Reversible Object-Oriented Interpreters

Henry Lieberman

Artificial Intelligence Laboratory, Massachusetts Institute of Technology
Cambridge, Mass. 02139 USA
Electronic mail (Arpanet): `Henry@AI.AI.MIT.Edu`

Abstract

The "programs are data" philosophy of Lisp uses Lisp's *S-expressions* to represent programs, and permits a program written in Lisp itself to implement the interpreter for the language. Object-oriented languages can take this one step further: we can use *objects* to represent programs, and an *object-oriented interpreter* takes the form of responses to a protocol of messages for evaluating programs. Because objects are a richer data structure than simple S-expressions, the representation of programs can have more built-in intelligence than was feasible using simple list manipulation alone.

This paper surveys techniques and applications for object-oriented interpreters. We focus particularly on object-oriented interpreters that are *reversible*, those that, unlike conventional interpreters, remember their history of evaluation. We illustrate the techniques involved with two applications of reversible object-oriented interpreters: a reversible stepper for Lisp, and a programming environment which constructs Lisp programs from examples.

---

1. Representing programs as active objects is a key to increased power in programming environments

As we strive to embed more intelligence in the programming environment, it is inevitable that we must change the way we think about programs. If the machine is to be more helpful in the design, coding, debugging, maintenance and documentation of programs, it must move beyond considering a program as simply a text string, denoting constructs in a programming language. The obvious place to locate the more sophisticated knowledge and behavior we will require of our programs is to distribute it in active objects representing the components of the programming language itself. The responses to messages received by objects representing programs will constitute a new generation of interpreters, some of which will exhibit novel properties.

To illustrate some techniques for exploiting object-oriented representations for programs, we will discuss interpreters that are *reversible*, those that maintain a *history* of the computation as they interpret programs. These interpreters, in addition to conventional program execution, also make it possible to run programs *backwards*, and provide new power for incremental program construction and debugging.

2. Representing programs as objects: the basics

To convey the flavor of the technique, we start by describing the structure of a relatively conventional interpreter for a Lisp-like language, then move on to consider more radical departures. We will show the structure of a simple interpreter based on responses of programming language objects to `EVAL` messages, which ask the objects to evaluate themselves. Whereas the Lisp interpreter is a monolithic function, the object-oriented interpreter is *distributed* by the response of the various objects to messages like `EVAL` or `APPLY`. Alternative interpreters can co-exist with the standard one, based on introducing messages other than `EVAL`. They may make use of the behavior provided by `EVAL` if they wish.

A major advantage of using an object-based representation for programs over Lisp lists is that behavior-sharing mechanisms such as *inheritance* can be used to capture commonalities between programming constructs. New

kinds of program objects can be defined to inherit the behavior of previously existing objects, and add new behavior for EVAL or other messages.

The root of the hierarchy for all objects representing programs is an object called FORM [after the Lisp terminology for an argument to the Lisp EVAL function]. For many interpreters, this object may not have much interesting behavior by itself, but at least serves to identify those objects which have procedural semantics from those that do not. Built directly upon FORM are the "atomic" constructs of the language. Numbers, text strings, and "self-evaluating constants" are all instances of the object CONSTANT. When a CONSTANT receives an EVAL message, it returns itself.

A VARIABLE is an object that has one [instance] variable, its NAME, a symbol or text string. [Note that we don't include the *value* of the variable, since we're talking about unevaluated variables]. The argument to the EVAL message is an ENVIRONMENT object, an object that accepts LOOKUP messages to retrieve the value of a VARIABLE. When a VARIABLE receives an EVAL message, it sends a LOOKUP method to the ENVIRONMENT.

Composite program objects are either FUNCTION-CALLs, SENDs, [if these differ in syntax], or SPECIAL-FORMS which represent other syntactic constructs. Each of these contains a list of ARGUMENTS, each of which is another FORM object.

The response of a FUNCTION-CALL object to an EVAL message is the guts of the interpreter. Its responsibility is to send EVAL messages to each FORM object in its list of arguments. They ultimately send an APPLY message to an object representing the DEFINITION of the function, as a LAMBDA object which sends EXTEND messages to the ENVIRONMENT to bind variables, and evaluates its BODY. Message SENDs work analogously, with MESSAGE objects, and METHODs.

Each of the special forms needs to have its own kind of object. Conditionals evaluate their TESTS, then evaluate their THEN or ELSE parts depending on the outcome of the test. Macros invoke their definitions and initiate another round of evaluation.

