The Cyan Object-Oriented Language

José de Oliveira Guimarães[a]

[a] *UFSCar, Rod. João L. dos Santos, Km 110, Sorocaba, Brazil*

Abstract

Cyan is a statically-typed prototype-based language that supports non-nullable types, partially safe object initialization, gradual typing, method overloading, anonymous functions, an object-oriented exception handling system, generic types with concepts, and metaprogramming at compile-time through a Metaobject Protocol. The language offers innovative features that support each of these constructs. The goal is to increase developer's productivity without sacrificing security.

Keywords: object-oriented language, metaprogramming, prototype-based language, generic programming, gradual typing, exception handling system

1. Introduction

Interest in programming language design has increased in recent decades. The new languages incorporate some features that help produce better and safer code. This includes support for metaprogramming, gradual typing, constructs that allow more expressive code (less boilerplate code), support for functional programming and embedded Domain-Specific Languages. The availability of good IDEs and tools (such as LLVM [1] and virtual machines) makes it easier to build compilers. Although there are lots of good languages nowadays, there is still room for improvement on several ubiquitous constructs.

In this article, we present the Cyan object-oriented language that supports innovations in some common constructs. Cyan programs consist of declarations of *prototypes*, which are literal objects that play the role of classes. A prototype defines the structure of its objects as a class, and it is also an object when used inside an expression. A prototype has the same structure as its objects. For these reasons, we consider that Cyan is a prototype-based object-oriented language, although it does not have two characteristics commonly associated with this type of language: dynamic typing (which is optional in Cyan) and structural reflection at runtime. The latter allows changes in the structure of objects at runtime. For example, structural reflection allows adding fields and methods to regular objects and prototypes. Cyan was initially based on the language Omega [2], a statically typed prototype-based language that does not support runtime metaprogramming.

Email address: `josedeoliveiraguimaraes@gmail.com`, `jose@ufscar.br` (José de Oliveira Guimarães)

Cyan is statically typed with support for gradual typing, compile-time metaprogramming, non-nullable types, safe object initialization, and generic prototypes. The language considers almost everything as an object, supports a limited form of pluggable type system [3], and has an innovative object-oriented exception handling system. Many of these features are implemented using compile-time metaprogramming. Since 2016, there has been a stable compiler that produces Java source code. That implies that the two languages interoperate well. The compiler and all the language documentation are available at `www.cyan-lang.org`.

The remainder of this paper is organized as follows. The next Section shows the basic Cyan syntax required to understand the rest of the text. Section 3 describes everything related to types in the language: message passing, inheritance, method overloading, gradual typing, and so forth. The initialization of objects and its rules are described in Section 4. Section 5 explains why not all prototypes can be considered as objects. Generic prototypes are discussed in Section 6. Section 7 presents the object-oriented exception handling system.

2. Basic Syntax of Cyan

This Section presents the basic syntax of Cyan that is necessary to understand the rest of the paper. A *class* in languages such as Smalltalk [4], Java [5], C♯ [6], and C++ [7] declares the fields and methods that its instances will have. Hence, a class is a *template* for the creation of its objects. A class, even when considered an object, does not have the fields and methods it describes. In Cyan, a prototype is a type and also a template for the creation of its instances. In addition, when a prototype is used inside an expression, it is considered an object that has the structure it declares. For example, when used inside an expression, prototype `Int` is an object of type `Int` which is the same as `0`.

```
assert Int*Int + 1 + Int == 1;
assert String == "";
    // call String method 'size'
assert String size == 0;
```

Inside an expression, prototype `String` has type `String` and value `""`. Methods `*` and `+` are declared in prototype `Int`, and method `size` is declared in `String`. The following paragraphs present the basic syntax and semantics of Cyan, which are fundamental for understanding the rest of the article.

Listing 1 shows a prototype `Person` in Cyan with one constructor, "`init:`". It can also have a parameterless `init` method. Method declaration starts with `func`, followed either by a single identifier (for *unary methods*, without parameters) or a sequence of

```
ident: parameters
```

`ident` is called a *method keyword* or just a *keyword*. There may be more than one *method keyword* followed by parameter declarations, as in this example.

```
package main
object Person
  func init: String name, Int age {
    self.name = name;
    self.age = age
  }
  func name: String name  age: Int age {
    self.name = name;
    self.age = age
  }
  func getName -> String = name;
  func setName: String name { self.name = name }
  func getAge -> Int = age;
  func setAge: Int age { self.age = age }
  var String name
  var Int age
end
```

Listing 1: *`Person` prototype in Cyan*

```
  func at: Int line, Int column
       put: String text { ... }
```

This method is called as in

```
  panel at: 20, 75  put: "A";
```

This is called a *message passing* or, more specifically, a *keyword message passing.* A call without parameters, such as `person getName`, is called a *unary message passing.* The Cyan syntax for method declaration and message passing is similar to Smalltalk [4], although with one important difference: in Cyan, after a method keyword, there may appear zero or more than one formal parameter declaration or, in case of a message passing, zero or more than one argument. In Smalltalk, every keyword is linked to exactly one parameter. The return type of a method is given after `->`. If not provided, `Nil` is assumed. Methods are public by default.

Prototype `Person` declares get and set methods for the private *fields* `name` and `age`. Fields are sometimes referred to as instance variables, data members, or attributes. Fields declared with the Cyan keyword `let` instead of `var` are read-only fields; they can only be assigned to in their declarations or in constructors. Fields declared without `var` or `let` are read-only. The fields and methods shared by all objects of the same prototype are preceded by the keyword `shared`. A shared method can only access shared fields. `self` is a pseudo-variable that refers to the object that received the message. It is the same as `this` in Java [5], C♯ [6] and C++ [7] or `self` in Smalltalk and Swift [8].[1]

The Java language supports single-class and multiple-interface inheritance. The constraints in Cyan are the same as those in Java, even though the concept names are different. Let us explain that. In Cyan, keyword `interface` is used instead of `object` to specify a prototype that declares only method signatures (methods without bodies). The compiler adds bodies that throw exceptions when called (they should not be called). Interfaces function primarily as types.

To make the explanation of inheritance precise, in this paragraph non-interface prototypes will be called *ni-prototypes* (those declared with "`object`"). A ni-prototype can inherit from just one other ni-prototype using the keyword `extends` (like in Java, where a class can only inherit from a single superclass). However, it can inherit from multiple interfaces through keyword `implements` (as in Java). An interface cannot implement any other interface, but it can inherit from several other interfaces (as in Java). A ni-prototype should define all methods declared in its superinterfaces unless it is an *abstract* prototype (declared with the keyword `abstract` before `object`). Therefore, Cyan supports only single inheritance (in the common use of these words) because there may be just one superprototype that is not an interface. However, it supports multiple inheritance in relation to interfaces. Hereafter, superprototypes that are not interfaces are called *superprototypes*.

There is a top-level prototype called `Any` that is inherited (directly or indirectly) by all prototypes, except `Nil` (this prototype will be detailed later). The basic prototypes of Cyan are `Byte`, `Short`, `Int`, `Long`, `Char`, `Boolean`, `Float`, `Double`, and `String`. They all inherit from `Any` and, therefore, are *reference* types. Conceptually, a variable of type `Int` does not hold an integer. Instead, it *refers to* a dynamically allocated `Int` object. Similar to Scala [9] and unlike boxed objects in Java, methods `==` and `!=` compare the values of the basic type objects (not their pointers).

For every `init` and `init:` constructor of a prototype `T`, the compiler adds methods

```
func new -> T { ... }
func new: paramDeclaration -> T { ... }
```

to `T`, which are responsible for allocating memory to the object and initializing it. Hence, object creation in Cyan is achieved using methods. This is unlike most class-based object-oriented languages because they use operator `new` or special constructs. As a result, a `new` or `new` method can be called using runtime metaprogramming as any other method.

