PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

Control of Heuristic Search in a PROLOG—based Microcode Synthesis Expert System

Michael D. Poe

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, Calif. 95014 USA

this work was performed at:

Digital Equipment Corporation
77 Reed Road
Hudson, Mass. 01749 USA

ABSTRACT

It is often suggested that the built-in search strategy of conventional PROLOG implementations makes difficult writing heuristic search-based programs. Experience from writing a large search-based program in PROLOG to perform microcode synthesis suggests that very intricate heuristics can be easily implemented.

1 INTRODUCTION

It is widely argued [3], [20] that the built-in search strategy of conventional PROLOG [2], [15] implementations makes artificial intelligence programming intrinsically inefficient or awkward. This paper discusses why this is not necessarily true, and argues that PROLOG's search mechanism can be considered to be an extremely convenient default. The discussion is based upon the author's experience implementing the expert system UMS (Universal Microcode Synthesizer) in over 15,000 source code lines of PROLOG [16]. The synthesizer consists, in approximately equal proportions, of a compiler, a global data and control flow analyzer, and a rule based code transformation system. This experience has found that PROLOG's built-in search strategy to be very convenient.

2 THE PROBLEM DOMAIN

One of the main ideas of UMS is that a procedural description of an instruction set (see [18] page 125) is a completely accurate, but inappropriately implemented, microprogram. The challenge of UMS is to apply program **TRANSFORMATIONS** to evolve this untargeted microprogram so that it can be used on the target microengine. Insight about which transformations should be applied where is obtained from analysis of the target hardware description (see [18] page 224 for the AM2901 based microengine description used to experiment with UMS). The hardware description for UMS many be at different levels of detail, as UMS accepts arbitrary hardware models. A more detailed hardware model allows for more cleverly produced microcode, but costs extra synthesis effort.

Previous work with the MIXER system [17] has shown that a manually guided transformation based system can produce microcode. MIXER uses three types of transformation rules supplied by a user for a specific microengine, called macro-knowledge, semantic knowledge, and micro-knowledge that explicitly describes how to translate the source microprogram. UMS automatically discovers the same knowledge by inspection of the hardware description.

3 INFERENCE ENGINE IMPLEMENTATION

The synthesis problem discussed here is based upon a representation of programs in terms of **ARCS** representing data and control dependencies, and **NODES** representing operations upon data (see figure 1). This type of program representation is commonly used in the optimization phases of conventional compilers [1], [4]. The first two parts of UMS translate ISPS source code to the arc and node form before transformation begins. Both the instruction set description and the hardware description are compiled into arc and node form.

In this discussion, variables of the source code will be simply termed variables, and variables in the PROLOG implementation will be termed logical variables. The hardware nodes (clauses with head "hwnode") of figure 1 have 4 main parameters; first a node operation type, then a list of inputs, a list of outputs, and last an unique node number. Each input or output list consists of a sublist for each operand. Each of these sublists consists first of an operand name, second a list of attributes, and last a list of arc connections to other nodes. These connections are represented as lists of node numbers that are implicit pointers to other nodes. An operand name of "_", representing the anonymous logical variable of PROLOG, means a control flow arc. An operand name of "[]"

represents an unnamed **DIRECT LINK** to the other nodes which make up a source code fragment. An operand with any other name represents an arc to a source code data dependency involving a variable with the same name.

The main activity of the synthesizer is to take the arcs and nodes from a code fragment of the instruction set specification, and **MATCH** them to a portion of the hardware specification. If no appropriate match can be found, then the code fragment from the instruction set specification is transformed into something different, and the process repeated. The nodes are stored in the global database as PROLOG clauses. The matching process has two parts; matching the nodes and their interconnections (such as the add instruction being **MAPPED** to be performed on the ALU), and mapping the variables (such as the program counter being mapped to a specific internal register). When each of the instruction set nodes is completely mapped to a hardware node, then the code generation phase of microcode synthesis is complete.

Two different implementation techniques could be used in PROLOG to accomplish the pattern matching between two nodes. One technique would be to write an interpreter that reads two specified nodes from the data base, and performs a program of data access operations which check if node sections correspond. Alternatively, a skeleton of a node could be built which is unified with clauses from the data base, with further processing of any ununifiable details. The second approach is taken in the synthesizer, which invokes a search of clauses in the data base by a conventionally implemented PROLOG interpreter. The techniques used to heuristically guide this PROLOG based search will be the topic of the rest of this paper.

