Implementation of an Interactive Programming System

Mark Kahrs

Computer Science Department
University of Rochester
Rochester, NY 14627

Abstract: Design and implementation experience with an Interactive Programming System are described. Reflections are made on various design decisions and on the effect of using a Tree Factored Interpreter as the core of the system.

Key Words: Incremental Compilation, Tree Factored Interpreter, Interactive Programming Systems

1. Introduction

This paper is concerned with the creation of an efficient and flexible interactive programming system (IPS) designed along the lines of the system discussed by Mitchell [Mitchell, 1970].

For Mitchell, an IPS must exhibit interactive control, plasticity, contextual control and human interfacing. For example, an IPS might include an editor, a compiler (or interpreter) and a debugger. Most interactive programming systems draw a definite line between code that is interpreted and code that is compiled. For example, a LISP system such as Interlisp [Teitelman, 1978] has the ability to compile code into machine code. However, the decision to switch from interpretation to compilation is made by the user and not by the system. This paper is about a system that makes this decision automatically.

The system is centered around an interpreter that interprets parse trees. Execution is divided between walking the parse tree (executing semantic routines and calling code generators) and the actual runtime component of execution.

In fact, interpretation and compilation are looked upon as endpoints on a spectrum of execution. The IPS slides back and forth on this line depending on the "constancy" of the program. The more "constant" (i.e., unchanged or unedited) a program is, the more previously compiled code is run instead of creating and interpreting new code.

The design of the system is also concerned with the creation of a usable programming tool. The system includes a display editor. The paper is also concerned with the interaction between a programming system and the user.

2. Previous Work

This work is based on the system described in [Mitchell, 1970], which was discussed, but never implemented. Implementation of parts of his thesis can be found in an interactive ALGOL system for TSS/360 called LC^2 [Mitchell, et al, 1969] Swinehart [Swinehart, 1974] implemented the simpler parts of Mitchell's design but attacked a different area of interest: a man machine interface assisted by multiple processes and a display. Other work on ALGOL IPSes included the work of Lock [Lock, 1965], [Peccoud, 1968] and [Atkinson and McGregor, 1978]. Early work on interactive interpreters was done by the Joss group [Baker, 1966] at Rand as well as the BASIC group [Kemeny, 1966] at Dartmouth. [Braden and Wulf, 1968] wrote an incremental BASIC system for the B5500 that combined both interpreted and compiled code. Each of these systems did little to improve the runtime efficiency of the interpreter.

3. The Tree Factored Interpreters and Compilers (Part 1)

In an interpreter, an internal representation of the program (such as the source text) is parsed and executed immediately. If the internal representation is changed to the parse tree generated by parsing the text, then the interpreter becomes a Tree Interpreter or TI.

Semantic actions can be decomposed into two classes: semantic analysis and code generation (denoted S-action and X-action by Mitchell). Let a further distinction be that no S-action can perform any X-action and vice versa. That is, no semantic action can effect either program or data and no code generator can effect semantics. This separation is commonly found in compilers, but is uncommon in interpreters. A tree interpreter with separate S-actions and X-actions is called a Tree Factored Interpreter or TFI. If the parse tree is used to compile a program by following all the branches (not only those on the control path), then this is a Tree Factored Compiler or TFC. There are two classes of TFCs: If only the subtrees of the parse tree which were changed are compiled, then the TFC is called a Tree Factored Incremental Compiler (TFIC) or Tree Factored Partial Compiler (TFPC). Otherwise, the TFC is a Tree Factored Total Compiler (TFTC). Clearly, a TFTC is the same as a "standard" compiler.

Once a statement has been interpreted, the code can be saved and used again. However, if the semantics change, then the X-actions are invalid and the S-actions must be be reinterpreted. When this technique is extended to parse trees, then the following changes must be made: when the semantics associated with a parse tree node are changed then the code that is generated by that node must be changed. Both semantic information and code "float up" from leaf nodes to their parent nodes. In a TFPC, these nodes continue to bubble information up until the root node has been reached and all the code is ready for execution. A TFI does not need to have all the code ready for execution, just the code needed at the particular point on the parse tree.

What are the additional computational requirements of such a scheme? In terms of space, the system must keep information about which lines have been modified as well as keeping the parse tree present. The extra time involved in the TFI or TFPC is principally the time it takes to walk the parse tree. This overhead is minimized as the parse tree changes less and less.