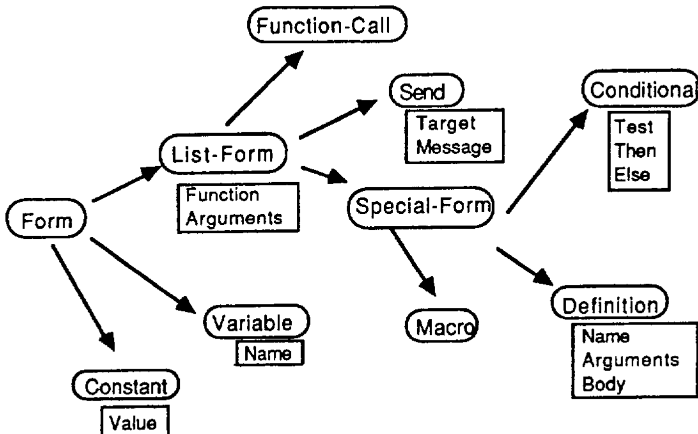

Figure 2-1: The object hierarchy for programs

The object protocol is transparent to control structure decisions in the interpreter. The object interpreter could rely on the stack of the host language, or explicitly create objects for continuations or activation records.

3. FORM objects respond to interpreters other than EVAL

Once we have a suitable object representation for programs, interpreters driven by messages other than EVAL play an important role. They can mediate between the object representation and other representations. Chief among them are the interpreters that correspond to the functions of READ and PRINT. These perform coercions in both directions between the object representation and either text strings or conventional Lisp S-expressions, in a distributed fashion. Each FORM object responds to messages asking it to convert a text string or S-expression to a program object, or to deliver a string or S-expression representation of itself.

These coercions are important, as they allow nonstandard object-oriented interpreters to coexist in a programming environment with more conventional interpreters. An object-oriented interpreter such as is described above may perhaps be slower than a conventional Lisp interpreter by factors of hundreds. But the ability to dynamically and incrementally convert between program representations means that at any point we can "boil away" object representation to produce a representation acceptable to conventional interpreters and compilers. This means that we needn't pay a permanent efficiency penalty for the use of object-oriented interpreters to enhance interactive programming environments.

4. Applications of object-oriented interpreters: A reversible object-oriented stepper

ZStep [Zstep] is a stepping debugger for Lisp designed to provide explicit support for the task of *localizing* bugs which appear in a large body of code, especially in cases where the investigator cannot make good guesses as to where the bug might be. ZStep's goals contrast with traditional debugging tools such as tracing and breakpoints, which are oriented toward the *instrumentation* of suspect pieces of code.

ZStep presents evaluation of a Lisp expression by visually replacing the expression by its value, conforming to an intuitive model of evaluation as a substitution process. The control structure of ZStep allows a user to "zoom in" on a bug, examining the program first at a very coarse level of detail, then at increasingly finer levels until the bug is located. ZStep keeps a history of evaluations, and can be run either forward or backward.