[1]The language from Apple, not the original Swift language (http://www.swift-lang.org).

The *name* of a unary method is its sole identifier, such as `getName`. The *name* of a *keyword method* is the concatenation of all its keyword names, each followed by the number of parameters and a white space.

```
func at: Int n, String s
     with: Person p { ... }
```

The name of this method is `"at:2 with:1"`. The *selector* of a unary method is its identifier. The *selector* of a *keyword method* is the concatenation of its keywords. In the last example, the selector is "`at:with:`".

A subprototype can *override* an inherited method with the same *name* by using the keyword `override`, which is required. The parameter types in the subprototype method should be equal to the corresponding types in the superprototype method, unless keyword `overload` is used (explained later). The return type of the subprototype method can be a subtype of the return type of the superprototype method (covariant rule [10]). Because the *name* of a method uses only its keywords and the number of parameters associated with each keyword, the compiler can differentiate between two methods without examining any type.

Cyan supports compile-time objects called *metaobjects* that can change the compilation process by temporarily adding code (the source files are not altered) and doing additional checks beyond those made by the compiler. The Cyan Metaobject Protocol (MOP) [11] describes the interactions between metaobjects, the compiler, and the source code. Although the Cyan MOP is not the main subject of this article, some basic knowledge about metaobjects should be explained because many Cyan features benefit from and interact with them. In the source code, a metaobject is linked to an *annotation* attached to a declaration (prototype, method, field), statement, or other syntactic elements. In the following example, `immutable`, `init`, and `restrictTo` are *annotations*.

```
@immutable object Earth ... end
@init(name, age) object Person ... end
  // inside a method
  var Int@restrictTo{* self >= 0 *} age;
```

An annotation can have arguments such as `name` and `age` in `init` and an *attached DSL* between a pair of sequences of symbols, usually `{*` and `*}`, as in `restrictTo`. In this example, `immutable` checks whether the prototype is immutable, `init` generates a constructor for `Person`, and `restrictTo` inserts checks to ensure that no negative value is assigned to `age`. It is important to stress that, for each annotation, there is an *associated* metaobject, and vice versa.

3. Type Checking and Gradual Typing

The main Cyan types are shown in Figure Figure 1. A dashed line indicates that the bottom type is a subtype of the top type, although no inheritance relationship exists between them. A type is a subprototype

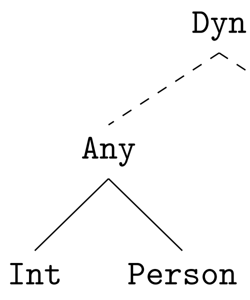

Figure 1: Type and prototype hierarchies in Cyan

of a type that is above it if there is a regular line between them. Subprototypes are always subtypes.

Type `Dyn` is the supertype of all prototypes and even of all Java classes when mixing Cyan and Java code. `Dyn` is not a prototype, and there is no source code associated with it. For these reasons, it is called a *virtual type*. `Dyn` cannot be used inside expressions and it disables type checking (more on that later). Prototype `Nil` cannot be inherited, and it does not inherit from any other. A prototype that does not explicitly inherit from any prototype inherits from `Any`. Therefore, `Any` is the top-level prototype. Figure 1 exemplifies that every basic type, such as `Int`, and every developer-defined prototype (without an explicit superprototype), such as `Person`, inherits from `Any`. The decisions associated with the choice of this hierarchy design will be presented later.

Union and Intersection Types. A *union type* is defined as two or more types separated by `|`, as shown in this example:

```
var String|Int|Any|Nil every;
var Int|Dyn id;
```

In a union $\mathtt{T}_1|\mathtt{T}_2|\ldots|\mathtt{T}_n$, supertypes should appear later in the list ($\mathtt{T}_i$ cannot be supertype of $\mathtt{T}_j$ if $i < j$). Therefore, union `Any|Int` is illegal. This restriction is intended to make unions easier to read.

An intersection type is composed of interface names separated by &, as in

```
var Closeable & Printable ca;
```

The compiler guarantees that at runtime, in an assignment `ca = expr`, the `expr` type implements both interfaces. *Unions* and *intersections* are considered *virtual types* because there are no source code associated with them.

Subtyping Rules. A Cyan type is either a prototype (including `Nil`), a union type, an intersection type, or `Dyn`. A type `S` is a subtype of `T` if an object of `S` can be used whenever an object of `T` is expected. The Cyan subtyping rules are given next in which `S` <: `T` means that `S` is subtype of `T`.

(a) `T` <: `T` for every type `T`.

(b) If prototype `S` inherits from prototype `T`, then `S` <: `T`.

(c) If a prototype `S` implements interface `I`, then `S` <: `I`.

(d) if `S` <: `T` and `T` <: `U`, then `S` <: `U`.

(e) If, for each $\mathtt{U}_j$, $1 \leqslant j \leqslant m$, there is a $\mathtt{T}_i$, $1 \leqslant i \leqslant n$, such that $\mathtt{U}_j$ <: $\mathtt{T}_i$, then $\mathtt{U}_1|\mathtt{U}_2|\ldots|\mathtt{U}_m$ <: $\mathtt{T}_1|\mathtt{T}_2|\ \ldots\ |\mathtt{T}_n$. There are two special cases of this rule: (i) if $\mathtt{U}_i$ <: `T` for every $1 \leqslant i \leqslant m$, then $\mathtt{U}_1|\mathtt{U}_2|\ldots|\mathtt{U}_m$ <: `T` and (ii) if there is an i, $1 \leqslant i \leqslant n$, for which `U` <: $\mathtt{T}_i$, then `U` <: $\mathtt{T}_1|\mathtt{T}_2|\ldots|\mathtt{T}_n$.

(f) If, for each $\mathtt{T}_i$, $1 \leqslant i \leqslant n$, there is a $\mathtt{U}_j$, $1 \leqslant j \leqslant m$, such that $\mathtt{U}_j$ <: $\mathtt{T}_i$, then $\mathtt{U}_1\&\mathtt{U}_2\&\ldots\&\mathtt{U}_m$ <: $\mathtt{T}_1\&\mathtt{T}_2\&\ \ldots\ \&\mathtt{T}_n$. There are two special cases of this rule: (i) if, for each i, $1 \leqslant i \leqslant n$, `U` <: $\mathtt{T}_i$, then `U` <: $\mathtt{T}_1\&\mathtt{T}_2\&\ldots\&\mathtt{T}_n$ and (ii) if, for some j, $1 \leqslant j \leqslant m$, $\mathtt{U}_j$ <: `T`, then $\mathtt{U}_1\&\mathtt{U}_2\&\ldots\&\mathtt{U}_m$ <: `T`.

(g) `T` <: `Dyn` for every type `T`.

(h) `Nil` <: `Dyn` and, if `T` <: `Nil`, then `T` is `Nil` (`Nil` has no subtypes but itself).

When a method has no explicit return type, it returns `Nil`. As this prototype does not have subtypes, the return value is always the same. Hence, the compiler knows the value returned at the call site.

Safe Downcasts. Downcasting is the conversion of an expression whose type is `T` to a subtype `S` of `T`. There are two statements for safe downcast in Cyan: `type-case` and `cast`. The syntax of `type-case` is:

```
type expr case T v { ... }
```

where the `case` clause can be repeated several times. At runtime, if `expr` can be cast to `T`, its value is assigned to `v`, and the statements inside `...` are executed. The cast test is done in the textual order of the `case` clauses. The `cast` statement is a short form of `type-case`. Statements `type-case` and `cast` are the only ways of casting a reference to a union type value to a more specific type (such as casting `String|Nil` to `String`).

