Concepts for Generic Prototypes in Cyan

Anonymous Author(s)

ABSTRACT

Generics are classes, prototypes, or algorithms parameterized by types. Concepts are constraints on real arguments to generics that express the generics' restrictions on the use of the type parameters. Their main goal is to give clear error messages to developers. Concepts are supported as a feature by several languages. This paper shows how the prototype-based object-oriented language Cyan supports concepts by employing a compile-time metaobject linked to a code annotation. A simple Domain Specific Language (DSL) code attached to the annotation allows a comprehensive set of concept constraints. As a result, not only concepts are easy to specify but it would be easy for developers to change and modify our metaobject to support a different concept DSL.

KEYWORDS

Object-oriented programming, Generic programming, Concepts, Metaprogramming, Prototype-based language

ACM Reference Format:
Anonymous Author(s). 2021. Concepts for Generic Prototypes in Cyan. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
package main
object Pair
    func init: String first, Int last {
        self.first = first;
        self.last = last
    }
    func getFirst -> String = first;
    func setFirst: String first {
        self.first = first
    }
    func getLast -> Int = last;
    func setLast: Int last {
        self.last = last
    }
    var String first
    var Int    last
end
```

Listing 1: Prototype `Pair` in Cyan

1 INTRODUCTION

Types can be abstracted over other types and algorithms using the technique of *generic programming* which is supported by generic classes in Java [10] and C♯ [7], templates in C++ [19], type classes in Haskell [22], and similar mechanisms in other languages. As a representative of constructs for generic programming, we will use *generic classes*, which are classes with type parameters that are replaced by *real arguments* at compile-time. A real argument is a real type that, when supplied, causes the creation of a specialized version of the class. For example, type `Int` is supplied to `SortedList<T>` in `SortedList<Int>` for creating an `Int` list. The code of the generic class may be duplicated for each instantiation or not.

The generic class may implicitly assume that the real argument has some characteristics such as the declaration of some public methods or some specific semantics. The code of the class assumes that the supplied types obey some *contraints*. If they do not, there is a compilation or runtime error that is usually unclear because it is issued in the context of the generic class code.[1] Therefore, the developer that is a user of the generic class has to understand its source code to correct the error. *C*oncepts are constraints on types that act as a filter between the generic class and the types that are its real arguments. A concept may point out that a real argument, a type, does not satisfy one or more constraints before the real argument replaces the type parameter. This results in clearer error messages. If the language duplicates the generic class code for every new set of real arguments, the resulting class is compiled after the real arguments replace the formal parameters. At this point, there may be compilation errors caused by non-compliance with the restrictions.

Concepts are supported by several object-oriented languages such as Java, the new C++ version [1], C♯, G [17], C$\sharp^{cpt}$ [6], Magnolia [3], JavaGI [23], and Genus [24]. However, no language has ideal support for concepts [9] [5]. In every one of these languages, there are missing features that, for example, prevent some constraints to be expressed or some valid types to be used as real arguments.

This paper presents concepts in the language Cyan. Unlike any other language, concepts are implemented using metaprogramming through a compile-time metaobject. The result is a very expressive system that features a Domain Specific Language (DSL) for constraint specification. This paper is organized as follows. Section 2 introduces the language Cyan and metaobjects. The metaobject that supports concepts is presented in section 3. Section 4 compares Cyan with other languages with relation to support for concepts.

2 CYAN AND METAOBJECTS

The Cyan language is a statically-typed prototype-based language that relates closely to class-based languages such as Java [10] and C♯ [7]. A prototype describes fields and methods as shown in Listing 1. Prototype `Pair` defines a constructor, which is always

[1] In case of a mismatch in the expected semantics, there may be no runtime warnings or errors. But the error is caused by the generic class code anyway.

```
1 package main
2
3 @init(first, last)
4 object Pair
5     @property
6     var String first
7     @property
8     var Int    last
9 end
```

Listing 2: Prototype `Pair` with annotations

called `init` or `init:` (if it has parameters), and get and set *methods* for `first` and `last` (called *fields* or *instance variables*). A prototype is both a type and an object. Prototypes `String` and `Int`, for example, are used as the types of fields `first` and `last`. They are also objects and can be used in expressions:

