Coroutine Semantics in the TOOL Programming Environment

Sergio Eduardo Rodrigues de Carvalho
Departamento de Informática, Pontificia Universidade Católica RJ
Rua Marques de São Vicente 225, Gávea, Rio de Janeiro, 22453
Telephone: (55)(21)274-4449 Fax: (55)(21) 511-5645
e-mail: sergio@inf.puc-rio.br

ABSTRACT

In this paper we discuss two situations, in object oriented programming, where the semantics of coroutines is useful: in the implementations of loop iterators and of dynamic data structure navigators. In both of these operations the receiving object is a loop control object. Iterators are applied to produce, outside of the loop's environment, the next value for the loop control object; navigators are applied to indicate, outside of the loop environment, the next element in a dynamic data structure.

KEYWORDS AND PHRASES

object orientation, coroutine, iterator, generic class, loop statement, data structure.

INTRODUCTION

Most object oriented languages today offer a single semantics for the application of operations to objects: that of procedures. Even though procedures, usually called *methods*, may be late bound to receiving objects, it is typically the case that the environment where the application takes place has its execution suspended while the applied operation is executing, and that control data such as return addresses, dynamic links and the like must be saved on the execution stack in order to allow for control flow.

Other semantics may also be useful. For example, in object oriented languages for the construction of applications dealing with graphical user interfaces, where users asynchronously interfere with the execution, the sending of a message to an object does not need to block the execution of the environment where the operation was applied. In this case the message is usually placed on a queue, to be eventually processed by a corresponding message handler. The Windows® operating system from Microsoft [10] is a software platform where this semantics is available, though at a very low level. In the TOOL programming system [2, 12], to be described in some detail below, the asynchronous behavior of objects can also be declared in classes [3].

As another example, consider exception handling. Here it may be convenient to attach exceptional behavior to objects, by placing, within classes, exception handlers to be executed when local handlers for raised exceptions are not available. Such class exception handlers would then provide an intermediate handling level, between that of local handlers and that associated with the call to the raising operation. An exception handling mechanism along these lines has been proposed in [4].

In this paper we discuss two situations, in object oriented programming, where the semantics of coroutines is useful: in the implementations of loop iterators and of dynamic data structure navigators. In both cases the receiving object is a loop control object. Iterators are applied to produce, outside of the loop environment, the next value for this loop control object; navigators are applied to indicate, outside of the loop environment, the next element in a dynamic data structure. This is clearly advantageous to the understanding of programs, since the size of the loop's body is reduced accordingly. Moreover, it would be nice to be able

to use loop control objects already possessing such iterative behavior, declared in their modeling classes.

We further characterize iterators and navigators:

- an *iterator* produces the next value of a loop control object by *constructing* this next value from the existing value and its own execution context (referencing environment plus code);
- a data structure *navigator* obtains the next value of a loop control object by *displacing a cursor* through an existing data structure, using the current cursor position and its own execution context (referencing environment plus code).

In the latter case the data structure accessing discipline is a primary factor: for example, the next value obtained by a single link list navigator must obey the linking discipline adopted by the single link list.

As an example of an iterator, we offer the arithmetic progression iterator present in most conventional programming languages: the initial-value-to-final-value iterator, which produces, without programmer intervention, the next value in the range specified. As an example of a data structure navigator, consider a first-to-last single link list navigator, which at each loop iteration produces the next list element for processing.

In this paper we discuss the implementation of iterators and navigators in object oriented languages. As context we use the programming language TOOL. This can be considered a second generation object oriented programming language, since different semantics for operations may be defined in classes, the syntactic and compile time entities modeling objects. Classes in TOOL already allow the definitions of the procedural and asynchronous behaviors for the objects they model. In the next section we present some aspects of TOOL relevant to the topic at hand.

THE LANGUAGE TOOL

TOOL is an object-oriented, event-driven programming language, designed to simplify the task of building application programs using the extensive facilities provided by

Windows® and similar environments. It incorporates some of today's most important features for the development of modern software, such as object orientation, message passing and graphical user interfaces.

The main syntactic entity in TOOL is the *class*. Classes are the meeting grounds for all properties associated with the objects they model. A TOOL class is a collection of (possibly absent) sections. Relevant to this discussion are the inheritance, object representation and behavior definition sections.

In the *inheritance* section we specify the reutilization of some existing class in the declaration of a new one. This is a simple inheritance mechanism, also establishing a superclass-subclass relationship that reflects on modeled objects. Thus if in the design of some application a new class Student must be defined, and if a class Person already exists, we can specify inheritance (and software reuse) by declaring

```
CLASS Student FROM Person;
    ...
END CLASS
```

This also states that if 's' is a Student then 's' may appear wherever a Person is expected.

