

SEMANTIC CODE ANALYSIS

Walter G. Wilson
Chester C. John

IBM Data Systems Division
Processor Architecture and System Structure
P. O. Box 390
Poughkeepsie, N. Y. 12602

ABSTRACT

Logic programming is applied to code analysis in a large IBM development center. It allows general queries about unmodified assembler programs, including data and control flow analysis. It saved weeks of time over the span of a few months.

Unmodified assembler listings are converted into PROLOG clausal notation. The operational semantics of the code is captured partly by analysis rules and partly by interactive dialog.

Some implicit relations, such as control flow, are made explicit, for efficiency in subsequent queries. These queries are not restricted. They may encompass anything that is provable from the axioms representing the assembler code.

The system proved its potential as substantial foundation for software development assistance.

I. Introduction

Logic programming is applied to code analysis in a large IBM programming development center. The MVS design group, specifically the Subsystem Design/Experimental Laboratory, wanted to prototype the IMS Cross Memory Local Storage Option. The effort was successful. The IMS Development group in Santa Teresa, California subsequently developed the product. This was announced as part of IMS 1.2.

Multiple Virtual memory System is IBM's largest and most advanced operating system. It is intended primarily for the largest IBM processors. See for example [1]. Information Management System is IBM's most popular data base management system. See for example [2].

We were presented with many very large and complex assembler language routines. The programmers had to understand these and modify them to work with a new 'cross memory'

architecture. The routines typically had many tens of thousands of instructions.

Time was of the essence. There was only a small window in time during which the IMS product plan could be changed to include the desired function. A successful prototype was critical to demonstrating the feasibility and degree of effort required to change the IMS product.

This need for a quick and effective tool was one of the primary factors influencing the development of SCAN. The goal was to get IMS prototype running, not to develop a new tool. Time could be spent on the tool only to the extent it facilitated the prototype. Consequently, the human factors of the tool would have to be considerably changed in order to make it available to users not having a good understanding of how the tool was constructed. We do not view this as a defect in the tool, since that was not its purpose.

We needed a tool which would be able to locate patterns and otherwise aid in analysis of the assembler code. It seemed that the PROLOG language might be fruitfully applied to the problem. We had Grant Roberts PROLOG interpreter on loan from the University of Waterloo for experimental purposes.[3] A pilot project was conceived to test the feasibility of applying logic programming to a 'real-world' problem. It was named Semantic Code Analysis (SCAN), to reflect the fact that it could reason about the meaning of programs. Due to the nature of logic programs, its understanding could be increased by the simple process of adding rules to the SCAN database.

The system extracts the information implicit in assembler output listings, and places it in relational form in a PROLOG database. It then performs interval analysis on the extracted information. All of this is placed in the database for querying. Queries may be expressed in the so-called Horn-clause form of first-order logical expressions.[4]

The system proved its potential as a substantial foundation for software development assistance.

II. Semantic Code Analysis

II.A. Objectives of the System

Semantic Code Analysis (SCAN) is an interactive aid to program understanding and analysis. It understands such things as control and data flow, and the effects of operations on machine states. It is intended to be used by people modifying S370 assembler language programs[5], although it is extensible to handle any language or combination of languages.

It is based on logic programming. The information contained in the assembler output listing of a program is converted into a PROLOG database. A flow graph at all levels of detail is deduced from the statements. That information becomes part of the database. PROLOG can then be used to interactively query this data. PROLOG is more than a query language. It can reason and apply rules which capture 'system wisdom'.

Semantically significant conditions such as using register 0 as a base register can be simply and easily expressed in the PROLOG language. (In the S/370 architecture, register 0 is treated as always containing 0 when used as a base in computing addresses.) A given query which finds a set of instructions (e. g. a path) can be easily modified to select that subset which satisfies such semantically significant conditions.

Rules for this sort of qualification of a given query (read 'program') in PROLOG require only the addition of a few clauses. This ability to further qualify existing predicates is one of the features that makes programming in PROLOG so easy.

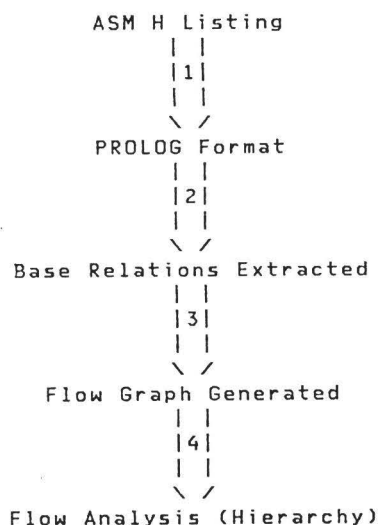


Figure 1. The analysis procedure

II.B. The Analysis Procedure

1. The Assembler (H) output listing is put in a format which can be read by Waterloo PROLOG. A small PL/I routine simply quotes all the strings and forms them into a well-formed PROLOG term.
2. The listings are then 'digested' and the information placed in the database. A PROLOG program is provided which does this.
3. The base relations are then analyzed to make some implicit relations explicit. This is for reasons of efficiency. The control flow graph was found to be a frequently used implicit relation. Therefore it is made explicit. The flow analysis algorithms we use are taken from [6].