In Cyan, `Nil` is not a subtype of any prototype. Therefore, this object can only be assigned to a variable if its type is `Dyn`, `Nil`, or a union in which one of the types is `Nil`, such as in `Int|Nil`. To retrieve the possible non-`Nil` value referenced by the variable, we use `type-case` or `cast` statements. Hence, Cyan is a `Nil`-safe language; runtime errors never happen because of `Nil` and prototypes.

Method Overloading. A language supports method overloading if two or more methods declared in the same class or prototype have the same name. This is usually a source of confusion for the developer [12] [13] because it is not always clear which methods are associated with a message passing. Some, but not all, reasons for this follow.

(a) For a message passing, two methods declared in the inheritance hierarchy can be called for a given argument. A more specific method in a superclass and a closer one in a subclass. An example is shown

```
1    // Y inherits from X
2    // X defines  void m(A obj) and
3    // Y defines  void m(Object obj)
4  X x = new Y();
5  x.m( new A() );
6  Y y = new Y();
7  y.m( new A() );
8  y.m( (Object ) new A() );
```

Listing 2: This code uses Java syntax. Its semantics vary from language to language.

in Listing 2 using the Java syntax. In it, a class `X` defines a method `void m(A obj)`, and its subclass `Y` defines a method `void m(Object obj)`. Which method should be called in the message passing of line 7? The nearest one, in `Y`, or the more adequate method, in superclass `X`? In the general case, there may be several suitable methods for a message passing, considering subtype relationships between the real arguments of the message and the formal parameters of the method.

(b) Two message passings that use the same objects at runtime may call different methods because at compile time the objects have different types. In Listing 2, lines 7 and 8 may call different methods (they do in Java).

(c) For a message passing, there may not be a method with the same formal parameter types as the real arguments of the message. But there may be various adequate methods if *automatic casting* between basic types is allowed. For example, `x.p(0, 1)` could call a method `p(double, byte)` or `p(byte, double)` (among many other possibilities).

(d) Default values for parameters and overloading may create ambiguity and confusion.

```
void m(Int a, Int b = 1) { ... }
void m(Int a) { ... }
```

Which method should be called for `m(0)`?

(e) Interfaces (like those of Java and Kotlin) and multiple inheritance makes the confusion even worse.

Cyan restricts the declaration of overloaded methods in several ways to avoid some problems with this construction. When a method is declared with keyword `overload`, the prototype may declare several methods with the same name but different parameter types. The return type of all methods with the same name in a prototype should be equal. The overloaded methods should appear after the first method with keyword `overload`. Hence, this is legal:

```
overload
func at:  Int n   put: Student s { ... }
func at:  Any n   put: Person p  { ... }
```

An overloaded method can be overridden in a subprototype. The parameter types of the overridden methods may differ from those of the superprototype. However, the return types of all overridden subprototype methods should be equal.

At runtime, a message passing triggers a search for an adequate method in the prototype of the receiver object. This search is trivial for regular methods and a bit more complex for overloaded methods. If an overloaded method can be called, the search also starts at the prototype of the object and looks for a method there with the expected name that *can accept* the real arguments. That is, if the type of each real argument is a subtype of each formal method argument. The first method tested is the first one *textually* declared in the prototype. The search for an adequate method continues in *textual* order. If the method is not found in the prototype of the receiver object, the search continues in the superprototype. A method is always found at runtime because the compiler ensures that. Note two important things:

(a) the compile-time types of the message passing arguments and of the receiver object are not important in the search for a method;

(b) in a prototype, the runtime search for a method follows the *textually* declared order of the methods.

A consequence of this search mechanism is that Cyan supports a restricted version of multi-methods, like those of CLOS [14], through overloaded methods. The great difference from true multi-methods is that the methods are tied to a hierarchy of prototypes, and there is a special place for the message receiver.

The design of overloaded methods in Cyan was oriented to avoid the usual problems of this feature. Was it successful? Let us examine each of the causes of misinterpretations. The three first ones, (a), (b), and (c), are caused by the use of compile-time types for selecting a method at runtime. This does not occur in Cyan because the compile-time types of arguments are never important when selecting a method to be called at runtime. Issue (c) is also avoided because there is no automatic casting between basic types. Cyan does not support default values for parameters. Thus, the issue (d) does not happen. Item (e) cannot be avoided. However, this issue is not serious because the language does not support multiple inheritance, and overloaded methods cannot be defined in interfaces. Keyword `overload` cannot be used in interfaces, and a prototype cannot implement interfaces declaring methods overloaded in it.

Some languages, such as Java, choose the method to be called at compile-time (if inheritance is not used) based on the types of the message passing arguments. This is better for efficiency reasons because the compiler reduces the number of methods that can be called at runtime. There is more static information that is known by both the compiler and the developer. In this sense, Java and similar languages are more readable than Cyan.

The method to be called, in all object-oriented languages, depends on the type of the receiver argument. In languages supporting multi-methods, the method to be called depends on all argument types. That means runtime information is more important than in languages that do not support this mechanism. Cyan tries to give precedence to runtime information in such a way that the developer is never in doubt if the method to be called for a message passing will be decided based on runtime types of objects or compile-time types. It is always based on runtime types.

Gradual Typing. Cyan supports *gradual typing* [15] [16], which allows a mixture of dynamically typed and statically typed code in the same program, even in the same expression. Expressions of type `Dyn` can be mixed with expressions of other types. The compiler will do all possible type checkings and insert code to do the checks that can only be done at runtime.

```
  var Dyn dyn = 0;
  var extremes = [ 'a', 'z' ];
  var ch = extremes at: dyn*2;
  assert ch == 'a';
```

The message passing `dyn*2` has type `Dyn`. However, `extremes` has type `Array<Char>` and the compiler is able to deduce that to `ch` is assigned a `Char` value. The compiler can calculate the return type of `at:` of `Array<Char>` because all methods with the same name should have the same return type. This design decision helps contain the spread of `Dyn` types in expressions.

The compiler stops doing type checking and inserts code for runtime type checking in two situations:

(a) an expression of type `Dyn` is used where an expression of type `T` (different from `Dyn`) is expected. The compiler inserts code that checks if the object resulting from the expression has a type that is a subtype of `T`. For example, in an assignment of the kind "`T = Dyn`" or when an expression of type `Dyn` is used after the keyword `if` or `while`;

(b) a message passing whose receiver has type `Dyn` at compile time. The compiler inserts a test to check if, at runtime, the object has the method corresponding to the message passing.

To make dynamically-typed programming in Cyan easy, the developer does not need to supply type `Dyn` on two occasions: when declaring method parameters and in the declaration of local variables (without assigning an expression to the variable).

```
     // n has type Dyn
func at: n   with: String s {
  var k;  // has type Dyn
  var m = 0; // has type Int
  ...
}
```

One of the goals of gradual typing is to make it easy for the developer to code the program using dynamic typing and later convert the program to static typing, at least partially. This is done by replacing type `Dyn` with another type (a union or a real prototype) in the declaration of local variables, parameters, fields (called collectively *variables*), and in the return type of methods. After that, the compiler can associate non-`Dyn` types with some expressions that had previously type `Dyn`. This replacement can cause compile-time errors, and the code may need to be corrected by the developer. However, we set a reasonable language design goal that, when there are no compile-time errors, the semantics of the new program should not be different from the old one. Let us study whether this goal was achieved or not. Type changes occur in variables and method return types. This also causes expressions to change types, which influences:

(a) statements checked at compile time, such as assignments, `if` statements, and method `return` statements. By design, their semantics are not attached to the static type of the expressions or variables used;

(b) message passings. The receiver and message arguments may change types. The semantics is not changed because the runtime search for a method after a message passing does not use the static type of the expressions. This is true even if the method is overloaded.

