http://www.cyan-lang.org

The Cyan Language Metaobject Protocol — Extended Version

JOSÉ DE OLIVEIRA GUIMARÃES, Federal University of São Carlos, Brazil

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

Additional Key Words and Phrases: object-oriented languages, metaprogramming, metaobject, computational reflection, prototype-based languages, compilers

ACM Reference Format:
José de Oliveira Guimarães. 2025. The Cyan Language Metaobject Protocol — Extended Version. 1, 1 (August 2025), 56 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Metaprogramming is the coding of programs, called *metaprograms*, that treat code as data. The program that is treated as data is called *the base program* or simply *program*. A *metaprogram* can generate new code, change existing programs, or do checks in them. Metaprogramming offers mechanisms for code reuse that go beyond that offered by traditional software libraries. It can generate families of related code, as in the case of C++ [Str13] templates, separate functional

Author's Contact Information: José de Oliveira Guimarães, Federal University of São Carlos, Computer Science Department, Sorocaba, SP, Brazil, josedeoliveiraguimaraes@gmail.com.

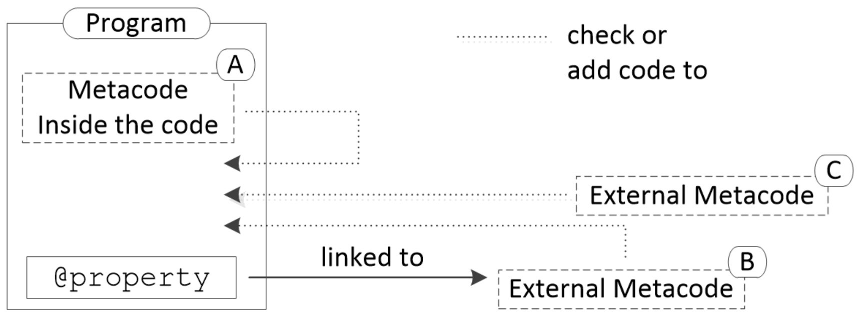

Fig. 1. The program and the metaprogram

and nonfunctional concerns, as in AspectJ [KHH+01], generate code based in specifications, as ANTLR 4 does [Par13], support new syntax (macros as in Scala [Bur13]), detect program bugs through static analyzers [Spo20] [Err20], implement new type systems using a pluggable type system [Bra04], run a program in multiple stages [Tah07], each stage generating and running a new program, change the program at runtime [RC02] [KCJ03], and support embedded Domain Specific Languages [RAM+12] [BIS16].

In this paper, the main focus is *language support* for Compile-Time Metaprogramming (CTMP), which is the handling of a program by a metaprogram at compile-time. Therefore, this text does not deal with metaprogramming using tools like ANTLR 4 [Par13] and SpotBugs [Spo20]. Or with metaprogramming made at a preprocessing-time (changes made in the source code before the program is parsed), loading time (changes in the program are made when the binaries or bytecodes of a virtual machine are loaded in the computer memory), or runtime. Runtime metaprogramming occurs during the program execution with the consequence that there are few or no static guarantees relating to the metaprogram.

To discuss specific characteristics of compile-time metaprogramming supported by programming languages, we will define some terms. The *program* is the code that implements the desired functionality for the application. A *metacode* is each of the pieces of code that compose the *metaprogram*. The metacode work as *compiler plugins* that interchange data with the compiler, in both directions, and can change the compilation process. The compiler calls metacode at specific points of compilation. Therefore, metacode can replace the type checker, code generator, parser, and any other algorithm used by the compiler. They can also add, delete, or replace code of the *program*. In practice, languages restrict what metacode can do to a few things. The metaprogram is designed to help the program achieve the desired functionality.

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

[1]Function as in language C

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

The goal of this article is to present the Metaobject Protocol of the Cyan language [Gui24]. 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. Cyan language allows the definition of prototypes, which are the counterpart of classes of class-based languages as C++ [Str13] or Smalltalk [GR83]. The compile-time Metaobject Protocol (MOP) of Cyan specifies the relationships between the compiler, the metaprogram, and the program. *Metaobjects* from the metaprogram can add code to the program, which includes new prototypes, fields and methods to prototypes, and statements and expressions to methods. Besides that, they can intercept message passing, field access, subprototyping, method overriding, etc. *Metaobjects* in Cyan have limited power, they cannot delete program code or replace any compiler algorithm as the type checker. Additional checks can be added to a program but no one can be bypassed. The code of Cyan *metaobjects* are what we called previously *metacode.* A *metaobject* is an object that exists at compile-time with methods called by the compiler at one or more compilation phases. An annotation in the source code, as `@property`, is associated with a metaobject.

The contribution of this article is to show how the Cyan MOP addresses total or partially each of the problems with Compile-time metaprogramming described previously. The characteristics of the Cyan MOP follow, relating them to the problems.

Metaobject methods return source code, as strings, added to the program by the compiler. The compiler traces which metaobject asked for code to be added to the program. If the generated code has errors, the compiler will know whom to blame. Code is only added by returning strings in metaobject methods, the Abstract Syntax Tree (AST) is never changed directly. An innovative fixed-point algorithm guarantees that, in an important compilation phase, metacode order in the base code is not important and that there are no circular information dependencies among metaobjects associated with the same prototype. The textual order of the annotations in the code is largely non-important. After the semantic analysis, metaobjects cannot change the code and, therefore, checks made after this phase are never invalidated by other metaobjects. The compiler has a security mechanism preventing metaobjects from accessing private information of prototypes of other source files. Code added by metaobjects in one compilation phase may have annotations. However, these will only be activated in the next compilation phase. The finiteness of the number of phases prevents a possible infinite compilation process.

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

2 Motivation

To compare Cyan with other languages, we describe some problems with metaprogramming, mainly CTMP which is the main topic of this paper. The problems are those considered general enough to be applied to various metaprogramming systems; specific drawbacks are not listed. Each problem has a name, placed in boldface. Cyan jargon is used in the descriptions: a *prototype* is a template from which *objects* are created. A prototype belongs to a *package* as in Java. A metacode *associated with* a source file is either inside source file, as `A` in Figure 1, or is linked to an annotation that is inside the source file, as `B` in the figure. A metacode *associated with* a prototype (or method) is inside the prototype or linked to an annotation inside the prototype (or method).

MessWithOthers A metacode associated with a source file changes another source file, which is called *obliviousness* [CL03]. 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 metacode it uses because other source files can change it. There is no way the developer can know, looking at the source code, which other source files can change it. A light version of this problem happens even inside a prototype because a metacode associated with 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.

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

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

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

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

(a) it demands a deep knowledge of the design and implementation of the compiler, which includes details of all the compilation phases and the data structures used. The metacode may require complex AST transformations that should keep compiler invariants (often undocumented);
(b) incorrect AST handling may crash the compiler or make it generate incorrect code;
(c) metacode may bypass compiler checks causing the acceptance of flawed source code. That is, metacode may add code after the compiler does some checks that will never be done in the added code.

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

WhoDidWhat A metacode that handles the compiler data structures directly leaves no traces of its activities. Therefore, if a metacode generates invalid code, detected in later compilation phases, the compiler will issue an error. But it will be unable to point out the metacode that generated the invalid code.

OrderMatters
If a prototype has many metacode associated with it, they can be called in an order that is not clear to the metaprogrammer [PS11] or they may be called in an order that prevents them from producing correct code or doing the intended checks.

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

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

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

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

Nontermination

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

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

NoGeneratedCodeGuarantees

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

NoContracts

A metacode may demand specific features from the base code it is attached to and vice-versa [LE16]. For example, the metacode may demand the base prototype `T` declares a method for comparing two `T` objects. And the base code may demand that the metacode adds to the prototype a method `sort` (built with the comparison method). Ideally, there should be a *contract DSL*[2] to specify the agreements between the metacode and the base code. The contract could be enforced by the compiler. If there is no contract DSL, metacode can check by themselves the demands they places on the base code. However, these demands would be more precisely described using a DSL code that is easily examined by the developer. Without a contract, the demands that the base code places on metaobjects are not verified. Thus, the causes of compilation errors are more difficult to discover. Errors may appear only in the final version of the source code which is a mix of base code with that added by metacode. To discover the

[2]Domain Specific Language

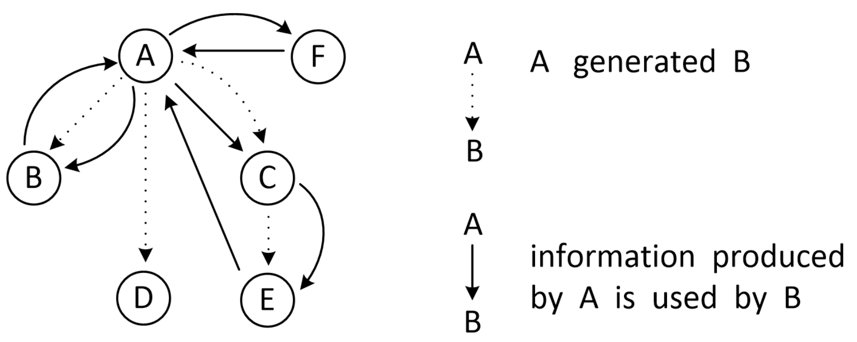

Fig. 2. Two graphs: one showing metacode generation and the other showing which metacode uses information produced or changed by others

errors, the developer has to examine the source file and scrutinize code generated by metacode, which exposes their implementation details.

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

The compiler has to chose a first metacode in a cycle to be the first to be run. Suppose the first one is metacode B. Since this vertex is in a cycle, there will be an ingoing edge from another vertex, say A, to B. By the definition of *dependence graph*, A produced or changed information used by B. The problem is that the compiler first runs B generating code or doing checks based on information that will be later changed when the compiler runs metacode A. The CircularDependency problem cannot be solved by an adequate choice of a vertex in a cyle.

Figure 2 shows an example of a *code generation* tree, using dashed edges, and of a *dependence graph*, using solid edges. The simplest cycle[3] has only two vertices as that composed of A and F in the figure. Therefore, metacode A depends on information produced by F and vice-versa. The one that runs first will produce bad code because it does not have the information produced by the other. Note the F metacode produces code without embedded metacode, as there is no dashed arrow coming out of it. The problem CircularDependency is an extended version of OrderMatters in which there is no adequate order of annotations.

We will give an example of *circular dependence* involving two annotations, `rr` and `ss` (the metacode here are annotations). `rr` is in the source code of prototype P that does not declare any fields (instance variables). This annotation generates

[3]A vertex that has an edge to itself is a cycle. There is a problem only if the metacode was not correctly implemented. If it was, the metacode would consider the consequences of the code it produces in itself. Thus, we consider that the simpleste cycle having this problem has two vertices.

```
@ss
var Int numFields = 1;
```

Field `numFields` is initialized with the number of fields of prototype `P` (including `numFields`). Annotation `ss` generates the declaration of field `fieldCount` initialized with the number of fields of `P`, which is now `2`. The resulting code produced by `rr` and `ss` is

```
@ss
var Int fieldCount = 2;
var Int numFields  = 1;
```

This is wrong because the number of fields of `P`, in the final code, is `2` and, therefore, both fields should be initialized with `2`. The problem here is that metacode associated with `rr` and `ss` depend on information, the number of fields, changed by both. Graphically, this is shown by

$$\begin{tikzcd}
\boxed{rr} \arrow[r, bend left] \arrow[r, dotted] & \boxed{ss} \arrow[l, bend left]
\end{tikzcd}$$

Note that a cycle can be more complex, like `A-C-E` of Figure 2.

Other Problems

There are other problems with metaprogramming that this paper will not further discuss. One of them is the unintended capture of identifiers by the code generated by metacode [KFFD86]. The generated code uses identifiers already in use in the environment where the code is inserted. Their semantics is equaled by accident. An example, cited by Duba et al. [KFFD86], is the macro expansion of

```
(or e1 e2)
```

made by

```
(let v e1 (if v v e2))
```

The expansion of

```
(or nil v)
```

results in

```
(let v nil (if v v v))
```

The last `v` in this expression was supposed to be different from the other occurrences of `v`. This problem is solved by renaming identifiers. When the metacode is a Lisp-like macro, this is called *macro hygiene*.

The language may not enforce a direct link between annotations in the program and the metacode. In Java [Dar06], annotation processors (AP) are passed in the compiler command line. They can do checks in the program. Each annotation in the program is passed to each AP. If the processing method of one AP returns `false`, the annotation is passed to the next annotation processor. Therefore there is not a hard link between annotations and APs, making it difficult to associate semantics to an annotation.

