si—An Interpreter for the C Language

Alan R. Feuer

55 Wheeler Street
Catalytix Corporation
Cambridge, Mass 02138

(617) 497-2160
cataly!arf

ABSTRACT

An interpreter for the C language has been developed that offers several advantages during program development over the traditional C compiler/linker/debugger: faster turnaround for modifications, finer control over program execution, better debugging facilities, and more complete run-time error checking. The interpreter is named.si, the Safe C Interpreter.

This paper begins by describing those features of **si** that distinguish it from the traditional C development environment. Next it contains a sample session that illustrates the interaction between a programmer and the Interpreter. It concludes with a discussion of a few key design decisions made during the development of **si**.

INTRODUCTION

The advantages of highly interactive program development environments have been described widely [GOLD83, KAY69, REIS84, SAND78, TEIT81]. All promise more productive software development at the expense of more computer cycles; a good tradeoff, these days.

This paper describes **si**, an interpreter for the C language that, while not a full-fledged programming environment, shares many attributes with the more ambitious projects. Rather than supplanting its host, **si** has been designed to blend into its operating environment.[1] Its major advance is that it provides for C the fast turnaround, good debugging tools, fine control over execution, and extensive error checking of an integrated programming system without sacrificing the utility of existing program development tools.

MAJOR FEATURES

si was designed to be a programming tool for application programmers.[2] It handles multi-file, multi-directory programs, it works with any text editor or special preprocessor, it can access compiled code, and it understands the full C language.

The first attribute of the Interpreter that attracts people used to compiling C is the short cycle between editing a program and running it. By removing from the edit-run cycle compiling, assembling, and link editing, changing a program becomes a low-penalty operation.

---

1. The Interpreter is running under UNIX, VMS, MS/DOS, and on the Macintosh. However, not all implementations have all features.
2. A lean version of the Interpreter is being packaged with a textbook for use in teaching C.

Link editing is done at run time as needed. One consequence is that pieces of a program can be run in isolation. As the functions of a program are developed they can be tested without the need to generate temporary driver routines.

Run-time errors, such as using a variable before it is defined or indexing/pointing beyond the end of an array (even for dynamic arrays), are trapped. Compiled programs that work by taking advantage of system-dependent circumstances, e.g., a NULL at location zero or physical adjacency of separately defined variables, generate run-time errors while being interpreted.

Parameters to functions are also checked at run time. The number and types of the parameters in a function call are checked against the function definition. For the formatted I/O routines of the **printf/scanf** family, the conversion specifications in the control strings are checked against the types of the remaining parameters passed. For routines in the standard libraries, the value of the passed parameters are checked for sanity. For example, in a call to **fseek**, if the origin is from the start of the file then the offset is checked to make sure it is positive.

There are several ways to stop a program in the Interpreter during its execution: A run-time error will halt the program immediately; an interrupt from the keyboard will stop the program at the next semicolon in the source text; a program can be single-stepped, stopping at each semicolon; assertions inserted into the source code will halt the program if they fail; explicit breakpoints can be set either in the code or from the command line.

When a program is stopped, the Interpreter accepts C expressions from the standard input. Each expression entered is evaluated in the context of the stopped program just as though it were inserted at the current point of execution.

A program can be traced while it executes. Function call/return, statement execution, and expression evaluation can each be traced separately or in combination.

SAMPLE SESSION

The sample session that follows shows the Interpreter running on the UNIX system. In the displays, user input appears in **bold** font and comments are printed in *italics*.