In the *object representation* section we declare the data structure, or *state*, of the objects modeled by the class. For example, the state of a point in a plane could be declared as a pair of real values:

```
CLASS Point;
    -- no inheritance
    REPRESENTATION
        REAL x, y;
    END REPRESENTATION
    ...
END CLASS
```

The *behavior* of objects is also specified in classes. This behavior can be *procedural* or *asynchronous*. We specify the procedural behavior for class objects by using *methods*. These are procedure-like units, consisting of a signature, local declarations and statements. A method signature has a name, an optional list of parameters, and an optional result value. Method parameters can be full fledged objects or merely argument identifiers. Object parameters correspond to arguments with the passing of either a value or of a reference. In both cases subclassing is enforced. For example, a value object parameter modeled by Person may receive the value of an argument modeled by Student; a value object parameter modeled by Student may provide a value to a Person argument. Argument identifiers are not local entities, but just aliases for the corresponding arguments.

To exemplify, consider a method Distance, declared in class Point, which returns a real value indicating the distance between two points:

```
METHOD  Distance (IN Point s) RETURNS REAL r;
        -- local declarations, if any;
BEGIN
        -- statements, producing a value for r;
END METHOD
```

Methods are applied to receiving objects. If in some context we declare

```
Point  p, q;
```

then eventually

```
... p <- Distance (q)  ...
```

would produce the distance between the receiving object 'p' and the input parameter 'q'. The receiving object of a method application is indicated in the method's body by the generic designator SELF.

Methods may be redeclared in subclasses, provided the method's signature is preserved. Methods may be also late-bound to receiving objects, provided these objects are declared with the polymorphic discipline. Methods may be declared with the attribute VIRTUAL, which means that their implementation is deferred to subclasses.

To specify the asynchronous behavior of objects we use *message declarations* and *message handlers*, also declared within classes. Message and handler declarations split the responsibility for the full treatment of messages in TOOL: the former tells the system about the messages that objects of this class can send; the handlers of this class tell the system in what way objects of this class treat messages they in turn receive. With respect to redeclaration and deferred implementation, message handlers behave exactly like methods. A complete description of the message passing facilities in TOOL can be found in [3].

We propose to consider iterators and navigators, discussed below, also as operations declared within classes, and therefore applicable to receiving objects. In the next section we discuss the context in which iterators and navigators are used.

LOOP STATEMENTS

A single *loop* statement is used to specify repetition in TOOL. The execution of a loop statement can be controlled by conditional *exit* and *repeat* terminators, placed anywhere in the loop's body, and in any number, as indicated below:

```
LOOP
        -- body of statements, including
        -- EXIT [WHEN expression];   and/or
        -- REPEAT [WHEN expression];
END LOOP
```

The EXIT terminator terminates the loop's execution, transferring control to the statement immediately following the loop. The REPEAT terminator terminates a loop iteration, transferring control to the beginning of the loop's body. Both terminators are only valid in loop bodies.

The association of loops and iterators is illustrated in the skeleton below. The same syntax would be used to control loops with navigators.

```
LOOP FOR receiving_object <- iterator_name (argument_list);
        -- statements, possibly including loop terminators;
END LOOP
```

In the remainder of this section, in order to continue to illustrate the semantics of TOOL loops, we shall use iterators as models. The use of navigators is similar to that of iterators, and will be discussed in detail at the end of the paper.

The reserved word FOR indicates that the loop is controlled through the application of an iterator. The iterator must be applicable to the receiving object, being declared in its class or superclass thereof. For security reasons, the control object is read-only in the loop's body. This means that:

- it can only be passed as an input value parameter to other operations;
- it cannot be assigned to;
- it cannot be subject to method applications.

The value of the control object after loop termination is the value obtained after the last activation of the corresponding iterator, even if loop termination is caused by an EXIT statement. In the case of a REPEAT statement, the iterator is applied again and the control object will have the value obtained in this last iterator activation.

Nested loops with identical or different iterators are allowed. In either case the control objects used in each FOR clause must be distinct, to preserve the read-only property of loop control objects. In the iterator's body, loops with or without other iterators are also allowed, as is recursion.

ITERATORS

Iterators are a control abstraction available in some programming languages, notably CLU [7], and more recently in Sather [9]. They permit users to iterate over arbitrary types of data in a convenient way, preserving abstraction by specification [8]. Iterators in TOOL have first been proposed in [11].