```
1     assert Int == 0 && Int*Int + Int == 0;
2     assert String == "" && String size == 0;
3     assert !(String endsWith: "abc");
```

"`String size`" in line 2 is a message passing whose *selector* is `size` and whose receiver is `String` (here, `String` is an object). This statement would be `String.size()` in most OOL. Line 3 shows another message passing whose receiver is also `String`. "`endsWith: "abc"`" is a *keyword message* with `"abc"` as argument. Cyan supports Java-like interfaces whose declaration starts with keyword `interface` instead of `object`. Interfaces are prototypes and therefore objects like any other.

Listing 2 illustrates the use of annotations, which are identifiers that start with `@` and may be followed by arguments between parentheses. An annotation like `init` in line 3 is linked to a metaobject at *compile-time*.[2] The compiler calls methods of each metaobject according to a *metaobject protocol*, MOP, which defines exactly the metaobject methods (their names and parameters) are called in which compilation phase. As an example, method

`afterResTypes_codeToAdd`

is called after the first semantic analysis phase. In this example, the metaobjects associated with both `init` and `property` define this method and they generate a constructor and get and set methods for the two fields. The result is as if prototype `Pair` were declared as in Listing 1. From now on, instead of writing "metaobject associated to annotation" `property`, we will use simply "metaobject `property`".

An interesting metaobject is `insertCode`, whose annotations take Cyan statements between delimiters that usually are `{*` and `*}`. The Cyan statements are interpreted at compile-time. The text between the delimiters is called *attached Domain Specific Language (DSL) code* of the annotation or simply *the attached text* of the annotation.

```
var Array<Char> alphabet = Array<Char>();
@insertCode{*
    for ch in 'a'..'z' {
        insertCode: "alphabet add: '$ch'; ";
    }
*}
```

[2]This paper will not detail how the compiler links the name to the metaobject.

When used inside a method body this annotation can be used for generating method statements. In the above example, it generates statements like

`alphabet add: 'a';`

from `'a'` to `'z'`.[3] When used outside a method body, metaobject `insertCode` can generate fields and methods. This metaobject can be used for checking any property of the prototype such as "the prototype defines a method `add: Int` that has an `if` statement", "the prototype inherits from prototype `X` and implements interface `IA`", or any other property that can be checked using the prototype Abstract Syntax Tree (AST).

3 CONCEPTS IN CYAN

Listing 3 shows a generic `Pair` prototype in Cyan with two *formal parameters*, `First` and `Last`. To use this prototype, one has to supply two *real arguments* as in this example:

```
var Pair<String, Int> p =
    Pair<String, Int> new;
```

When the compiler finds `Pair<String, Int>`, it creates a new prototype, `Pair<String, Int>`, in which every occurence of `First` and `Last` is replaced by `String` and `Int`, respectively. This is a textual replacement, a new source file is created for every new combination of *real arguments*. The language itself does not restrict what the parameters can be, any errors are discovered when the new prototype is compiled. The process of replacing *formal parameters* by *real arguments* and compiling the new prototype is called *instantiation* of the generic prototype. Note that, in the above example, there are two occurrences of the generic prototype `Pair`, one as type and one as an expression, but only one instantiation.

The Cyan compiler replaces a formal parameter `T` by the real argument if `T` appears:

(1) after `#`, which is a *symbol*, a short form of string. Therefore, `#cyan` is the same as `"cyan"`. If the formal parameter `T` is replaced by `cyan`, `#T` is replaced by `#cyan`;
(2) anywhere a type may appear;
(3) as a package name or a type name in a full-specified type as "`packageNameA.T.Person`". If `T` is a formal parameter, it would be replaced by a real argument in an instantiation;
(4) as a method keyword and an unary method name, both at the declaration or inside an expression. In this case, `T` would be part or all of a method name;
(5) as argument to an annotation or inside *the attached annotation text*.

A generic prototype instantiation `A<X>` may cause another instantiation `B<Y>`, which may cause yet another instantiation `C<Z>`. Hence, there may be a sequence of nested instantiations. If an error occurs in `C<Z>`, the Cyan compiler will print an error message and the stack of instantiations. For everyone, the compiler will print the line and file in which it is.

[3]`$ch` inserts the value of the variable `ch` into the literal string.

