17–33. Issue 5. https://doi.org/10.1016/j.entcs.2009.09.038
592 Conrad Barski. 2010. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* (1st ed.). No Starch Press, San Francisco,
593 CA, USA.
594 Shigeru Chiba. 1995. A Metaobject Protocol for C++. In *Proceedings of the Tenth Annual Conference on Object-oriented*
595 *Programming Systems, Languages, and Applications (OOPSLA '95)*. ACM, New York, NY, USA, 285–299. https://doi.org/
596 10.1145/217838.217868
597 Dotty 2018. Dotty Documentation. http://dotty.epfl.ch/docs/ http://dotty.epfl.ch/docs/.
598 Peter Feigl. 2011. *An Efficient Meta-Object Protocol for Scheme*. Ph.D. Dissertation. Johannes Kepler Universitat.
599 GNU. 2018. Guile Reference Manual. https://www.gnu.org/software/guile/docs/ https://www.gnu.org/software/guile/docs/.
600 Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman
601 Publishing Co., Inc., Boston, MA, USA.
602 James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8*
603 *Edition* (1st ed.). Addison-Wesley Professional.
604 Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. 2006. Concepts:
605 Linguistic Support for Generic Programming in C++. *SIGPLAN Not.* 41, 10 (Oct. 2006), 291–310. https://doi.org/10.1145/
606 1167515.1167499
607 Groovy 2017. Runtime and compile-time metaprogramming. http://groovy-lang.org/metaprogramming.html http://groovy-
608 lang.org/metaprogramming.html.
609 Gregor Kiczales, J.Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow. 1993. *Metaobject protocols: Why we*
610 *want them and what else they can do*. MIT Press, Cambridge, MA, USA. 101–118 pages.
611 Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. 1991. *The Art of Metaobject Protocol*. MIT Press, Cambridge, MA,
612 USA.
613 Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. 2007. *Groovy in Action*. Manning Publications
614 Co., Greenwich, CT, USA.
615 Nemerle 2017. The Nemerle Programming Language. http://nemerle.org/ http://nemerle.org/.
616 Andreas Paepcke. 1993. Object-oriented Programming. MIT Press, Cambridge, MA, USA, Chapter User-level Language
617 Crafting: Introducing the CLOS Metaobject Protocol, 65–99. http://dl.acm.org/citation.cfm?id=166848.166855
618 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical Pluggable
619 Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM,
620 New York, NY, USA, 201–212. https://doi.org/10.1145/1390630.1390656
621 Paolo Perrotta. 2010. *Metaprogramming Ruby* (1st ed.). Pragmatic Bookshelf.
622 Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM*
623 *SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691
624 Bjarne Stroustrup. 2013. *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.
625 Laurence Tratt. 2005. Compile-time Meta-programming in a Dynamically Typed OO Language. In *Proceedings of the 2005*
626 *Symposium on Dynamic Languages (DLS '05)*. ACM, New York, NY, USA, 49–63. https://doi.org/10.1145/1146841.1146846
627 David Ungar and Randall B. Smith. 1987. Self: The power of simplicity. *SIGPLAN Not.* 22, 12 (Dec. 1987), 227–242.
628 https://doi.org/10.1145/38807.38828

629
630
631
632
633
634
635
636
637