4 ARC SEARCH

The synthesizer operates upon one source code fragment of the instruction set specification at a time. Each of these nodes has the clause name "instnode" to represent its origin. UMS takes a list of the nodes of the code fragment, and creates a second list with the most common (such as ASSIGN) types of nodes occurring last. Thus the search order of the nodes are heuristically derived. The motivation for this heuristic is to prune away as much of the search tree as early as possible by initially basing the search on any unusual operations within the instruction set specification code.

In the new node order described by this second node list, the synthesizer creates a data structure similar to the instruction set node (see figure 2) called a **TEMPLATE**. This structure has a different clause name, "hwnode", to correspond to clauses of the global data base that represent the hardware description. The template has the same operation name, the same number of input and output sublists, and a logical variable replaces the node number.

During the activity of creating the template, a list of arcs is created. Each member of this **ARC LIST** has three components; the arc of the instruction set node, a list of logical variables of the corresponding "hwnode" template arc, and a list of descriptive information useful later in the synthesis process. Note that sections of this arc list still correspond to the original node ordering heuristic. When an arc is successfully matched with, or compared to, an arc of the hardware, the arc is said to be **VALIDATED**. Next, the arcs in the arc list are ordered for heuristic search by copying them into other lists called bins. Part of the **BOOKKEEPING** information within each arc records which bin that copy of the arc is placed into. The six bins are for control flow, simple direct links, previously mapped variables, previously unmapped variables, difficult to map variables of either type, and difficult direct links.

Those arcs that represent control flow input are placed into the **FIRST BIN**. Direct link arcs to other nodes go into the **SECOND BIN**, and a copy of the same arc goes into the sixth bin. The **THIRD BIN** contains arcs of variables that had previously been mapped to hardware registers, with a copy to the fifth bin. In the **FOURTH BIN** are those arcs that the synthesizer had not mapped to hardware, also with a copy to the fifth bin. Copies of both previously mapped and previously unmapped variable arcs are in the **FIFTH BIN**. The **SIXTH BIN** contains copies of the direct links. Finally, the contents of bins are concatenated in order to form an arc stream. Although this stream is implemented as a sequence of arcs, each element of the stream also describes the bin it originally comes from. Different code is used to process arcs from each bin. Note that if the synthesizer would process all the arcs in arc stream, then the arcs would be analyzed in the order shown in figure 3. This is one example of where PROLOG's built-in search strategy does not preclude heuristic search.

4.1 CONSERVATION OF COMPUTATIONAL EFFORT

The different bins represent different types of processing heuristics used to validate the use of that arc in synthesis. The order of the bins represents the amount of computational effort needed to process that arc. The bookkeeping information within each element of the arc stream becomes especially important in those arcs represented by more than one bin.

A common logical variable called a TAG, within each bookkeeping region of the duplicated arc pair, is set to an atom when the processing of the first duplicate of the arc stream is successful. If unsuccessful, then the logical variable remains unchanged. This indicates that a larger amount of computational effort is needed to validate the arc. Code processes an arc only when the tag is not an atom. Thus the tag can act as a message, communicated between one arc and its duplicate further down the arc list, to inhibit further and redundant processing of the arc. Note that the use of two attempts to search part of a search space has an effect similar to the PROLOG technique of Intelligent Backtracking [8], [9], [10], [11].

The bins described above were introduced in order of increasing computational effort. The first bin for control flow arcs requires a trivial amount of computational effort to validate. It is only necessary to note down the control dependencies, such as that a microword field needs to have a specific value.

The direct links between two nodes, as represented within the second bin, are usually very easy. However, when complex arithmetic or logical operation is to be performed on a simple ALU, intermediate values of the calculation need to be temporarily stored and later fetched. In such a case, the initial direct link would fail validation, and the tag would be set to an atom. Further processing would later continue upon this arc as it is represented in the sixth bin.

It is almost as easy with the contents of the third bin to match a previously mapped hardware variable with one from the instruction set. Such a match already exists as a historical reference for a previously synthesized code fragment.