```
1  package main
2  object Pair<First, Last>
3      func init: First first, Last last {
4          self.first = first;
5          self.last = last
6      }
7      func getFirst -> First = first;
8      func setFirst: First first {
9          self.first = first
10     }
11     func getLast -> Last = last;
12     func setLast: Last last {
13         self.last = last
14     }
15     @doc{*
16       return -1 if self < other,
17       0 if self == other, and
18       1 if self > other. Last is
19       only used in the comparison
20       if 'first == other getFirst'
21     *}
22     func <=> Pair<First, Last> other {
23         var fc = first <=> other getFirst;
24         if fc != 0 { return fc }
25         return last <=> other getLast
26     }
27     var First first
28     var Last  last
29 end
```

Listing 3: Generic prototype `Pair`

3.1 Concepts with Examples

Method <=> of prototype `Pair<First,Last>` follows the expected semantics for methods with this name, as asserted by the documentation given in annotation doc. Lines 23 and 25 of the code demand that `First` and `Last` should also define a <=> method and both of them should follow the same semantics as the prototype method. The Cyan compiler cannot check whether both real arguments have a <=> method before the instantiation. This task could be made by *concepts*, which are *predicates* or *constraints* on template/generic parameters. Concepts could be just comments in the code and, in this case, the programmer is responsible for using appropriate real arguments. Or they could be defined in the generic prototype code through an ad-hoc language mechanism that would trigger the compiler to do the checks. In the latter case, if the real argument for `First` does not define a <=> method, the*concept* feature would issue an error. This is done before instantiation, which means the developer views the error message issued by the concept feature, not the possible confusing error message of the compiler after processing the code of the generic prototype.

Concepts were proposed by Stroustrup [18] [21] and recently incorporated into the current version of language C++ [1]. From this point on, *concepts* always refer to the language feature, never as simply comments in the code that does not have the power of doing checks. There are two main reasons for using concepts: better error messages and code documentation. The compiler can point out errors before the *instantiation* and, therefore, the user will not view error messages in the context of the code of the generic prototype or template class. Concepts also play the role of documentation since they describe the requirements for the use of the generic code. In some languages, concepts are also used for test case generation.

Although the Cyan language does not have special language features for supporting *concepts*, they can be implemented using metaobjects. In particular, there is a metaobject whose name is `concept` defined in package `cyan.lang`. Since this package is automatically imported by any Cyan source code, this metaobject is available everywhere.

Metaobject `concept` can be used to guarantee that both `First` and `Last` define a <=> method, as shown in Listing 4. In the annotation `concept`, between `{*` and `*}`, there should appear the code of a *concept DSL*. In this example, only one statement kind is used, "`has`". After this keyword, there should appear a list of methods that the formal parameter, put before `has`, should have. Optionally, after `]`, there may appear a literal string with a tailored error message. If not present, a standard message is used. The parameter names that appear inside the literal strings are replaced by the real arguments to the generic prototype. Therefore, if prototype `P` does not define a method

```
func <=> First -> Int
```

the instantiation

```
Pair<P, Int>
```

will fail because metaobject `concept` will issue the compile-time error message

```
"P should define method <=>"
```

Between `[` and `]`, after a `has` keyword, there may appear a list of comma-separated `func` declarations, each one optionally succeded by a literal string. As an example, we could have, in another prototype, the following code in annotation `concept`:

```
T has [
    func <=> T -> Int "Error 1",
    func < T -> Boolean
  ], "Error 2"
T has [ func > T -> Boolean ]
```

If, in an instantiation, the real argument that replaces `T` does not define method <=> (with the adequate parameters), message `"Error 1"` is issued. If < is missing, message `"Error 2"` is issued. If the real argument does not define >, metaobject `concept` signals the standard error message for `has`, which names the missing method.

In the instantiation of `Pair<P, Int>`, the compiler first replaces the formal parameters by the real arguments `P` and `Int`. This replacement takes place even in the DSL code of the annotations of the prototype `Pair<First, Last>` (that between `{*` and `*}`). Hence, every "`First`" inside the text of annotation `concept` of Listing 4 is replaced by `P` (even inside a literal string). After that, the compiler calls a method of metaobject `concept` that *interprets* the DSL code (which does the checks and issues error messages, if any).

```
package main
@concept{*
  First has [ func <=> First -> Int ],
    "First should define method <=>",
  Last  has [ func <=> Last -> Int ],
    "Last should define method <=>"
*}
object Pair<First, Last>
   // elided, same as before
end
```

Listing 4: Generic prototype `Pair` with annotation concept

```
...
@concept{*
    OptimizeFor in [ speed, memory ]
*}
object BinTree<T, OptimizeFor>
  func add: T value {
    @insertCode{*
      if #OptimizeFor == #speed {
        insertCode: " /* use array */"
      }
      else {
          // use linked list
        insertCode: " /* use root  */"
      }
    *}
  }
  ...
end
```

Listing 5: Use of real arguments to control code generation using `insertCode`

All formal parameters to a generic prototype should start with an uppercase letter. The real arguments can be prototypes or identifiers that start with a lowercase letter, which are called *identifier parameters.*

```
var BinTree<Int, speed> binTreeForSpeed;
var BinTree<Int, memory> binTreeForMemory;
```

Here, `speed` and `memory` are *identifier parameters*. Of course, the second formal parameter to `BinTree` of both declarations cannot be used as a type because Cyan assumes that all types start with an uppercase letter. However, this formal parameter can be used in all other situations a formal parameter can. In particular, it can be used in the text of an annotation as shown in Listing 5. If the real argument is `speed`, the `BinTree` methods use an array, which is faster than using a linked list. The concept of this generic prototype demands that `OptimizeFor` be one of `speed` or `memory`.

Listing 6: Example with all statement kinds of the concept language