Iterators in TOOL are active units that specify how the sequence of values of loop control objects is to be produced. An iterator yields, one at a time, a value in this sequence. The policy that selects the next element of the sequence is hidden from the user of the iterator and implemented by the iterator itself. The iterator is responsible for producing the items that the loop is supposed to consume. Iterators in TOOL follow very closely the semantics of CLU, as will be seen below.

TOOL iterators are operations declared in a class, applicable only to objects modeled by this class or subclass thereof, just as a method or a message handler. Such declarations specify the iterative behavior of objects, as a method defines the procedural behavior and a message handler the asynchronous behavior. Iterators in TOOL are thus considered "first class operations". Iterators may be redeclared in subclasses, but their signatures must be preserved. Their definition may also be deferred to subclasses. The syntax of iterators in TOOL can be exemplified with the skeleton below.

```
ITERATOR iterator_name (parameter_list);
        -- local declarations, if needed;
BEGIN
        --statements, including YIELD and RETURN statements;
END ITERATOR
```

Iterator parameters must be exclusively of input mode. This restriction is due to the very nature of iterators, which must solely produce a sequence of values to control a repetition process; in this sense no side effects should be generated through the activation of iterators. Single objects and/or structures of objects may be locally declared within iterators, just as in any method or handler. Their life cycle is bound to that of the enclosing iterator. Iterators do not return values, as may be the case with methods. They are constructed specifically to alter the state of a loop control object, freely accessed, through the generic designator SELF, in the iterator's body, where its new value is produced.

In order to discuss semantics and implementation issues, it is useful to compare iterators with methods. The execution of a method terminates either normally, after the

execution of the last statement in its body, or abruptly, when a possibly conditional RETURN statement is executed. In either case, the last value associated with the return object, if there is one, is returned to the calling environment. At method termination, its local environment is lost. Several RETURN statements may be placed anywhere in a method's body.

At each YIELD statement the iterator is suspended, providing a new value for the control object. The iterator's context is not lost, and neither is the iterator's access to the loop control object. On the next iteration, the iterator will continue from the point where it stopped, executing the first instruction just after the most recently executed YIELD statement. Several YIELD statements may be placed in an iterator's body.

When the iterator terminates, a signaling flag in the iterator's environment is set, and the corresponding loop terminates. This flag is tested each time the iterator is resumed. An iterator terminates either by itself, after the production of the last loop control value in its body, or if an EXIT statement is executed in the corresponding loop's body. A RETURN statement within the iterator may also be executed to terminate its environment. This statement, however, should only be used as a shortcut to termination; the use of an iterator supposes the need for all values it is specified to produce.

The implementation of iterators in TOOL follow the guidelines described for example in [6]. The iterator's environment is allocated on the execution stack, containing, besides space for parameters and local objects, and the usual control data, the following information:

- an entry point;
- a termination flag;
- access to the loop control object.

As mentioned before, at each yield statement the iterator is temporarily suspended. After the loop's body is executed with the value provided, the control is passed back to the iterator, which resumes from the instruction following the last executed yield. This instruction is given by the entry point. When the iterator terminates, the termination flag is set, and the next attempt to loop will fail. The iterator's environment is then abandoned and the execution continues from the first instruction following the END LOOP delimiter.

If a loop controlled by an iterator is terminated by an EXIT statement, then the iterator's environment is destroyed at this time. A REPEAT statement in an iterator controlled loop causes the iterator's environment to be resumed.

To exemplify the use of iterators, consider again class Point, augmented with the OriginToGiven iterator, declared below. The purpose of this iterator is to allow a point controlling a loop to assume point values between the origin of a two dimensional coordinate system and a given point 'p'. Also as a parameter, the iterator receives a real value indicating the 'x' component displacement to be used in the production of the next point for the loop.

```
ITERATOR OriginToGiven (IN Point p, IN REAL delta_x);
        REAL theta;
BEGIN
        theta := p.y / p.x;   -- computing segment inclination.
        SELF.x := 0.0;    SELF.y := 0.0;  -- producing initial value for control point.
        LOOP  -- for producing other values.
                YIELD;  -- control to the application loop.
                SELF.x := SELF.x + delta_x;  -- preparing next value.
                EXIT WHEN SELF.x > p.x;  -- last value has been produced.
                SELF.y := theta * SELF.x;
        END LOOP
END ITERATOR
```

This iterator could be used in the following loop, supposing the declaration of points 'p' and 'q':

```
-- assign a value to p;
LOOP FOR q <- OriginToGiven (p, 0.5);
        -- use value of q;
END LOOP
```

NAVIGATORS

Iterators, described above, are associated to loop statements to obtain, from the current value of the loop control object, the value to be used in the next loop iteration. A special kind of iterator, which we are calling a navigator, can be associated to loop statements to produce, from the element being currently accessed in a dynamic data structure, the next element for the next loop iteration.