Basic Interaction

Interaction with the Interpreter is modeled after program development on the UNIX system. The Interpreter operates in two modes: *command* (the Shell) and *execution* (C). On invocation, the Interpreter is in command mode:

```
$ si      invoke si
si.$
```

Command mode is used to establish the environment for running a program. In the file system, programs are kept in files; within the Interpreter they are packaged in modules. A *module* is the internal representation of a file. When a file is read into the Interpreter a corresponding module is created:

```
si.$ r echo.c    read a file and create a module
loading "./echo.c"
si.$ p    print the current module
echo (in file ./echo.c)
 1   main(argc,argv)
 2   int argc;
 3   char *argv[ ];
 4   {
 5     int i;
 6
 7   STOP( );
 8     for( i = 1; i<argc; + +i )
 9             printf("%s%c", argv[i], i<argc—1?' ':'\n');
 10  }
si.$
```

Programs can be executed in the Interpreter using syntax modeled after the Shell. Shell quoting, environment variables, pattern matching, and I/O redirection are all supported:

```
si.$ echo \'Twas brillig "at $HOME"
STOP on line 7 in function "main" of module "echo"
```

Execution mode is entered whenever a program is stopped. One way to stop a program is to call the built-in routine **STOP**, as on line seven of **echo**. In execution mode the Interpreter accepts C expressions from the standard input, evaluates them, and prints the resulting value and type. Any expression can be entered, including calls to functions and references to active variables:

```
. argc
= (int) 3
. argv[0]
= (char *) 643492 (0x9D1A4) "echo"
. PR(argv)
argv (char **)
        [0]     = 643492 (0x9D1A4) "echo"
        [1]     = 643504 (0x9D1B0) "'Twas"
        [2]     = 643516 (0x9D1BC) "brillig"
        [3]     = 643528 (0x9D1C8) "at /usr/arf"
        [4]     = <NULL>
PR on line 7 in function "main" of module "echo"
= (void) <UNDEFINED>
```

The built-in function **PR** expands the first level of structure for aggregate data. It does not return an explicit value, so the result of the call is **<UNDEFINED>**.

Variables can be modified using standard C syntax:

```
. argv[1] = "'Twasn't"
= (char *) (644044,644053) 644044 (0x9D3CC) "'Twasn't"
```

The pair of numbers printed in the parentheses above are the bounds of the pointer **argv[1]**.

Execution of the stopped program continues following an end-of-file (**<eof>**):

```
. <eof>
'Twasn't brillig at /usr/arf
si.$
```

To change a module, the Interpreter invokes a user-specified text editor on the file associated with the module. In this example I will use the **ed** editor because it matches the paper medium best:

```
si.$ e  edit the current module
134
7d      delete the call to STOP
w
126
q
loading "./echo.c"
si.$
```

After editing, the Interpreter rereads those files that have been changed since the last time they were read. **echo** now reflects the change, i.e., it no longer stops on line seven:

```
si.$ echo my text editor is called $editor
my text editor is called /bin/ed
si.$
```

Watching a Program Execute

A program can be traced on any of three levels: expression, statement, and function. Each level is controlled by the setting of a flag:

```
si.$ +te        enable tracing of expressions
—d —l —ss +te —tf —ts
```

In an expression trace, the operands to an operator appear before the operator:

```
si.$ x  enter execution mode
. 3.4 + 5*6
        3.4 = (double) 3.4
              5 = (int) 5
              6 = (int) 6
        5*6 = (int) 30
3.4+5*6 = (double) 33.4
= (double) 33.4
. 1 || 0
        1 = (int) 1
1||0 = (int) 1
= (int) 1
. 1 && 0
        1 = (int) 1
        0 = (int) 0
1&&0 = (int) 0
= (int) 0
```

The levels of tracing can be mixed in any combination. Here is a statement and function trace of a recursive power function:

```
si.$ r power.c
loading "./power.c"
si.$ p    power is now the current module
power (in file ./power.c)
1   power(x,n) int x, n; { /* return x to the nth power */
2       if( n>0 ) return( x*power(x,n—1) );
3       else return(1);
4   }
si.$ —te, + tf,ts          expression trace off, function and statement trace on
—d  —l  —ss  —te  +tf  +ts

. power(3,2)
        power(3,2)
power:  n = (int) 2
        x = (int) 3
if( n>0 )
return( x*power(x,n—1) )
power:  n = (int) 1
        x = (int) 3
if( n>0 )
return( x*power(x,n—1) )
power:  n = (int) 0
        x = (int) 3
if( n>0 )
return( 1 )
power = (int) 1
power = (int) 3
power = (int) 9
= (int) 9
```