```
@concept{*
  typeof(T size) is typeof(R length),
  typeof(T getProduct: 0) implements
    IProduct<R>,
  typeof(IProduct rep: Long)
    subprototype Map<T, R>,
  Element<T> superprototype Item<T>,
  IProduct<R> interface,
  Element<T> noninterface,
    // S is a formal parameter too
  T has [ func S -> typeof(T set: Int) ],
  typeof(R getId) in [ Short, Int, Long ],
  S in [ speed, memory ],
  S identifier,
  ! typeof(T set: String) interface
*}
object GenProto<T, R, S>
  ...
end
```

3.2 The Concept Language

This subsection describes the *concept language* used by metaobject `concept`. The description uses the compile-time function `typeof` which returns the *type* of its argument, which should be an expression.

```
typeof(T getProduct: 0) is
  typeof( (R at: Int) getName )
```

Here, `typeof(T getProduct: 0)` is the type of the message passing "`T getProduct: 0`" whose receiver is `T` (this prototype name is used as an expression here). This is the same as the return type of method `getProduct: Int` of prototype `T`. Instead of `0`, we could have used `Int`, which would be considered as an expression whose type is `Int`. If Cyan were not a prototype-based language, different syntactic constructs would be used to express types and expressions. "`(R at: Int) getName`" is the return type of method `getName` of the prototype that is the return value of "`R at: Int`".

In the following explanation of the *concept language*, letters `S`, `T`, and `U` are types which can be *formal parameters* to the generic prototype, a real prototype (such as `Int`), or a call to the compile-time function `typeof`. The kinds of statements of the *concept language* of metaobject `concept` and their meanings follow.

(1) `T is U`, `T` should be equal to `U`;
(2) `T implements U`, `T` is a prototype that implements interface `U`. Cyan interfaces are very similar to Java and C♯ interfaces. A Cyan interface is a prototype that declares methods with empty bodies[4]. An interface may be *implemented* by a noninterface prototype `P` meaning that `P` should define all methods declared in the interface;
(3) `T subprototype S`, `T` should be a subprototype of `S`;

[4]Methods with default bodies are not allowed yet.

Listing 7: Example of use of *concept files*

```
1 @concept{*
2   cyan.lang.arithmetic(Matrix<R>),
3     "Matrix<R> should define +, -, *, and /",
4   cyan.lang.arithmetic(R)
5 *}
6 object Matrix<R>
7    // elided
8 end
```

(4) `S superprototype T`, `S` should be a superprototype of `T`;
(5) `T interface`, `T` should be an interface;
(6) `T noninterface`, `T` should be a prototype that is not an interface (as `Pair`);
(7) `T has [ method signature list ]`, `T` should implement the methods of the list. Each method signature starts with keyword `func` followed by the *method keywords* and parameter types (parameter names are optional). In a declaration
    `func add: Int value account: Long c`
`add:` and `account:` are *method keywords*;
(8) `T in [ prototype list ]`, `T` should be one of the prototypes in the list;
(9) `I in [ list of identifiers]`, the identifier parameter `I` should be in the list;
(10) `I identifier`, `I` should be an identifier;
(11) `! any predicate`, the negation of the predicade, any of the above, should be true;
(12) a *concept file call*, described in subsection 3.3. This allows the reuse of concept statements;
(13) axioms, explained in subsection 3.4.

Note that each kind of statement is a kind of *predicate* or *constraint*. The last one is a *semantic* constraint and it cannot be verified at compile-time. The others are *syntactic* predicates. Listing 6 presents an example of each possible statement kind of the *concept language* (except axioms).

3.3 Reusing Concept Code

Some predicates on types are largely used, such as those that demand that a type defines the comparison operators, arithmetical operators, iterator methods, methods object constructors, and so on.[5] Instead of defining the same predicates in the `concept` attached annotation texts of several generic prototypes, it is possible to create a library of concepts. Hence, concepts can be reused just like procedures and functions.

Listing 7 shows a `Matrix<T>` prototype. The attached text of the `concept` annotation demands that both the prototype itself (line 2) and the real argument `T` (line 4) have arithmetic operators. This is demanded by *concept file calls* in lines 2 and 4. There is a file "`arithmetic(T).concept`" in a special directory of package `cyan.lang`, which is called a *concept file*. Listing 8 presents a sketch of this file. It is just composed of predicates on type `T`, the same parameter that appears in the file name.

[5]See this list of C++ concepts: https://en.cppreference.com/w/cpp/header/type_traits.

Listing 8: A *concept file*