The main difference between iterators and navigators is that iterators change the state of the loop control object, whereas navigators simply traverse data structures, their receiving objects, without altering their states.

Syntactically and semantically, however, both operations are similar. A navigator in TOOL is declared as follows:

```
NAVIGATOR navigator_name (parameter_list);
        -- local declarations, if needed;
BEGIN
        --statements, including YIELD and RETURN statements;
END NAVIGATOR
```

The navigator context is preserved between element indication cycles, as is the access to the receiving object (through SELF). Possibly conditional YIELD statements can also be used at will in a navigator's body; each indicates a possible entry point. Navigator parameters must also be exclusively local entities initialized with argument values; we want no side effects from navigator executions. Also, navigators do not return values, may be redeclared in subclasses with the same signature, and may be deferred to subclasses for full declaration.

There are, however, two notable differences between navigators and iterators. First, navigators can only be declared in classes modeling dynamic data structures, as for example lists, stacks, trees, and so on. These structures are defined as generic classes, modeling objects instantiated as for example in

```
SingleList sl OF Person;
```

which establishes 'sl' as a possible receiver for SingleList methods and navigators (below we present the declaration of a FirstToLast navigator for single lists).

Secondly, there must be an extra link, besides SELF, between the loop environment and the navigator context. This link is needed since the loop control object (SELF in the navigator's context) is the entire data structure, and the navigator must yield, for loop processing, one at a time, elements in this structure. To make an element available in the loop, the CURRENT generic designator is used, both in the loop and navigator environments, to indicate the data structure element being currently accessed (in the navigator) and processed (in the loop).

The CURRENT designator actually promotes a change of worlds, in a way similar to the **cvt** operator in CLU. In the client application environment, where the loop is constructed, CURRENT indicates an instance of the component class of the data structure controlling the loop: in the 'sl' example above, an instance of Person. This is fine from the application programmer point of view, since in the client world only objects of class Person and the list 'sl' need to be known. In the class constructor or server world, on the other hand, in order to displace CURRENT from element to element, an access to the next element is needed, as illustrated in the FirstToLast navigator presented below.

We can imagine, from the implementation point of view, and still considering the 'sl' example, that in the client world CURRENT accesses a space in the heap, enough to hold a value of class Person; in the server world, CURRENT accesses an element of the loop control object, actually an instance of another class, containing in its representation a pair (Person, access to Person). It is the execution of an YIELD statement that copies the value of Person, accessed from the server's CURRENT, to the heap area pointed at by the client's CURRENT, thus bridging the gap between the two environments.

The copy semantics associated with the YIELD statement indicates that changing the contents of the area accessed from the client's CURRENT, which is a legal operation, does not change the loop control object. To begin with, this object (for example a list) can only be operated upon (for example changed) by operations available in its own class. Again using 'sl'

as an example, CURRENT in the loop is in fact an instance of Person (or subclass thereof); the loop control object is a single list of persons, an instance of another class. Furthermore, as stated above in the case of iterators, the loop control object must not be changed within the loop, this in fact being the primary reason for the use of a navigator.

We now present a FirstToLast navigator that could be declared in the class SingleList. This definition is incomplete; to be presented in full, a description of the generic TOOL classes modeling data structures would have to be presented. We include a few comments after the code, hoping they will suffice to promote understanding.

```
NAVIGATOR FirstToLast;
BEGIN
        RETURN WHEN SELF.count = 0;
        CURRENT <- NOW_IS (SELF.first);
        LOOP
                YIELD;
                RETURN WHEN CURRENT.next <- IS_NULL;
                CURRENT <- NOW_IS (CURRENT.next);
        END LOOP
END NAVIGATOR
```

Every SingleList in TOOL has a header containing the field 'count', an integer indicating the number of elements in the list. The first RETURN statement kills the navigator (and consequently the loop) if the list is empty.

The NOW_IS method, implicit in every user-defined TOOL class, promotes polymorphism [2] and the copy of a reference. The result of the statement

```
CURRENT <- NOW_IS (SELF.first);
```

is that the navigator's CURRENT now accesses the first list element. This element provides a value of Person that the yield statement transfers to the loop, as explained above.

The RETURN statement within the navigator loop terminates the navigator when the last list element has already been processed. If this is not the case, the following operation displaces CURRENT to the next list element. This is so because every SingleList element in TOOL has, unknown to application programmers, besides a field modeled by the component class of the list (Person in our running example), an access to the next list element.