To ease the transition from dynamic to static typing, `Nil` is used as the return type of methods that do not declare a return type. Hence, we can always assign the value returned by a method to a variable of type `Dyn`, even if the return type is unknown at compile time.

As a complement to gradual typing, Cyan offers two alternative mechanisms for message passing. The first one allows the calling of a method using its name as a string (`` obj `messageAsString: args ``). The second one suspends compile-time type checking for the message passing when the message keywords or the unary message name is preceded by `?`, as in `obj ?at: 0`.

Anonymous Functions. An anonymous function (or just function) with parameters of types T_1, T_2, ..., T_n and a return type `R` is defined as

$$\{\ (:\ T_1\ t_1,\ \ldots\ T_n\ t_n\ \texttt{->}\ R\ :)\ \texttt{/* stats */}\ \}$$

The compiler transforms this function into a hidden prototype that inherits from $\texttt{Function<}T_1,\ T_2,\ \ldots\ T_n,\ R\texttt{>}$ which is an instantiation of a generic prototype (to be seen later). The hidden prototype overrides an `eval` (no parameters) or `eval:` method inherited from `Function<...>`. The compiler puts inside this method the statements `stats` of the function.

The code inside an anonymous function can access variables and fields visible at the static scope of its declaration. Keyword `return` is used to return a value from a method. It cannot be used inside an anonymous function. A function returns a value by supplying an expression after `^` as in the code

```
var zero = { ^0 };
```

```
object Prod(Int &product) extends Function<Int, Nil>
  override
  func eval: Int x {
    product = product * x;
  }
end
```

Listing 3: `Prod` is an context object

The code of anonymous functions cannot be reused because they are *literal* objects. *Context objects* are a generalization of functions in which fields can be bound to local variables and fields visible at the creation of an object. Listing 3 shows a context object `Prod` that inherits from `Function<Int, Nil>`, the type of functions that take an `Int` as parameter and returns `Nil`. `Prod` declares a field `product` preceded by `&`, which suggests that this field is some kind of reference. It is, and it refers to the variable or field passed as a parameter at the object creation time:

```
var p = 1;
  // calls Prod passing every array element as an argument
[ 2, 3, 5 ] foreach: Prod(p);
assert p == 30;
```

Any changes to the field `product` of `Prod` are immediately reflected in the local variable `p`. Context objects cannot cause runtime errors. There will never be a case in which a local variable removed from the stack is referenced in a context object.

Related Works. Unions, intersections, `Dyn`, `Any`, `Nil`, and the subtyping rules of Cyan are not novelties. All or some of these concepts appear in several languages, such as C♯ [6], Kotlin [17], Ceylon [18], Scala [9], and Scala 3 [19]. However, there are many differences between Cyan and other statically typed languages that support some form of null values. In Cyan, (a) there is no bottom type such as `Never` of Dart, `null` of Java, or `Nothing` of Scala; (b) like in Smalltalk and Dart, and unlike most statically typed languages, `Nil` is a real type (there are methods defined in it);[2] and (c) the return type of methods that do not return anything is `Nil`. The last item is particularly important because the caller knows that the return value can only be the object `Nil` (this prototype has no subprototypes).[3]

By its definition, method overloading causes uncertainty because it associates two or more methods with the same name (they can be considered a single method with numerous implementations). The confusion

[2]Even its source code is available!

[3]For efficiency reasons, the generated code does not really return object `Nil`.

increases when the methods belong to different classes and interfaces that are inherited and implemented. Cyan limits this problem by: (a) demanding that the top-level declaration of an overloaded method be preceded by the keyword `overload`; (b) prohibiting overloaded methods in interfaces. In any object-oriented language, the method called at runtime because of a message passing is always the most specific in the receiver class hierarchy. The design of method overloading in Cyan ensures that this is valid even when using overloaded methods.

Gradual typing in Cyan is rather conventional, except for one point: the semantics is not changed when types are added or removed from the code. We have seen that in overloaded methods: the runtime search does not use the compile-time types. The meaning of all other statements of the language remains the same when `Dyn` is replaced by a prototype, and vice versa. This can be checked by a tedious process of examining all statements (not done here).

With *environmental acquisition* [20], objects can acquire behaviors from their containers at runtime. As an example, a `Door` object of a car can acquire its color from the `Car` object in which it is part of. This specific example is simulated in Cyan using *context objects* (prototypes with reference fields). In the constructor of the prototype `Car`, each `Door` object would be created by passing the field `color` as an argument. Prototype `Door` could be declared as

```
object Door(Int &color)
```

Any changes in the car color would be reflected in each door object.

To our knowledge, *context objects* are not directly related to any construction of other languages. They can be simulated in a language that allows inheritance from function types and supports C-like pointers. In the example of Listing 3, `Prod` inherits from `Function<Int, Nil>` (a function type) and `product` is a field implemented as a pointer to an `Int`. The importance of context objects is that they permit the *safe* reuse of code. Instead of using an anonymous function in one place, the developer can put its code in the `eval:` method of a context object reused multiple times. Even the access to local variables can be abstracted in the context object. Of course, this code reuse can be done using regular objects, albeit in a more complex way.

4. Object Initialization

Prototypes have *fields*, which are *instance fields* or instance variables, and *shared fields*, shared by all objects of the same prototype. Two different mechanisms initialize these two categories. Let us study non-shared fields first.

Initialization of Object Fields . Fields are initialized in their declarations or in methods `init` or `init:`, which are called *constructors*. The latter should have at least one parameter. Every `init` and `init:` method should initialize every field that is not set in its declaration. The declaration of a field can assign to it a *Safe*

```
object Dangerous
  func init {
    LeakSelf leak: self;
    liquid = 0;
  }
  func getLiquid -> Int = liquid;
  var Int liquid;
end
```

Listing 4: Leaking `self`

Expression (SE) that is a basic type (such as `Int`), a basic value optionally preceded by a unary operator (`5` or `-5`), a literal array, a map, a tuple, or an object creation. The arguments or elements of these arrays, maps, etc should be SEs (such as `Array<Char>(4)`, `4` is a SE).

A prototype may not define any constructors if all fields are initialized in their declarations. In this case, the compiler creates an empty `init` method for it. If a superprototype `S` of a prototype `P` defines an `init:` but not an `init` method, every `P` constructor should call the method `init:` of `S` as its first statement. The compiler inserts a call to `init` of `S` in `P` constructors if the superprototype does not define any `init:` method.

One problem with object initialization is *self leakage*, which is to pass it as a parameter to a method inside the constructor. Then the method can use an uninitialized field of `self`. As an example of this case, in Listing 4, the constructor calls method `leak:` of `LeakSelf` before initializing the field `liquid`. If method `leak:` calls method `getLiquid`, it would access an uninitialized `liquid` field (as will be seen, the compiler issues an error in this example). To prevent this and other problems, there are restrictions on the statements inside constructors:

(a) a field can only be used in expressions after it has been initialized;

(b) `self` can only be used in two situations: (i) the prototype is *final* (it cannot be inherited) and `self` appears after statements that initialize all fields; or (ii) `self` is the receiver of a message passing in which the method to be called is preceded by the annotation

    `@accessOnlySharedFields`

    The metaobject associated with this annotation ensures that the annotated method accesses only shared fields and does not leak `self`. This method can be overridden in subprototypes.

If these rules were not obeyed, a field could be used before its initialization.

Method Overloading for init: . There are special rules for constructor overloading in Cyan. First, the keyword `overload` does not need to be used because `init` and `init:` methods cannot be overridden in subprototypes. Hence, the compiler knows which methods should be called at runtime. There may be several constructors with the same number of parameters. However, for every two `init:` methods with the same number of parameters of the same prototype, there should be at least one n such that the type of the nth parameter of a method should not be a subtype or supertype of the type of the nth parameter of the other method. This avoids any ambiguity in the call to an `init:` method.