A sufficiently powerful metaprogramming system permits deep changes in the program by the metaprogram. For example, the metaprogram can remove methods and fields, alter 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. Profound changes make code unreadable. The developer has to know the meaning of every metacode in a source file before assuring very basic facts about that code. Therefore, the developer cannot be sure that a method that is in the source code does exist in the final code, produced after all changes made by the metacode. She or he cannot be sure a statement in a method will remain there in the final code.

Few metacode need to do deep changes, in our own experience. And they can be properly documented, lessening their impact on code readability. An example of a powerful Metaobject Protocol is that of CLOS [KdRB91], created to simulate several Lisp dialects. In this paper, we do not consider a *too powerful metaprogramming system* as a problem but as a *design decision* that, if abused, can bring problems.

3 The Cyan Language

Cyan is a statically-typed prototype-based object-oriented language. A *prototype* is a template from which other objects may be created, the same role played by *classes* in Java [GJS+14], C++ [Str13], C♯ [Csh24], and Smalltalk [GR83]. The difference is that the prototype itself is an object like any other if it declares a constructor without parameters. This restriction is unique to Cyan and it is requirely to properly initialize the prototype fields, made with the no-argument constructor. In all other prototype-based languages, any prototype is an object.

The look and feel of Cyan is that of a class-based language. The compiler translates Cyan to non-legible Java code. Thus, many language constructs are directly translated into Java, as inheritance, method overriding, message passing, assignment, and prototype declaration (each prototype is translated to a Java class). The two languages interoperate: Cyan code can import Java packages and classes and vice-versa.

The basic types of Cyan are `Short`, `Int`, `Long`, `Char`, `Boolean`, `Float`, `Double`, and `String`. They are all *reference types*: their instances are dynamically-allocated and variables *refer* to the objects. All basic types inherit from `Any`, the top-level prototype.

Listing 1 shows the declaration of prototype `Student` of package `university`. A *package* in Cyan is a named set of prototypes and has no important conceptual differences from Java packages. A *compilation unit* is a single source file composed of *import declarations*, with the imported packages for this file, and a single prototype. The file name is composed of the prototype name and extension "`cyan`". Fields that can change their values are declared with keyword `var`, as `number` in the example. The type precedes the field name. Read-only fields are declared with `let`, as `name`. If neither `var` or `let` is used, `let` is assumed. Fields are always private to their prototypes.

`Student` is declared with word `open` before `object`. Without `open`, it would be a *final* prototype, it could not be inherited. Inheritance is done with the *Cyan keyword* `extends`. A prototype that does not explicitly inherits from any other prototype inherits from `Any`, the superprototype of every other prototype but `Nil`. Hence, `Nil` cannot be assigned to a variable whose type is `Student`. There is no sub or superprototype of `Nil`, which means there is no type relationship

Listing 1. Prototype Student

```
package university

open
object Student
    let String name
    var Int number

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

    func getName -> String = name;

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

end
```

between Any and Nil. A variable that can be either a non-Nil value of type T or Nil should have type T|Nil as in

```
    var Student|Nil s;
```

T|Nil is a *union type*. Variable s can receive objects of Student or Nil in assignments. There is a type-case statement to safely retrieve the value of s.

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

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

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

This is a single method with three *keywords*, add:, at:, and doc:.

A *unary message passing* is composed of a *receiver* and an identifier that should be the name of a *unary method*:

```
aStudent getName
```

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

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

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

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

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

A Student object is created as

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

or just Student("Newton", 1).

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

```
var m = IMachine;
    // throws an exception at runtime
m turnOn;
```

It is illegal to send a message to an interface like "IMachine turnOn".

Cyan supports *gradual typing* [ST07]. The type of a method parameter, if omitted, is considered to be Dyn, a *virtual type* supertype of every other type. The compiler does not check a message passing if the receiver type is Dyn. Therefore, if every variable, field, and method return type is Dyn, Cyan becomes a dynamically-typed language. Whenever a non-Dyn type is expected and a Dyn expression is supplied, the compiler inserts a runtime type check. A *message keyword* preceded by ? disables the compiler typechecker. Thus, "obj ?getName" typechecks and method getName will be searched for only at runtime. An *anonymous function* is a nameless literal function. This is the same concept of Smalltalk *blocks*, Lisp *lambda expression*, and Java *lambda abstraction*. An *anonymous function* has the type

Listing 2. The generic prototype with varying number of parameters Tuple

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

```
Function<T1, T2, ..., Tn, R>
```

in which the Ti's are the parameter types and R is the return type. This is the syntax of the instantiation of a *generic prototype* Function with the real parameters Ti's and R.

Generic prototypes in Cyan take one or more parameters.

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

T is a *generic parameter* used, inside GroupList, in any place a type is expected: as the type of variables, parameters, fields, return type of methods, and inside expressions.[4] A generic prototype is *instantiated* when real arguments are supplied to it:

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

Assume GroupElem is a prototype. The *instantiation* is the process of creating a new prototype by replacing, textually, the *generic parameters* by the real arguments. In this example, T is textually replaced by GroupElem. A new source file is created and compiled. Therefore, the semantics of Cyan *generic prototypes* is close to *class templates* of C++.

The Cyan compiler parses the generic prototype GroupList<T>, before any instantiations, but it is unable to do the semantic analysis because T is an unknown type. Therefore, semantic errors are possible after an instantiation like GroupList<GroupElem> when the newly-created source file is fully compiled.

As an example of error, suppose a local variable p, inside GroupList, has type T replaced by GroupElem in the instantiation GroupList<GroupElem>. A message passing p getName causes a compilation error if GroupElem does not define a unary method getName. This kind of error is difficult to understand if the user of the generic prototype is not the same developer that instantiates it.

A generic prototype with a varying number of generic parameters has just one parameter followed by +, as shown in Listing 2. This is the real code of prototype Tuple. Parameter T cannot be used inside the prototype using the Cyan syntax. But the metaobject associated with annotation createTuple generates code using the real arguments. The language supports literals of arrays, tuples, and maps:

```
var Array<Int> primes = [ 2, 3, 5 ];
var Tuple<String, Int> nameAge = [. "Newton", 85 .];
var IMap<String, Int> map = [ "one" -> 1, "two" -> 2 ];
```

Cyan is similar to statically-typed class-based languages and to the statically-typed prototype-based language Omega [Bla94]. Thus, at runtime, there is no structural reflection: methods

[4]T can be used in other places like method keywords but this is irrelevant in this paper.

Listing 3. Prototype Person that uses metaobject annotations

```
package human

@init(name)
object Person
    @property var String name
    func test {
        let Array<String> list = @compilationInfo("field list");
        list println;
    }
end
```

and fields cannot be added to a prototype at runtime, inheritance cannot be changed, and so on. This is unlike other dynamically-typed prototype-based languages like Self [US87]. The preferred way of creating an object in Cyan is using method `new`, not to call the `clone` method.

There are other features of the Cyan language that are not presented in this section: anonymous functions, the exception handling system (made only with message passing), safe object initialization (fields are initialized before used, except in a few circumstances), and a generalization of anonymous functions called *context objects*.

4 The Cyan Metaobject Protocol

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

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

4.1 A Complete Example Explained

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

Listing 4. Prototype Student

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

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

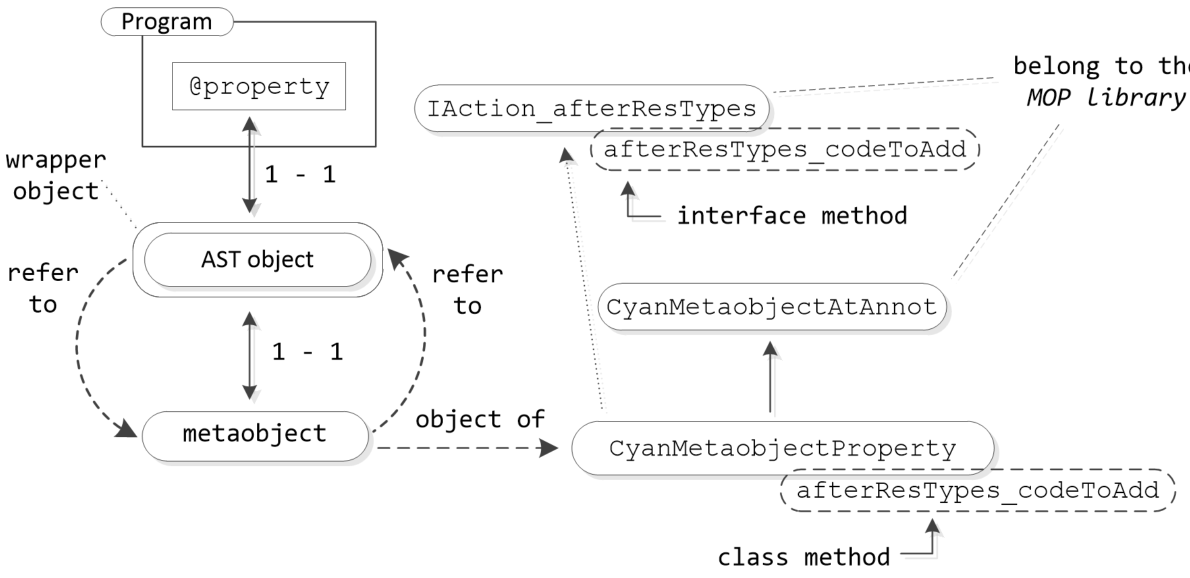

Fig. 3. Relation between metaobjects, annotations, metaobject classes, MOP interfaces, and compilation phases

Basic values (3, `3.14`, `'A'`), literal arrays, literal tuples, literal maps, and any combination of these can be parameters to annotations. An annotation may be followed by a text given between two delimiters, as shown in Listing 4. This text will be called *attached text* or *attached DSL*[5] *code*. In this example, the *attached DSL code* of doc and `replaceCallBy` are documentation in English and an expression, respectively. There are many possible variations of delimiters, described by Guimarães [?], so that the DSL and the delimiters do not clash. As a rule, the right delimiter should mirror the left one. Annotations `replaceCallBy` and doc are attached to method "`twice:`". `replaceCallBy` takes one parameter.

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 that inherits from prototype `CyanMetaobjectAtAnnot`.[6] We will soon describe how the compiler links an annotation to the metaobject prototype in Cyan. A metaobject can also be implemented in Java. In this case, it will be an object of a *Java class*. Throughout this paper, we will refer to both "prototype of a

[5]Domain Specific Language
[6]Suffix "AtAnnot" means "annotation that starts with an @". Some annotations do not use this syntax, but they are not described in this paper.

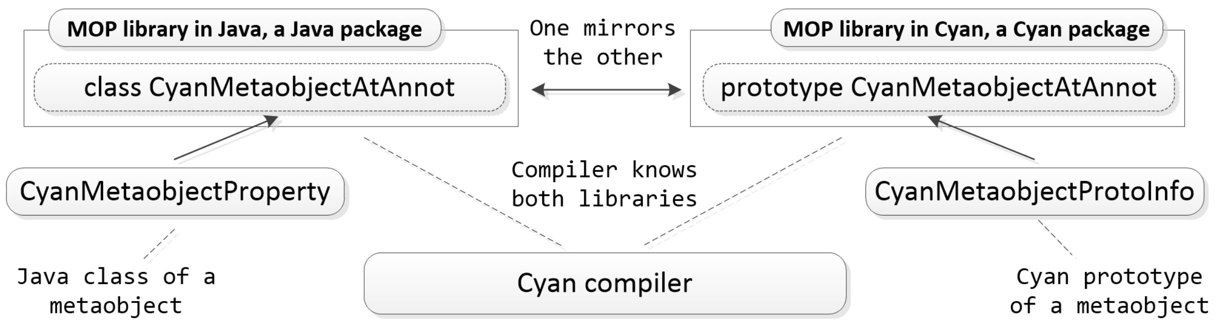
MOP library in Java, a Java package
class CyanMetaobjectAtAnnot
One mirrors the other
MOP library in Cyan, a Cyan package
prototype CyanMetaobjectAtAnnot
CyanMetaobjectProperty
Compiler knows both libraries
CyanMetaobjectProtoInfo
Java class of a metaobject
Cyan compiler
Cyan prototype of a metaobject

Fig. 4. The relation to the compiler to the MOP libraries

metaobject" and "class of a metaobject" to indicate that a metaobject can be implemented in either Cyan or Java.