4. The Editor and code constancy

The editor is one of the common ways the semantics of a program may be made invalid. If the user deletes, inserts or modifies a statement, the TFI must be made aware of the modification. Other ways of affecting the semantics of the program include user actions such as changing type declarations at an execution break and other actions. However, this paper is mainly concerned with modifications made with the editor.

Each line in the editor has a block of information that contains a bit about whether a lexical level change (such as those indicated by a BEGIN/END statement) is contained within the line. It also contains other flags that indicate whether the line is parsable from the beginning of that line. A line is parsable if the parser is able to restart the parse by starting the scan on the beginning of that line. This bit is used in the reparsing algorithm, which will be discussed in the parser section.

The editor knows nothing about the language it is editing. Both the scanner and the parser interact indirectly with the information block associated with each line, but the editor does not use this information.

5. The symbol table

The symbol table is another critical tool in determining whether a statement should be reinterpreted. If the type of a variable changes, it is the symbol table handler's responsibility to find which lines contains that variable and to invalidate the code associated with that line. This process is aided by a linked list of pointers to statements which contain the variable. This list is attached to each instance of a variable name in the symbol table. The implemented language is block structured and "modular" in the sense of Modula [Wirth, 1976] Therefore, the symbol table entry (bucket) for each symbol also contains fields for the current block pointer as well as a pointer to the current module.

6. The parser

The parser was implemented using production language. This decision was made for the following reasons:

- Production language parsers are relatively easy to implement
- Error recovery is easy to program
- It is possible to implement different parsing strategies
- It is easy to restart the parser
- It is possible to call semantic actions at parse time which can assist incremental parsing later.

Currently, the parser is implemented in a top down fashion.

Incremental parsing involves both the parser and the editor. As mentioned before, each line contains information on whether it contains a BEGIN in it as well as whether a statement starts this line. Reparsing takes place in the following manner: When a changed line is found, an attempt is made to find a line preceding it which can be used to start the parse. Clearly, there must always be such a line (in the worst case, the first line). Reparsing starts from that line. Before starting the parse however, it is necessary to restore the lexical environment. This can be easily done by backing up through the lines detecting BEGINs. (1) The new parse tree can be grafted onto the node that has the changed text.

Each line with a BEGIN/END has a lexical level associated with it as well as a pointer to the symbol table with a list of all identifiers declared at that level. If this block was lexically active, the pointer to the identifier list is put onto the lexical level stack and the search proceeds for the previous level (until the lexical level pointers are restored). Then the system can reparse and accommodate the changes. The system can stop reparsing if all of the following are true:

- No declarations have been detected
- No new BEGIN/ENDs were detected
- The current line is unchanged
- The parser is back up at statement level

Because of the extensive modifications that the parse tree may undergo as the result of the insertion or deletion of a lexical level change, it is necessary for the reparsing algorithm to restart parsing at the level where the new change was inserted. Note that insertion and or deletion is detectable because the system knows which lines are inserted or deleted as well as whether this statement has a BEGIN or END in it. However, the implemented system chooses to give up and restart parsing from the top line.

Incremental parsing in a more general context has been attempted by many others including [Lock, 1965] and [Lindstrom, 1970], but none have really succeeded. A restricted approach such as the one discussed here is a nice intermediary. Note that reparsing procedures could save the implementation complexity of searching for lines which begin a statement.

----

(1) Currently only a bit is used to signal that a BEGIN is contained in that line. This naturally restricts lines to have one BEGIN statement. This restriction could be removed by placing a linked list of lexical level pointers for a line with multiple BEGINs.

7. The Tree Factored Interpreters and Compilers (Part 2)

A TFI can operate between two endpoints of the "spectrum of compilation". Interpretation involves executing semantic routines along a path of control. If a line is changed under interpretation then the TFI will discover it during the parse tree walk which occurs with execution. Execution begins by calling the parse tree root node's semantic routine which calls a procedure (Mitchell called it PERFORM) that calls the TFI for the subnodes on the path of execution. If the lines associated with a parse tree node are unchanged, then the code associated with that node is called. If some lines where changed or if its semantics change, then the S-actions attached to the subnodes are called. Therefore, the semantic routine for a parse tree node assures that code found below it in the parse tree is correct.