Initialization of Shared Fields . Every *shared field* should be initialized either in its declaration or in a special private method called `initShared`. In both places, only *Restricted Safe Expressions* (RSE) can be assigned to shared fields. An RSE is defined as a SE except that only object creations from package `cyan.lang` prototypes are allowed. The `initShared` method cannot have any statements that are not initializations of shared fields.

Related Works . Any attempt to solve the *initialization problem* locally is doomed to failure. Its causes are non-local to a prototype because objects may reference each other. Objects in a cycle of references should be set up simultaneously.

A group of objects that should be simultaneously initialized go through a series of states, starting with "no field has a value" and ending with "all fields of all objects have a value" [21]. In each state, only some object methods are available, which are those that only access the fields already set. These are the methods that can be called to help with the initialization process. Therefore, a mechanism for object initialization that takes care of all possible cases should link methods to the fields they access, clearly breaking encapsulation. There are other elements to consider:

(a) languages supporting null-safety do not allow null or equivalent values to be temporarily assigned to non-null fields during the initialization;

(b) in the initialization of a cyclic object structure, at least one object should be used before it is fully initialized;[4]

(c) immutable objects demand each field be assigned precisely one time;

(d) invariants inside and between objects should be kept;

(e) the leaking of `self` and message passings to `self` inside a constructor may call methods that access non-initialized fields;

(f) the number and classes of the objects that should be simultaneously initialized may not be known at compile time;

[4]If objects `x` and `y` should refer to each other, one of them should be passed to the constructor of the other before being fully initialized.

(g) the combination of all the above elements in a sole initialization.

There are several proposals [22] [23] [24] [25] [26] [27] that address some issues related to object initialization. They rely on annotations in the code and code analysis to prevent the full use of partially initialized objects.

Cyan provides mechanisms that prevent most problems associated with unsafe object initialization. These mechanisms are: (a) annotation `accessOnlySharedFields`; (b) limitations on the expressions that can be assigned to fields; and (c) restrictions on accesses to fields and `self`. In our limited experience, these mechanisms are not a burden to developers, since most constructors are simple. According to a Java data set studied by Gil and Shragai [28], only 8% of constructors can potentially send messages to `this` and, hence, call methods in subclasses. In practice, subclass methods are called inside constructors in only 1% to 2% of the cases. This is because either the class is not inherited or the methods are not overridden in subclasses. These numbers seem to suggest that constructors, in general, do not need to process their arguments. If we extrapolate these numbers to Cyan, which uses a Java-like inheritance mechanism, we can conclude that the `accessOnlySharedFields` annotation will be used in a few cases.

None of the solutions to the initialization problem are easy to use and cover all cases nicely. This is because of its inherent complexity. The Cyan solution is minimal and works in most cases we have found in practice. When it does not work properly, the developer is forced to change some field types to unions `T|Nil` (instead of just `T`) and make the initialization in (usually) two steps. The first one assigns `Nil` to some fields in the constructor, and the second step assigns a non-`Nil` object to them. It is important to note that Cyan partially solves the initialization problem. A field can be used before initialization in the following scenario: a prototype `InitBug` declares a non-shared field `value` and defines only an `init: Int` constructor. In the first line of the following example, the `prototype` method returns a reference to `InitBug`.

```
a = InitBug(5) prototype;
a getValue println;
```

In the second line, the field `value` is accessed without being initialized because there is no `init` constructor.

5. Limitations for Prototypes

Class-based languages that consider everything as objects [4] are forced to supply a class for a class, its metaclass. And then a class for a metaclass, and so on, which gives rise to a problem called "the infinite regression of metaclasses" [29]. In a prototype-based language, a prototype is both a template for the creation of objects and an object itself, thus avoiding infinite regression. However, a new issue is created, described in this Section, which is a tension between safe object initialization and the main feature of prototype-based programming, which is to consider prototypes as ready-to-use objects. Without loss of generality, assume that only constructors do object initialization. If all or some fields are initialized in their declarations, this is the same as adding all declaration assignments at the start of every constructor. If there is no developer-declared constructor, all fields should have been initialized in their declarations. The compiler uses these initializations to create a parameterless constructor.

Non-prototype objects are created by developer code that can choose the arguments for constructors. That is not the case with *prototypes* when considered as objects. They should be created by the Runtime System (RTS) before their first use using an `init` or an `init:` method. If there is no parameterless `init` constructor, the RTS should choose default arguments for one of the `init:` methods. For example, the empty string and `0` would be used as arguments for a `Planet` constructor, representing the name and mass of a planet. Clearly, default arguments may result in the assignment of non-valid values to fields.

There is another problem with this solution: which values should be used as arguments if the corresponding types were created by the developer? It cannot be `Nil` because all types but some unions are non-nullable. One solution would be to use the prototype itself. For example, prototype `Company` would be used if a constructor of prototype `Employee` took a `Company` parameter. Again, non-valid values could be assigned to fields, and object invariants could not be obeyed. We conclude that there is a conflict between the ability to use prototypes as objects and safe object initialization.

Some prototypes can only be properly initialized with constructors that take parameters. Because of that, Cyan does not consider prototypes without an `init` constructor as objects, except in a few special situations:

(a) the prototype receives a message whose name is `new` or `new:`;

(b) the prototype receives a message whose corresponding method is inherited from `Any` and annotated with `canBeCalledOnPrototypes`. The compiler guarantees that the method does not access any prototype field. Among these methods, there are `isA:` (returns `true` if the receiver is an object of the argument, which should be a prototype) and several methods declared in `Any` for introspective runtime reflection (they give information on the prototype itself);

(c) the prototype is `Nil`.

In all these cases, no field of the prototype is accessed, and hence there is no risk of a runtime error. We consider Cyan a prototype-based language because prototypes with an `init` method can be used as objects. However, someone could consider that, because of this restriction, Cyan is in the intercession of prototype and class-based languages (or none of them).

6. Generic Prototypes

A generic prototype takes one or more parameters that allow the generation of a specialized version of the prototype at compile time. An example is `Box` of Listing 5. An *instantiation* of `Box` creates a new and

```
package main
  // creates a constructor
@init(value)
object Box<T>
    //create get and set methods
  @property var T value
end
```

Listing 5: The generic prototype `Box`

specialized version of this prototype by supplying a *real argument* at compile time:

```
  var intBox = Box<Int> new: 0;
  var Box<String> strBox;
```

Here, `Int` and `String` are the real arguments to `Box`. For each set of arguments, different prototypes are created that do not relate to each other.

Generic prototypes may occasionally use the concept of *identifier*, which is a sequence of letters and numbers starting with a lowercase letter. For example, `write`, `speed`, and `year2022`. These will be called *lowercase identifiers*, which are never confused with types. A type name starts with an uppercase letter optionally preceded by the package name as `Person` or `main.Tree`. Both lowercase identifiers and types can be real arguments for generic prototypes. The documentation of a prototype defines whether a certain parameter should be one or the other.

In the *instantiation* process, the compiler replaces a formal parameter by the corresponding real argument, in the body of the prototype, if the parameter appears: (a) where a type is expected (including expressions); (b) in an annotation (either as an argument or in the *attached annotation text*); (c) after `#` (`#str` is the same as `"str"`); (d) as a method name.

A real argument for a generic prototype instantiation may be a *lowercase identifier*. For example, the instantiation `MyList<Int, speed>` may create a list optimized for speed. A metaobject associated with an annotation inside `MyList` would create the prototype code based on the second argument, which could be `speed` or `space`.