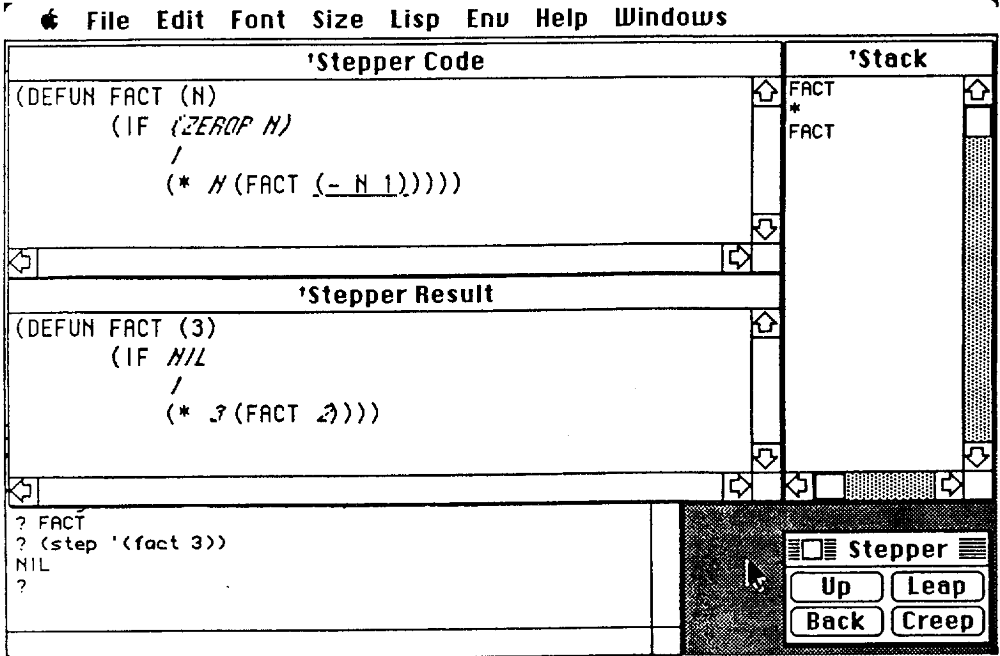

Figure 4-1: A reversible stepper displaying evaluation as substitution

When ZStep presents an expression for evaluation, the user is given a choice about whether to next view the details of evaluation [the *creep* command], or simply see the result of evaluation [the *leap* command]. Successive creep commands create a *linear* structure of incrementally generated evaluation events, whereas the leap commands induce a *tree* structure on the history of evaluations.

Either command can be invoked either in the forward or reverse direction at any point. Thus, the interpreter that generates the successive displays as the program is executed must be a *reversible* interpreter. We will show how the technique of object-oriented interpreters makes this easy to implement an interpreter that runs in parallel with Lisp's, and provides the necessary reversibility property.

An important property of a debugger like ZStep is that the control structure of the stepper must be *decoupled* from the control structure of the program being debugged. The traditional technique for implementing steppers is to piggyback the stepper's control structure onto the control structure of the program, using hooks into the interpreter provided by the EVALHOOK feature of Common Lisp [CommonLisp]. This won't work for a debugger like ZStep, where the control structure of the program always runs "forward", recursively, despite the fact that the debugger may run, iteratively, in either direction.

5. EVENTs provide a reversible representation of evaluation steps

The basic kind of object manipulated by ZStep is called an EVENT. Different kinds of EVENT objects represent different states the interpreter can be in. An EVAL-EVENT is produced when the interpreter is just about to

evaluate a form, while a RESULT-EVENT says the interpreter is just about to return a value for a form. Similar objects exist for APPLY.

Events are linked in four ways: First, every event is doubly linked to its "leap" event, EVAL-EVENTs to EVAL-RESULT-EVENTs and vice versa. Every form thus knows its value and every value knows what form produced it. Every EVENT is linked to its UP-EVENT, the one for which it is evaluating a form, or to which it is returning a value. The chain of UP-EVENTs thus constitutes the "stack".

Each event is linked to its CREEP and PREVIOUS events, strictly in the order of "single-stepping" evaluation. The CREEP events are computed lazily; The first time an event receives a CREEP message, it computes the next step of the interpreter, then stores it to answer future CREEP queries.

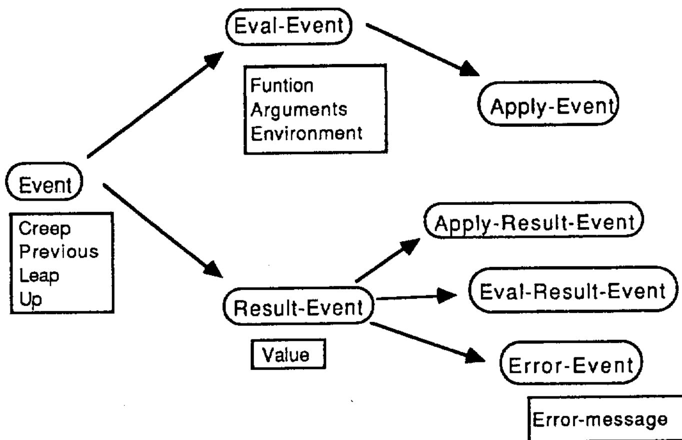

Figure 5-1: The hierarchy of events

The ZStep object-oriented interpreter is a distributed one, working by responses of each FORM object to a SINGLE-STEP message. It takes an event as argument, and returns the next event in the sequence of evaluation, if it has not already been computed. This becomes the next event accessed by the creep command, and initialization of the event object automatically links the event objects in both directions.

To illustrate the flavor of the interpreter code, we present the response of a FUNCTION-CALL object to a SINGLE-STEP message.

```
If I'm a FUNCTION-CALL object, and I get a SINGLE-STEP message,
 with an EVENT, asking me to produce the next EVENT:
  If I don't have any ARGUMENTS,
   I return a new APPLY-EVENT
    whose FUNCTION and ARGUMENTS are copied from mine.
  Otherwise, I return a new EVAL-EVENT
   evaluating the first element of my ARGUMENTS list,
   with an ARGUMENT-EVALUATION object
    that will eventually evaluate
    all the other ARGUMENTs and apply the FUNCTION.
  I copy the ENVIRONMENT and APPLY-EVENT for the new event
   from the old EVENT,
   and the UP-EVENT for the new event is the old EVENT.
```

Figure 5-2: Part of a reversible single-stepper

Argument evaluation objects are like continuations, in that they receive control after values are returned, and initiate the remainder of the computation. They carry any state that must be preserved over the recursive evaluation of argument expressions.