Much more effort is needed in arcs of the fourth bin; those with variables had never previously been mapped. The characteristics of the variables must be checked for compatibility, such as matching bit width, read-only (constant ROM) or read-write (registers) use, number of cells in memory arrays, etc. Failure of validation for arcs from the third or fourth bin would also potentially trigger further and much more costly processing in the fifth bin.

The processing for arcs of the fifth bin will attempt to deal with problems in matching an instruction set variable with one of hardware. Often in the machine description, the representation of an individual value passing though a bus will be represented by chains of assignment statements. In other cases, a previously mapped variable will not be "readable" and "writeable" by hardware resources (such as both the ALU and the shifter) within the directed graph of hardware data flow. In this case, the current instruction set variable use, and any previous uses of that variable, will have to be reanalyzed and REMAPPED to meet this new set of simultaneous constraints. Furthermore, if some but not all of these remappings are possible, attempts will be made to use transformation rules to modify those instruction set arcs that cause the problems. The recursive nature of the remapping activity can potentially cause the disruption and reanalysis of all previous work, at a cost much greater than the original effort. Arcs in the sixth bin create intermediate variables for complex calculations, at great computational expense.

The processing of an arc stream is potentially a very expensive computational process, especially for those arcs in the fifth and sixth bins. At each step in the way, heuristic based cost functions analyze the success rate of the progress up to that point. The cost functions take into account the total number of each type of arc, the grand total of arcs, and the individual and collective success rate of these categories. If at any time the computational effort has not been successful enough relative to the amount of effort expended, then that the line of reasoning (or that part of the search space) is abandoned.

The bookkeeping information is also used to implement automatic commutativity of operands within the arc stream. This is based upon a table of operation types that describes which are commutative (addition, inclusive or exclusive or, etc.). When a template for these types of nodes is made, both operand orientations are put into the arc stream. The bookkeeping information then allows matching to be sequentially attempted on each. Commutativity further enriches the flexibility of search in this Prolog-based implementation (see figure 4). Automatic generation of temporary variables is also facilitated by this bookkeeping. Originally both commutivity and temporary variable creation were implemented as individual rules. However, for efficient synthesis of irregular hardware architectures, these were directly added to the search processing.

5 NODE SEARCH

The heuristics involved in matching a set of arcs within an instruction node set to those within a hardware node set have been described. Next, the choice involved in determining which hardware nodes to be involved in the matching processes will be described. The nodes of both the instruction set specification and the hardware specification are sequentially stored in source code order in PROLOG's global data base. If a template for a node was referred to by the variable name X, then a simple minded

approach to using these templates would be to simply call X from PROLOG, for all nodes, and process the arc list. This would cause exhaustive search.

To provide a large reduction of the search space, UMS tries to use in the search for hardware nodes any insights derivable from processing of the arc list. A match candidate node is specified one at a time, as needed, during the processing of the arc list. Specifically, direct links between nodes are often used to match the instruction set fragment nodes to the hardware. It is equally important to relate any experience gained from previous synthesis activity to this decision making process.

When processing begins on each arc (except those arcs from the first bin), if a hardware node is not yet associated with that arc, then one is fetched from the data base. A list of previously used hardware resources (such as a node from the ALU) is kept in the global data base, and ordered as a stack. The use of a stack is based upon the assumption that the hardware resources needed for the current synthesis may be most similar to the synthesis just completed. To fetch a hardware node, the hardware resource stack is scanned for a node of the appropriate operation. If such a node is not found, then the global data base is searched in its source code order. Figure 5 shows how this search heuristic overrides the sequential search order in a PROLOG data base. Note that the use of this hardware use stack produces microcode with a much more vertical style. Other work [7], [12], [13], [14], [19] addresses the issue of optimization for horizontal micro-engines.

6 CONCLUSION

In conclusion, although the top-down, left to right search strategy of PROLOG is always present as a default, it is possible to write heuristic searches with radically different search patterns. An existence proof of the success of these programming techniques has been developed in the implementation of an expert system for microcode synthesis. This implementation did not require the implementation of a separate interpreter, but only specialized code to fit the problem domain. The results from this work are consistent with other efforts [5], [6], which have shown PROLOG to be as efficient as other Artificial Intelligence programming languages.

7 REFERENCES

[1] Aho, A. V., Ullman, J. D., Principles of Compiler Design, 1-Jan-77, Addison-Wesley, Pages 1 to 604.