In a TFPC, execution begins by calling the TFPC with the root node of the parse tree. The TFPC executes a depth first search of the parse tree. If the lines associated with a parse tree node are unchanged, then the code is left unchanged. On the other hand, if one of the lines was modified or its semantics changed, then the TFPC algorithm is applied to the node. Code is ready for execution when all the nodes in the parse tree have valid code attached to them.

The implemented system is both a TFI and TFPC. By setting a flag, it is possible to ignore calls to PERFORM. In this mode of operation the TFI is called for all of the subnodes of a node. This corresponds directly to a TFPC.

Note that since each node generates code and new code can be inserted at any point, the code linkages between nodes can become quite complicated. One way out of this problem is to generate the code for a parse tree node as a procedure. The ancestors of the node can then integrate the code by generating an indirect procedure call through a vector kept in the parse tree node. Changes to the program can be made by patching the new code's address into the address vector.

A reasonable question to ask at this point is how the system would detect that a node had been changed, given the fact that the user changes text lines and not parse tree nodes created by the parser. This problem is solved in the following manner: Each node has a first and last line pointer. Then the tree is being checked end-order, it is possible to examine all the subranges of a given node by processing the children of that node.

When a modified line is found (2), then that node is marked as being modified. That line is then reparsed, according to the reparsing algorithm described above. The TFI can then proceed with the processing. If on the other hand, if a type change has occurred that would effect the semantics of the code belonging to a line, then that line is marked for reinterpretation. When the TFI algorithm comes across that line, it will reinterpret the node that belongs to that line.

If program text is modified during a program break, then the new text might be inserted near the line parsed into the parse tree node that generated the code executing at the time of the break. The strategy to return from a break after such a modification is not simple. For this reason, Mitchell (pp. 4C3-7) used the following idea to re-establish the proper context: recompile up to the point at which the break occured, but never directly change a piece of code where the break occurred. If the code around a break point is changed, then switch into interpretation mode but keep the old code around to proceed from. Of course, after proceeding from the break, the old code can be replaced by the new code.

8. Writing Semantic routines

There are several considerations that must be followed in writing semantic routines for a TFI. If the user would like partial compilation (using a TFPC), then it may be possible for the compiler to skip over nodes on its way down to the node that has the changed lines. For this reason, the semantic routines must not depend on "inherited" attributes [Knuth, 1969] or must save the results so that the compiler can pick them up on its way down the tree. On the other hand, if the user uses a TFI, then saving semantics is not necessary because the interpreter will have the full state at any given node. Of course, "synthesized" attributes are perfectly acceptable to either scheme.

After calling PERFORM, the next step for a TFI semantic routine is to coerce the types of any arguments on the stack into their proper type. After that, code is generated. Lastly, the code is marked as being "complete" and passed to the parent node (in reality, pointers to the code are just passed). Before the code is passed up, it is executed.

----

(2) Note that this search can be shortened by bringing changes up the parse tree at the time the changes were made.

In order to separate the S actions from the X actions, Mitchell pointed out that FSL's [Feldman, 1964] notion of "code brackets" is useful. This syntax provides a mechanism to separate these two actions. Ideally, any compiler compiler or semantics language should provide such a facility for the semantic routine writer.

9. Debugger considerations

Satterthwaite [Satterthwaite, 1974] discusses source language debuggers at length. Interpreters have a great deal of useful information for a source language debugger. In Satterthwaite's system two interpreters were used to control tracing and maintain frequency counts. The X interpreter (not to be confused with the X actions mentioned above) executed machine code, the R interpreter simulated machine code execution and the E interpreter displayed and edited source text. Satterthwaite uses two markers called "alpha" and "beta" to synchronize the two interpreters. When a "alpha" is encountered, control switches from the X interpreter to the R interpreter. When a "beta" is encountered, control switches from the R interpreter to the X interpreter. These markers are explicity introduced for conditional statements. In a TFI, the X interpreter and R interpreter may still be desired for tracing the execution of programs. The facilities of the E interpreter may be easily provided since the source text is still present at debug time.

Note that it is easy to find the point at which a program break occurs because the range of code can be found in the parse tree nodes. Therefore, given a code address, it suffices to find the node in the parse tree node that generated the code and then print out the text line which generated that node. However, this would entail a search of the entire parse tree. Therefore, one stategy is to place a back pointer to the parse tree node in front of the code generated by that node.