Other useful formulations are available. For example see [7].

4. When higher levels of abstraction are desired, the flow analysis is repeated on each succeeding level, until an irreducible form is obtained. Again we follow Hecht.

The base relations include assertions defining the statements and links among them. For some instructions (e.g. ADD), the LINK relationships are clear: one LINK to the next sequential instruction is generated. Also, for conditional branches to internal labels it is clear that two LINKS are needed -- one to the label and one to the next sequential instruction.

There are other situations where it is not so clear where the LINKs should go. For some of these cases we were able to add PROLOG rules which embody some conventions. For example:

- unconditional branch on register 15 is the standard convention for leaving these routines
- Supervisor call 13 (SVC 13 -- Abnormal end) is considered an EXIT.
- Branch And Link Register (BALR 14,15) is a subrouting call, and no link is generated.
- SVC 10 (Get main storage) is also considered a subroutine call, and no link is generated.

There are other things which cannot be deduced from the code. For example, user defined supervisor calls may be exits or subroutine calls. This makes a difference in whether a LINK tuple is generated for that statement. In cases

like this, the user can add a rule defining the semantics. This avoids annoying the user with useless repetition of the same question.

We could not simply assert the rule automatically after the first answer, because the effect of the Supervisor call could be different depending on its input.

The PROLOG code which does this may be of interest. The pertinent part of GETER (which is the main routine) is:

```
GETER(*QRS)<-
  SUBSTMT(*A,*OP,*OPR,*B,*C,*)&
  .
  .
  .
  STRING(*B,*Q.*R.*S.*T.NIL)&
  ((EQ(*Q,'0')& /*test for 0X ops*/
  ((EQ(*R,'5')&BALR(*A,*OPR,*B))|
  (EQ(*R,'6')&BCTR(*A,*OPR,*B))|
  (EQ(*R,'7')&BCR(*A,*OP,*OPR,*B))|
  (EQ(*R,'A')&SVC(*A,*OP,*OPR,*B))))|
  (EQ(*Q,'4')&
  .
  .
  .
```

When the opcode is seen to be an SVC (0Axx), the rules for handling SVCs are invoked. There are special rules for known SVCs, such as 0A0D (Abnormal Termination) -- which is asserted to be an EXIT:

```
SVC(*A,**,*,'0A0D')<- ADDAX(EXIT(*A))&
  /(SVC(**,*,,*))&FAIL.
SVC(*A,**,*,'0A07')<- ADDAX(EXIT(*A))&
  /(SVC(**,*,,*))&FAIL.
SVC(*A,**,*,'0A7A')<-
  /(SVC(**,*,,*))&FAIL.
SVC(*A,**,*B)<- WRITERR(SVC,3,*A) &
  GENA(*A).
```

The last rule applies if none of the others do. It tells the user that an unknown SVC has been encountered, and asks what to do (this is done by the subroutine GENA):

It is here that additional rules could be added specifying the semantics for other SVCs that the user knows about. A simple ADDSVC(number, exit or subroutine or link) function would be provided the general user would not have to know this internal representation. This would most likely be added as a subfunction of GENA.

Other means of building or adding to the database may of course be used. A proposed design could be entered by hand; or our short (50 line) PROLOG analysis routine extended for PL/I or COBOL.

Assembler H Listing

```
000061C 07FF          1466+LABEL BR 15   GO TO ROUTIN  01-RESUM
000126C 4740 608A 3438      BM PXCT010 ENQ ECB  73142280
```

PROLOG Format

```
STMT(DFSREP00,S,1466,+, 'LABEL',BR,15,
  'GO TO ROUTINE','01-RESUM',
  00061C,
  '07FF',' ',' ').
STMT(DFSREP00,S,3848, , ' ',BM,PXCT010,
  'YES-GO ENQ ECB','73142280',
  00126C,
  '4740','608A',' ').
```

Base Relations

```
EXIT(1466).
LINK(3848,3822). LINK(3848,3849).
```

Flow Graph

```
NODE(115). GLINK(115,293). NODEND(115,1466).
NODE(245). NODE(246).
GLINK(245,222). GLINK(245,246).
NODEND(245,3849). NODESTART(246,3848).
```

Figure 2. Examples of data formats

II.C. The Interval Analysis Algorithm

PROLOG allows an easy, straightforward expression of Hecht's interval analysis algorithm. In his book, Hecht first expresses the definition of an interval in a semi-declarative form. He then spends considerable time converting this into a PASCAL program. Much effort is taken to demonstrate that the procedural PASCAL version is equivalent to the declarative definition. The dual declarative/operational semantics of PROLOG enabled us to bypass this effort, and use his definition as the program.