Detecting Run-Time Errors

Run-time errors such as indexing beyond the end of an array, marching a pointer off the end of an object, passing the wrong arguments to a function, or using a variable before it is defined are caught by the Interpreter. When a run-time error is detected, the program is stopped and a new expression evaluator, in Lisp-like fashion, is spawned:

```
. $      $ is a generic variable
= (void) ⟨UNDEFINED⟩        initially $ is undefined
. $ = malloc(—10)
bad argument to "malloc"
        int arg should be >= 0
STOP
. .
```

Entering end-of-file returns to the previous evaluator:

```
. . ⟨eof⟩
= (void) ⟨UNDEFINED⟩
. $ = malloc(10)          $ takes on the type of the value assigned to it
= (char *) (403024,403034) 403024 (0x62650) ""
. PR($)
$ (char *)
        [0]     = '^@' (00)
        [1]     = '^@' (00)
        [2]     = '^@' (00)
        [3]     = '^@' (00)
        [4]     = '^@' (00)
        [5]     = '^@' (00)
        [6]     = '^@' (00)
        [7]     = '^@' (00)
        [8]     = '^@' (00)
        [9]     = '^@' (00)
 PR
 = (void) ⟨UNDEFINED⟩
```

The generic variable **$** is now a pointer to a 10-element character array. A pointer to an object is bounded by the object, hence stray references can be detected:

```
. $[9]
= (char) '^@' (00)
. $[10]
pointer/index out of range
STOP
. . *($ + 10)
pointer/index out of range
STOP
. . ⟨eof⟩
 = (void) ⟨UNDEFINED⟩
```

The result of an erroneous operation is always **⟨UNDEFINED⟩**. **⟨UNDEFINED⟩** is also the return value of a function that returns no value and the initial value of uninitialized automatic variables.

Linking to Compiled Functions

Compiled modules are manipulated similarly to interpreted ones. First the compiled file is read and a module is created:

```
si.$ !cc —c power.c    create a compiled file
si.$ r power.o    read the compiled file
"power" already loaded, type o to overwrite: o
loading "./power.o"
```

Since the interpreted version of **power** was still loaded, the Interpreter asked for confirmation to replace it with the compiled one. When a compiled module is loaded, user-specifiable libraries are searched to satisfy unresolved references. Thus the compiled module may contain functions that were not in the source:

```
si.$ f power    list the functions in power
power (in file ./power.o) compiled
        lmul( ) : (int)
        mul( ) : (int)
        power( ) : (int)
```

The compiled version of **power** can be executed just like the interpreted one:

```
si.$ x
. power(3,2)
= (int) 9
```

DISCUSSION

A few fundamental design decisions have been responsible for much of the nature of **si**.

Tools versus Environments

Innovation in programming tools continues [HEND84]. The current vogue is to weave these tools into an integrated environment to achieve conceptual cleanliness and added efficiency. Our goal with **si** is not so ambitious.

Without joining into the tools vs. environment fray, it is clear that **si** is a tool, not an environment; it can be used alongside a programmer's other tools. Making the decision to build a tool made other design decisions easy:

> *Should* **si** *contain its own editor?* No. There are real advantages to building an editor into an interpreter, such as immediate syntax checking and faster context switch between editing and execution. Nevertheless, editors live in our fingers and not our heads; everyone has their favorite editor. And if you like a particular syntax-directed editor, **si** won't stop you from using it.
>
> *Should* **si** *contain its own preprocessor?* No. C is evolving in a direction that encourages embedding **cpp** into the translator. On some systems, embedding **cpp** results in a nice performance gain. But the existence of special preprocessors has shown that having **cpp** as a separate program makes C more malleable. For instance, C++ [STRO83] was first implemented as a special preprocessor to C. The Safe C Runtime Analyzer [FEUE85] is a source-to-source transformer that runs after **cpp** and before the first pass of the C compiler.
>
> *Should* **si** *control the screen like a visual editor?* No. The Interpreter would be easier to use if the different kinds of output were directed to different places on the screen. Some development environments divide the screen into different windows, one showing the currently executing text, another showing the values of interesting variables, a third showing the output of the running program, and another used for commands to the system.
>
> In the implementation of **si**, we have tried to separate the user-interface from the Interpreter engine. To assist in building a clean interface, the different types of output from the Interpreter are sent to distinct streams. If the underlying system has the flexibility to attach those streams to different windows, then **si** can be given a multi-window front-end. The interface built into **si** is easily transportable across systems and does not compete with the program being interpreted.