A prototype that is not generic may be declared with the generic prototype syntax, such as `Box<Char>` of Listing 6. If each parameter between `<` and `>` is a type or a lowercase identifier, as in this example, we have a *generic prototype with real arguments* (GPRA). The compiler differentiates between a GPRA and a regular prototype using the file name in which the prototype is (`Box(Char).cyan` and `Box(1).cyan`). The same base name `Box` can be used for generic prototypes with different numbers of parameters (there could

```
package main
@init(value)
object Box<Char>
  func toUpperCase { value = value toUpperCase }
  @property var Char value
end
```

Listing 6: The generic prototype with *real arguments* `Box<Char>`

be, in package `main`, a `Box` with two parameters, for example). A generic prototype with a varying number of arguments (GPV) is declared as

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

In an instantiation of this prototype, the real arguments can only be accessed through a metaobject. The above code is the full definition of prototype `Tuple` used for the creation of literal tuples. The methods and fields are created by metaobject `createTuple` based on the real arguments. For example, when the compiler finds

```
[. 0, "zero", '0' .]
```

it creates an object of

```
Tuple<Int, String, Char>
```

During the instantiation, metaobject `createTuple` generates the fields and methods of the prototype.

A package may have regular generic prototypes (with various numbers of parameters), GPRA, and GPV with the same base name. In an instantiation, like `Box<Int>` or `Box<Char>`, the compiler will look for an adequate prototype in the following order: list of GPRA, regular generic prototypes, and GPV.

Concepts [30] [31] are constraints on real arguments to generic classes, functions, and prototypes. An attempt to pass an invalid argument to a generic structure results in a compile-time error. Cyan itself does not support concepts, which are implemented through metaobject `concept`. Listing 7 shows a new version of the generic prototype `Box` with the annotation `concept`. In line 3, the metaobject demands that the real argument `T` have a method `<` with the given signature.[5] If the requirement is not satisfied, there is a compilation error issued by the metaobject. There may be a sequence of instantiations that leads to `Box<T>`.

[5]The signature of a method is composed of its keywords, parameter types, and return type.

```
package main
@concept{*
    T has [ func < T -> Boolean ]
*}
@init(value)
object Box<T>
  func < (Box<T> other) -> Boolean =
      value < other getValue;
  @property var T value
end
```

Listing 7: `Box` with annotation `concept`

For example, prototype `Program` instantiates `A<Int, Person>` that instantiates `B<Person>` that instantiates `Box<Person>` that causes the error. The compiler shows this sequence in the error message and shows the filename and line number of each instantiation.

The *concept language* of the metaobject `concept` has several kinds of statements that test whether a certain condition is satisfied or not. The types used may be formal generic prototype parameters (as `T`), regular prototypes (as `String` or `Array<Int>`), or calls to the compile-time function `typeof` (which returns the type of an expression). The available statements of the *concept language* test whether: two types are equal; a type implements an interface; a prototype is a superprototype or subprototype of another; a type is an interface or a non-interface prototype; a type declares a list of methods (used in the `Box` example); a type is in a list of prototypes; a generic prototype parameter is a lowercase identifier; the negation of any of these statements is true. Furthermore, a file with concept language code may be imported, which allows the reuse of concepts. The `concept` metaobject also supports *axioms*, which automatically generate test cases in some special directories managed by the compiler.

Related Works. Concepts are supported by languages G [32], C++ [33], JavaGI [34], Java [5], Scala [35] [36], C♯^{cpt} [37], Haskell [38], Rust [39], C♯ [6], Swift [8], and Genus [40]. These languages are vastly different in the level of concept support and in the features they offer, as verified by Belyakova [41], Garcia et al [42], and Siek [43]. The comparisons made by these authors are not repeated here (that would require a whole article). Instead, we emphasize what is already supported in Cyan [44]. Metaobject `concept` supports most features of concepts of the languages cited above except:

(a) *modeling.* A type may model a concept in two different ways. That is, there is more than one way for a type to adapt to a concept (`<` for strings may use lexicographic order or just the size of the string).

This is supported by C$\sharp^{cpt}$ and Genus;

(b) *retroactive modeling*, to adapt a type to a concept after its release. For example, supplying a method to a class through *extension methods.* This is supported by C$\sharp^{cpt}$, Genus, Rust, Swift, Haskell, G, and JavaGI;

(c) *associated types*, which are types derived from the generic prototype parameters (a concept can place restrictions on types of formal method parameters, for example). This is supported by Haskell, Scala, Rust, Swift, G, and C$\sharp^{cpt}$. Cyan supports associated types only partially.

Some characteristics of the `concept` metaobject that are unique to Cyan or supported by only a few languages are:

(a) *axioms*, which are used for generating test cases (also supported by Magnolia [45]);

(b) customization of error messages (the custom message can be put after the concept statement);

(c) the developer can change it; she or he can implement his or her own concept metaobject. Unlike all other languages we know, concepts are not part of the language. Since metaobjects have access to the AST of the prototype, almost everything can be checked by the associated metaobject DSL code (written in the concept language);

(d) the `concept` metaobject is not limited to generic prototypes. Its annotations can also be used in non-generic prototypes (as C++ concepts), thus playing the role of a compile-time specification language.

Generic prototypes in Cyan benefit from metaprogramming. Metaobjects are used for dealing with a varying number of parameters and generating code. This allows the creation of type-specific code for each instantiation. As an example, a metaobject creates methods for the `Tuple` prototype based on its real type arguments. Another interesting example is the `createArrayMethods` annotation that is attached to generic prototype `Array`. At compile time, the metaobject associated with this annotation adds some methods to the prototype, such as `hashCode`, `asString:`, and `sort` (if the type parameter defines a method `<=>`).

7. The Exception Handling System

The exception handling system (EHS) of Cyan was based on that of Green [46] [47], although with several improvements. Statement

```
throw expr
```

throws the exception `expr`, whose type should be a prototype inheriting from prototype `CyException` of package `cyan.lang`.

The format of the `try-catch` statement in Cyan is shown in Listing 8. After each `catch` *clause*, there should appear an expression called *the catch expression.* Usually, this expression is an anonymous function, as shown in Listing 9. In this case, both the syntax and the semantics of Cyan are very similar to

```
try
  statement-list
catch expr1
catch expr2
...
catch exprN
finally {
  statement-list
}
```

Listing 8: The `try-catch` statement

Java/C++/C♯/etc. In particular, the statements of the `finally` *clause* (see Listing 8) are always executed, even if an exception was thrown and no `catch` clause caught it. Both the `catch` clauses and the `finally` clause are optional, but there should be at least one of them.

Let us explain how the example of Listing 9 works. Exceptions are thrown in lines 4 and 7. In line 4, prototype `ExcDivZero`, which is an expression, is thrown.[6] In line 7, the negative number is passed to the constructor of `ExcNegNumber`.

At runtime, if exception `ExcNegNumber` is thrown (line 7 is executed), the `try-catch` statement searches, among the objects of the `catch` clauses, for one that defines an `eval:` method that accepts an argument of type `ExcNegNumber`. Note that both `ExcNegNumber` and `ExcNegNumber` should inherit from `CyException`. The expression of the `catch` clause in line 10 is an anonymous function that declares a method

```
func eval: ExcDivZero e { "Division by 0" println }
```

Since `ExcNegNumber` is not a subprototype of `ExcDivZero` (assume this), the search continues in the next `catch` object (line 13), which declares method

```
func eval: ExcNegNumber e {
 Out println: (
   "Illegal negative number: " ++ (e getNumber));
}
```

This method can accept an argument of type `ExcNegNumber`. Hence, it is called to treat the exception. If the `catch` expressions are all anonymous functions, the semantics is exactly the same as in the languages Java/C++/C♯. The novelty in Cyan is that the `catch` expressions do not need to be functions. They can be

[6]There is no need to create a new object because there is no information associated with the exception.

```
...
try
  if num == 0 {
    throw ExcDivZero
  }
  if num < 0  {
    throw ExcNegNumber(num)
  }
  share = value/num;
catch { (: ExcDivZero e :)
   "Division by 0" println
}
catch { (: ExcNegNumber e :)
   Out println: (
     "Illegal negative number: " ++ (e getNumber) );
};
```

Listing 9: Catching an exception

any expressions whose types declare at least one `eval:` method that accepts a subprototype of `CyException` as a parameter.