6. Objects representing errors are another example of unconventional interpretation techniques

When evaluating some code with ZStep, if the code causes an error, instead of substituting the value for the code in the result window, ZStep *substitutes the error message for the code which caused the error.* This is important because it makes visually clear the context in which the error happened. The user can use the commands which browse the event structure to discover the cause of the error.

Obviously, if an error occurs in the program being debugged, we don't want a corresponding error to happen in the debugger! This is a grave problem for traditional debuggers that couple the control structure of the debugger to that of the target program. To deal with this, we introduce *objects explicitly representing errors* into the language.

ERROR-EVENTs are similar to RESULT-EVENTs, except that they hold an error message instead of a returned value. They also must be linked to their corresponding EVAL-EVENTs. The rule for propagation of error events says that if any expression contains an error event, that expression must itself yield an error event, including the back pointers that maintain the reversibility of all ZStep events. Error events linked by the UP-EVENT link preserve the "stack" at the time of the error, leaving the potential for fixing and resuming the computation at any point.

Tinker, the example oriented programming system described below, also makes use of error objects, albeit somewhat differently, to implement non-standard semantics for error situations.

7. A reversible object-oriented interpreter provides the means for implementing example-based programming

Tinker [Seeing], [Aisbed] is a novel programming environment which synthesizes Lisp programs from examples. To write a program with Tinker, a user presents an example of data the program is to operate on, and demonstrates the steps of the computation on the example data. As each step is performed, the results in the

concrete example are displayed, and the step is also remembered to become part of the final program. By indicating which objects represent arguments to the procedure to be constructed, the user informs Tinker what generalizations to make. A "teaching sequence" of examples may be presented, each building upon the partial definition constructed from previous examples. When several examples are presented, Tinker asks the user to provide criteria for distinguishing between them, resulting in procedures containing conditionals. Tinker has a menu-based graphical interface, as shown below.

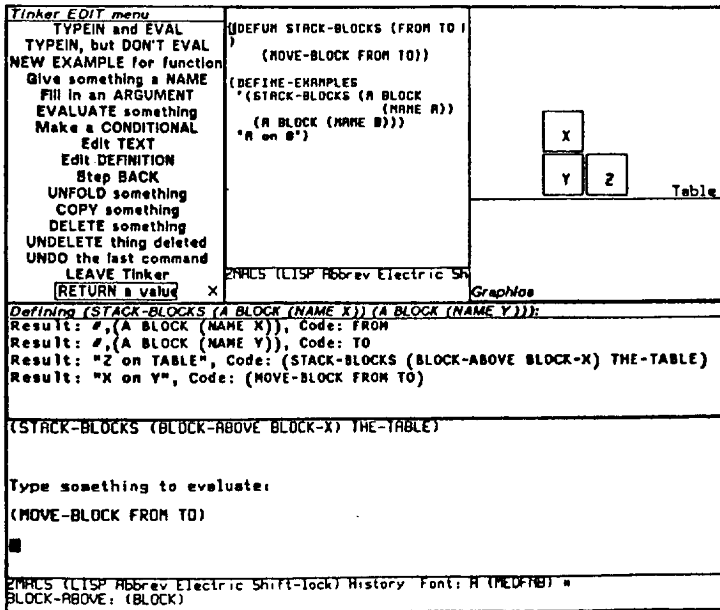

Figure 7-1: Writing a blocks world program by example with Tinker

The heart of Tinker, like ZStep, is a reversible object-oriented interpreter. While ZStep's interpreter starts by running forward, beginning with Lisp code and unfolding the sequence of events that take place for a particular computation, Tinker's *normally runs backward*. Starting with an example computation, Tinker's interpreter eventually generates Lisp code that corresponds to the user's intended program.

Values are input to Tinker in one of two ways: either by evaluating typed Lisp expressions, or as the result of demonstrative actions like menu selections. In any case, Tinker converts the expression to a FORM object. The evaluation of this FORM object via EVAL messages produces what Tinker calls a RESULT object. This object contains both the value, and the original FORM that produced it. Both the value and code are displayed for each such object, appearing in the center window of the Tinker screen.

