Annotation-Based Program Stepping[1]

Glen Randolph Parker
Palladian Software
4 Cambridge Center
Cambridge, MA 02142

Abstract

Traditionally, stepping is achieved by modifying the evaluator. In the approach presented here, stepping is achieved by annotating the user's program with calls to stepping procedures. This provides greater flexibility in the selection of which program sections the user is interested in stepping. In addition, the sections not selected are executed without overhead. Stepped program sections are visually highlighted inside the user's program editor, providing a sense of context. Stepping control mechanisms, such as the ability to execute until a certain point and then step, permit the user to navigate through a program in a precise and flexible manner. A prototype stepper based on this approach has been implemented.

1 Introduction

A program stepper allows the user to execute code section by section in steps of his own choosing. The typical stepper executes code, stopping and waiting for user instruction before and/or after each section is executed. Any values returned by a section of code are displayed by the stepper. The user can investigate the state of various data structures in search of anomalies, and can easily follow the control path of the executing program. Much of the importance of steppers in debugging arises from the fact that most program errors are harder to locate than to fix. It has also been claimed in [Plattner 81] that monitoring the execution of a program is more effective in the debugging process than studying the program text, because the programmer's intuition about the program state space is more highly developed than his intuition about the program text.

At each stop the code section currently being executed is displayed, and the stepper waits for command input. The user must tell the stepper which section to step next. *"To step"* a program statement means to stop and display that statement before its execution, and then, barring explicit instruction to the contrary from the user, to display any values returned by it after execution. In the event that a statement is not stepped it is just executed normally. The following describes some of the key commands supported by the typical LISP stepper:

[1]This paper is taken from MIT AI Lab Working Paper 283 (available only from the author), which is a revised version of a thesis submitted in May 1986 to the Department of Electrical Engineering and Computer Science at MIT in partial fulfillment of the requirements for the degree of Bachelor of Science. ©G. R. Parker, 1987.

- *STEP-WITHIN* Evaluate the current form, stopping before evaluating any subforms and after evaluating the current form.
- *STEP-NEXT* Evaluate the current form without stepping and stop before evaluating the next form at the current level.
- *STEP-UP* Evaluate the current form and stop after its superior is evaluated.
- *STOP-STEPPING* Evaluate back to the top level without stopping.
- *RETURN-VALUE* Return a user-supplied value as the result of the current form's evaluation.
- *STEP-THROUGH-NEXT* Evaluate until after the next form at the current level is evaluated.

Whenever a stepper stops, a section of code is either just about to be evaluated or is just about to return a value. If the user chooses the command *STEP-WITHIN*, the stepper then stops immediately before the next evaluation occurs. At each stop the user has the flexibility to view or change values.

1.1 Key Features of the Annotation-Based Approach

Annotation-based program stepping is the addition of program statements to the code to be stepped, in such a way that the result of the code is the same, but systematic control of the execution is permitted.

The annotation-based stepping approach provides the following advantages over traditional steppers:

- *Unmodified evaluator*
  This approach departs from standard stepper methods in that it controls evaluation without modifying the evaluator. The program annotations themselves, as opposed to hooks into the evaluator, invoke the stepper functionality.
- *Efficiency*
  Since only that code which is to be stepped is annotated and the evaluator operates normally, all other program statements are unmodified and there is no overhead when not stepping. This combined with the ability to step compiled code enables runtime-bound programs to be stepped.
- *Language/environment independence*
  The annotation approach is not dependent on the language used in the environment. Stepper control instructions can be introduced into the execution stream in any language. In particular, this approach does not require the existence of an interpreter, and thus can be applied to compiled code. [Balzer 69] demonstrates the use of annotation in a debugging system for PL/I code.

- *Convenient selectivity*
  Existing steppers either give the user too little information or too much. The annotation approach allows the user to conveniently select exactly which code sections are to be stepped. The user may choose to step all the functions in a file or only a particular section of a single function. This also permits the user to avoid stepping through system functions.

A prototype stepper based on this approach has been implemented. VisiStep, the prototype, has several other key features which are not fundamentally part of the annotation approach but which interact synergistically with it.

- *Flexible navigation*
  A drawback of standard steppers is that long chains of stepping commands are often needed in order to get to specific points within the code.