For example, one argument can be an object `CatchAll` of package `cyan.lang`.

```
object CatchAll
  func eval: CyException e { }
end
```

Therefore, the example could be changed to

```
try
  // same statements
catch CatchAll;
```

Now, only one `catch` clause is necessary because the `eval:` method parameter type (`CyException`) is at the top of the hierarchy of exceptions. This `eval:` method does nothing, and this last code is equivalent to the following Java code.

```
try { ... }
catch( Throwable e ) { }
```

The anonymous functions of Listing 9 that are expressions of the `catch` clauses can be transformed into method bodies of `eval:` methods, as shown in Listing 10. Keyword `overload` is required because there are two methods with the same name and number of parameters in the sole selector. The code below has exactly the same semantics as the Listing 9 code, assuming the statements are the same.

```
try
  // same statements
catch CatchErrNum;
```

`CatchErrNum` could have declared just one `eval:` method whose parameter type is the union

`ExcDivZero | ExcNegNumber`

A type-case statement would be necessary for taking different actions for each type.

The iterations between `throw` and the two other statements that cause sudden changes, `return` and `break`,[7] may be confusing. For this reason, some use patterns are prohibited in Cyan.

(a) There cannot be any `return` in the statements between `try` and `catch` if there is a `finally` clause. If the `return` appears only inside a `finally` clause, it is legal.

(b) A `break` keyword cannot be used to end a repetition statement if the `break` is between `try` and `catch` and there is a `finally` clause.

[7]`break` works like in language C: it finishes the repetition of `while`, `repeat-until`, and `for` statements.

```
object CatchErrNum
  overload
  func eval: ExcDivZero e { "Division by 0" println }
  func eval: ExcNegNumber e {
     Out println: (
        "Illegal negative number: " ++ (e getNumber));
  }
end
```

Listing 10: Catching an exception

The reason for both prohibitions is that it is not clear if the `finally` statements should be executed or not.

Related Works. The *exception handling system* (EHS) of Cyan is different from the EHS of most object-oriented languages in one point: after a `catch`, there should appear an expression whose type has at least one `eval:` method. If an anonymous function is used after the `catch` keywords, both the syntax and semantics of Cyan are very similar to languages Java/C++/C♯/Python.

Cyan has rules to prevent code from mixing exception capturing (with `try-catch`) with statements `return` and `break`. The mixing would be confusing because it is not always clear what would be the result of the interaction [12].

In Cyan, the EHS interacts with many language features:

(a) type system. The compiler checks whether every `catch` expression has an `eval:` method accepting one parameter whose type is a subprototype of `CyException`;

(b) method overloading. Conventional `try-catch` statements look for an adequate catch clause in the textual order of the declaration. To mimic this behavior, Cyan uses overloaded `eval:` methods in catch objects. The behavior is the same by design: the overload method to be called is also the first adequate method found in a textual order search;

(c) inheritance. A prototype that catches a family of related errors (for example, those related to file handling) can be subprototyped, and some catch methods can be overridden;

(d) polymorphism. The argument to a `catch` clause may be an expression. The prototype of the resulting object may vary every time the expression is evaluated at runtime.

```
try ...  catch catchVar;
```

Hence, error treatment can be easily changed at runtime due to polymorphism;

(e) generic prototypes. Several prototypes of package `cyan.lang` catch only some specified exceptions that are type arguments to them.

```
try ... catch CatchWarning<E1, E2>;
```

Exceptions `E1` and `E2` are caught; the others are propagated. The generic prototype `ExceptionConverter` takes an even number of type parameters that should be exceptions. It catches the exception of position `i` (even) and throws the exception of position `i+1` (odd).

```
try ...
catch ExceptionConverter<ExcNegNumber, ExcOutOfLim>;
```

Exception `ExcNegNumber` is caught, and `ExcOutOfLim` is thrown instead. That is used when combining two libraries that use different exception prototypes for equivalent errors;

(f) context objects. In some cases, the exception treatment needs to change local variables to correct the error. Usually, that is done using an anonymous function as the `catch` expression.

```
var Int n = 1;
try ...
catch { (: ExcNegNumber e :) n = 0 };
```

The anonymous function code can be reused if put inside a context object.

```
var Int n;
try ...
catch CatchNeg(n);
```

8. Conclusion

Cyan supports gradual typing, method overloading, partially safe object initialization, anonymous functions, generic types with concepts, and an exception handling system. Although these constructs are well known, Cyan introduces several novelties in their support. In relation to gradual typing, the semantics of the code does not change by replacing a prototype with `Dyn`, and vice versa. The language grammar was designed such that omitting the types of method parameters (making them of type `Dyn`) does not cause any ambiguity. In relation to method overloading, the compile-time types of message arguments are never considered when choosing a method at runtime.

The initialization of objects is partially safe. There are draconian rules for shared fields and more relaxed ones for regular fields. The rules ensure that, except in one particular known case, non-initialized fields are never used. *Context objects* are a safe way to reuse code that can access local variables. They can replace non-reusable anonymous functions. Generic prototypes can take an undetermined number of parameters, and the metaobject *concept* can check them, issuing customized error messages (if necessary). Test cases can be automatically generated. Regular developers can change this metaobject, adapting it to their needs.

The exception handling system (EHS) is object-oriented. Consequently, error handling code can be reused, and there may be a hierarchy of prototypes for error treatment. Besides that, the EHS interacts with other language features such as polymorphism, generic prototypes, and context objects.

A language is used because of its features and programming environment. Although the compiler is user-friendly (clear error messages), there is no IDE for Cyan. To address this flaw, we plan to implement a Language Server for Cyan. A Language Server Protocol (LSP)[8] defines a bridge between an IDE and a *language server* to provide features such as "go to definition" (declaration), code completion, information on hovering an element, quick fixes for errors, an outline of the program (its packages, files, types, and type hierarchies), and automatic build. Once the server is created, it can be used with all the main IDEs. Another alternative would be to use Xtext [48], a framework for implementing languages and their integration in the Eclipse IDE.[9] Although Xtext is associated with DSL, it can also be used with any language. Since Cyan is closely linked to Java, Xbase [49] could be used for implementing parts of the Cyan language expressions. Xbase is an expression language that can be used in Xtext through the concept of language inheritance. In this way, an implementation can inherit not only the syntax for expressions but also the associated tools such as a parser, an unparser, a compiler, and an interpreter.

Acknowledgments. This project was partially financed by FAPESP (São Paulo - Brazil) under Process number 2014/01817-3.

References

[1] LLVM, The llvm compiler infrastructure (Aug. 2022).
URL `https://llvm.org`

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

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

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

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

[6] C# language specification (Sep. 2023).
URL `https://learn.microsoft.com/en-us/dotnet/csharp/`

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

[8] Swift, The Swift Programming Language (Swift 5.6), 1st Edition, Apple Inc., 2022.
URL `https://docs.swift.org/swift-book/`

[8]https://microsoft.github.io/language-server-protocol

[9]https://www.eclipse.org

[9] M. Odersky, P. Altherr, V. Cremet, G. Dubochet, B. Emir, P. Haller, S. Micheloud, N. Mihaylov, A. Moors, L. Rytz, M. Schinz, E. Stenman, M. Zenger, The Scala Language Specification (2004).
URL `http://www.scala-lang.org`

[10] L. Cardelli, A semantics of multiple inheritance, in: Information and Computation, Springer-Verlag, 1988, pp. 51–67.