Figure 3 shows the relationships among all of the previously explained elements. On the left, there is an "AST object" representing the annotation. It is wrapped by another object, which is represented by a rectangle with round borders around another rectangle. The inner AST object is used by the compiler and the outer, by the MOP. Both are called "AST objects". The outer one is a read-only wrapper object. This figure also shows that the AST object and the metaobject refer to each other. There are two one-to-one relationships: between metaobjects and wrapper objects and between wrapper objects and their annotations. `CyanMetaobjectProperty` is the Java class of the metaobject associated with annotation `property`. This class can be used to create any number of metaobjects. It inherits from `CyanMetaobjectAtAnnot` and implements interface `IAction_afterResTypes`, overriding the interface method `afterResTypes_codeToAdd`. This is the method that creates the get and set methods for field `name`. Although `CyanMetaobjectProperty` could be user-defined, it is supplied with the Cyan basic libraries. A metaobject prototype should inherit from `CyanMetaobjectAtAnnot`. Appendix A shows the complete code, in Cyan, of metaobject `myproperty` that is a simplified version of `property`.

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 is no confusion.

A Cyan *package* contains one or more prototypes in a directory with the name of the package. A special subdirectory of the package may contain the compiled version of a Java metaobject class or a Cyan metaobject prototype — a "`.class`" file. Whenever the package is imported, the annotations associated with the package's metaobject classes or prototypes can be used in the source file. Every metaobject class or prototype has a `getName` method that returns the annotation name. This links the annotation to the metaobject class or prototype. All the metaobjects used in Listing 3 belong to package `cyan.lang` imported automatically by every Cyan source file.

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

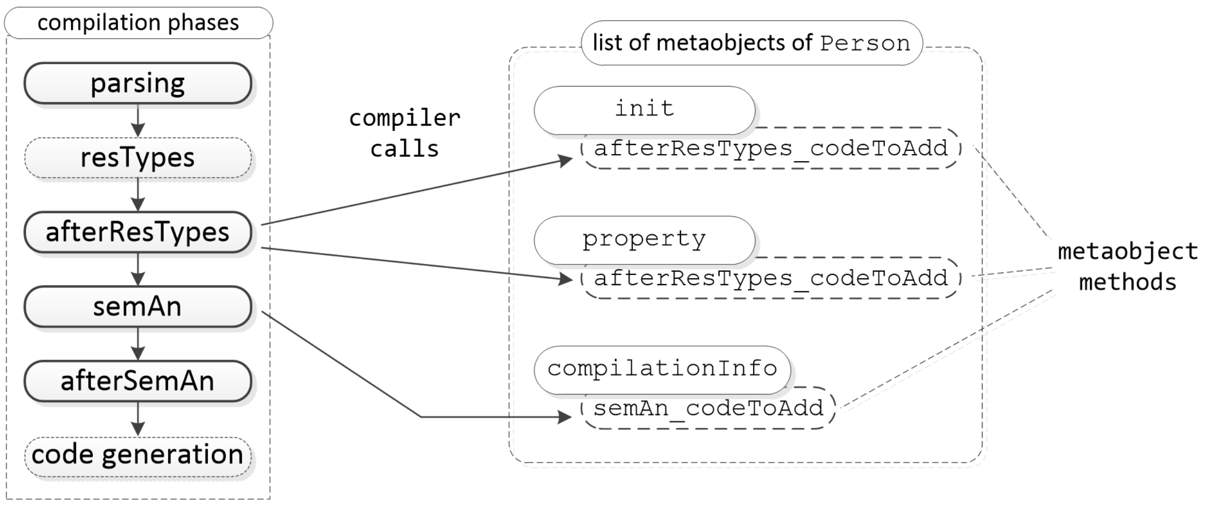

Fig. 5. The compilation phases and their links to methods of metaobjects at compile-time

used for building metaobject prototypes (classes). Prototypes `CyanMetaobjectAtAnnot` and `IAction_afterResTypes` belong to the Cyan *MOP library*. There are classes in the Java *MOP library* with these same names. If a metaobject is implemented in Java, there is a Java class for it and therefore the Java package representing the *MOP library* is used. The same applies to Cyan. The Cyan *MOP library* mirrors the Java *MOP library* as shown in Figure 4. The compiler knows and is capable of interacting with the two libraries.

The Cyan compiler goes through six compilation phases for each source file, shown inside the rectangle with dashed lines in the left of Figure 5. The flow of control is from the top to bottom. Phase **parsing** does the syntactical analysis and builds the Abstract Syntax Tree (AST) of the source file. Some AST objects are associated with a type and have a `type` field, initially set to `null`. For example, AST objects representing method parameters, prototype fields, implemented interfaces, the superprototype, message passings, and expressions have a `type` field.

There are two kinds of AST objects associated with types: those representing expressions, which are always inside method bodies, and those outside method bodies. The `type` field of the later AST objects is set in phase **resTypes** (resolving types). Thus, field `name` of `Person` of Listing 3 is represented by an AST object whose field `type` is `null` at the beginning of phase resTypes. During this phase, the compiler sets the `type` field to the AST object representing the prototype `String`. Phase resTypes, therefore, does part of the *semantic analysis* of the source code. The compiler goes through phase resTypes on a source file only after parsing all source files referenced in this file or loading the jar file with the referenced prototypes.

Phase **afterResTypes** means *after resolving types*, which is used only by the Cyan MOP. Some methods of metaobjects are called in this phase. For example, the method of metaobject `property` that adds the get and set methods to the prototype. AST objects that represent expressions in the Cyan code have a `type` field set in phase **semAn** (semantic analysis), which is the remainder of the *semantic analysis*. The compiler also does the remainder checks, demanded by the language, in this phase. Phase **afterSemAn** means *after semantic analysis*. It is

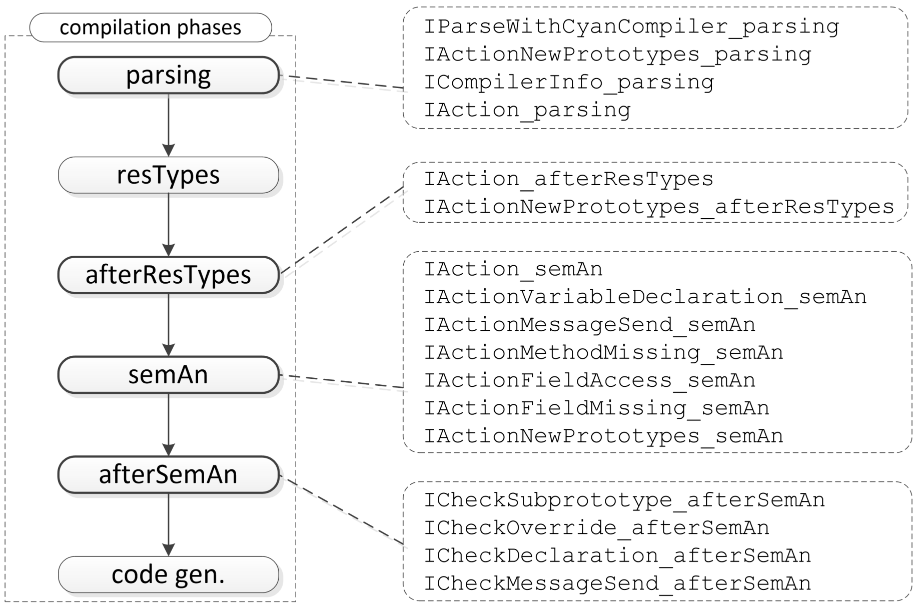
compilation phases
parsing
resTypes
afterResTypes
semAn
afterSemAn
code gen.
IParseWithCyanCompiler_parsing
IActionNewPrototypes_parsing
ICompilerInfo_parsing
IAction_parsing
IAction_afterResTypes
IActionNewPrototypes_afterResTypes
IAction_semAn
IActionVariableDeclaration_semAn
IActionMessageSend_semAn
IActionMethodMissing_semAn
IActionFieldAccess_semAn
IActionFieldMissing_semAn
IActionNewPrototypes_semAn
ICheckSubprototype_afterSemAn
ICheckOverride_afterSemAn
ICheckDeclaration_afterSemAn
ICheckMessageSend_afterSemAn

Fig. 6. The compilation phases and their links to the interfaces of the *MOP library*

used only by the MOP: some metaobject methods are called in this phase. The last compilation phase is code generation. Currently, no metaobject method is called in this phase.

The non-dashed rectangles of Figure 5 represent phases parsing, afterResTypes, semAn, and afterSemAn associated with interfaces of the *MOP library*. The interfaces associated with each phase are depicted in Figure 6. A phase is associated with multiple interfaces but each interface is associated with just one phase. The interface name ends with the phase it is associated with.

During the parsing of prototype `Person` of Listing 3, the compiler creates a metaobject for each annotation. Then, in each of the phases parsing, afterResTypes, semAn, and afterSemAn, the compiler calls all of the metaobject methods, of all metaobjects, declared in interfaces of that phase. In the `Person` example, the dashed rectangle in the right of Figure 5 shows a list of metaobjects created for this prototype. There are three metaobjects represented by rectangles with the annotation name (`init`, `property`, and `compilationInfo`). In the compilation phase afterResTypes, shown in the left, the compiler calls methods `afterResTypes_codeToAdd` of metaobjects `init` and `property`. Method `afterResTypes_codeToAdd` is declared in interface `IAction_afterResTypes`. In the same way, the compiler calls method `semAn_codeToAdd` in phase semAn.

There is a missing point: at which point in each compilation phase does the compiler call each method? It depends on the interface the method is declared. Some interfaces are associated with *triggers*. For example, methods of

```
IActionMethodMissing_semAn
```

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

Listing 5. Prototype Person that uses metaobject annotations

```
package human

object Person
    var String name
    func test {
        let Array<String> list = [ "name" ];
        list println;
    }
    func getName -> String = name;
    func setName: String name { self.name = name }
    func init: String name {
        self.name = name
    }
end
```

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 3, metaobject `property` generates methods "`getName`" and "`setName:`". Metaobject `init` generates code for a method "`init:`", a *constructor*, setting field `name`. The compiler inserts the code generated by the two metaobjects in the `Person` prototype that goes through the parsing and resTypes phases again. Phase afterResTypes is skipped, the compilation proceeds to phase semAn. Then, metaobject `compilationInfo` generates a string whose contents is

```
    [ "name" ]
```

This is the code of a literal string array with one element: the name of the `Person` field. The compiler inserts the code produced just after the annotation. The resulting prototype `Person` is equivalent to the one shown in Listing 5. It is not exactly equal because some auxiliary annotations introduced by the compiler are not shown.

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

4.2 The Interfaces of the MOP Library

This subsection explains how metaobjects direct the compilation of Cyan code. That is, how the compiler chooses methods of metaobject classes/prototypes to be called at specific phases of the compilation. Although metaobjects can be implemented in either Java or Cyan, this subsection assumes they are implemented in Cyan. Therefore, the *MOP library* used is composed of *prototypes*, including `CyanMetaobjectAtAnnot` and the Cyan interfaces of Figure 6.

