# Closures for Statically-Typed Object-Oriented Languages

José de Oliveira Guimarães
Departamento de Computação
UFSCar, São Carlos - SP, Brazil
jose@dc.ufscar.br

**Abstract**

Closures increase considerably the level of a language by mixing access to local variables with remote execution of a set of locally-defined statements. However, to date closures have not been added to statically-typed languages because it is difficult to type them and runtime errors occur if local variables that no longer exist are accessed. We proposed a limited but quite general kind of closure for statically-typed object-oriented languages. They can be used in most situations normal closures can without introducing any runtime errors.

**Keywords:** closure, Smalltalk blocks, object-oriented languages, Green.

## 1 Introduction

A closure is a kind of routine that can be assigned to a variable or passed as a parameter to another routine. A closure can access the local state (local variables, parameters, methods, and instance variables) visible in the place it was defined. In an object-oriented programming language, a closure would be an object. Smalltalk [1, 3] supports a kind of closure called block. A block is an object composed by parameters and a sequence of statements specified between square brackets. For example, the statement

```
sumAll := [ sum := sum + each ].
```

assigns to `sumAll` a block — since blocks are objects, they can be assigned to variables. The statements of a block are executed when it receives a message `value`:

```
sumAll value.
[sum := sum + each] value.
```

`each` may be a local variable, parameter, or instance variable visible in the method in which the block is defined. In fact, it may be any variable visible at the block declaration — it may be a local variable of an enclosing block,[1] for example. A block can also send messages to `self` and `super`. To access variables visible at the block declaration is a great asset but also a dangerous operation: the block may refer to a local variable of a method call that has returned. In this case, when the block receives the `value` message and tries to access the local variable, a runtime error occurs. As an example, let us study the following code:

```
aBlock := anObject getBlock.
aBlock value.
```

Method `getBlock` is defined as

```
| sum |
...
^[ sum := sum + 1 ].
```

`sum` is a local variable of method `getBlock`, declared between vertical bars. The return value of the

[1]We use "enclosing method" for the method in which the block is defined. This is the same as "defining block method". Of course, the "enclosing block" of a block B is the block in which B is defined, if B is defined inside other block.

method follows the character `^`. The method returns a block that accesses local variable `sum`. The ellipsis means any piece of code.

When `aBlock` receives message `value`, the block refers to local variable `sum` of `getBlock` that no longer exists — a runtime error.

Blocks may have parameters, which are `a` and `b` in the example below.

```
    [:a :b | a > b]
```

The return value of the block is its last and sole expression, "`a > b`". The execution of a block with parameters is made with the `value:` selectors:

```
    greaterThan := [:a :b | a > b].
    ok := greaterThan value: 1 value: 2.
```

To `ok` is assigned `false`.

A block defined inside a method `m` is an object created at runtime when method `m` is called. To be more precise, if the block is in a statement S inside `m`, it is created when S is executed. As said before, we call `m` the "enclosing method" of the block. For short, blocks are associated to method calls, not methods.

A return statement inside a Smalltalk block refers to the enclosing method. If the block survives the method call that created it, the return statement refers to a method call that no longer exists. In this situation, a runtime error will occur if message `value` is sent to the block. Let us show an example of that. Method `setBlock` is defined as

```
| n |
...
b := [ n >= 0 ifTrue: ^1 ifFalse: ^0 ].
```

in which `b` is an instance variable. After calling `setBlock`, a method of the same class may be called and may execute the above block:

```
    b value.
```

Inside the block there will be a return (`^1` or `^0`) from a method call of `setBlock` that no longer exists — a runtime error.

To our knowledge, only dynamically-typed languages support closures. In fact, it seems impossible to add true closures to a statically-typed language because of the runtime errors just described. Although these errors cannot be strictly considered "type errors", they make the language unsafe, which is against the philosophy of statically-typed languages.

We have designed a limited kind of closure that never cause runtime errors. They are described in the next section.

## 2 A Limited Kind of Closure

The problem with closures is that they can live past the method calls that created them. Therefore, to get rid of all runtime errors described in the previous section, we just need to assure that a closure object is only alive while its defining method is. How can we achieve that?

