ACTOR, A Threaded Object-Oriented Language

Charles B. Duff
The Whitewater Group
906 University Place
Technology Innovation Center
Evanston, IL. 60201
(312) 491-2370

Object-oriented programming, in which objects encapsulate private data and respond to generic messages, has become increasingly popular in the last several years. OOP is particularly effective when implemented as a consistent programming environment in the manner of Smalltalk.

This paper describes a new language, ACTOR, that attempts to address the central problem of object-oriented language design: achieving a practical, efficient language without compromising the integrity of the runtime environment. ACTOR uses a token-threaded interpreter as an efficiency measure, permitting a more flexible optimization strategy. While other researchers have suggested threading as an optimization technique [Deutsch83], to the author's knowledge, no other pure threaded OOLs have been implemented.

Threading

Actor's token-threading model integrates the object table of Smalltalk with the token table of threaded languages. The result is that all objects are separately relocatable and have executable behavior, unlike Smalltalk, in which objects are "dead" entities that rely on the bytecode interpreter to give them life. Each ACTOR object has a family code embedded in its object table entry that describes its executable behavior. Most objects simply place their object pointer on the stack.

Syntax

Unlike the author's previous threaded OOL, Neon, ACTOR uses an Algol-like infix syntax. There is a tremendous amount of resistance in the world at large to RPN syntax, justified or not. More importantly, it is impossible for the compiler to protect the runtime environment using RPN, because the state of the stack cannot be accurately predicted or controlled. This would cause an unacceptable breach in the safety of the system, which is directly at odds with object-oriented philosophy. Neon is also a hybrid OOL, in that not all entities are treated as objects (for instance, numbers).

In the development of ACTOR, the author did employ an RPN object-oriented language as a bootstrapping mechanism. Only the stack operations DUP and DROP were provided in addition to a local variable facility. This language was successful in its own right, and demonstrated the feasibility of a Forth-like pure OOL, although with the caveat mentioned above.

ACTOR's infix parser is a state machine driven by tables generated with the Unix utility yacc. The parser is described by a class, permitting easy parsing of custom grammars by the user. Yacc output must be transliterated into ACTOR array literals using a text-processor. These arrays are then used to initialize a descendant of class LR1, generating a custom parser.

Efficiency Considerations

Because ACTOR was designed with real-time AI development in mind, an efficient and non-obtrusive garbage collector was considered mandatory. ACTOR uses a modification of the Baker/Lieberman/Hewitt scavenging collectors, incorporating sensitivity to object lifetimes. ACTOR is thus able to perform incremental garbage collection with no lengthy pauses to disrupt time-sensitive code.

The central optimization strategy that ACTOR provides involves an optional typing mechanism. The programmer can assign types (class names) to variables and functions, allowing the compiler to perform early function binding in some cases. For instance, in the following declaration,

```
Def copyFrom(self, startIdx:Int, endIdx:Int)
```

the arguments startIdx and endIdx are stated to be of class Int. The compiler must then ensure that any early-bound calls to copyFrom pass arguments of class Int. Given this assurance, any messages to these two arguments within copyFrom can potentially be early-bound. Even with method caching, early binding produces a performance increase of five to ten times in a particular call.

Because late binding is so desirable from a maintenance and debugging standpoint, the programmer can use exclusive late binding until the application is debugged. Selected variables can then be typed based on profiling information that ACTOR provides. The most heavily used functions can be early bound, until performance is acceptable. As a final measure, functions can be implemented as primitives in assembler or one of the Microsoft high-level languages.

Development Environment

Like Smalltalk, ACTOR provides a very rich set of development tools for the programmer. A browser allows one to peruse the class tree, and shows the functions defined for each class with their arguments. If a function is selected, the text for the function is pretty-printed into an edit window so it can be read or modified. A menu provides formatted templates for all of the control structures. Since functions are compiled one at a time, the parser can provide immediate and accurate feedback on syntax errors.

If an error occurs at runtime, a dialog is shown in which the function activations that led up to the error are presented. A given activation can be converted into an object and inspected via the Inspector, described below. This allows examination of local variable values in any function activation prior to the error.

A trace facility allows single-stepping through code. Each function can have an attached error handler that is consulted if an error occurs. The handler can selectively clear the error flag for errors that it wishes to handle. Possible actions including interaction with the user via a dialog, resuming, aborting to the caller, aborting to ACTOR's interpreter, or fatal abort.

The inspector is a tool that presents a graphical, window-based interface to the programmer. The inspector lists the private variables belonging to its target object in a list box. If a variable is selected with the mouse, its contents are formatted into an edit window. Collections list their keys or indices in another list box, subject to the same manipulation. Objects that have special printing behavior, such as graphics objects, can be asked to show their contents in a special display window.

Because ACTOR's object-oriented environment is so highly organized, the programmer can learn a lot by simply interacting with objects in the Workspace window. For instance, the following message shows the programmer the private variables in class TextWind:

```
TextWind.variables
```

The variable names are stored in a dictionary, which knows how to print itself when sent a sysPrint message by the interpreter. The message TextWind.methods would show the methods available to a TextWind object.

Artificial Intelligence

OOLs are generally a very good starting point for doing AI work. Interestingly, most of the major Lisp systems that are being used for AI have object-oriented extensions (e.g., Loops, Flavors, Scheme, KEE). OOP is an ideal approach for problems involving complex data structures that must grow organically.