10. An example of the operation of the TFI

In order to illustrate the operation of a TFI, a factorial program with two bugs introduced is used as an example.

```
[1]  procedure Factorial
[2]   (n: integer);
[3]   begin
[4]         var result,i: integer;
[5]         while i > 0 do
[6]         begin
[7]               result := result*i;
[8]               i := i
[9]               + 1;
[10]        end;
[11]        Return(result);
[12]  end;
```

Text lines are denoted by [n], where n is the line number. Note that line 8 has been split across two lines. The parse tree constructed from this program is shown below. The pointers to the lines are shown as well.

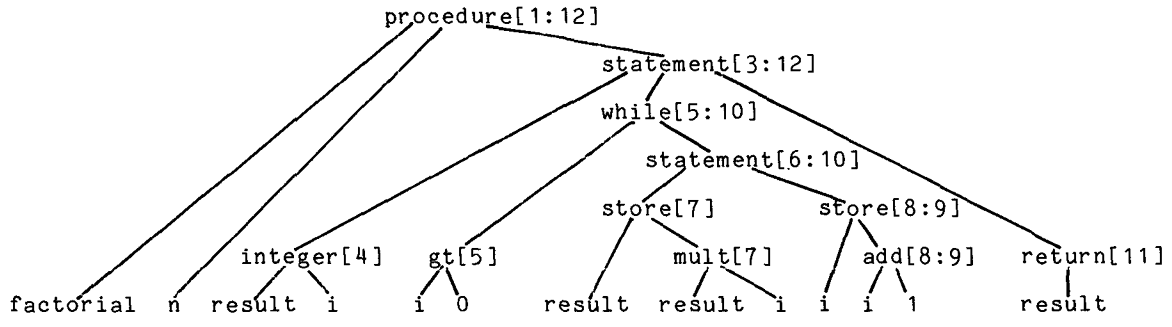

Figure 1

The code can now be executed and the walk of the parse tree continues.

Since i will increase, clearly the program as constructed will never terminate. In order to correct this, the user will change statement 9 from i + 1 to i - 1. Note that the correction comes on a line that is impossible to cleanly restart the parse. When the editor modifies line 9, the line is marked invalid. The parse tree is left in its present state until an attempt is made to run the program.

When this happens, then under partial compilation, the parse tree is walked in end-order, checking the range of lines associated with each statement node. If any of the lines have been modified then the algorithm checks to see whether that line can be used to restart the parse. In the case of line 9, this is not the case, because + 1 couldn't possibly start a statement. Therefore, the previous line is checked. Fortunately, the TFI need not look any further, the parse can be started from that line. A new tree is constructed and is grafted onto the existing tree. However, another bug still lurks. The variable i is never initialized to n. So, the user would insert a line (Call it [4.5]) that sets i to be n. This new tree is also grafted in and then the tree would look as follows: (the new branches are shown with doubled lines)

11. The state of the implementation

The system described here is implemented in BCPL [Richards, 1969] on a 16 bit minicomputer. Versions of the parser, scanner, symbol table routines, a display editor and the TFI algorithm are working. Currently, a formal semantic language known as ISLE is being developed. A grammar exists and the semantic actions are being developed. In addition to FSL's primitives, ISLE will provide the notion of separation between abstraction and implementation as discussed in the language ALPHARD. [Wulf, 1978]

12. Conclusion

Since semantic analysis is separated from parsing, parse time "semantic actions" can be used to help the interactive system. In particular, this was used to implement the incremental parsing hooks as well as assists for the editor.

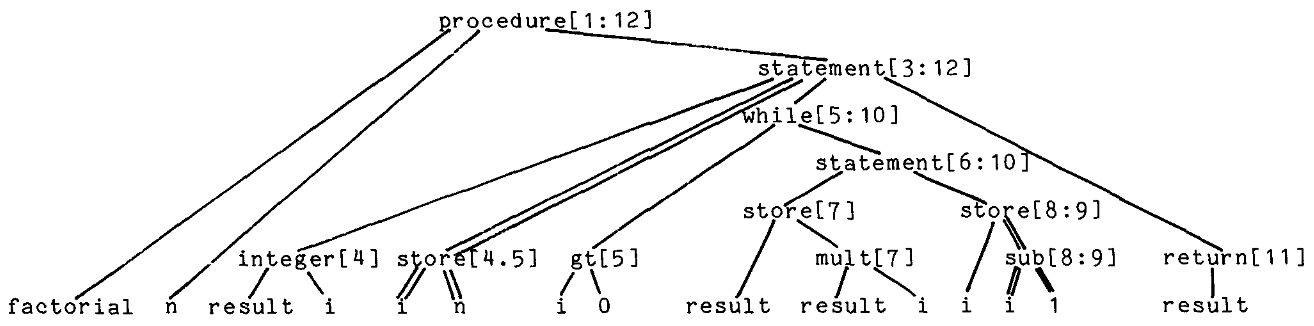

Figure 2

A clear advantage of the TFI over other organizations is the fact that the user gets the advantages of an interpreter and a compiler together in one integrated system. This means that the system writer can use the extensive information present in a interpreter to write a useful editor and a helpful debugger.