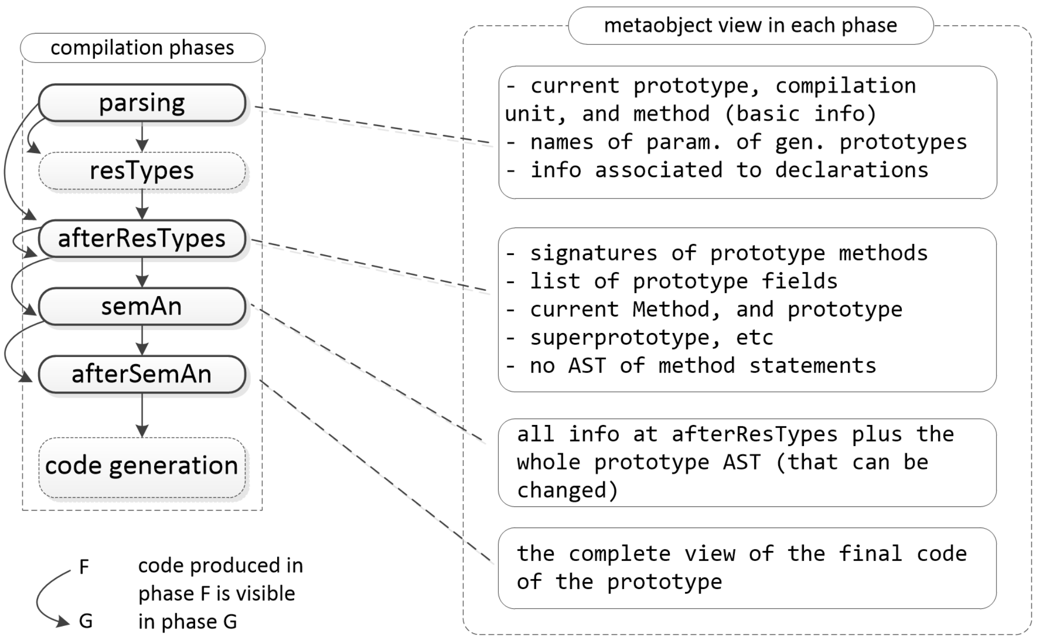
compilation phases
parsing
resTypes
afterResTypes
semAn
afterSemAn
code generation
F
G
code produced in phase F is visible in phase G
metaobject view in each phase
- current prototype, compilation unit, and method (basic info)
- names of param. of gen. prototypes
- info associated to declarations
- signatures of prototype methods
- list of prototype fields
- current Method, and prototype
- superprototype, etc
- no AST of method statements
all info at afterResTypes plus the whole prototype AST (that can be changed)
the complete view of the final code of the prototype

Fig. 7. The information available in each compilation phase

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

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

The design of a metaobject prototype starts by defining its goals. Then, the programmer chooses one or more interfaces to satisfy these goals. If the metaobject should add fields and methods to the *current prototype*, it should implement interface `IAction_afterResTypes`. If it should intercept access to a prototype field, the metaobject prototype should implement interface `IActionFieldAccess_semAn`. And so on. This is very important: the functionality of a metaobject prototype is driven by the interfaces it implements. In some other languages, a metacode decides what to do during its execution, which is at compile-time of the base program. This is more prone to errors since what to do depends on runtime decisions.

The metaobject prototype should override the interfaces' methods. These methods may need information on:

(a) the annotation. A method inherited from `CyanMetaobjectAtAnnot` returns the AST object representing the annotation (see the right arrow labeled "refer to" in Figure 3). Through this object, one can retrieve all information related to the annotation: its parameters, the attached DSL code, and the declaration it is attached to (as the AST object of `Person` for annotation `init`). The annotation AST object also has a method returning the metaobject (the arrow "refer to" in the left of the figure);

(b) the environment in which the metaobject annotation is. The information is encapsulated in a parameter passed to each method overridden from the interfaces. Through this parameter, the metaobject method discovers the details of the Cyan source code in which the annotation is: the current method, the current prototype, the imported packages, the AST of the current prototype or method, and so on. The type of the parameter with this information varies according to the phase of the corresponding interface. The parameter type restricts the amount of information available. For example, the AST of method statements is not available in methods declared in `IAction_afterResTypes`. But it is in methods of `IAction_semAn`. The most important information a metaobject has in each compilation phase is shown in figure Figure 7.

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

4.2.1 Interfaces for Creation of New Prototypes. There are three interfaces to create new prototypes during phases parsing, afterResTypes, and semAn (the interface names are composed of `IActionNewPrototypes_` and the phase name). In each interface, the sole interface method, when overridden in the metaobject prototype, should return a tuple consisting of a prototype name and code. Both as strings. The new prototype is created in the same package as the prototype in which the annotation is.

Why do three interfaces for prototype creation are needed? Why not just one? There are two reasons: (a) the semantic analysis is made in phases, first resTypes (outside method statements) and then semAn (method statements); (b) The latter phases provide more information than the former ones. Therefore, a prototype used, for example, as the type of a method parameter should exist in phase resTypes. It either exists in the original program or is created by a metaobject in phase parsing. Consider a prototype that is the type of a local variable whose declaration does exist at the start of phase semAn. That is, the local variable declaration was not added by metaobjects acting in phase semAn. This prototype should exist at the start of phase semAn. It either exists in the original program or it was created in phases parsing or afterResTypes.

A local variable declaration could have been created by metaobjects acting in phase semAn. If the variable type (a prototype) did not exist before phase semAn, it can be created in this phase. Ideally, a prototype is created as late as possible because more information is available in later phases. However, it may be necessary to create it earlier because it is used in early compilation phases.

4.2.2 Interfaces of Phase parsing. The interfaces of this phase are used to parse the *attached DSL code* of an annotation, generate code after the annotation, and pass information, like documentation, from the annotations to declarations.

Interface `IParseWithCyanCompiler_parsing` has a method with a parameter that is a restricted view of the Cyan compiler. The parameter has methods for lexical analysis and parsing of Cyan types, expressions, and statements. It is the ideal tool to use when the attached DSL code is similar to Cyan. Interface `ICompilerInfo_parsing` is used to pass information, like documentation, from the annotations to declarations. As an example, metaobject `doc`, cited in subsection 4.1, uses this interface. Interface `IAction_parsing` declares a method to add code after the annotation (it will be removed in the next MOP version).

4.2.3 Interfaces of Phase afterResTypes. Interface `IAction_afterResTypes` declares four methods. One is used to add statements at the beginning of methods of the *current prototype*.

Another method is used to rename methods.

It should return a list of tuples, each one with the old name and the new name.

However, whenever a method is renamed, the metaobjects should add another *base method* with a name equal to the old *base method* name. This is to prevent difficult-to-understand compilation errors caused by the renaming of methods.

The other two methods of `IAction_afterResTypes` need the concept of *signature*. A *method signature* is the method declaration without its body. Parameter names are optional. Hence, the signatures of the method `getName` of `Student` (Listing 1) and method `add:at:doc:` of section 3 are

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

The *signature of a field* is composed of its declaration, preceded by `var` or `let`, without the optional expression assigned to it. For example, a *field signature* can be “`var String name`”.

Method `afterResTypes_codeToAdd:` of interface `IAction_afterResTypes` returns a tuple composed of the code of *base fields and methods* (to be added to the *current prototype*) and the signatures of these *base fields and methods* (separated by “`;`”).

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

The first parameter is a restricted version of the compiler. It has methods to return the *current prototype, the current compilation unit*, methods and fields of the current prototype, and so on. The second parameter, `infoList`, is an array of tuples, each one composed of an annotation and an array of *base method and field signatures.*[7] The compiler passes an empty array, as the second argument, in the first time this *metamethod* is called. This parameter is only useful when the metaobject depends on code generated by other metaobjects. That is, the code generated by other metaobjects acting in the same prototype should change the code generated by the metaobject. An example will be used to explain that.

[7]A *slot* is either a method or a field. Interface `ISlotSignature` represents a field or method signature.

Metaobject `addFieldInfo` adds to the current prototype a field whose name is the first parameter, initialized with the number of prototype fields. The metaobject class[8] implements interface `IAction_afterResTypes` and defines a method `afterResTypes_codeToAdd`. In prototype `TestField`, there are two annotations `addFieldInfo`.

```
@addFieldInfo(fieldNum)
@addFieldInfo(numOfFields)
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 `afterResTypes_codeToAdd` of each of the metaobjects passing an empty array as the second argument, `infoList`. Both methods return 2 because they view the original prototype, without *base fields and methods* added by other metaobjects.

To work correctly, the metaobject class should define a method `runUntilFixedPoint` returning `true`. This method and `afterResTypes_codeToAdd:` act together. If `runUntilFixedPoint` returns `false`, then the compiler calls `afterResTypes_codeToAdd:` just one time. Otherwise, this method is called multiple times according to the algorithm `FixMeta` of Listing 6.

This algorithm takes two lists as input. The first, `fullList` is the list of all metaobjects of the *current prototype* that implement `IAction_afterResTypes`. The second, `roundList`, is a sublist of the first containing the metaobjects whose methods `runUntilFixedPoint` return `true`. The algorithm first collects all *base method and field* signatures generated by all metaobjects into a list `infoList`. This is made by the `for` statement of lines 3-9. Then it makes *rounds* of calls (lines 12-30), each *round* composed of calls to method `afterResTypes_codeToAdd:` of all metaobjects of `roundList` (lines 16-26). The *rounds* end when all metaobjects produce the same code as in the previous round or the number of rounds is greater than 5. In the later case, the compiler issues an error because the metaobjects were not able to reach an agreement in five rounds. This number can be changed by a compiler option.

The flow of control of algorithm `FixMeta` when used with two metaobjects is shown in Figure 8. The output of one round, the dashed rectangle with the signatures, is given as input to all methods called in the next round. Each `afterResTypes_codeToAdd:` method also generates code (base fields and methods) that will be added to the *current prototype* after phase afterResTypes.

Let us return to the `addFieldInfo` example. Method `runUntilFixedPoint` of the metaobjects returns `true`. Because of this, the compiler calls all methods `afterResTypes_codeToAdd:` again.

[8]This metaobject is implemented in Java.

Listing 6. Algorithm FixMeta