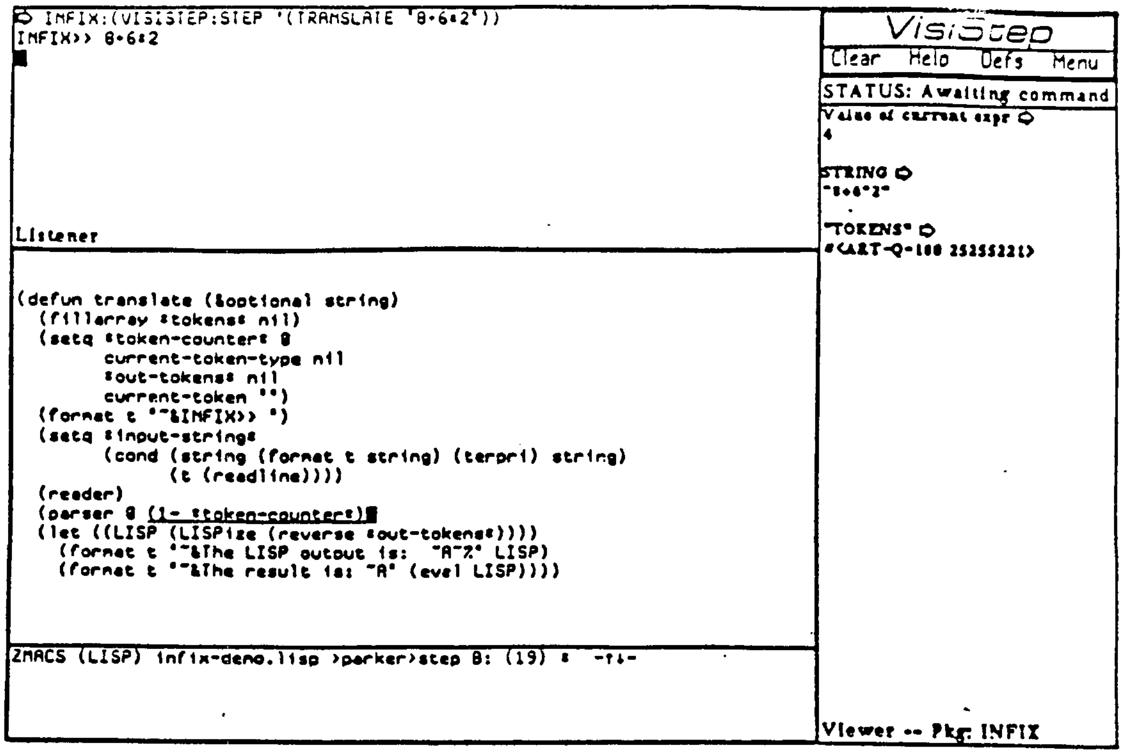
  VisiStep includes several navigational commands, such as the ability to point to specific places in the program to step. For example, one can choose to step just one critical expression inside a large loop.
- *Contextual display*
  The current expression is highlighted inside the user's original source code, so that it appears alongside the surrounding code and documentation. This contrasts with the line-oriented display mode of most traditional steppers. Only the user's code is stepped, so the execution of system functions is not seen by default.
- *Editor integration*
  VisiStep operates inside the user's program editor in a nonintrusive manner. Editing commands and stepping commands are available for simultaneous use.

2 The VisiStep Prototype

VisiStep does not act until the user decides to step some code. Before initiating stepping, the user must select which definitions to prepare for stepping. When he is ready to start stepping, the user evaluates a form that will result in the execution of one of the prepared functions. While stepping, the user has access to all of the canonical stepper commands described earlier, as well as some more complex navigational commands. He may also prepare the definition of a function at any point before it is called.

The user may also prepare just a subform of a function; all other subforms of that function will execute normally. At all times the program's execution is being displayed in the context of the user's source code, making it more understandable and easier to debug. The currently executing form is highlighted in the editor.