If a closure object, which we will call a block, is never assigned to a variable or passed as a parameter, then no runtime error will ever occur. A closer look at the subject reveals that the language does not need to be so restrictive to be safe. A block may be passed as a parameter to a method if the method does not assign the parameter to any variable or returns it. The called method may still execute the block. In fact, the method may pass the block as a parameter to another method that follows the same restrictions. In this way, the defining block method will exist whenever the block is executed. Any references in the block to variables of the defining method will be legal.

Figure 1 shows the method call stack of a program. Method `m` defines a block `[sum := sum + 1]` in which `sum` is a local variable. Method `m` calls `p` which calls `r`, always passing the block as a parameter.

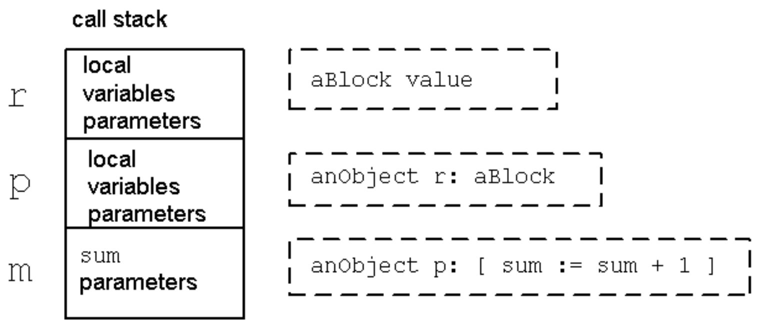
call stack
r
local variables parameters
p
local variables parameters
m
sum parameters
aBlock value
anObject r: aBlock
anObject p: [ sum := sum + 1 ]

Figure 1: Method call stack showing a block execution in method `r`

Method `r` executes the block defined in `m`. No runtime error occurs because local variable `sum` is in the stack.

We will now show how a limited kind of closure can be added to a statically-typed object-oriented language by using the idea of limited parameters described above. Our limited closures will be called blocks and will be presented in the Green language [2]. However, the ideas exposed in this paper are applicable to other languages as well. No specific Green feature is used.

To support blocks in Green, we will use the concept of `#types`. A `#type` is a special kind of type declared by putting a `#` before a class name: "`var p : #Person;`". Variable `p` is declared with type `#Person` — Green employs a Pascal-like syntax. `#types` are more restrictive than regular types. Only local method variables and parameters can have #types. Therefore a #type cannot be the type of an instance variable or a method return value.

#types follow more restrictive rules for type-checking than normal types. To show them, `typeOf(x)` will be used for the compile-time type of variable or expression `x`. An assignment "`a = expr`" is legal if `typeOf(a)` is supertype[2] of `typeOf(expr)` and one of the three situations occurs:

1. neither `typeOf(a)` nor `typeOf(expr)` is a #type;
2. both `typeOf(a)` and `typeOf(expr)` are #types;
3. `typeOf(a)` is a #type and `typeOf(expr)` is not a #type.

These rules assure that a #type variable will always refer to an object created in its declaring method[3] or in a method call currently in the call stack. Since, as will soon be seen, blocks have #types, a block will only be referenced in its enclosing method or in methods called by the enclosing method. Therefore whenever the block is called, its environment (enclosing method and class) will be alive. That is, the method call in which the block was created will be in the call stack. To see why, remember that only local method variables and parameters can have #types. Therefore, a local variable with a #type will only refer to objects created in the method call or passed as parameter. If the object the variable refer to was passed as a parameter, it was created by a method call currently in the call stack. In any case, the method call that created the block the variable refer to is in the call stack. That means it is safe to call the block through the variable since the block environment will be alive.

A block is a literal object defined using the following syntax:

```
    [ parameters | local variables | commands ]
```

The parameters and local variables are optional. A parameter is declared as in "`:variableName type`". `commands` are a sequence of commands optionally ended by a return statement of the kind "`-> expr`". A block can access all variables and call all methods visible in the place of its declaration. So this includes the local variables and parameters of the method, pseudo-variable `self`, and the enclosing-class instance

[2]Assume a supertype is a superclass. This is not always the case in Green but this detail is not important in this paper.
[3]The method that created the variable.

variables and methods. As an example, the block

```
[:a integer :b integer | :sum integer | sum = fat(a) + b;  -> sum ]
```

takes two integer parameters, has a local variable `sum`, and returns the sum of `fat(a)` with `b`. `fat` is a method of the enclosing class. Note that "`->`"returns from the block, not from the enclosing method. A return statement (from the method) inside a block is illegal.