```
T has [
    func +  T -> T
    ... // -, *, /
    ],
axiom ...
```

In an instantiation `Matrix<Int>`, the concept metaobject replaces `R` by `Int` and line 2 of Listing 7 becomes
    `cyan.lang.arithmetic(Matrix<Int>),`
The metaobject reads file "`arithmetic(T).concept`" and replaces all identifiers `T` inside it by `Matrix<Int>`. After that, the resulting file is compiled by the metaobject and its predicates become part of the list of predicates of the `concept` annotation. Hence, the file is treated as a poor man generic file parameterized by the parameter between parentheses. There may be two or more parameters separated by commas. A *concept file* may contain *concept file calls*.

Package `cyan.lang` has several *concept files* for common sets of requirements such as `addable` (has a `+` method), `arithmetic`, `comparison` (has the comparison methods), `init`, `iterator` etc.

3.4 Test-case Generation

The text of an annotation `concept` can specify simple type-related requirements, which are syntactic constraints. However, the text cannot demand any semantic properties because they mean *runtime properties*. By the Rice theorem [13], the runtime properties of programs cannot be checked at compile-time. For example, the text of Listing 4 could not be improved for demanding that method <=> of `First` be commutative. The alternative that metaobject `concept` offers for semantic checking is a mechanism that automatizes part of the generation of test cases. This is made through the use of *axioms*, which are semantic specifications of types [20].

In the text of an annotation `concept`, an axiom starts with keyword `axiom` followed by a keyword method declaration (there should be at least one parameter).

```
axiom spaceShipTestFirst: First a, First b {%
  if (a <=> b) != (b <=> a) {
    return "Method <=> of First is not " ++
      "commutative"
  }
  return Nil
%}
axiom spaceShipTestLast: Last a, Last b {%
  if (a <=> b) != (b <=> a) {
    return "Method <=> of Last is not " ++
      "commutative"
  }
  return Nil
%}
```

Instead of using `{` and `}` to delimit the method statements, use[6] `{%` and `%}`. The method should not declare a return type because it is always the union `String|Nil`. That is, the method should either

[6]There are alternatives for these delimiters but this is not important here.

return `Nil` (if there is no error) or a literal string (with an error message).

If annotation `concept` takes a parameter "`test`",

```
@concept(test){* ... *}
```

the metaobject generates a prototype with a method for each axiom in a special directory. It is up to the programmer to create test cases that call the test methods. This task cannot be automatized.

The statements of a `concept` attached annotation text may be inconsistent. For example, they may demand that `T` is a prototype and superprototype of `R` and, at the same time, that `T` is an interface and subprototype of `R`. Or that `T` has a method `get -> Int` and another `get -> String`. When an annotation of metaobject `concept` takes a `test` parameter, the metaobject generates a package and prototypes used to help to discover inconsistences in the code of the `concept` attached annotation text. When the generic prototype is instantiated with these metaobject-created prototypes, there may be instantiation errors that reveal failures in the code attached to annotation `concept`.

Let us explain how these test prototypes are created. The metaobject creates a new package, a test prototype, and a prototype for every formal parameter of the generic prototype (with the same name as the formal parameter). For example, the metaobject creates a prototype `T` in the test package if there is a formal parameter with this same name in the generic prototype. Prototype `T` obeys the statements of the annotation text in which the formal parameter appears. As an example, for the `Pair<First, Last>` prototype of Listing 4, the metaobject would create a prototype `First` with method `<=>`. If the attached annotation text had the statements

```
First subprototype Elem,
First implements IElem
```

the metaobject would make `First` inherit from `Elem` and implement interface `IElem`. Of course, some `concept` statements cannot be checked, such as those that involve `typeof`, `in`, identifiers, and the use of the formal parameters in the right-hand side of a statement. For example, the following statements are not used for test-case generation.

```
  // assume T is a formal parameter
Person subprototype T,
typeof(T get) implements IElem
```

Therefore, to partially test the consistency of a `concept` annotation test, the developer should use the real argument `test` and, after the compilation, compile the test package produced by the metaobject. If this compilation fails, there is at least one inconsistency in the annotation statements.

4 COMPARISON WITH OTHER LANGUAGES

In this section, a *constraint* (or a *predicate*) is a restriction on one or more types (or identifiers, in Cyan) and corresponds to a statement in the language of our metaobject. As usual in the literature, the word "concept" will be used with two meanings: (a) the set of constraints (or statements of a concept language) or (b) the programming language features supporting concepts in the sense (a). Garcia et al [8], Siek [16] and Belyakova [5] have made detailed comparisons of support for concept features by a *set of languages* (SL), which are G [17], C++ [1], JavaGI [23], Java, Scala [12] [15], C♯^{cpt} [6], Haskell [22], Rust [11], C♯ [7], and Genus [24]. The comparisons are not repeated here because that would be redundant (and there is no space for it too). Unless stated otherwise, it should be assumed that most languages of SL (set of languages) support a concept feature when it is first described.