As shown in the sample display, there is a viewer window that allows the user to continuously monitor the values of LISP expressions. In the event that an error occurs while trying to evaluate the expression, such as in the case of a lexical variable being evaluated out of its context, the error message is displayed in place of the value.



```
INFIX:(VISISTEP:STEP '(TRANSLATE "8+6*2"))
INFIX>> 8+6*2
```

```
(defun translate (&optional string)
  (fillarray *tokens* nil)
  (setq *token-counter* 0
        current-token-type nil
        *out-tokens* nil
        current-token "")
  (format t "~&INFIX>> ")
  (setq *input-string*
        (cond (string (format t string) (terpri) string)
              (t (readline))))
  (reader)
  (parser 0 (1- *token-counter*))
  (let ((LISP (LISPize (reverse *out-tokens*))))
    (format t "~&The LISP output is:  ~A~%" LISP)
    (format t "~&The result is: ~A" (eval LISP))))
```

```
ZMACS (LISP) infix-demo.lisp >parker>step 8: (19) *  -↑↓-
```

```
STATUS: Awaiting command
Value of current expr
4

STRING
"8+6*2"

*TOKENS*
#<ART-Q-100 25255221>
```

Figure: a sample VisiStep display

VisiStep supports several advanced navigational commands which permits the user to point to a form and cause the system to evaluate normally to that point and then to continue stepping.

One of the features of VisiStep is that it supports stepping at several levels of detail, or granularity. Most ordinary steppers show everything to the user, all at a constant level of detail. VisiStep can provide this, but provides several other choices. The user may begin stepping a function that previously was beneath his level of interest without having to step through system internals. With VisiStep, one can also choose to step selected functions, and even selected forms within individual functions. Ultimately, the more advanced navigational commands permit the user to direct the stepper to an exact location for stepping to resume.

The extended version of this paper [Parker 86] presents a full scenario of usage of the VisiStep stepper.

3 Implementation Issues

3.1 The Stepping Wrapper

Most traditional steppers temporarily bind the definition of the system evaluation function to a stepper function that displays the form and values, accepts command input, allows manipulation of the environment, and eventually calls the system evaluation function (EVAL) to actually perform the evaluation. Most contemporary LISPs (e.g. MACLISP [Pitman 83], ZETA-LISP [Symbolics 85], and Common LISP [Steele 84]) provide a system variable, *EVALHOOK*, which, when non-NIL, is the alternative evaluation function.

In contrast, the annotation-based approach leaves the evaluator completely unchanged, instead augmenting each form of the user's function definitions with the information required to operate the stepper.

As an example, here is a definition of a LISP absolute-value function, ABS:

```
(DEFUN ABS (X)
  (COND ((MINUSP X) (- X)) (T X)))
```

After preparation for stepping, that definition will look like:

```
(DEFUN ABS (X)
  (WRAPPER
    (COND ((WRAPPER (MINUSP (WRAPPER X ABS (3 1 1 0 1) T))
                    ABS (3 1 1 0) T)
           (WRAPPER (- (WRAPPER X ABS (3 1 1 1 1) T))
                    ABS (3 1 1 1) T))
          ((WRAPPER T ABS (3 1 2 0) T)
           (WRAPPER X ABS (3 1 2 1) T)))
    ABS (3)))
```

That is the conceptual representation; in the actual implementation the arguments for each call to WRAPPER are stored in a table for ease of access.

The stepper function shown above as WRAPPER is called the *stepping wrapper*. Its arguments are, respectively, the form being wrapped, the name of the function in which the form appears, a path specification of which form in the parent definition to highlight, and a condition flag that signals whether to step the form.

The stepper wrapper is implemented as a LISP special form, which calls one internal stepping function to handle entering a form and then calls a second with the result of the first when returning from that form. The first function determines whether the form is to be stepped, highlights the form in the editor and accepts commands from the user if appropriate, and then returns a value to the second function that controls whether or not it should stop and display the value when returning from the form. The second function evaluates the form normally, and, if appropriate, displays the value and accepts commands. For these actions to occur it is necessary for the wrapper to receive several arguments that are determined when it is first created, enveloping a form inside a user definition.

The condition flag allows forms to be wrapped, but not necessarily stepped without explicit instruction from the user. It also permits user-defined conditional stepping. The internal stepping functions also update the values in the viewer pane and maintain global information for the preemptive navigation commands.

3.2 Annotating a Definition

When the user specifies that a definition is to be stepped, a central annotating function is called. In order to prepare the definition, it must determine the name of the definition, the definition itself, and, if only a subexpression is being prepared for stepping, a unique path specification for that form. One of the requirements of a LISP environment in order to support VisiStep is the existence of editor functions that handle LISP syntax, enabling VisiStep to determine the definition name and the subexpression path for the current location in the editor buffer and permitting easy navigation amongst S-expressions in the buffer. The editor must also provide source code location facilities to display and read the body of the desired definition.