The FirstToLast navigator above could be used as in

```
-- create the list 'sl';
LOOP FOR sl <- FirstToLast;
        -- access to 'sl' persons through CURRENT
END LOOP
```

CONCLUSIONS AND RELATED WORK

There are several meanings that can be attached to the important word "class" in object oriented languages. One of the simplest, and possibly the most easily understood by conventional language programmers, is the one stating that:

- a class is a model for objects, defining their structure and behavior;
- a class is a compile-time entity, allowing the compiler strict checking on the use of application objects.

Once this meaning is chosen, and considering that the behavior of objects is defined by units that can be invoked at execution time, one begins to wonder why, in most object oriented languages today, the only semantics associated to object behavior is that of procedures.

In this paper we presented two situations where the semantics of coroutines is useful, both in the modeling of loop control abstractions, as is done in CLU. We distinguish, however, between loops controlled by objects that have their states altered during execution and loops controlling the traversal of dynamic data structures.

We conclude that the coroutine semantics, by adding functionality to classes, increases their chance of reuse. We also feel that placing the burden of controlling loop iterations on class constructors should improve the productivity of application programmers. Clearly, however, not all traversals can be pre-constructed for all data structures; in such situations the application programmer must step in to provide that extra functionality, as is done with methods.

Since its origin [5], the semantics of coroutines in programming languages has been used mainly as a mechanism to implement some form of concurrency, as for example in SIMULA [1], BLISS [14] and INTERLISP [13]. Its use as a control abstraction, specially in object oriented languages, is less popular. The language Sather [9] includes iterators, but not as first class operations. Several differences exist between TOOL and Sather operations:

- Sather **iter**s can be invoked from within the loop body, and in any number, whereas in TOOL each loop may be associated to a single iterator, associated to the loop's header;
- Sather programmers can choose the environment to be retained between iterator invocations, whereas in TOOL the entire iterator environment is always retained;
- the predefined local variable **res** is used in Sather to hold the value produced by the iterator, whereas in TOOL the iterator changes the value of the receiving object;
- in Sather a loop can only be terminated by the exhaustion of an iterator, whereas in TOOL EXIT statements in the controlled loop and RETURN statements within the iterator can terminate the loop.

ACKNOWLEDGEMENTS

The TOOL Programming System is the result of a joint research and development project conducted by SPA (Sistemas, Planejamento e Análise) and a group from the Departamento de Informática of the Pontificia Universidade Católica, both in Rio de Janeiro. This report must be understood as another result from this same team effort, although the author takes sole responsibility for its contents. Thanks are due to Otavio P. Coelho, Nelson Gorini, and Werther J. Vervloet, early day team members, and to Mario S. Cardoso, Eduardo Galucio and Toacy C. Oliveira. Also acknowledged are the contributions of Stella C. Porto, introducing iterators in TOOL.

BIBLIOGRAPHY

[1] Birtwistle G, Dahl O-J, Myhrhaug B, Nygaard K (1973) SIMULA Begin. Petrocelli-Charter, NY.

[2] Carvalho SER (1993) The Object and Event Oriented Language TOOL. Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 6/93.

[3] Carvalho SER (1993) Asynchronous Behavior in the TOOL Programming System. Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 18/93.

[4] Carvalho SER (1992) Exception Handling in Object Oriented Languages: a Proposal. Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 23/92.

[5] Conway M E (1963) Design of a Separable Transition Diagram Compiler. Comm. ACM 6(7): 396-408.

[6] Ghezzi C, Jazayeri M (1982) Programming Language Concepts; Wiley.

[7] Liskov B, Atkinson R, Bloom T, Moss E, Schaffert J, Scheifler R, Snyder A (1981) CLU Reference Manual. Springer Verlag, Berlin.

[8] Liskov B, Guttag J (1986) Abstraction and Specification in Program Development. The MIT Electrical Engineering and Computer Science Series; The MIT Press and McGraw-Hill Book Company.

[9] Murer S, Omohundro S, Szyperski C (1993) Sather Iters: Object-Oriented Iteration Abstraction. The International Computer Science Institute, Berkeley, CA.

[10] Petzold C (1990) Programming Windows. Microsoft Press, Redmond, WA.

[11] Porto S C, Carvalho S E R (1992) Introducing Iterators in TOOL. Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 41/92.

[12] SPA Sistemas Planejamento e Análise (1992) TOOL: the Language. Rio de Janeiro.

[13] Teitelman W (1975) INTERLISP Reference Manual. XEROX Palo Alto Research Center, CA.

[14] Wulf W, Russell D B, Habermann A N (1971) BLISS: a Language for Systems Programming. Comm. ACM 14(12): 780-790.