As Swinehart pointed out (pp. 162), it may be necessary for a TFI interpreter to intervene as often as every statement. This is not the case with a TFI that operates as an incremental compiler.

Of course, it should be clear that one must pay a price for the combination of an interpreter and a compiler. The most substantial cost is that of space. In the implemented system each text line occupies at least 7 words and each parse trepies at least 7 words and each parse tree node has at least 3 words of overhead. For a program of any size, this can be substantial. Another expense is the space used by the dependency lists. For a program with many variables and variable references, this begins to occupy space.

Another problem is that generated code may become quite disjointed. Because of this, code optimization across statements may be difficult. It is possible to get around this problem by coalescing the code through copying code attached to subnodes.

The purpose of this work is to build a flexible and efficient interpreter as well as to explore the interface between interactive languages and line editors. The system is flexible because it allows portions of the program to be incomplete. Certain statements will be compiled while others will be interpreted. The efficiency of the TFI (as discussed in the TFI section above) is slightly below compiling, but far above interpretation.

13. Acknowledgements

Peter Deutsch of Xerox PARC suggested the implementation of a TFI as an interesting topic to investigate. Peter also provided advice at nearly every stage. Dan Swinehart, also of Xerox PARC, tolerated my questions about both his and Mitchell's thesis as well as read several versions of this paper. Many people have read this paper in various forms and at various times. I thank them all for their help. Of course, I am solely responsible for the content. This research was supported through the generosity of the Xerox PARC Computer Science Laboratory. Preparation of this report was sponsored in part by the NSF and the Sloan Foundation.

14. Bibliography

1. Atkinson, L.V. and McGregor, J.J. "CONA - A Conversational ALGOL System" Software Practice and Experience, vol.8, 699-708 (1978)

2. Baker, C.L., "Joss: Introduction to a Helpful assistant", Rand memo RM-5058-PR, July 1966

3. Braden, B. and Wulf, W., "The implementation of BASIC in a Multiprogramming environment", CACM, v.11, no.10, pp. 88-692

4. Feldman, J.A., "A formal semantics for computer oriented languages", PhD thesis, Carnegie Institute of Technology, May 1964

5. Kemeny, J. BASIC manual, Dartmouth College, 1966

6. Knuth, D.E., "Semantics of Context Free Languages", Mathematical System Theory, v.2 no.2, 1967, pp. 127-145

7. Lindstom, G. "Variability in Programming Languages", PhD thesis, Carnegie Mellon University, 1970

8. Lock, K., "Structuring programs for multiprogram time-sharing on-line applications", AFIPS FJCC 1965, pp. 457-472

9. Mitchell, J.M., "The design and construction of flexible and efficient interactive programming systems", PhD thesis, Carnegie Mellon University, 1970

10. Mitchell, J., Perlis, A.J., Van Zoeren, H.V., "LC^2 - A language for interactive computing", in "Interactive Systems for Experimental Mathematics"

11. Peccoud, M. et al, "Incremental Interactive Compilation", IFIP 1968, pp. B33-B37

12. Richards, M., "BCPL - A language for systems programming", AFIPS SJCC 1969, pp. 557-566

13. Satterthwaite, E.H., "Source language debugging", PhD thesis Stanford University, 1974

14. Swinehart, D.C., "Copilot: A multiple-process approach to Interactive Programing Systems", PhD thesis, Stanford University, 1974

15. Teitelman, W., ed., Interlisp reference manual, Xerox PARC report, 1978

16. Wirth, N., "Modula: a language for modular multiprogramming", Software Practice and Experience, vol.7 no.1, 1977

17. Wulf, W., ed., "An informal Definition of Alphard (preliminary)", Carnegie-Mellon University Computer Science Report CMU-CS-78-105