```
Algorithm FixMeta(fullList, roundList)
    infoList = empty list
    for every metaobject in fullList {
        call method afterResTypes_codeToAdd of the metaobject
           passing an empty array as the second parameter.
           This method returns a tuple. Add the slot
           signatures of the second tuple element to
           list infoList.
    }
    somethingChanged = true;
    count = 1;
    while somethingChanged {
        newInfoList = empty list
        // each 'while' loop is a round
        somethingChanged = false;
        for every metaobject in roundList {
            call method afterResTypes_codeToAdd of the metaobject
               passing infoList as the second parameter.
               This method returns a tuple. Add the slot
               signatures of the second tuple element to a
               list newInfoList.
            if the code produced by this call are
            different from the code produced by the
            same metaobject in the previous round, stored
            in infoList, set somethingChanged to true
        }
        infoList = newInfoList;
        ++count;
        if count > maxNumRoundsFixMetaDefaultValue { issue error; }
    }
```

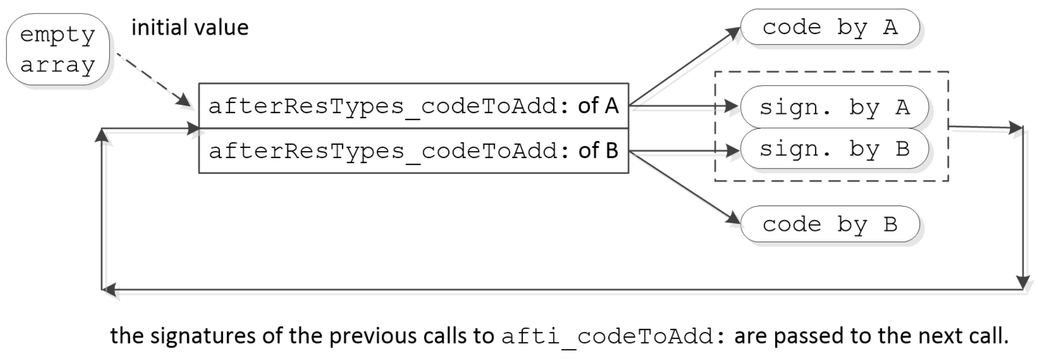
empty array
initial value
afterResTypes_codeToAdd: of A
afterResTypes_codeToAdd: of B
code by A
sign. by A
sign. by B
code by B
the signatures of the previous calls to afti_codeToAdd: are passed to the next call.

Fig. 8. Flow of control in algorithm **FixMeta** with two metaobjects

In this round, parameter `infoList` is an array of tuples, each one composed by the AST object of an annotation and a list of base field and method signatures. The metaobject associated with the annotation, first tuple element, generated the base 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 `numOfFields`.

In this second round of calls to all `afterResTypes_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 `afterResTypes_codeToAdd` methods. Now each method returns exactly the same code as in the second round. Because the generated code is equal, the algorithm `FixMeta` ends. The compiler makes a new round of calls even if just only one metaobject returns a code different from that of the previous round.

The Cyan compiler checks whether the elements of the tuple returned by

    afterResTypes_codeToAdd:

match. Thus, the compiler checks if the base fields and methods of the first tuple element are in the second tuple element and vice-versa. Method `afterResTypes_codeToAdd:` should return a tuple with empty strings, if used only for checks.

4.2.4 Interfaces of Phase semAn. The sole method of interface `IAction_semAn` returns a string with code to be added after the annotation. If used only for checks, the method should return an empty string. Some annotations are expressions, like `compilationInfo` of Listing 3. Their associated metaobject prototypes should inherit from `IAction_semAn`.

Annotations associated with metaobject prototypes implementing interface

    IActionVariableDeclaration_semAn

should be attached to local variable declarations. The sole interface method adds code after the variable declaration and has access to the variable name, type, and expression used to initialize the variable (if any).

Message passings are intercepted by metaobjects that implement interface

    IActionMessageSend_semAn

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

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

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

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

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

When the compiler analyzes a message passing in phase semAn, it first finds the type of the receiver expression. Suppose it is `T`. If there is no adequate method for the message passing, the compiler collects into a list all metaobjects that implement interface `IActionMethodMissing_semAn` and whose annotations are in prototype `T`. This interface declares two metamethods, one for unary and the other for keyword messages. Then, a metamethod of each metaobject is called. The metamethod is declared in the interface and overridden in the metaobject prototype.

Each metamethod called may return code (an expression), as a string, to replace the original message passing. Again, there are three possibilities based on the number of metaobject methods returning a non-empty code string:

(1) more than one. The compiler issues an error;
(2) exactly one. The returned code replaces the message passing. This replacement is visible in the next compilation phase, afterSemAn;
(3) none. The same algorithm is applied to the superprototype of `T` (implemented interfaces are not taken into account).

Metaobjects whose prototypes implement interface `IActionFieldAccess_semAn` intercept field access. This interface has a method called when the value of the field is retrieved and another one called when the field is set. Each metamethod returns code that replaces the get and set of the field. The annotation should be attached to the prototype field whose access is intercepted.

There may be more than one annotation attached to a field whose associated metaobject prototype implements interface `IActionFieldAccess_semAn`. In this case, the compiler calls all appropriate metamethods. If more than one metamethod returns a non-empty code string, the compiler issues an error.

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

In phase semAn, the Cyan compiler resolve types in a method's body in the textual order of statement declarations. The metaobject associated with an annotation has access to the types resolved in lines that come before the annotation. This information can be used for checks or code generation.

Metaobjects have access to Abstract Syntax Tree (AST) objects from parameters passed to overridden interface methods and from the associated annotations. By calling methods of AST objects, metaobjects have access to information on the current prototype, method, etc. AST objects can be visited using the *Visitor* Design Pattern [GHJV95]. Every metaobject has a method `replaceStatementByCode` for replacing a statement (which includes expressions) by a code given as string. The statement is given as an AST object. As an example of the use of this metamethod, the demonstration metaobject `shout` visits the AST objects of the current prototype and replaces all strings by the equivalent ones in upper case. Because of the safety features of the Cyan compiler, `replaceStatementByCode` can only be used to replace statements of the *current prototype*.

4.2.5 Interfaces of Phase afterSemAn. The compilation phase afterSemAn comes after semAn. In it, no code can be changed. Therefore, this is the ideal phase for checks because they will not be invalidated by metaobjects that change the code already checked.

Inheritance should be planned [Blo18] 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 [Sny86] — the subprototype designer should know internal details of the superprototype methods.

Interface `ICheckSubprototype_afterSemAn` is used to partially solve this problem.

Annotations of metaobjects whose prototypes implement interface
`ICheckSubprototype_afterSemAn`
can only be attached to prototypes. Whenever the prototype is inherited, even indirectly,[9] the sole method of this interface is called. That is, the compiler calls the sole metamethod of the interface overridden in the metaobject prototype. A parameter of the metamethod is the AST object representing the subprototype inheriting the prototype with the annotation. Through it, the metaobject can do checks. For example, the metaobject may require that an interface be implemented only by prototypes that also inherit from another class, as done by a feature of language Hack [Hac20].

A metaobject annotation whose prototype implements interface
    `ICheckOverride_afterSemAn`
can only be attached to a base method. Whenever the base method is overridden, even in a sub-subprototype, the compiler calls the metaobject metamethod that overrides the sole `ICheckOverride_afterSemAn` method. The compiler passes, as an argument to this metamethod, the AST object of the subprototype base method.

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

[9]If prototype `C` inherits from `B` that inherits from `A`, thus `C` indirectly inherits from `A`.

Message passings can be checked in phase semAn using interface

`IActionMessageSend_semAn`

However, this is flawed because metaobjects can introduce new code in this same phase and this new code will not be checked by the metaobject whose prototype implements `IActionMessageSend_semAn`. The correct procedure for message passing checks is to make the metaobject prototype implement interface `ICheckMessageSend_afterSemAn`. The metaobject methods are called as described for interface `IActionMessageSend_semAn` except that methods of all metaobjects are called, including those of superprototypes.

4.2.6 Interface for Metaobject Communication at Compile-Time. Metaobjects of the same prototype whose prototypes implement interface

`ICommunicateInPrototype_afterResTypes_semAn_afterSemAn`

can communicate before phases afterResTypes, semAn, and afterSemAn. 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. Then, by calling a metaobject method, overridden from the interface, it collects the objects each of these metaobjects want to share. After that, the compiler calls another metaobject method with the list of shared objects.

4.3 The Cyan MOP and the Problems with Metaprogramming

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

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

Metaobjects can create new prototypes, each one in a new source file. This does not cause this problem because it only occurs when there are two or more source files created by the developer and one changes the other.

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

A metaobject that does not implement the interfaces cited in the previous paragraph can only replace or add code to its associated prototype. That is, a metaobject whose annotation is inside a prototype can only replace or add code to this prototype. This is assured by several mechanisms:

(a) the AST is read-only. Therefore, even if a metaobject has a reference to the AST object representing a prototype that is in another source file, it cannot change it;
(b) metaobjects define methods that return source code as strings. For example, method `afterResTypes_codeToAdd:` declared in interface `IAction_afterResTypes` return code

of fields and methods to be added to the current prototype. There is no method in any interface to add code to an external prototype;

(c) method `replaceStatementByCode`, described in subsection 4.2.4, asks the compiler to replace a statement by a code given as a string. The statement is an AST object. This method can only change the *current prototype* because there is no way of a metaobject whose current prototype is `P` has a reference to an AST object representing a statement that is in another prototype `Q`. That happens because some methods of the AST classes have security checks that prevent access to private parts of other prototypes.

As an example, suppose the metaobject inside `P` has a field whose type is `Q`. Therefore it can ask for the field type, resulting in the `Q` AST object. The metaobject can now ask for the AST object of a public method of `Q`. This is fine. There is a method in the AST class representing a *Cyan method* called `getStatementList`. If the metaobject in `P` calls this method on the `Q` public method, it throws an exception. At the start of `getStatementList` there is a check that compares the prototype of the *Cyan method*, `Q`, with the prototype in which the metaobject annotation is, `P`. Since they are different, this method throws an exception. Note that is an exception thrown at compile-time of the Cyan program being compiled and at runtime of the metaprogram.

There is no need to put this type of test on all AST methods. For example, there are no checks in method `getStatementList` of the AST class of Cyan statement `while`. If a metaobject has a reference to an AST object representing a `while` statement, it has already passed a check previously.

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

Suppose a metaobject annotation is inside prototype `P`. Whenever the metaobject gets a reference to another prototype, the compiler adds the dependency from `P` to the prototype in a *dependency table.*[10] This is made with checks at the start of several methods of the *MOP library*, including AST methods.

For example, an environment object is passed as an argument to some metaobject methods. This object has a method that searches for a prototype given its name as a string. If prototype `Q` is found, the compiler adds the dependency from `P`, the prototype in which the metaobject annotation is, to `Q`. As another example, the metaobject inside `P` may walk in its AST and, after calling several AST methods, get a reference to another prototype `R`. The compiler will add the dependency to the table. All AST methods that are related to *dependencies* between prototypes have statements to add entries to the *dependency table.*

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

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

[10]Currently, this table is not used by the compiler. It will be in future versions.

(a) methods of the AST return more or less information according to who is asking for it. The information degree varies to match the *current prototype* view of the program. For example, class `WrProgramUnit` of the AST represents a prototype and declares a method `getMethodDecList` returning the list of methods of the prototype. Suppose a metaobject whose annotation is in prototype `P` sends message `getMethodDecList` to an AST object representing prototype `Q`. This method takes an argument that is a compilation environment. Through it, the method can identify prototype `P`. `getMethodDecList` returns a list of Cyan methods that includes the `Q` public methods and: (a) the `package`[11] methods of `Q` if `P` and `Q` are in the same package; (b) the `protected`[12] methods of `Q` if `P` is a subprototype of `Q`.

(b) methods of the AST throw exceptions if the metaobject is trying to retrieve private information of other prototypes; that is, a metaobject whose annotation is in `P` tries to retrieve private information of `Q`.
An example of that was given in MessWithOthers with method `getStatementList`. Another example is method `getFieldList` of `WrProgramUnit`. It also takes an argument that is a compilation environment. If a metaobject whose annotation is inside a prototypes calls this method of the AST object representing another prototype, it throws an exception.

The checks cited above are made with a compilation environment object of class `WrEnv` that is passed as an argument to metaobject methods or retrieved from other metaobject method arguments. It cannot be user-created because its constructor takes an object of a class hidden to metaprogrammers. If a developer could create an object of `WrEnv`, she or he could build it to falsify the original object. Hence, a metaobject whose annotation is inside `P` could call, without errors, method `getFieldList` of a `Q` AST object because it pretended to be inside `Q`.

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

Metaobjects do not use the Cyan compiler data structures. They use wrapped versions of these structures, including a wrapped version of the compiler AST. We consider that this problem is addressed in Cyan for several reasons:

(a) the wrapped data structures are a *simplified* version of the compiler structures. Thus, the developer does not need to know very complex structures. The wrapped AST classes mirror the language features they represent. That means they are not highly subject to change. They are modified only when the language change;

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

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

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

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

Metaobjects never replace or add code to the base program directly. They ask the compiler to do the changes in the source program. And, when the compiler does that, it keeps track of the annotation associated with the metaobject that asked for replacement or addition of code. If there is an error in the source code replaced or added by a metaobject, the compiler can point out the line and the source file of the annotation associated with the metaobject.

OrderMatters *The order metacode is called inside a source file changes metacode behavior.*

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

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

Metaobjects can generate new prototypes but these are created in a new file. Therefore, their creation order is irrelevant. If two metaobjects try to create prototypes with the same name, the compiler issues an error. The calling order of metaobject methods in phase parsing is not important for two reasons: (a) metaobjects can add code which will be visible by other metaobjects only in the next phase and (b) metaobjects can add information to declarations (such as documentation) but this data cannot be read in phase parsing.

In phase semAn, metaobjects can only add code, in the *current prototype*, after the annotation. The code added by other metaobjects in this phase will only be visible in the next phase, afterSemAn. Thus, the calling order of the methods is not important concerning code generation.

The sole method of interface `IActionVariableDeclaration_semAn` can add code after a local variable declaration. If several annotations add code to the same local variable declaration, the code added follows the textual annotation order. However, this is not a serious problem because the annotations and the added code are textually very close to each other.

Methods of interfaces `IActionMessageSend_semAn`, `IActionMethodMissing_semAn`, `IActionFieldAcces` and `IActionFieldMissing_semAn` replace a message passing or field access by some other code. The calling order is not important because at most one metaobject can replace the message passing or field access. If two or more try to do that, the compiler issues an error.

Methods of interfaces of phase afterSemAn do checks in an immutable program. Since they cannot add code, the calling order of their methods is irrelevant.

A method of interface `IAction_afterResTypes` is used to rename methods. The order of calls is not important because the compiler issues an error if two metaobjects try to rename the same base method. Another metamethod of this interface adds statements at the beginning of base methods of the current prototype. There may be two or more metaobjects that try to add statements to the same base method. In this case, the textual order is important, the statements are added in the textual order of the metaobject annotations. If necessary, a metaobject may demand it is the only one to add statements to a given method.

The other two methods of `IAction_afterResTypes`, `afterResTypes_codeToAdd:` and `runUntilFixedPoi` work together. Algorithm `FixMeta` of Listing 6 calls method `afterResTypes_codeToAdd:` in rounds. In each round, every method can view the information produced by all of the calls of the previous round. Thus, the call order is not important.

Subproblem **DifferentViews** only happens in phase semAn. In this phase, a metaobject knows the types of all expressions that come before its annotation in a method body. Hence, if there are two annotations in the same method, the one that comes textually after has more information than the first.

The information available to metaobjects, in phase parsing, cannot be changed by them and therefore all metaobjects have the same program view. In phase afterResTypes, all methods view the AST build in the previous compilation phase, resTypes. And `afterResTypes_codeToAdd:` methods of metaobjects that participate in algorithm FixMeta view also what other methods have produced in each round of the algorithm. Therefore, all of them share the same program view. Some metaobjects chose not to participate in this algorithm because the code produced by other metaobjects is unimportant for them. In phase afterSemAn, all metaobjets view the AST produced in the previous phase. Therefore, all have the same program view.

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

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

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

Nontermination *Metacode may not finish its computation.*

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

Nondeterminism *Metacode is nondeterministic.*

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

Metaobjects can read and write to files, get the current time, call a random number generator, interact with the network, and so on. That makes metaobjects nondeterministic. There is no easy way to make them deterministic. That could only be done if they use a special language in which any interactions with the world are prohibited. A restricted version of interpreted

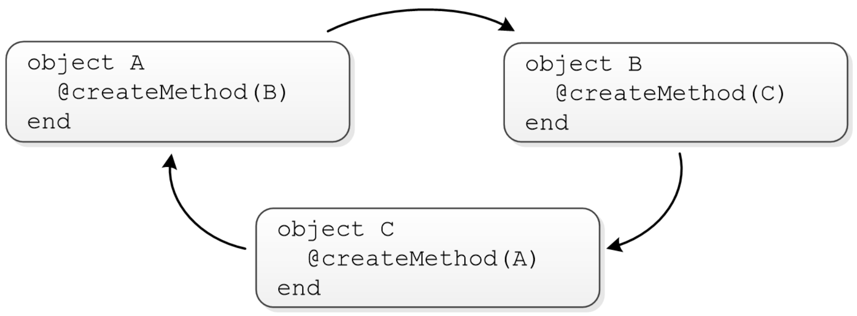

Fig. 9. Dependencies among prototypes

Cyan could be used for that. However, this would place such great limitations on metaobjects that we prefer not to use this solution.

NoGeneratedCodeGuarantees *Metacode may generate defective code.*

Metaobjects in Cyan can produce defective code. However, the compiler will point out the errors because, whenever code is added to a prototype, it is compiled again.

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

There is no way of specifying a contract between base code and metaobjects in the Cyan MOP. Note that NoContracts is similar to the problem that motivated the creation of *concepts* [GJS+06] for C++ templates (See subsection B.3). *Concepts* are predicates on generic prototype parameters. They restrict what a parameter can be, like "parameter `T` should have a unary method `init`".

A solution to problem NoContracts would be to add a concept-like DSL to specify: (a) the restriction a metaobject expect from the *current prototype* and (b) the code a metaobject should generate. This DSL will certainly be more complex than concept DSLs because the diversity of code generation and checking of metaobjects is much greater than that of generic prototypes. Thus, we have chosen not to add this contract DSL to the Cyan MOP.

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

Metaobjects cannot access any information produced by other metaobjects in phase parsing, preventing this problem from occurring. In phase afterResTypes, circular dependencies are dealt with algorithm FixMeta of Listing 6. This algorithm assures that all metaobjects that participate in the algorithm have the same information on the current prototype. However, FixMeta is not useful for some exotic metaobjects. For example, suppose a metaobject generates a field for each prototype method and another generates a method for each field. This results in an endless loop, FixMeta does not solve this problem.

FixMeta only considers information on the current prototype. There may be dependencies among metaobjects associated with annotations of different prototypes as shown in the example of Figure 9. Assume that, if annotation `createMethod(Y)` is in prototype `X`, the metaobject adds to `X` the method