II.D. Possible Queries

Queries are limited only by the imagination and skill of the user. An idea of the capabilities may be formed by noting the following list of some possible queries.

- What possible paths exist between two statements?
- What is the locking status at a specific point on a path?
- What registers are available for use at an instruction?
- Where is a variable referenced? Set?
- Does a certain code sequence occur in the module?
- What are the comments on statements branched to by a conditional branch based on a condition code which may be set by some instruction using one of a set of DC'd constants?

Queries include, but are not limited to, the full range of register usage, including EQUated synonyms. In addition, all other information implicit in the program listing is available as the subject of a suitable query.

Queries may be built up. Simpler ones, such as finding a path, may be used in more complex ones, such as finding locking status. Once they are constructed (and saved), they become available for future use, and for use by others.

Examples of PROLOG format for queries.

Here are two of the queries used, and their PROLOG realization.

1. Where is any symbol equated to 0 used as a base register?

We illustrate for those statements whose operands are of the form D(R), for any displacement D, and check to see if R is equated to 0. There are two other possible operand forms, each of which requires an alternate query.

Step 1 in Figure 1 makes the operand field of every statement

available as the 7th of 13 fields in the STMT relation. Since it is frequently used, we define an OP(statement#, operand) relation which simplifies extracting the operand. Traditional database literature terms this a 'view':

```
OP(*stmt,*op)          <-
STMT(*,*,*stmt,*,*,*,*op,...*,*).
```

We have defined the relation PARSE(string,tree) which parses the operand string and returns the parse tree. For example, operands of the form 'D(B)' are parsed into the tree in Figure 3.

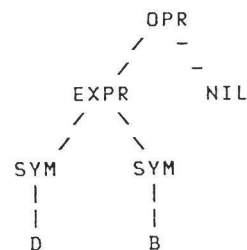


Figure 3. The Parse tree for D(B) So to pick out those operands of this form with 0 as base, we require the operand be parsed into this tree with the symbol B equated to 0.

This tree is an instance of

```
OPR(EXPR(*,SYM(*R)),NIL)
```

so the entire query to find 0 used as a base is

```
OP(*stmt,*op)
& PARSE(*op,OPR(EXPR(*,SYM(*r)),NIL))
& EQU(*r,0).
```

When successful, the variable *stmt is bound to a statement whose opcode parses into the looked-for tree, where the register symbol is equated to 0.

A symbol may be a constant, equated to a constant, equated to a symbol equated to a constant, ...

```
/*NUMERIC CONSTANT*/
EQU(*s1,*s1) <- INTEGER(*s1).
/*IMMEDIATE EQUATE*/
EQU(*s1,*s2) <-
  STMT(*,A,*,*,*s1,EQU,*s2,*,*).
/*CHAINED EQUATE*/
EQU(*s1,*s2) <- EQU(*s1,*s3) &
  EQU(*s3,*s2).
```

2. Where is a register used as a branch target base, but never set?

Define BRANCHTARGET(r,s), which holds when register r is the base of a branch target at statement s.

Define SET(r,s) which holds when register r is set at statement s.

Define SETONPATH(r,p), which holds when register r is set on path p.

We now have enough to define our primary query.

We assume we are interested in paths which are from a given entry point. NOTSET(ENTRY,R,S) holds when

- the statement number associated with label ENTRY is STRT,
- and there is a path from STRT to S,
- and register R is a branch target base at S,
- and register R is not set on any path from STRT to S.

```
NOTSET(*ENTRY,*R,*S) <-
    STMT(*S,*STRT,*ENTRY,*S) &
    PATH(*STRT,*S,*PATH1) &
    BRANCHTARGET(*R,*S) &
    NOT( PATH(*STRT,*S,*PATH2) &
        SETONPATH(*R,*PATH2) ).
```

III. SIZES

The SCAN system was quite small compared to 'normal' programs, moderately large compared to common PROLOG programs. Approximate sizes are:

- 300 PL/I statements for pre-processor quoting routine
- 200 PROLOG clauses to parse assembler operands
- 200 PROLOG clauses to extract base relations and perform flow analysis

Of more interest is the size of the data used to represent a program. This ranged typically from 5,000 - 10,000 unit PROLOG clauses per module for the base relations.