One mode versus many

A central feature of some programming environments is that they operate in one mode; expressions in a universal language are always acceptable [DELI84, GOLD83, SAND78]. The traditional C development environment on the UNIX system embodies several modes and at least two languages: the Shell and C. With the Interpreter we had the opportunity to drop one of the languages, presumably the Shell, but chose instead to stick with the UNIX model.

Our feeling is that C does not make a wonderful command language. Since a command language is interactive, we prefer short commands with little extraneous punctuation. Typing

```
r("power.c")
```

instead of

```
r power.c
```

does not seem worth the conceptual purity.

In addition, the Shell has shown us the utility of I/O redirection and pattern matching in a command language. Since these are not features of C, using C as a command language would force us to give up the convenient and familiar Shell syntax for something bulkier.

On the other hand, C seems like the natural choice for the language to explore a C program. With this decision we leave ourselves slightly vulnerable because we have just argued that C does not make a good command language. Browsing surely requires a command language of some sort.

Our experience with debuggers that have opted for an interactive language instead of C convinces us that, in this context, it is worth the extra keystrokes to have the versatility of C. In providing all of the peeking and poking commands, debugging languages invariably become too complex. C has all the requisite power plus the advantage of familiarity.

Translation-time versus run-time checking

Standard practice when designing a compiler is to do as much checking at translation time as is possible, thus saving run-time overhead. We observe that during program development, more time is spent editing, compiling, assembling, and linking than running. By delaying some checks until run time, translation time is reduced.

In addition, during program development it is advantageous to have complete checking. For some important classes of errors, run-time checking is required for completeness. Since the Interpreter links on demand allowing pieces of a program to be executed in isolation, the referent for a function is not known (and may not exist!) until run time. Thus type checking of function calls across modules cannot be done at translation time. The use of a variable before it has been assigned a value, indexing outside the bounds of an array, and indirection through a stray or dangling pointer could each be detected in some cases at translation time. Complete checking for these errors, however, can only be done at run time.

CONCLUSION

si is a young program, it has been available outside of Catalytix for just five months. si continues to evolve based on feedback from its users. The most active areas of evolution deal with the user-interface and the performance of interpreted programs.

Within Catalytix, the Interpreter has become the translator of choice for developing new programs. It offers superior error checking and good tools for debugging at a tolerable cost in run-time performance. Our compilers have been relegated to what they do best, translating already debugged programs for fast execution.

REFERENCES

DELI84 Delisle, Norman M., David E. Menicosy, and Mayer D. Schwartz, "Viewing a Programming Environment as a Single Tool," *Proc. ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

FEUE85 Feuer, Alan R., "Introduction to the Safe C Runtime Analyzer," Catalytix Corp. Technical Report, January 1985.

GOLD83 Goldberg, A. J. and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

HEND84 Henderson, Peter (editor), *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* Pittsburgh, PA, April 1984.

KAY69 Kay, Alan Curtis, *The Reactive Engine*, Ph.D. Thesis, Department of Computer Science, University of Utah, 1969.

REIS84 Reiss, Steven P., "Graphical Program Development with PECAN Program Development Systems," *Proc. ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

SAND78 Sandewall, Erik, "Programming in an Interactive Environment: the 'Lisp' Experience," *Computing Surveys*, vol 10, no 1, March 1978.

STRO83 Stroustrup, B., "Adding Classes to C: An Exercise in Language Evolution," *Software—Practice and Experience*, vol 13, pp 139-61, 1983.

TEIT81 Teitelbaum, T. and T. Reps, "The Cornell program synthesizer: a syntax-directed programming environment," *Communications of the ACM*, vol 24, no 9, Sept. 1981.