A literal block `[ ... ]` is an object of an anonymous class — each literal block has its own class. This class is a subclass of one of the "block classes" `Block0`, `Block0R`, `Block1`, `Block1R`, `Block2`, `Block2R`, and so on. All of these classes but `Block0` are parameterized by the parameter types and return value type (if one exists) of the corresponding block. A block class `Blockn` represents a block with n parameters and no return value. A block class `BlocknR` represents a block with n parameters and a return value. For example, block

```
[:a integer | -> 2*a ]
```

is an object of an anonymous class which is subclass of class `Block1R(integer, integer)`. Class `BlocknR` is defined as

```
abstract class BlocknR(T1, T2, ...  Tn, R)
   public:
      abstract proc eval(p1 :  T1; p2:  T2; ...  pn :  Tn) :  R;
      abstract proc blockClone() : BlocknR(T1,T2, ...  Tn, R)
end
```

`T`$_1$`, T`$_2$`, ... T`$_n$`, R` are parameters to the class. The corresponding real arguments must be types. `eval` is a method that represents the body of the block. Method declaration in Green begins with keyword `proc`. Method `eval` has parameters `p`$_i$ of type `T`$_i$ and returns a value of type `R`. Classes `BlocknR` and `Blockn` are abstract — no object of them can be created. Method `eval` is abstract — its body does not need to be defined. Method `blockClone` will be explained later. Class `Blockn` is similar to `BlocknR` although without parameter `R`.

A block

```
[:p1 T1 :p2 T2 ...  :pn Tn | :u1 U1 ...  :um Um | commands; -> expr ]
```

is an object of the anonymous class

```
class Anonymous subclassOf BlocknR(T1, T2, ...  Tn, typeOf(expr))
   public:
      proc eval(p1 :  T1; p2 :  T2; ...  pn :  Tn) :  typeOf(expr)
            var u1 :  U1; ...  um :  Um;
         begin
         commands;
         return expr;
         end
      proc blockClone() : BlocknR(T1,T2, ...  Tn, R)
         begin
         ...
         end
   private:
      ... // explained in the text
end
```

The class of the block just described is `Anonymous` but is type is `#Anonymous`. Therefore, the block object follows the restrictions applied to #types. This means a block will only be referenced by variables declared in its enclosing method or in methods called by the current method (when the block is passed as a parameter). The declaring block method will be in the call stack whenever a variable or parameter references the block. Therefore no runtime error will ever occur because of blocks and a block object can be deallocated as soon as the method that creates it returns.

The block body is put in method `eval`. Therefore, to execute the block one should call method `eval`:

```
nine = [:x integer | -> x*x].eval(3);
  // declares variable incMonth of type #Block0 and assigns a block to it
var incMonth : #Block0 = [ ++month; month = (month-1)%12 + 1; ];
incMonth.eval();
[:a char | print(a)].eval('A');
```

The constructor of class `Anonymous` was not shown. It takes as parameters pointers to all local variables, parameters, and instance variables used inside the block. It also takes, if necessary, a reference to `self` and pointers to some enclosing-class methods used inside the block.

Although blocks are objects, `self` inside a block refers to the `self` pseudo-variable of the enclosing method. Another pseudo-variable could be defined to refer to the block object but this is not proposed in this paper. This feature would rarely be necessary.

A block

```
[:a integer |
    sum = sum + a;
    self.add(sum);
    super.put(a);
    self.show();
]
```

uses a variable `sum` and calls methods `add`, `put`, and `show` of `self`.[4] Assume that `sum` is a local variable of the enclosing method, `add` is a private method, and `put` and `show` are public methods. The anonymous class representing this block defines a constructor taking as parameters:

**(a)** a pointer to `sum`;

**(b)** pointers to methods `add` and `put`;

**(c)** `self`

The constructor of the anonymous class assigns the parameters to instance variables of the class. Therefore, the class for this block would have four variables — one for `sum`, two for `add` and `put`, and one for `self`.