An annotation dispatcher is called on the body of this definition, and then recursively on each subform. It determines what the appropriate method for preparing each form is, depending on such things as whether it is a lambda-expression, a call to a special form, or a symbol. For normal function-calling forms the form and each argument to the function are wrapped. However, some functional forms, among them special forms and macros, do not necessarily evaluate all their arguments, but instead control the evaluation themselves. VisiStep cannot, for instance, wrap the arguments to a `COND`, as each of these arguments is itself a list of forms to be evaluated in a select way. Instead, it relies on knowledge (like most code-walkers) about the evaluation of the arguments of system special forms and macros, and thus is able to properly wrap them.

In the event that the user has specified when wrapping that only a certain section of a definition should be stepped, the annotation mechanism wraps the whole definition but only activates the stepping switch in the desired forms. The deactivated prepared forms evaluate normally, albeit with the slight overhead of testing a flag.

4 Limitations and Extensions

4.1 Simplifying Assumptions

As explained in Section 3, system special forms are able to be stepped due to extra information found in a special form template provided for each of them. No user-defined special forms are handled, as it is considered improper for the user to create his own special forms in LISPs currently in use (e.g. Common LISP).

A significant limitation of the current prototype is in its source code highlighting in instances of macro usage. This is due to the nature of macros in that the evaluable components of the macro are not known until the macroexpansion time, which occurs either at compile time or, if interpreted, at evaluation time. A user can inform the system of how to step the arguments of a call to a user macro by creating a special form template for it. The system uses several tricks to determine whether a form in the macroexpansion is an evaluable form present in the user's original source code or whether it is a product of the macroexpansion. Unfortunately, this is a difficult decision in the case of atoms in the arguments to the macro, and certain assumptions are made regarding macroexpansions.

Support for the viewer display of lexical variables currently takes advantage of the fact that special forms (such as the stepping wrapper) are passed an argument that specifies the lexical environment in which they are to be evaluated or expanded (when being compiled). However, this environment does not exist in the compiled code, and so the ability to display local variables while stepping a compiled function is reduced. In particular, the local variables are determined at the time the wrapper is compiled, as opposed to being determined at run time.

The current system relies heavily on the state of the editor and the user's editor buffer, and an undetected change in these can create problems. For instance, editing the code that is current being stepped may invalidate the path information stored in the stepping wrapper, and as a result the wrong forms may be highlighted in the buffer.

4.2 Future Extensions

There are several proposed extensions to the operation of VisiStep. The first is the ability to make stepping conditional upon the values of arbitrary LISP expressions. Support for this exists currently in the format of the stepping wrapper, but a user interface to this option is necessary. Also, it would be helpful if the navigation feature allowed the user to select an arbitrary number of points to which execution should continue without stepping. Finally, several steppers have had at least some limited "backup" capability, whereby the user can return to the beginning of a form that is an ancestor of the current form and restart the computation.

Since the use of macros pervades modern LISP programming, the limitation of the approach regarding macros is a serious one. Specifically, more study is needed with respect to atoms in macros.

The system is written for a LISP implementation with both a window system and a powerful editor, and such LISP environments are now found on conventional hardware as well as on LISP machines. As a result, the system will be ported to other machines in the near future.

It is envisioned that eventually most of the debugging capabilities of advanced LISP environments, including the debugger and the tracer, can be enhanced and incorporated into the VisiStep framework.

5 Related Work

Stepping is an old idea. The research related to VisiStep consists of two main areas: the development of other LISP steppers and the use of program monitoring tools. Much of this work has originated from the debugging tools used in old assembly-language programming systems. The result of the development of other LISP steppers was the generation of a standard stepper command set. More recent attempts in this category have focused on integrating steppers with the advanced user interfaces available under current LISP implementations.

Most of the related stepping tools were developed during the late 1970's in the MACLISP environment at MIT. Several were implemented, each providing some functionality not yet available. VisiStep has included the best ideas from each. The typical stepper command set presented earlier in the paper applies to most of the following.