```
    func numMethodsY -> Int = numMet;
```

in which `numMet` is the number of methods of Y (a literal). In the figure, an arrow from prototype X to Y means metaobject `createMethod` whose annotation is in X uses information about Y (its number of methods). There is a circular dependency among the prototypes shown in Figure 9. Each one depends on another prototype for generating code. Currently, a metaobject whose annotation is associated with a prototype has a view of other prototypes, in phase afterResTypes, that do not consider methods added in this phase. Therefore, in the current compiler version, the `createMethod` metaobjects of prototypes `A`, `B`, and `C` would generate incorrect code. In a future version of the compiler, methods added to a prototype in phase afterResTypes will be visible to other prototypes in this same phase. This is the correct strategy because other prototypes should know the final version of a prototype as soon as possible. However, the problem would persist. If one of the prototypes of the figure, say `A`, undergoes phase afterResTypes before the others, then metaobject `createMethod` associated with `A` will generate wrong code: `num` will be `1` less than the real number of methods of `B` (the method created by `createMethod` in `B` would not be considered).

Circular dependency among metaobjects of different prototypes in phase afterResTypes could be addressed by extending algorithm FixMeta to deal with all prototypes of a *dependency cycle*. But how to build this cycle? It could be the dependency graph built by the compiler, before phase afterResTypes, based on the types used by the prototypes (considering every type appearing outside method bodies in the prototype). This would not work in the example of Figure 9. Metaobject `createMethod` adds a dependency from its current prototype and its parameter. This dependency is not discovered by the compiler in phase resTypes because the parameters to the annotations are just symbols, they do not represent the types with the same name. The dependency would be discovered in phase afterResTypes during the execution of algorithm FixMeta, when it should be added to the algorithm data. The dependencies added during FixMeta execution could be removed in the next round of calls. Any language solution to this complex mesh would make the Cyan MOP too complex for regular use.

There is no *circular dependency* in phase semAn because changes caused by metaobjects are only visible in the next compilation phase. In phase afterSemAn, all metaobjects view the same code and, therefore, there cannot be any *circular dependency*.

A metaprogramming system with severe restrictions on how it changes the code and does checks will have few or none of the problems described in this section. This is not the case with the Cyan MOP. It is powerful enough to implement complex metaobjects as demonstrated in Appendix B which presents some non-trivial metaobjects that could not have been done in a limited metaprogramming system.

4.4 Shortcomings of the Cyan MOP

A metaobject cannot generate code with annotations whose metaobjects generate code in the same compilation phase. The metaobject must generate by itself all of the source code. As an example, suppose a metaobject `propertyAll` takes pairs (name, Type) and generates fields with that name and type and get and set methods for them. This metaobject cannot generate

```
    @property var Type name
```

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

Some MOP features are missing in Cyan, as to intercept compiler error messages and code generation. These features are planned to be added soon. Currently, there is no interface for adding code to a subprototype whenever the *current prototype* is inherited. Although this would be an intrusive feature, it may be added to the Cyan MOP.

Some metaobjects cannot be done with the Cyan MOP because metaobjects, in phase semAn, cannot view the code generated by others. For example, suppose two annotations are inside the same method. In phase semAn, a metaobject associated with one of them may generate "`n println;`" in which `n` is the number of statements of that method. The other may generate "`let numStat = n;`". Each one will generate wrong code because it does not consider the code added by the other, visible only in the next phase. This could be resolved if algorithm FixMeta were adjusted to work in phase semAn. However, we think it would add a lot of complexity to MOP in terms of the benefits it would bring.

5 Comparison with Related Work

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

5.1 How Code is Generated and Represented

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

As text. Code is generated in string format. Metaobject `mypropery` of Appendix A exemplifies how this works. The metaprogram does not usually check that the generated code is error-free. Therefore, the code may have lexical, syntactic, and semantic errors. This is the mechanism used by Cyan which will be compared with the following ones.

Handling of the program Abstract Syntax Tree. Code is generated by creating objects of the AST representing it, if the compiler is implemented in an object-oriented language. Therefore, the developer has to know a great number of classes (more than one hundred in Cyan). Code generation is difficult because it demands the mapping, by the metaprogrammer, of human legible source code into the creation of AST objects. AST handling has the advantage that the metaprogram compiler catches usually all syntactic errors of the generated code. The remaining errors, if any, are caught by the base compiler. If the metacode inserts the generated

AST objects directly into the AST, the base compiler will not be able to point out the metacode that generated the offending code. Cyan generates code as strings and any errors in them are not caught either at compile-time or runtime of the metaprogram (metaobject code). However, errors are discovered in the next Cyan compilation round of the base program. The compiler will give precise error messages, pointing out exactly which annotation is associated with the metaobject that produced the offending code.

Quoting. A special language syntax transforms text into AST objects. Therefore, the metaprogram handles text that is converted into AST objects. The later are passed to the compiler. The quoting mechanism will be presented using examples in Converge [Tra08], a Python-based language. A *quasi-quoted* expression `[| code |]` builds the AST of code. Inside code there may appear annotation `${ code2 }` meaning that `code2` is to be inserted into `code`. Annotation `$< code3 >` evaluates `code3` at compile-time, generating an AST that replaces the annotation. It is as if the result of the evaluation were inserted in the source code. Any variables visible are renamed to ensure there is no variable capture from the environment. In the next example [LS15], the AST of 5 is assigned to code in the first line. In line 2, code, an AST object, is used twice to build the AST of a multiplication expression. Thus, `square` refers to the AST of `5 * 5`. In line 3, `square` is evaluated at compile-time, resulting in 25, assigned to `result`.

```
1    code := [| 5 |]
2    square := [| ${code} * ${code} |]
3    result := $<square>    // 25
```

In Cyan, the result equivalent of quasi-quoting would be to use strings in metaobject methods. The insertion of a quasi-quote into another, using `[| ${ ... } |]` in Coverge, is translated into string concatenation in Cyan. Therefore, the two first lines of the above code in Cyan would be

```
var String code   = "5";
var String square = code ++ " * " ++ code;
```

The last line of the previous example demands the interpretation of `square` by the Cyan interpreter, supplied as a prototype in package `cyan.reflect`. An alternative mapping of this example is to use metaobject `insertCode` of Appendix B. Since the attached DSL code to `insertCode` is interpreted at compile-time, there is no need of quasi-quotes.

```
@insertCode{*
    var Int code = 5;
    var Int square = code * code;
    insert: "result = " ++ square ++ ";"
*}
```

The code this annotation produces is `"result = 25;"`, inserted after the annotation. Using the Cyan syntax, a quasi-quote with the contents “`var Int n;`” would be ambiguous because it could represent the declaration of a field or a local variable. This is addressed by some metaprogramming systems by supplying several different kinds of quasi-quotes [BLS98] [Tra08]. There is no ambiguity in Cyan because generated code is represented as strings. Therefore, `"var Int n;"` becomes a field if it is inserted outside a method, in phase afterResTypes, or a local variable if it is inserted in phase semAn.

Quasi-quotes are, therefore, simulated with string handling in Cyan, which is much simpler for two reasons. First, because it uses operations known by every programmer (string handling). Second, there is no confusion between metacode with base code, the base code is wrapped in strings and it is not an AST object. The downside of the Cyan approach is that any checks are delayed until the compilation of the code produced by a metaobject method. This code is inserted into a *prototype* which is compiled again and errors in code inserted by metaobjects are detected. Code within quasi-quotes is checked at compile-time, although usually only for syntactical errors.

The code snippets produced by metaobject methods, which are just strings, are not checked when the method is running. Thus, the code below is perfectly valid in Cyan, even considering `dec` is returned as the code generated by the metaobject method.

```
var String partial = "var Int n =";
var String dec = partial ++ " 0;";
```

In languages that use quasi-quotes, the equivalent code would cause a parsing error in line 1 because the literal string would be represented using quasi-quotes and the compiler would check if this is a valid statement or expression. It is not because the expression assigned to `n` is missing.

Macros. Macros in high-level languages were first introduced in Lisp [Har63]. In this language, a macro is a function called at compile-time[13] to produce code that then replaces the macro call. This is the definition of "macro" used in this paper. Currently, there are more sophisticated versions such that of Nemerle [SMO05], Rust [KN22], and Scala [Bur13].

Skalski, Moskal, and Olszta [SMO05] give an example of a `for` statement added to Nemerle using a macro. The macro defines the syntax and how code is to be generated to a `for` statement. Quasi-quotes are used to express the generated code.