More complex expressions are built out of simple ones by utilizing the RESULT objects as arguments in further computation. The user can point to an already-computed value, and pass it as an argument to some other function, using the menu operation **Fill in an argument**. The function will be executed using the indicated value, but the RESULT object produced will contain the code that created the value instead of the value itself. Thus every Tinker computation represented by a RESULT object records a complete history of its computation, which is finally read out in the opposite direction when Tinker constructs the final program. While conventional

interpreters analyze expressions from the outside in, Tinker builds up an expression "inside out".

Tinker has no need of the doubly-linked linear history of events produced by ZStep. But Tinker does have another novel data structure for interpretation of programs, motivated by the need to handle multiple examples. A RESULTS object is similar to the RESULT object introduced earlier, but records the correspondence between a piece of code and its execution in *more than one environment*. As multiple examples are presented to Tinker for the same function, a RESULTS object is sent a MERGE message to incorporate each successive example. The RESULTS object contains a list of VALUES and ENVIRONMENTS, instead of just a single value and environment. For subexpressions where the examples differ, Tinker assumes the user intended to generate a conditional. The set of values and environments for that subexpression becomes *split*, with one subset following the THEN part of the conditional, and the other subset following the ELSE part.

8. Object-oriented interpreters have application in language design

One reason object-oriented languages are popular among system developers is that they provide a good substrate for further experimentation in developing new languages and systems. A major category of applications of object-oriented interpretation techniques is in the implementation of new languages on top of existing object-oriented ones. One can use an object-oriented representation of programs in the embedded language, and define an interpreter which implements the semantics of the language by "unfolding" it into the implementation language. This is another kind of reversibility that can appear in object-oriented interpreters. Scripter [ApiarySimulator] and its successor Pract [Pract] interpret actor programs by transforming them from a recursively nested expression form to unidirectional message passing using generated *continuation* objects. Symbolic evaluators, "code walkers", indexers, and partial evaluators are also examples of alternative evaluators that can profit from implementation in an object-oriented fashion.

The idea of representing programs as objects, by itself, is not new. The most radical object-oriented languages, such as the actor languages [Coop] and Smalltalk [SmalltalkLanguage] represent *everything* as objects, and so have no choice if they wish to represent programs at all. The actor languages have long had object-oriented interpreters written in the manner we describe. Smalltalk-80, a compiler-based system, lacks an interpreter, but implements its compiler through compilation messages, though perhaps not as accessibly as one would like.

It has been pointed out that the *denotational semantics* technique for defining the meaning of programming languages [DenSem] bears some conceptual similarity to object-oriented interpretation techniques. Syntactic categories of language constructs are the "objects", and equations involving them constitute the "interpreter". However, these formalisms are weak on dynamic representations, don't explicitly consider reversibility, and have little to say about the consequences for programming environment issues such as incremental program construction and debugging.

The examples presented here far from exhaust the uses of reversible object-oriented interpreters. We hope that these examples give some indication of the power and utility of the techniques. Computer programs are complex entities which can be viewed in many different ways, and object-oriented representation of programs provides a convenient and effective means of realizing some of the possibilities.

9. Acknowledgments

Major support for this work was provided by by the System Development Foundation. Related work at the MIT AI Lab was sponsored by DARPA under ONR contract N00014-80-C-0505.

References

1. Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass., USA, 1983.

2. Henry Lieberman. An Object Oriented Simulator for the Apiary. AAAI-83, American Asociation for Artificial Intelligence, Washington, D. C., USA, August, 1983.

3. Henry Lieberman. Steps Toward Better Debugging Tools for Lisp. Proceedings of the Fourth ACM Conference on Lisp and Functional Programming, Austin, Texas, USA, August, 1984.

4. Henry Lieberman. "Seeing What Your Programs Are Doing". *International Journal of Man-Machine Studies 21*, 4 (October 1984).

5. Henry Lieberman. "An Example Oriented Environment for Beginning Programmers". *Instructional Sciences 14* (1986), 277-292.

6. Carl Manning. Traveler: the Apiary Observatory. ECOOP-87 [this volume], Paris, France, June, 1987.

7. Guy Steele, et. al.. *Common Lisp: The Language.* Digital Press, Maynard, Mass., USA, 1984.

8. Joseph Stoy. *Denotational Semantics.* MIT Press, Cambridge, Mass., USA, 1977.

9. A. Yonezawa and M. Tokoro, eds.. *Concurrent Object Oriented Programming.* MIT Press, Cambridge, Mass., USA, 1987.