```
class GraphVertex { ... };
class GraphEdge { ... };
class Graph {
  public:
    typedef GraphVertex Vertex;
    typedef GraphEdge Edge;
};
class GraphAdj {
  public:
    typedef int Vertex;
    typedef pair<int, int> edge;
};

template<typename T>
std::list<typename T::Vertex>
  *connectedTo(T *g, typename T::Vertex v ) {
    ...
}
```

Listing 9: Associated types in C++

A concept is *multi-type* if several types are constrained. For example, the concept annotation of Listing 6 falls in this category because it restricts the formal parameters `T`, `R`, and `S`. Therefore, it will be asserted that metaobject `concept` supports *multi-types*. The annotation text of the listing place several restrictions on type `T` in lines 2, 3, 6, 7, 9, 11, and 15. Hence, metaobject concept supports *multiple constraints*. An *associated type* [9] is obtained from other types. For example, types of edges and vertices are associated to the `Graph` type. Listing 9 shows an example in C++. The template function `connectedTo` accepts a template parameter `T` that should be either an object of `Graph` or `GraphAdj` (assume that, otherwise, there would be a compilation error in the template instantiation). These two classes have associated types `Vertex` and `Edge`. The associated type `Vertex` to parameter `T` is refered to using the syntax

```
typename T::Vertex
```

Hence, when the first parameter to `connectedTo` is a `GraphAdj`, the compiler will demand that `GraphAdj::Vertex` is the second parameter, which is `int`.

Cyan offers a partial support to associated types through the use of method return types and the compile-time function `typeof`. For example, a variable inside a method of `GenProto` could be declared as

```
var typeof(T getProduct: 0) aProduct;
```

The support is partial for two reasons: (a) only method return types can be *associated types* and (b) `typeof` cannot appear as method parameter type or return type. Therefore, a method similar to `connectedTo` is illegal in Cyan, it should belong to a generic prototype with *formal parameters* `T` and `Vertex`. That is, `Vertex` should be explicitly given, it could not be deduced using `T`. Associated types decrease the number of real argument types the developer is required to supply to a generic class or algorithm. In the C++ example, instead of supplying two types, a Graph and the vertex type, only the first needs to be specified (the compiler may be able to deduce the types). Java, C♯, JavaGI, and Genus do not support associated types.

A type *models* a concept when the type obeys the restrictions of the concept. For example, suppose a concept `C` parameterized by `T` demands that `T` has a method