```
macro for (init, cond, change, body)
syntax ("for", "(", init, ";", cond, ";", change, ")", body)
{
    // generate code here using quasi-quotes
}
```

Macros are used for local changes only, a macro call is replaced by code. They are not capable of the other code modifications and checks allowed by Cyan: add fields and methods to prototypes, intercept several operations, and check the final prototype code.

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

Cyan does support *macros* which are metaobjects with most of the power of other kinds of metaobjects. However, this feature is not discussed in this paper. Macros of Nemerle, Rust, and Scala can be roughly simulated in Cyan by metaobjects whose prototypes implement interface

```
IActionMethodMissing_semAn
```

If the compiler cannot find an adequate method, a metaobject method is called. It can then apply any transformation to the message passing parameters and produce any code. Just like a macro whose syntax is that of a message passing. Metaobject `grammarMethod` presented in subsection B.2 does just that.

Generic classes, functions, and prototypes. This encompasses C++ class templates [Str13] in which a new class is created for each new instantiation of the class. That is, for each new set of class parameters, a new class is created. This is also what Cyan does with generic prototypes. This mechanism does not include generic classes of languages such as Java in which all generic instantiations share the same class code. The C++ template mechanism offers a compile-time Turing-complete functional language for template generation [Vel03]. In Cyan, metaobjects are used for generating base code for generic prototypes. Metaobject `insertCode` of Appendix B, for example, takes an interpreted Cyan code as attached DSL text, interprets it, and adds the code it produces to the current prototype.

Specialized languages. Domain Specific Languages are used to generate code. AspectJ [KHH+01] [Asp23] is a Java extension for *Aspect-Oriented Programming* (AOP) [KLM+97]. In this paradigm, code for an *aspect* of a program, like error handling and logging, is grouped 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. This is specified through an *aspect language*, a DSL, resembling Java. The AspectJ compiler, directed by user-code, can add methods, fields, and constructors to classes and change inheritance and implemented interfaces.

Genoupe [DLW05b] [DLW05a] is a C♯ extension whose generic classes can make use of a language for code generation. There are a `foreach` and `if` statements used to generate code. In the example, adapted from [DLW05b], `C` declares a field for each field of `S`

```
class C(Type S) {
    @foreach(F in S.GetFields()) {
        @F.FieldType@  @F.FieldName@;
    }
}
```

Genoupe cannot add code to existing classes. It also does not guarantee the generated code is well-typed, although it offers a high degree of type safety at compile-time.

Generators written in SafeGen [HZS05], a metalanguage for Java, produce only well-formed Java code. SafeGen uses a theorem prover fed with first-order logical sentences representing properties of the generated code. If the prover cannot assure the generated code is well-formed Java code, an error is issued. SafeGen statements `#foreach` and `#when` are used for repetition and decision, much like the equivalent statements of Genoupe.

CTR [FCL06] extends C♯ with *transformers* which are constructs combining patterns and generation templates. Whenever a *transformer* matches a code, like a class, the generation template is applied. It can, for example, add a method to the class or create new classes. The well-formedness of the generated code is checked both by CTR and the compiler. In Cyan, just the compiler checks the generated code.

Generic classes in MorphJ [HS11] specify how to build other classes based on the fields and methods of their type parameters. This technique is called *morphing*. The classes instantiated from the same generic class may have different structures. MorphJ generic classes are checked without the knowledge of their real parameters. Hence, not well-formed code is detected early. The language offers positive and negative patterns for code generation. In the following example [HS11], class `Logging<A>` extends `A` and declares a method for every method in `A` that matches the pattern `R meth (Y)`. `R` and `Y` can match any non-`void` type and `meth`, any identifier. The `Logging<A>` method calls the superclass method and prints a message.

```
class Logging<class X> extends X {
    <R,Y*>[meth]for(public R meth (Y) : X.methods)
    // method below is added to class Logging<X>
    public R meth (Y a) {
        R r = super.meth(a);
        System.out.println("Returned: " + r);
        return r;
    }
}
```

MorphJ cannot add code to existing classes, it can only create new classes.

Trait functions in the model MTJ [RT07] take parameters and are composed of `requires` and `provides` clauses. A *trait function* is called on a class when real arguments are supplied. Then the fields and methods of the `provides` clause are added to the class. The `requires` clause impose constraints on the real arguments. The calling of a *trait function* works similarly to an annotation in Cyan whose metaobject adds fields and methods to the current prototype. PTFJ [MS12] extends MTJ with patterns borrowed from MorphJ. Miao and Siek [MS14] extend PTFJ introducing pattern-based code generation at the statement level. That is, method statements can be generated based on pattern matching. For example, a statement is generated only if a class has a given method.

cJ [HZS07] is a Java extension with predicates on the type parameters of generic classes. A predicate works as a *static if* for code generation. For example, a method is added to the generic class only if the parameter `X` is subclass of class `Y`. The type-checking of a generic class is modular, it can be made before any instantiations.

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

Identifiers starting with a lowercase letter are not considered types when passed as a parameter to a generic prototype. Therefore they can be used to give information to metaobjects. For example, a metaobject associated with prototype `MyList` could create a list optimized for space when instantiated with identifier `space` as in `MyList<Int, space>`. Hence, generic prototypes work like functions that take arguments and return a type.

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

5.2 Runtime Metaprogramming

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

Introspective reflection happens when a program can observe itself, discover its own structure. A language supports a kind of *reflection* called *intercession* if a program can change itself. Smalltalk [GR83] [NDP09] has a runtime Metaobject Protocol based on metaclasses, which are the classes of classes. Almost everything in Smalltalk is an object, and every object is an instance of a class. A class is also an object, an instance of its metaclass. The Smalltalk MOP is fundamentally different from that of Cyan because it supports *introspective reflection* but not *intercession*. A Smalltalk program can change itself at runtime using methods inherited from fundamental classes such as `Behavior` which are outside of the MOP.

Metaprogramming in Python 3 [Ram15] is supported by a MOP and other language features. Every class has a single metaclass that can modify its class. For example, it can add fields and methods to the class. Python metaprogramming has very different characteristics when compared with Cyan.

(a) Each class has only one metaclass in Python. This limitation drastically reduces the complexity of metaprogramming in Python and, at the same time, limits its usefulness. The equivalent restriction in Cyan would be to allow just one annotation to each prototype. If this were the case, some of the problems described in section 2 would not exist such as

OrderMatters, InfiniteMetaLoop, and CircularDependency. They only exist if there is more than one metacode acting on the same code;

(b) Annotations in Cyan can be expressions. Code can be added after an annotation inside a method. In Python, there is no similar functionality;

(c) Metaobjects in Cyan have access to the AST of the current prototype. In Python, the AST is not readily available. To obtain it, one has to get the bytecodes of a class or its source code and then built the AST;

(d) A metaobject method in Cyan can be called whenever a prototype is inherited or a base method is overridden in a subprototype. Python does not have a similar feature;

(e) Cyan supports a compile-time MOP and Python, a runtime MOP. Hence, Python can change classes with information only available at runtime. Hence, errors in metaprogramming are only discovered at runtime too.

5.3 Static Analysis Tools

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

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

5.4 Compile-Time Metaprogramming

The prime example of a Metaobject Protocol is that of CLOS [KdRB91] [KAR$^+$93] [Pae93] [BGW93] [DG87], an extension of Common Lisp [Ste90] with features for object-oriented programming. The CLOS MOP acts at runtime, allowing the intercepting of several operations: object creation, allocation of memory, calculus of superclass precedence,[14] method calls, field access, and many more. The MOP of this language uses *metaclasses* which are the classes of classes and methods,[15] which are objects too. By using a user-made metaclass for a class we

[14]The superclasses have to be ordered because the language supports multiple inheritance.
[15]CLOS have both *methods* and *generic methods*. To our goals, it is not necessary to distinguish them.

change its expected behavior. For example, a metaclass can introduce a field into a class that keeps how many objects were created. The method that creates instances of the class may increment this field every time it is called.

OpenC++ [Chi95] is a C++ extension in which metaclasses for classes and methods are given the opportunity of changing the AST after parsing. 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 interception of variable declarations, creation of objects, and reading and writing in fields.

OJ [TCIK00] [Tat99] is a Java extension in which a class may be associated with a user-defined metaclass. Methods of the metaclass have the opportunity of changing the AST. For example, a method called `translateDefinition` of a metaclass may add methods to the class. `expandFieldRead` can change the read of a class field. The user-defined metaclass can also define methods for intercepting object creation, array allocation, writing to fields, method calls, and casts to the class.

Languages Xtend [Xte20], Groovy [Kö07], and Nemerle [Nem18] [Ska05] support *compile-time metaprogramming* without a Cyan-like Metaobject Protocol. We will say that these languages support *metaprogramming features*. They share many similar characteristics, described below, and therefore will be considered together.

(a) annotations are attached to classes, methods, and other declarations;
(b) an annotation is linked to a *Processor Class* (PC) that can implement interfaces and define methods that change the compilation;
(c) methods of the PC are invoked in several phases of the compilation, like before parsing, after parsing, before typing members (similar to afterResTypes 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, using these AST objects, add methods to an annotated class, change inheritance, add statements to an annotated method, change method statements, and so on. Any AST object reachable from the method arguments can be changed. Therefore, a method can be added to a class that is not annotated or directly related to the annotated class. The class may 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. Consequently, the view of a class by methods of a PC is not well-defined.

A *compiler plugin* is composed of metacode called in *hooks* of a language's compiler. These plugins change the compilation process and, therefore, add compile-time metaprogramming features to the language. The difference, in usage, between the terms *compiler plugin* and *metaprogramming* is the emphasis in the implementation aspects of the first and conceptual aspects of the later. Languages Scala [ST20], Java [Ora23], X10 [NS07], Kotlin [Kot20], TypeScript [typ20], and Rust [Rus20] support *compiler plugins*. Java annotation processors [Dar06] are compiler plugins for Java that allow checks but not code modifications. They are used, for example, for implementing pluggable types [che18]. Project Lombok [Kim10] is a Java annotation processor whose supported annotations can add code to classes because it uses non-supported downcasts. Compiler plugins will not be discussed in depth in this paper because there is a shortage of good documentation about them. However, languages whose compilers accept plugins have all of the main characteristics of languages supporting *compile-time metaprogramming* without a Metaobject Protocol, discussed above.

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

(1) the language prohibits non-local changes. A metacode associated with a class can only change the class, a metacode inside a method can only change the method;
(2) the compiler detects conflicts between different parts of the metaprogram, like two metacodes trying to add code 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 its dependents and it can view the changes made to the AST by them. This is complex because a metacode can create itself metacode.

Metaprogramming extensions are frequently implemented using a non-extended language, which is called the *base language*. The former are implemented in terms of the later leading to the conflating of both. Lamping, Kiczales, and Chiba [CKL96] give an example: a metaclass adds a `history` field to a class and generates code that intercepts all field accesses that are recorded in the `history` field. The extension, also called the *implemented level*, has objects with field access history. The *implementing level* is the base language, which does support field access history. There are two problems with that caused by the conflation of both levels. The first is that users of the class will view the `history` field because the conflation mixes the original fields, the implementing level, with the extended fields, the implemented level. In most cases, this field should not be visible since it is just an implementation scheme. The second problem is that the metacode translating the implementor level into the implementing level also mixes up both levels. In this example, a careless code that records field accesses into `history` would also record the accesses to this field, resulting in an infinite recursion. The meta-helix architecture [CKL96] separates two or more levels of implementation automatically: the implementing level does not mix up with the implemented level. This prevents the problems with implementation level conflation.

In Cyan, there is a conflation of levels, which are not automatically separated from each other. However, metaobjects can differentiate fields and methods added by metaobjects from the original ones:

(a) in phase afterResTypes, the signature of added fields and methods are passed as parameters to method `afterResTypes_codeToAdd`;
(b) in phases semAn and afterSemAn, an AST object representing a field or method has a method that returns `true` if the slot was created by a metaobject. This is also true for statements, including expressions, in phase afterSemAn.

5.5 Metaprogramming Systems and their Problems

This subsection discusses the metaprogramming problems of section 2 that occur in languages with metaprogramming powers similar to Cyan and, occasionally, with other languages. Languages with more limited capabilities have, because of this lack of power, fewer problems:

(a) if the language generates code using class patterns, it cannot have any of the problems. Usually, a language use patterns and some other form of code generation and it may have some of the problems;
(b) languages supporting a single metaclass for each class cannot have the WhoDidWhat, OrderMatters, InfiniteMetaLoop, and CircularDependency problems because they only exist if there is more than one metacode acting on the same code;
(c) runtime metaprogramming cannot have the problems associated with compilation like WhoDependsOnWho, Compiler-Interactions, and CircularDependency;
(d) some languages allow the interception of operations like object creation and message passing but not the addition of code. Problems MessWithOthers and OrderMatters cannot occur with them.

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

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

Languages OJ, Xtend, Groovy, and Nemerle allow non-local changes by AST handling. CLOS, OpenC++, and BSJ limit the changes to the scope of the metaclass or metacode. Cyan addresses this problem by a series of mechanisms, a metaobject can only change external files if the change is expected.

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

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

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

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

We are unaware of any language other than Cyan that: (a) supply AST objects to metacode and (b) limits the visibility of the AST objects by security checks.

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

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

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

Converge [Tra08] tracks who produced which code to issue precise error messages. It goes beyond Cyan in two aspects: (a) every bytecode[16] knows its origin, which can be used in runtime error messages, and (b) an AST node can be associated with more than one location (an error may be associated with more than one source). Cyan keeps track of which metaobjects did what.

OrderMatters *The order metacode is called inside a source file changes metacode behavior.*

This problem occurs in all languages that allow direct handling of the AST:

(a) a metacode views the changes made by metacode executed before it. If the metacode call order is changed, the view is changed too;
(b) usually, there is no way of specifying that, after a certain compilation phase, the AST is read-only. Therefore, checks made by a metacode may be invalidated by code added by other metacode.

In AspectJ, a keyword may declare the execution order of the metacode. In BSJ, metacode may declare its dependencies. A metacode with clause `#depends label` is only executed after a metacode with a `#target label` clause. The later metacode can view the changes made by the previous one. Cyan addresses this problem except in two cases: (a) code addition at the start of base methods (the addition is made in annotation order) and (b) in phase semAn, metaobjects

[16]Source code is translated into bytecodes of a Converge VM.

whose annotations come later view a more detailed AST of the statements that come before (types are resolved).

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

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

Nontermination *Metacode may not finish its computation.*

SafeGen, MorphJ, and Meta-traits [RT07] ensure termination of code generation [SZ13]. In general, the termination of code generation is guaranteed only if the generated code is composed of code patterns or the metacode is limited to a few kinds of statements. Cyan does not address this problem.

Nondeterminism *Metacode is nondeterministic.*

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

NoGeneratedCodeGuarantees *Metacode may generate defective code.*

Only a few languages offer a high degree of safety in code generated at compile-time: Genoupe, SafeGen, CTR, and MorphJ. They are all pattern-based. DynJava [OMY01] is a Java extension that supports quasi-quotes with information on the context in which they can be used. The context includes the name of the base class, the local variables, the fields and methods, and so on. These typed quasi-quotes and rules of the language assure that code produced at runtime is type-safe. Cyan offers no guarantees in relation to the generated code.

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

SafeGen arguments to metacode may be restricted by predicates. For example, a metacode can accept only non-abstract classes as arguments. The pattern in a *transformer* of CTR limits the classes it can match, therefore working as a contract between meta and base code. The `requires` clause of a trait function of model MTJ imposes constraints on real arguments and the `provides` clause supplies the code added to a class. MTJ has the best solution to the

NoContracts problem. Metaobject `concept` of subsection B.3 can specify a contract between metacode and the base code. Since its use is optional, Cyan does not enforces this contract.

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

Circular dependency occurs in compiler plugins and language with metaprogramming features. In BSJ, the execution order of metacode may be specified. This does not solve this problem because there may be no correct order of execution — remember example with metaobject `addFieldInfo` in subsubsection 4.2.3. Cyan addresses this problem in all but one case: metaobjects associated with several prototypes in phase afterResTypes. A solution to this case would demand the extension of algorithm FixMeta of Listing 6 to several prototypes, a complex solution for a not-to-common problem.

6 Conclusion

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

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. Therefore, 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. As a result, the metaprogram acts passively in relation to the compiler, who is in control of the execution flow of the metaprogram. This architecture makes it relatively easy to build metacode when comparing with other metaprogramming systems with the same powers. In the later ones, the decisions are taken at metacode runtime with the help of the original compiler data structures.

The compile-time Metaobject Protocol of Cyan effectively allows the extention of the language with new functionality, brought to life using annotations. Metaobjects can be used to: code testing, optimize code, log events, document code, define embedded DSLs, support *concepts* for generic prototypes (subsection B.3), enforce code style, associate metadata to declarations, generate boilerplate code, assure static properties of prototypes (as immutability), support object replication in distributed systems, evaluate code at compile-time, and implement Design Patterns.

The Cyan MOP supports six 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"<s>XML code</s>"), (c) macros (each start with an identifier after which any syntax is allowed), (d) annotations to types that implement pluggable types [Bra04] [che18] [PAC⁺08] (like String@regex("a*[A-Z]") or Char@letter), and (e) Codegs (code + eggs), visual metaobjects that demand a plugin to an IDE[17] (an annotation @color(red) allows one to choose a color using a menu, during editing time). The metaobject classes or prototypes of all metaobject kinds can implement most interfaces of subsection 4.2. For example, the metaobject of a number annotation, like 101bin, could add fields and methods to the current prototype (it does not). It certainly generates number 5 as code in phase semAn.

There are several planned future works for the Cyan MOP. One of them is to allow metaobjects to change the original source files, if they ask for that. Other future work is to support variable ownership like language Rust [KN22]. The Cyan compiler is available for download at cyan-lang.org. There one can find the language manual, a complete description of the Cyan MOP, and a list of around one hundred metaobjects with examples.

References

[Asp23] The aspectj language, 2023. URL: https://www.eclipse.org/aspectj/doc/next/progguide/language.html.

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

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

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

[Blo18] Joshua Bloch. *Effective Java.* Addison-Wesley, Boston, MA, 3 edition, 2018.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, page 143, USA, 1998. IEEE Computer Society.

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

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

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

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

[CKL96] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The meta-helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Object Technologies for Advanced Software, Second JSSST International Symposium, ISOTAS '96, Kanazawa, Japan, March 11-15, 1996, Proceedings*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996. URL: https://doi.org/10.1007/3-540-60954-7, doi:10.1007/3-540-60954-7_49.

[17] Integrated Development Environment

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

[Csh24] C# language specification, September 2024. URL: https://learn.microsoft.com/en-us/dotnet/csharp/.

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

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

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

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

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

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

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

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

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

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

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

[Gui24] José de Oliveira Guimarães. The cyan language. 2024. URL: http://cyan-lang.org/docs.

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

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

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

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

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

[Kö07] Dierk König. *Groovy in Action.* Manning, New York, 2007.

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

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

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

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

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

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

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

[KN22] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* No Starch Press, second edition, 2022. URL: https://doc.rust-lang.org/book.

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

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

[LS15] Yannis Lilis and Anthony Savidis. An integrated implementation framework for compile-time metaprogramming. *Softw. Pract. Exper.*, 45(6):727–763, June 2015. doi:10.1002/spe.2241.

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

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

[NDP09] Oscar Nierstrasz, Stphane Ducasse, and Damien Pollet. *Squeak by Example.* Square Bracket Associates, 2009.

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

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

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

[Ora23] Oracle. Interface plugin, October 2023. URL: https://docs.oracle.com/en/java/javase/11/docs/api/jdk.compiler/com/sun/source/util/Plugin.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *SIGPLAN Not.*, 21(11):38–45, June 1986. doi:10.1145/960112.28702.

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

[ST07] Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 2–27, Berlin, Heidelberg, 2007. Springer-Verlag.

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

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

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

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

[SZN15] Amanj Sherwany, Nosheen Zaza, and Nathaniel Nystrom. A refactoring library for scala compiler extensions. In Björn Franke, editor, *A Refactoring Library for Scala Compiler Extensions*, volume 9031 of *Lecture Notes in Computer Science*, pages 31–48. Springer, 2015. doi:10.1007/978-3-662-46663-6.2.

[Tah07] Walid Taha. A gentle introduction to multi-stage programming, part II. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, volume 5235 of *Lecture Notes in Computer Science*, pages 260–290. Springer, 2007. doi:10.1007/978-3-540-88643-3_6.

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

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

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

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

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

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

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

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

Acknowledgments

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

A Metaobject myproperty Implemented in Cyan