The code the compiler generates for a block is the creation of a new object of the corresponding anonymous class. Something like

```
    new Anonymous(&sum, &A::add, &A::put, self);
```

using an ad-hoc C++-like syntax. "`&`" is the "address of" operator. "`&A::add`" is the address of method `add` of class `A`. Suppose class `A` is the class in which the block is defined.

Note that a copy of `self` is passed to the constructor, not the address of `self`. Through the pointer to `sum`, the block can change the value of this variable. Through the pointers to `add` and `put`, method `eval` of the anonymous class can call the corresponding private/superclass methods of the enclosing class. Public method `show` is called using the `self` copy passed as a parameter.

There may be blocks inside blocks and recursive blocks. These two features appear in the example that follows.

[4]Calls to `super` are in fact calls to `self` in which the method called is fixed at compile-time.

```
var fat : #Block1R;
fat = [:n integer | :result integer :comp Block0R(boolean) |
   comp = [-> n > 1];
   if comp.eval()   // calls block comp
   then
      result = n*fat.eval(n-1);    // recursive block call
   else
      result = 1;
   endif
   -> result
   ]
```

A block that does not access local variables, parameters, and instance variables could be assigned to a non-#type variable. No runtime error would ever occur. However, if the language permits this assignment, it would be discriminating between blocks that access local state and blocks that do not. They should have different types with different type-checking rules. The confusion would probably be greater than the gain.

Although a block cannot be assigned to a #type variable, a *copy* of the block can. Each of the `Blockn` classes declares a method

`abstract proc blockClone() : Blockn(`$T_1$`, `$T_2$`, ...  `$T_n$`)`

which returns a copy of the block. Observe that the return type of this method is not a `#` type. Of course, a class `BlocknR` declares a `blockClone` method returning a `BlocknR(T` $_1$`, `$T_2$`, ...  `$T_n$`, R)` object. When the compiler creates an `Anonymous2` class for the block

`[:a integer :b integer | -> a > b]`

it defines a method

```
proc blockClone() : Block2R(integer, integer, integer)
   begin
   return self;
   end
```

The block neither accesses local/instance variables nor calls private/superclass methods. Therefore, class `Anonymous2` has no instance variable. All `Anonymous2` objects are equal to each other and `self` itself can be returned as a clone. If the block accessed local variables, parameters, or instance variables, `blockClone` would return `nil`, the null pointer. Now a block can be assigned to non-#type variables and returned by methods:

```
var b : Block1R(real, real) = [:x real | -> x*x].blockClone();
return [:a char :b char | -> a < b].blockClone();  // return from a method
```

Suppose a block `b` does not access any variables but those defined in itself. Block `b` calls private/superclass/public methods using `self`. Therefore, the class for `b` will have instance variables which are pointers to the private and superclass methods that the block calls. Method `blockClone` returns a shallow copy of `self`. This is possible because all instance variables point to methods which, of course, are shared among all objects of the same class and will never cease to exist at runtime.

## 3 Conclusion

This paper presented a limited kind of closure that can be added to statically-typed object-oriented languages. Our proposal is not overly restrictive when compared with full closure implementations like those found in Smalltalk, for example. It only prohibits dangerous things as to assign a block to a variable that can live more than the method than created the block. The limited closures proposed in this article bring to statically-typed languages features only found in dynamic languages.

Although Green was used to show the idea, the limited closures were neither officially added to this language nor were they implemented yet. However, the implementation does not demand any sophisticated algorithm or special techniques. Closures are not supported by the Green compiler because it generates Java code and Java does not support pointers to variables, a feature necessary to implement closures (see the constructor of the `Anonymous` class).

## References

[1] Goldberg, Adele and Robson, David. Smalltalk-80: The Language and its Implementation. Addison-Wesley, 1983.

[2] Guimarães, José de Oliveira. The Green Language. Available at `http://www.dc.ufscar.br/~jose/green/green.htm`.

[3] Hopkins, Trevor and Horan, Bernard. Smalltalk: An Introduction to Application Development using VisualWorks. Pearson Education, 1995.