One of their fundamental limitations was that they could only be used with interpreted code, due to the use of the evalhook mechanism inside the interpreter. Although they suffered no overhead when turned off, these steppers did incur a small bit of overhead with each evaluation with this mechanism turned on. STEP [Rich 76] provided the ability to select certain functions for stepping, but did so only with a significant amount of overhead on each execution. It also provided the ability to step inside the expansion of a macro. STEPMM [Morganstern 76] was another of the early MACLISP steppers. It permitted the user to specify conditional stepping and conditional breakpoints. It also included the ability to substitute a form to be evaluated in place of the current form. STEP* [Waters 78] included the ability to reevaluate the arguments to a function, the ability to reevaluate a function with stepping, and the ability to stop after the evaluation of a form.

More recently, two steppers are significant for their attempts to integrate display of the program source code with the output from the stepper. These are DIDL [Halbert 78] and Zstep [Lieberman 84], operating under MACLISP and LISP Machine LISP, respectively. DIDL had two main objectives: (1) to be a totally comprehensive debugging tool, including breakpoints, tracing, and stepping, and (2) to use display terminals to display program text in a flexible manner. DIDL introduced the use of display terminals to highlight the user's code; however, it displays the internal representation of function definitions and not the original source code.

Zstep [Lieberman 84] extended the concept of contextual display by using the LISP Machine editor to display evaluation inside the editor. Zstep used two editor windows, one which maintained the unchanged source code being stepped, and the other which replaced the current expression with its value. Zstep did this in order to model evaluation as a substitution process. It also maintained a history of past expressions. It permitted the user to "backup" and reevaluate a prior form, but it did not handle the presence of side-effects. When an evaluation produced an error, an object representing the error was returned.

The EXDAMS system [Balzer 69] is one of the earliest departures from the assembly-language style of debugging tools. The EXDAMS philosophy is that the user first ascertains *what* is happening, decides if it is incorrect behavior, and, if necessary, determines *how* the program produced this behavior. In order to provide a flexible monitoring system that could be easily extended, EXDAMS executes a program and records a file of all history and execution information. Then, interactively, the user can go back and trace the path of execution, monitoring the values of variables in a viewer window. The user can move both forward and backward in the execution with relation to time. In order to facilitate the recording of this execution data the original program is annotated with history-generating statements. [Balzer 69] demonstrates the use of the system on PL/I code, underscoring the language independent nature of the annotation-based approach.

Acknowledgements

Not enough credit can go to Dick Waters, whose ideas provided the foundation for this paper. Critical commentary of drafts of this paper was provided by Dick Waters, JonL White, and Shane Hartman. Previous versions of this paper also benefited from discussion with Kent Pitman, Henry Lieberman, and Chuck Rich.

References

[Balzer 69] Balzer, R. M. EXDAMS – EXtendable Debugging and Monitoring System. pp. 567 - 580, in Proceedings of the AFIPS Spring Joint Computer Conference, Vol. 34, 1969.

[Halbert 78] Halbert, Daniel C. A LISP Debugger for Display Terminals. B.S. Thesis, Dept. of EECS, Massachusetts Institute of Technology, May 1978.

[Lieberman 84] Lieberman, Henry. Steps Toward Better Debugging Tools for LISP. Proceedings of the 1984 ACM Conference on LISP and Functional Programming. Austin: ACM, 1984 pp. 247 - 255.

[Morganstern 76] Morganstern, Matthew. MIT online documentation for the STEPMM MACLISP stepper, March 1976.

[Parker 86] Parker, Glen R. Annotation-Based Program Stepping. MIT AI Lab Working Paper 283, June 1986. Available only from the author.

[Pitman 83] Pitman, Kent M. The Revised MACLISP Manual, Saturday Evening Edition. MIT Laboratory for Computer Science TR 295, May 1983.

[Plattner 81] Plattner, Bernhard, and Jurg Nievergelt. Monitoring Program Execution: A Survey. pp. 76 - 93, IEEE Computer, November 1981.

[Rich 76] Rich, Charles. MIT online documentation for the STEP MACLISP stepper, November 1976.

[Steele 84] Steele, Guy L., Common LISP: The Language. Maynard, MA: Digital Press, 1984.

[Symbolics 85] Symbolics, Inc. Documentation for the Symbolics 3600 Release 6, 1985.

[Waters 78] Waters, Richard C. MIT online documentation for the STEP* MACLISP stepper, 1978.