When running analysis or query, we used a 6Mbyte stack, which in Waterloo PROLOG is used to hold programs and data, and to perform computations. Even with this stack size we too frequently encountered 'STACK OVERFLOW' conditions due to the size of the data. Since computing a path required much stack space and was frequently done at several levels in a computation, we could have used more space for the computation.

The 'small' stack size was most restrictive on the number of programs we could deal with at once. To handle all of IMS we estimate would take about 500Mbytes. A stack of this size will be available on MVS/XA.

Waterloo PROLOG is theoretically inefficient in its look-up mechanism for clauses when there are large numbers of clauses with the same predicate symbol, which was the case here. Surprisingly, we found the performance quite satisfactory. A 'STACK OVERFLOW' condition could be reached in just a few (5-15) seconds when an infinite computation was being pursued.

IV. Cost/Effectiveness of a Programming Knowledge Base

Costs are expressed in terms of equivalent person-hour costs. Assuming a machine-to-person cost ratio of 400, gives us a cost-savings ratio of 16:1 over base analysis of a given module. Our experience tells us that this is an extremely conservative estimate for a more fully developed system. The ratio of machine-to-person costs will continue to drop, which will further improve the cost savings.

An improvement in the human interface to the query system will also improve the cost/benefit ratio. Natural language query would seem to be the obvious next step. See for example [8], [9], or [10].

V. Other related work

There is currently much work being done of the 'programmer's apprentice' variety. Systems like PSL/PSA[11] allow specifications to be written in an entity-attribute language. These supply the user with a predefined set of relations and reports, which are hard to extend but suffice for a large class of problems.

SCAN allows these relations and reports to be defined and easily extended, but does not force users into preconceived structures. A prepackaged set of definitions (ie. PSL/PSA provided on a Logic Programming base) which could be easily extended would be an interesting facility.

The aim of these assistants is to go from specifications to code. SCAN as implemented and used worked from the other direction -- from written code to higher constructs. Howard Holley's GRASP system[12] is remarkably similar to SCAN (even to the name of one of the components). GRASP offers a full screen and graphics user interface, but a much more restricted query capability.

Shapiro[13] has investigated a system which debugs or generates PROLOG programs from processing examples. It would be interesting to see how well his algorithms apply to our PROLOG representation of an assembler program.

VI. Conclusion

Logic programming has been successfully applied to a 'real world' problem. The advantages of a strong pattern matching and reasoning system have been amply demonstrated. The pilot project points the way to the tool which may very well provide the industry with a badly needed quantum improvement in productivity and quality.

References

- [1] Auslander, M., et al., "The Evolution of the MVS Operating System", IBM Journal of Research and Development, Vol. 5, No. 5, September 1981, 471-482.
- [2] McGee, W., 'The Information Management System IMS/VS', IBM Systems Journal, Vol. 16, 1977, 84-168.
- [3] Roberts, G., An Implementation of PROLOG, MS Thesis, Department of Computer Science, University of Waterloo, 1977.
- [4] Kowalski, R., Logic for Problem Solving, North-Holland, New York, 1979.
- [5] OS/VS Assembler Language Order No. GC33 4010 available through IBM branch offices.
- [6] Hecht, M., Flow Analysis of Computer Programs, North-Holland Computer Science Library, Programming Language Series, New York, 1977.
- [7] Holley, H., and B. Rosen, "Qualified data flow problems", IEEE Transactions on Software Engineering, Vol. SE-7, No. 1, 1981, 60-78.
- [8] Pereira, L., P. Sabatier and E. Oliveira, "Orbi: an Expert-System for Environmental Resource Evaluation through Natural Language," Proc. of the First International Logic Programming Conference, ADDP-GIA, Faculte des Sciences de Luminy, Marseille, France, September 1982, 200-209.
- [9] Pereira, F., Logic for Natural Language Analysis, Ph.D. Thesis, University of Edinburgh, Scotland, 1982.
- [10] Warren, D., "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," Proc. Seventh International Conference on Very Large Data Bases, Cannes, France, September 1981m 272-281,
- [11] Teichroew, D., and Hershey, E. "PSL/PSA: a computer-aided technique for structured documentation and analysis of information processing systems," IEEE Transactions on Software Engineering Vol. SE-3, No. 1, 1977, 101-108.
- [12] Holley, H., "GRASP, an analytical tool for systems programming", IBM Programming Productivity Tools Symposium, Santa Teresa, November 1982, 646-672.
- [13] Shapiro, E., Algorithmic Program Debugging, Ph.D. Dissertation, Yale University, 1982.