[11] J. d. O. Guimarães, The cyan language metaobject protocol (2022).
URL `http://cyan-lang.org/docs`

[12] J. Bloch, N. Gafter, Java Puzzlers: Traps, Pitfalls, and Corner Cases, 2005.
URL `http://www.javapuzzlers.com`

[13] J. Chan, W. Yang, J. Huang, Traps in java, J. Syst. Softw. 72 (1) (2004) 33–47. `doi:10.1016/S0164-1212(03)00040-2`.
URL `https://doi.org/10.1016/S0164-1212(03)00040-2`

[14] D. G. Bobrow, R. P. Gabriel, J. L. White, Object-oriented programming, in: A. Paepcke (Ed.), Object-oriented Programming, MIT Press, Cambridge, MA, USA, 1993, Ch. CLOS in Context: The Shape of the Design Space, pp. 29–61.

[15] J. Siek, What is gradual typing, last accessed in September 21, 2021 (2014).
URL `https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing`

[16] J. Siek, W. Taha, Gradual typing for objects, in: Proceedings of the 21st European Conference on Object-Oriented Programming, ECOOP'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 2–27.

[17] Jetbrains, The kotlin language (Jun. 2022).
URL `https://kotlinlang.org/`

[18] G. King, The ceylon language (Jun. 2022).
URL `https://ceylon-lang.org`

[19] D. Pollak, V. Layka, A. Sacco, Beginning Scala 3: A Functional and Object-Oriented Java Language, Apress Berkeley, CA, 2022.

[20] J. Gil, D. H. Lorenz, Environmental acquisition: A new inheritance-like abstraction mechanism, OOPSLA '96, New York, NY, USA, 1996, p. 214–231. `doi:10.1145/236337.236358`.
URL `https://doi.org/10.1145/236337.236358`

[21] F. Liu, O. Lhoták, E. Xing, N. C. Pham, Safe Object Initialization, Abstractly, Association for Computing Machinery, New York, NY, USA, 2021, p. 33–43.

[22] F. Liu, O. Lhoták, A. Biboudis, P. G. Giarrusso, M. Odersky, A type-and-effect system for object initialization, Proc. ACM Program. Lang. (OOPSLA) (nov 2020). `doi:10.1145/3428243`.

[23] M. Servetto, J. Mackay, A. Potanin, J. Noble, The billion-dollar fix - safe modular circular initialisation with placeholders and placeholder types, in: G. Castagna (Ed.), ECOOP 2013, Vol. 7920 of Lecture Notes in Computer Science, Springer, 2013, pp. 205–229. `doi:10.1007/978-3-642-39038-8\_9`.
URL `https://doi.org/10.1007/978-3-642-39038-8_9`

[24] A. J. Summers, P. Mueller, Freedom before commitment: A lightweight type system for object initialisation, OOPSLA '11, New York, NY, USA, 2011, p. 1013–1032. `doi:10.1145/2048066.2048142`.
URL `https://doi.org/10.1145/2048066.2048142`

[25] M. Fähndrich, K. R. M. Leino, Declaring and checking non-null types in an object-oriented language, in: Proceedings of OOPSLA 03, OOPSLA '03, New York, NY, USA, 2003, p. 302–312. `doi:10.1145/949305.949332`.

[26] T. Etzel, Flexible initialization of immutable objects, SPLASH Companion 2016, New York, NY, USA, 2016, p. 53–54. `doi:10.1145/2984043.2998541`.

[27] X. Qi, A. C. Myers, Masked types for sound object initialization, in: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '09, ACM, New York, NY, USA, 2009, pp. 53–65. `doi:10.1145/1480881.1480890`.

[28] J. Y. Gil, T. Shragai, Are we ready for a safer construction environment?, in: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, Springer-Verlag, Berlin, Heidelberg, 2009, p. 495–519. doi:10.1007/978-3-642-03013-0_23.

[29] J. d. O. Guimarães, The Green language type system, Comput. Lang. Syst. Struct. 35 (4) (2009) 435–447. doi:10.1016/j.cl.2008.09.001.
URL http://dx.doi.org/10.1016/j.cl.2008.09.001

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

[31] A. Sutton, B. Stroustrup, Concepts lite: Constraining templates with predicates, retrieved march 2021 (feb 2013).
URL isocpp.org

[32] J. G. Siek, A. Lumsdaine, A language for generic programming in the large, Sci. Comput. Program. 76 (5) (2011) 423–465. doi:10.1016/j.scico.2008.09.009.

[33] ISO/IEC, Iso/iec 14882:2020 programming languages — c++ (2021).
URL https://www.iso.org/standard/79358.html

[34] S. Wehr, P. Thiemann, Javagi: The interaction of type classes with interfaces and inheritance, ACM Trans. Program. Lang. Syst. 33 (4) (2011) 12:1–12:83. doi:10.1145/1985342.1985343.

[35] M. Odersky, L. Spoon, B. Venners, F. Sommers, Programming in Scala, Fifth Edition, Artima Incorporated, 2021.

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

[37] J. Belyakova, S. Mikhalkovich, Pitfalls of c# generics and their solution using concepts, Proceedings of ISP RAS 27 (3) (2015) 29–46. doi:10.15514/ISPRAS-2015-27(3)-2.
URL http://mi.mathnet.ru/tisp134

[38] P. Wadler, S. Blott, How to make ad-hoc polymorphism less ad hoc, POPL '89, Association for Computing Machinery, New York, NY, USA, 1989, p. 60–76. doi:10.1145/75277.75283.
URL https://doi.org/10.1145/75277.75283

[39] S. Klabnik, C. Nichols, The Rust Programming Language, 2nd Edition, No Starch Press, 2022.
URL https://doc.rust-lang.org/book

[40] Y. Zhang, M. C. Loring, G. Salvaneschi, B. Liskov, A. C. Myers, Lightweight, flexible object-oriented generics, SIGPLAN Not. 50 (6) (2015) 436–445. doi:10.1145/2813885.2738008.
URL http://doi.acm.org/10.1145/2813885.2738008

[41] J. Belyakova, Language support for generic programming in object-oriented languages: Peculiarities, drawbacks, ways of improvement, in: F. Castor, Y. D. Liu (Eds.), Programming Languages, Springer International Publishing, Cham, 2016, pp. 1–15.

[42] R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, J. Willcock, A comparative study of language support for generic programming, Proceedings of OOPSLA '03 (2003) 115doi:10.1145/949305.949317.
URL http://portal.acm.org/citation.cfm?doid=949305.949317

[43] J. G. Siek, A language for generic programming, Ph.D. thesis, Indiana University (August 2005).

[44] J. d. O. Guimarães, Concepts for generic prototypes in cyan (2022).
URL http://cyan-lang.org/docs

[45] A. H. Bagge, Constructs & Concepts: Language design for flexibility and reliability, Ph.D. thesis, University of Bergen, PB 7803, 5020 Bergen, Norway (2009).

URL http://www.ii.uib.no/~anya/phd/

[46] J. d. O. Guimarães, The Green language, Comput. Lang. Syst. Struct. 32 (4) (2006) 203–215. doi:10.1016/j.cl.2005.07.001.
URL http://dx.doi.org/10.1016/j.cl.2005.07.001

[47] J. d. O. Guimarães, The green language exception system., Comput. J. 47 (6) (2004) 651–661.
URL http://dblp.uni-trier.de/db/journals/cj/cj47.html#Guimar04

[48] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, 2nd Edition, Packt Publishing, 2016.

[49] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, M. Hanus, Xbase: Implementing Domain-Specific Languages for Java, in: GPCE, ACM, 2012, pp. 112–121.