```
    func < T -> Boolean
```

In Cyan, prototypes `Int` and `String` model the `C` concept. Through a model declaration, allowed in C♯^{cpt} and Genus, a `String` may model concept `C` in two different ways. One is using the usual `<` method and another is considering just the string size (`"x" < "ab"`). Cyan does not allow multiple models although the *Concept Design Pattern* [14] simulates it to a certain extend.[7]

Retroacting modeling is the ability to change the modeling relationships of a type. For example, a type may not define a `<` method but some mechanisms may adapt the type to the concept. For example, in Rust, methods lay outside *structs* and can be added later to make a type adapt to a concept. The same reasoning applies to extension methods of Swift, which add *virtual* methods to existing classes (the source code is not changed). Hence, even if a class does not define a `<` method, an extension may add this method to it. Genus and C♯^{cpt} have special mechanisms for the declaration of models which can be created after a class to adapt it to be a real argument to a generic class. Therefore, both languages, Swift, and Rust support *retroacting modeling* and so do Haskell, G, and JavaGI. Cyan does not because there is no way of specifying a *model*.

A generic class or prototype may be compiled and put in a library before any instantiation. This occurs in Java because the code of a generic class is reused for all instantiations. In Cyan, the code of a generic prototype is parsed and any syntactic errors are discovered. However, a new prototype is created for each new set of real arguments to a generic prototype. Only when this newly-created prototype is compiled the semantic errors, if any, are discovered. Another dimension for comparing generic programming support is *concise syntax*, which can be achieved by several language mechanisms:

(1) type deduction by the compiler so the user do not need to supply all the generic types when using a generic class. In Java, the code
```
    Set<Int> s = new Set<>();
```
is legal. The compiler deduces the real argument in the right side. Cyan does not allow this but, if the type is not supplied in a variable declaration, it is assumed to be the initializing expression type:
```
    var s = Set<Int> new;
```
Hence, the real argument to `Set` is also supplied just one time.

(2) type deduction of the real arguments to a generic function or method. In C++, an example using Listing 9 could be
```
  list = connectedTo(g, v);
```
instead of
```
  list = connectedTo<Graph>(g, v);
```
Cyan does not support either functions (outside prototypes) or generic methods;

(3) the use of type aliasing features, such as `typedef` of C/C++, which is not supported by Cyan;

(4) associated types. Instead of supplying `Graph`, `Vertex`, and `Edge` to one instantiation, as in
```
  GraphTool<MyGraph, MyVertex, MyEdge>
```
one could supply just `Graph`, which has associated types for vertices and edges. Cyan supports associated types partially.

[7]Extra formal parameters are added to supply extra features, like a parameter that tells a sort method how to compare elements.

Can concepts place restrictions on associated types? They cannot in C♯, Java, JavaGI, and Genus. They can in Cyan, as shown in Listing 6. In line 2, the return type of method `size` of `T` should be the same as the return type of method `lenght` of `R`. These types are associated with `T` and `R`. In some languages such as Scala and C♯^{cpt}, a type may be restricted to be a supertype or subtype of another. Languages Java, C♯, and Swift offers just subtype constraints. That is, a real argument should be a subtype of a type that restricts the corresponding formal parameter. Metaobject *concept* of Cyan supports the relations "subtype of", "supertype of", and "implement". Another desirable feature for generics is the support for concept refinement or inheritance. That is, sets of constraints can be imported to a concept and reused. All languages used in this section support, through many different mechanisms, concept refinement. This is achieved in Cyan through the use of *concept files* (section 3.3).

C++, Rust, and G [17] support *concept-based overloading*. There may be multiple versions of a generic function and the compiler chooses the most specific one. Although Cyan does not support this feature, different versions of a prototype body are easily generated by metaobjects according to the real arguments, as shown in Listing 5. Metaobjects are used to generate code according to the parameters. This feature is used, for example, in prototype `cyan.lang.Array<T>`: if `T` defines method `<=> T -> Boolean`, a metaobject adds a method `sort` to `Array<T>`.

Separate compilation of generics is supported by languages Java, Scala, C♯, and G. Therefore, type checking is made before any instantiations which share the same code. Cyan duplicates the code for each instantiation with new arguments. It could not be different because metaobjects can add code (fields and methods) depending on the real argument, as occurs with `Array<T>`. The sharing of code by several instantiations is hard to achieve and can even lead to unsound type systems [2]. No type error is introduced in Cyan by either the metaobject `concept` or generic prototypes. The reason is that every new instantiation of a generic prototype is compiled as a regular prototype.

Bagge et al. [4] proposed a C++ extension for generating test cases based on *axioms*, which are part of the concept DSL. Axioms are conditional equations that are transformed and used for generating test cases automatically. Language Magnolia [3] was built based on this C++ extension. Cyan axioms are much less sophisticated: they describe methods for testing which are output to test prototypes almost unmodified.

5 CONCLUSION