```
package main

import meta
import java.lang
import java.util

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

  /* Cyan do not support enum types yet. Therefore, strings
     are used in the second and third parameters in the
     'super init:' message passing
  */
  func init {
    super init: "myproperty", "ZeroParameters", [ "field" ]
  }

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

      // cast a Java value of class IDeclaration to
      // the Java class WrFieldDec
    var WrFieldDec field = JavaCast<WrFieldDec>
        asReceiver: getAnnotation getDeclaration;
    var String fieldName = field getName;

    var nameUpper = (fieldName[0] toUpperCase) ++
            (fieldName substring: 1);
    var String ivTypeName = field getType getFullName;

    var String methodGet = " func get$nameUpper -> $ivTypeName ";
    var String methodSet = " func set$nameUpper: $ivTypeName other ";

    var methodsSignature = "$methodGet;\n $methodSet; ";

    var methodsCode = "$methodGet = $fieldName;\n" ++
        "$methodSet { self.$fieldName = other; }\n";

    return [. methodsCode, methodsSignature .];
  }

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

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

```
end
```

B Metaobjects in Action

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

B.1 Metaobjects in Interpreted Cyan

A metaobject prototype, after successfully compiled, should be put in a special directory of a package. To use the metaobject, a *compilation unit*[18] imports that package. To streamline this process, package cyan.lang supplies some metaobjects that accept attached DSL code in *interpreted Cyan*. For example, annotation onOverride takes an attached DSL code that is run whenever the associated method is overridden in a subprototype.

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

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

Package cyan.lang has more complex metaobjects whose attached interpreted Cyan code can do multiple tasks like add fields and methods to the current prototype, communicate with other metaobjects, create new prototypes, do checks in phase afterSemAn, and so on. The interpreted Cyan code can be put in files and loaded by metaobjects, thus reusing them. As a last example, Listing 7 shown anannotation that inserts 9 methods in the current prototype whose names run from power_2 to power_10. The insert: method accepts two arguments: the signature of a method and its full definition.

[18]A *compilation unit* is composed of a prototype and its import declarations.

Listing 7. Annotation insertCode adds methods to the current prototype

```
    @insertCode{*
        // adds to the prototype functions like
        //     func power_num: Int n -> Int = n*n ... *n;
        //  "= $s;" is equal to "= " ++ s ++ ";"
        for num in 2..10 {
            var sig = "func power_$num: Int n -> Int ";
            var s = "n";
            //  ++ is concatenation of strings
            for p in 2..num {
                s = s ++ "*n"
            }
            insert: sig,
                    sig ++ "= $s;"
        }
    *}
```

B.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)* )+
    *}
    func meet: Array<Tuple<String,
                 Union<some, String, none, Any>,
                 Array<String>>> p {
         // elided
    }
end
```

Message passings to expressions of type Schedule that do not match any methods are matched against the regular expression. If there is a match, method meet: is called passing the arguments packed as a single parameter. The following 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 metaobject will tell the correct type if a wrong one is given.

A *functional metaobject* is any metaobject whose class or prototype implements interface `IActionFunction` and declares an `eval` method. In the attached DSL to a `grammarMethod` annotation, a list of *functional metaobjects* may be given after the regular expression. 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.

B.3 Metaobject concept

Concepts were devised to help the compiler issue clearer error messages in the instantiation of a template class in C++. Gregor et al. [GJS+06] proposed this feature for the language C++, although it has not been accepted yet.[19] *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 subsection 4.2, have methods to read and write to files of these special directories. Each Cyan package can have the directories `--data` (for

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

DSL code like those of metaobject concept), --test (for tests), and others not described in this paper.

B.4 Metaobject in the Cyan Libraries

Package cyan.lang is imported by every Cyan source file and defines prototype Any, the top-level prototype, generic prototypes for tuples, unions, and anonymous functions, the Array<T> prototype, and all basic prototypes such as Int, Char, and String. Metaobjects are used extensively in this package. Since there is a large interaction between it and the Cyan language, we can assure that not only package cyan.lang but also the Cyan language would be very different without the Cyan MOP. A small list of metaobject 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. As a result, there is a method sum that returns the sum of all elements of an object of Array<Int>. Method isA: tests if the receiver object is an instance of the parameter. A metaobject tests whether the argument is really a prototype. Metaobjects of annotations attached to method == of Any check whether the argument is compatible with the receiver. For example, it is a compile-time error to compare an Int with a Char because the result will always be false. Another metaobject demands that, if == is overridden in a subprototype, hashCode has to be overridden too. And yet another metaobject generates code for testing the overridden method. This code is put in a special directory of the package.