Other facilities are very helpful in doing any kind of AI work [Carr86]. Figure 1 shows an implementation of Carr's symbol manipulation example in ACTOR. The principal difference between ACTOR and Lisp in this example is ACTOR's use of arrays for storing the expressions instead of lists. ACTOR also has a List class, but arrays provided a more transparent representation in this case. In general, an OOL provides a rich variety of representation options, within a consistent framework.

We are currently developing a frame-based knowledge representation system within ACTOR that exploits the full power of objects. This will allow integration of a knowledge base with a more conventional procedural application, without having to shift paradigms. It is our expectation that ACTOR's efficiency and incremental garbage collection will make it an excellent choice for real-time AI applications.

C. Mellish, one of the authors of the definitive book on Prolog, has stated that logic programming is a poor choice for many real-world problems, such as writing text editors and other naturally procedural applications [Mellish84]. He suggests that instead of trying to build a complete language around logic programming, it should be integrated as a facility into a language with procedural features. The host language can control backtracking, modify rules, and provide procedural implementations of the side-effects that are so essential for real applications.

We are examining the desirability of a logic-programming facility within ACTOR. Providing the facility is not difficult, but correct integration is a significant challenge. [Tokoro84] describes one approach to logic programming within Smalltalk.

OOP and Forth

Working in an object-oriented language can at first be a startling experience, because everything is so highly organized. The process of developing an application is channeled and made more comprehensible by class inheritance. The burden on the programmer is greatly reduced, because information is localized in the classes and objects themselves. As in Forth, there is a continuum between the "system" and the application; the working environment of an OOL, however, is much more sophisticated because of the consistent application of the object philosophy.

Forth is seriously limited by its lack of a sophisticated, structured and consistent data definition facility. CREATE/DOES is not adequate, yet it is the only standardized means of defining data [Duff84]. It does not support nested or composite structures, and can only associate a single behavior with a data structure. A common response from Forth programmers when presented with criticisms such as these is, "I can do that in Forth, and here's some code that proves it". [Carr86] raises some issues that are sure to provoke such a response from Forth hackers.

Yet, this response misses the point of these and other constructive criticisms. The issue is not one of power - Forth is certainly a very powerful and flexible language. Many researchers have concluded that the "extensible language" movement has been a failure, with the notable exception of Lisp. Lisp has been able to grow organically and channel its growth into new standards. Common Lisp, while far from ideal, was able to integrate many of the extensions that had proliferated into a standard, and perhaps rescue Lisp from a chaotic decline.

The Forth community must recognize that raw power is not always terribly desirable, particularly when many programmers need to exchange code. The Tower of Babel is a very dangerous scenario, but inevitable when every shop has its own way of defining extensions. Unless Forth is able to identify some of the most essential extensions and incorporate them into a standard, it is the author's opinion that any hope of general acceptance in commercial environments will be lost. Object-oriented techniques could provide a model for a more advanced data structuring facility within Forth.

Bibliography

Borning, A. and Ingalls, D., "A Type Declaration and Inference Mechanism for Smalltalk". In Proc. 9th Annual Principles of Programming Languages Symposium, ACM, 1982.

[Carr86] Carr, H., "Forth for AI?", Proceedings 1986 Rochester Forth Conference.

[Deutsch83] Deutsch, P. and Schiffman, A., "Efficient Implementation of the Smalltalk-80 System", in In Proc. 10th Annual Principles of Programming Languages Symposium, ACM, 1983.

[Duff84] Duff, C. and Iverson, N., "Forth Meets Smalltalk". JFAR Vol. 2 #2.

[Mellish84] Mellish, C. and Hardy, S., "Integrating Prolog in the POPLOG Environment". In "Implementations of Prolog", J. Campbell, Ed. Wiley, 1984.

[Tokoru84] Tokoru, M. and Ishikawa, Y., "An Object-Oriented Approach to Knowledge Systems". ICOT Proceedings, 1984.

```
/*  symbolic differentiation example  ref Carr 1986 Rochester paper */!!
/*
        C.B.Duff  7.12.86
        (c) Copyright, 1986
        The Whitewater Group
        Technology Innovation Center
        906 University Place
        Evanston, Il. 60201  (312) 491-2370        */

/*      Modification history:
        7.12.86 cbd
*/

/*  Conditional logic for the various object types is incorporated
        into the distinction between statements, rather than requiring
        explicit conditionals as it does in Lisp  */

/*  derivative of an Int is Ø  */
Compiler.curClass := Int;!!

Def deriv(self, var)
{ ^Ø  }!!

Compiler.curClass := Object;!!

Def deriv(self, var)
{       if  self == var
        then  ^1
        else  ^Ø
        endif;
}!!

/* composite expressions are stored in array objects.  Elements
        Ø and 2 are operands, and element 1 is the operator  */
Compiler.curClass := Array;!!

Def deriv(self, var | exp)
{       exp := self;

        if  exp[1] == #+ or exp[1] == #-
        then  ^triple( deriv( exp[Ø],var), exp[1], deriv( exp[2],var));
        endif;

        if  exp[1] == #*
        then    ^triple( triple( exp[Ø], #*, deriv(exp[2],var)), #+,
                triple( exp[2], #*, deriv(exp[Ø],var)));
        endif;
}!!

/*  Sample output:

        deriv( #(1ØØ * 3), #x)
        1ØØ * Ø + 3 * Ø

        deriv( #(1Ø * x), #x)
        1Ø * 1 + x * Ø
*/
```