The Cyan Metaobject Protocol gives to regular developers the power of adapting the language to their needs. Metaobjects work like plugins to the compiler and can change the compilation by adding new checks and generating code. The *new checks* feature is used by metaobject `concept` for constraining the real arguments to generic prototypes through a *concept language*. The parsing and interpretation of this language are made by the metaobject itself. Therefore, no new Cyan language construct or feature is needed for concept support. There can be other metaobjects for concept specifications. In particular, it is easy copy-and-paste metaobject `concept` and change it. In this new concept metaobject, the concept language may be different and it can have fewer or more statement kinds. None of the languages cited in this paper allows developers to change the support for concepts without changing the compiler. However, it should be noted that some features cannot be implemented using Cyan metaobjects: retroactive modeling, default method implementations, and multiple models. Future research could make the metaobject support associated types.

The concept language is expressive: there are 13 different statement kinds and all common uses are supported. There are 12 statements for syntactic checking and one, `axiom`, for helping to build test cases for the semantic restrictions. Through the use of `typeof`, the concept language can express constraints that are difficult or impossible to define in other languages. Developers can give their own error messages. Annotations of metaobject `concept` may be used with non-generic prototypes. Therefore, it may work as a checker for prototype characteristics such as the definition of methods with a given signature (using the `has` statement). An annotation can also generate test cases outside the current prototype.

Metaobject `concept` does not support all concept features cited in this article, like any other language. However, it supports a real DSL concept language and achieves an excellent balance between power and ease of use.

REFERENCES

[1] ISO/IEC JTC 1/SC 22. 2021. ISO/IEC 14882:2020 Programming languages — C++. https://www.iso.org/standard/79358.html

[2] Nada Amin and Ross Tate. 2016. Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers. *SIGPLAN Not.* 51, 10 (Oct. 2016), 838–848. https://doi.org/10.1145/3022671.2984004

[3] Anya Helene Bagge. 2009. *Constructs & Concepts: Language Design for Flexibility and Reliability.* Ph.D. Dissertation. University of Bergen, PB 7803, 5020 Bergen, Norway. http://www.ii.uib.no/~anya/phd/

[4] Anya Helene Bagge, Valentin David, and Magne Haveraaen. 2009. The Axioms Strike Back: Testing with Concepts and Axioms in C++. *SIGPLAN Not.* 45, 2 (Oct. 2009), 15–24. https://doi.org/10.1145/1837852.1621612

[5] Julia Belyakova. 2016. Language Support for Generic Programming in Object-Oriented Languages: Peculiarities, Drawbacks, Ways of Improvement. In *Programming Languages*, Fernando Castor and Yu David Liu (Eds.). Springer International Publishing, Cham, 1–15.

[6] Julia Belyakova and Stanislav Mikhalkovich. 2015. Pitfalls of C# generics and their solution using concepts. *Proceedings of ISP RAS* 27, 3 (2015), 29–46. https://doi.org/10.15514/ISPRAS-2015-27(3)-2

[7] Csharp 2020. C# Language Specification. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction

[8] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. 2003. A comparative study of language support for generic programming. *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications - OOPSLA '03* (2003), 115. https://doi.org/10.1145/949305.949317

[9] RONALD GARCIA, JAAKKO JARVI, ANDREW LUMSDAINE, JEREMY SIEK, and JEREMIAH WILLCOCK. 2007. An extended comparative study of language support for generic programming. *Journal of Functional Programming* 17, 2 (2007), 145–205. https://doi.org/10.1017/S0956796806006198

[10] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[11] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language* (second ed.). No Starch Press. https://doc.rust-lang.org/book/2018-edition/index.html

[12] Bill Venners Martin Odersky, Lex Spoon. 2016. *Programming in Scala: Updated for Scala 2.12 : a comprehensive step-by-step guide* (3ed. ed.). Artima, Incorporated., Artima Press [Imprint.

[13] P. Odifreddi. 1992. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers.* Elsevier Science. https://books.google.com.br/books?id=zgE-lQEACAAJ

[14] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes as Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 341–360. https://doi.org/10.1145/1869459.1869489

[15] Artem Pelenitsyn. 2015. Associated types and constraint propagation for generic programming in Scala. *Program. Comput. Softw.* 41, 4 (2015), 224–230. https://doi.org/10.1134/S0361768815040064

[16] Jeremy G. Siek. 2005. *A Language for Generic Programming.* Ph.D. Dissertation. Indiana University.

[17] Jeremy G. Siek and Andrew Lumsdaine. 2011. A Language for Generic Programming in the Large. *Sci. Comput. Program.* 76, 5 (May 2011), 423–465. https://doi.org/10.1016/j.scico.2008.09.009

[18] Bjarne Stroustrup. 2003. *Concept Checking - A More Abstract Complement to Type Checking.* Technical Report N1510=03-0093. C++ Standards Committee Papers. ISO/IEC JTC1/SC22/WG21. http://www.stroustrup.com/n1510-concept-checking.pdf

[19] Bjarne Stroustrup. 2013. *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.

[20] Andrew Sutton and Bjarne Stroustrup. 2011. Design of Concept Libraries for C++. In *Revised Selected Papers of the Fourth International Conference on Software Language Engineering (Lecture Notes in Computer Science, Vol. 6940)*, Anthony M. Sloane and Uwe Assmann (Eds.). Springer International Publishing, 97–118. https://doi.org/10.1007/978-3-642-28830-2_6

[21] Andrew Sutton and Bjarne Stroustrup. 2013. Concepts Lite: Constraining Templates with Predicates. isocpp.org retrieved march 2021.

[22] P. Wadler and S. Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

[23] Stefan Wehr and Peter Thiemann. 2011. JavaGI: The Interaction of Type Classes with Interfaces and Inheritance. *ACM Trans. Program. Lang. Syst.* 33, 4, Article 12 (July 2011), 83 pages. https://doi.org/10.1145/1985342.1985343

[24] Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. 2015. Lightweight, Flexible Object-oriented Generics. *SIGPLAN Not.* 50, 6 (June 2015), 436–445. https://doi.org/10.1145/2813885.2738008