[2] Clark, K. L., Tarnlund, S. A., Logic Programming, 1-Jan-82, Academic Press, Pages 1 to 366.

[3] Feigenbaum, E. A., McCorduck, P., The Fifth Generation Artificial Intelligence and Japan's Computer Challenge to the World, 1-Jan-83, Addison-Wesley.

[4] Gries, D., Compiler Construction for Digital Computers, 1-Jan-71, John Wiley and Sons, Pages 1 to 493.

[5] Mizogushi, F., PROLOG Based Expert Systems, 1-Sep-83, New Generation Computing, Pages 99 to 104, Vol. 1, #1.

[6] O'Keefe, R. A., PROLOG Compared to LISP?, 1-May-83, SIGPLAN Notices, Pages 46 to 56, Vol. 18, #5.

[7] Patterson, D., Goodell, R., Poe, M. D., Steely, S. G., V-Compiler: A Next-generation Tool for Microprogramming, 4-May-81, AFIPS Conference Proceedings, Pages 103 to 110.

[8] Pereira, L. M., Backtracking Intelligently in And/or Trees, 1-Jun-79, Technical Report Universidade Nova de Lisboa, Pages 1 to 25.

[9] Pereira, L. M., Porto, A., Selective Backtracking, 1-Jan-82, In: Logic Programming, Clark and Tarnlund, Editors.

[10] Pereira, L. M., Porto, A., Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Implementation, 1-Dec-79, Technical Report Universidade Nova de Lisboa, Pages 1 to 42, #13.

[11] Pereira, L. M., Porto, A., Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Theory, 1-Oct-79, Technical Report Universidade Nova de Lisboa, Pages 1 to 66, #2.

[12] Poe, M. D., Heuristics for the Global Optimization of Microprograms, 30-Nov-80, 13th Annual Workshop on Microprogramming, Pages 13 to 22.

[13] Poe, M. D., Goodell, R., Steely, S., Issues of the Design of a Low Level Microprogramming Language for Global Microcode Compaction, 12-Oct-81, 14th Annual Workshop on Microprogramming, Pages 88 to 94.

[14] Poe, M. D., Measurement and Manipulation of Potential Parallelism in Microcode, 8-Sep-81, 7th Euromicro Symposium on Microprocessing and Microprogramming, Pages 341 to 350.

[15] Poe, M. D., Nasr, R., Potter, J., Slinn, J., A KWIC (Key Word In Context) Bibliography on PROLOG and Logic Programming, 1984, Journal of Logic Programming (in press).

[16] Poe, M. D. Practical Synthesis of Microcode, forthcoming Ph. D. Dissertation, University of Massachusetts at Amherst.

[17] Shimizu, T., and Sakamura, K., MIXER: An Expert System for Microprogramming, 11-Oct-83, 16th Annual Workshop on Microprogramming, Pages 168 to 178.

[18] Siewiorek, D. P., Bell, C. G., and Newell, A., Computer Structures Principles and Examples, 1982, McGraw-Hill.

[19] Vegdahl, S. R., Local Code Generation and Compaction in Optimizing Microcode Compilers, 1-Dec-82, Ph. D. Thesis, Carnegie-Mellon University.

[20] Verity, J. W., PROLOG vs. LISP, Datamation, 1-Jan-84, Pages 50 to 51.

ISPS source code:

```
var2 = var1 + 1;
```

corresponding PROLOG node notation:

```
hwnode('CONSTANT',
        [],
        [[1, [], [14]]],
        13).

hwnode('ADD',
        [
         [_, [['CHOICE', _, 'SUBRDEF']], [12]],
         ['var1', [], [6, 11]],
         [1, [], [13]]
        ],
        [[[], [], [15]]],
        14).

hwnode('ASSIGN',
        [
         [_, [['CHOICE', _, 'SUBRDEF']], [12]],
         [[], [['FunctionName', _, 'ADD']], [14]]
        ],
        [['var2', [['WIDTH', _, [0, 3]]], [23]]],
        15).
```

node structure:

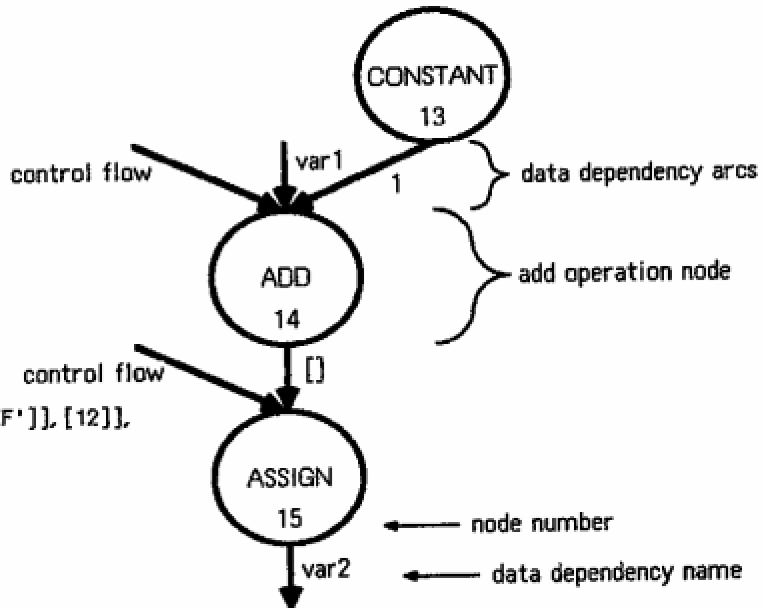

Figure 1: An ISP language statement and its corresponding representation as arcs and nodes.

PROLOG template:

```
hwnode('CONSTANT',
        [],
        [[_,_,_]],
        _).

hwnode('ADD',
        [
         [_,_,_],
         [_,_,_],
         [_,_,_]
        ],
        [[_,_,_]],
        _).

hwnode('ASSIGN',
        [
         [_,_,_],
         [_,_,_]
        ],
        [[_,_,_]],
        _).
```

node structure:

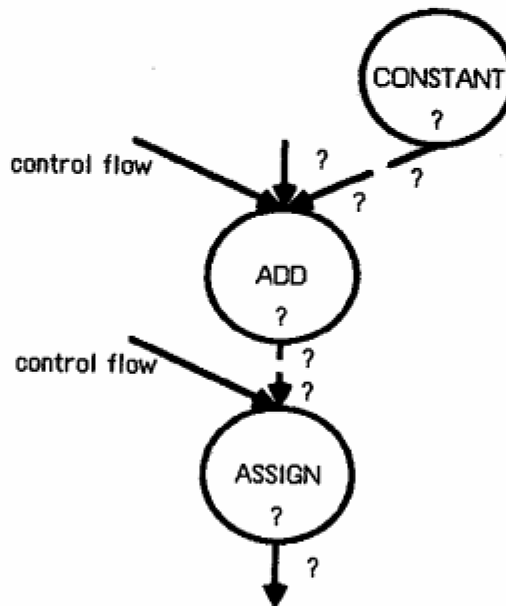

Figure 2: A template corresponding to the nodes of figure 1, and its arc and node representation.

```
                                                      Arc Processing Order
instnode('CONSTANT',
        [],
        [[1,[],[14]]],                                (7) (11)
        13).

instnode('ADD',
        [
         [_,[['CHOICE',_,'SUBRDEF']],[12]],           (1)
         ['var1',[],[6,11]],                          (5) (9)
         [1,[],[13]]                                  (6) (10)
        ],
        [[[],[],[15]]],                               (3) (13)
        14).

instnode('ASSIGN',
        [
         [_,[['CHOICE',_,'SUBRDEF']],[12]],           (2)
         [[],[['FunctionName',_,'ADD']],[14]]         (4) (14)
        ],
        [['var2',[['WIDTH',_,[0,3]]],[23]]],          (8) (12)
        15).
```

Figure 3: Potential order of arc processing.

```
                                                         Arc Processing Order

instnode('CONSTANT',
        [],
        [[1,[],[14]]],                                   (9) (15)
        13).

instnode('ADD',
        [
         [_,[['CHOICE',_,'SUBRDEF']],[12]],              (1)
         ['var1',[],[6,11]],                             (5) (8) (11) (14)
         [1,[],[13]]                                     (6) (7) (12) (13)
        ],
        [[[],[],[15]]],                                  (3) (17)
        14).

instnode('ASSIGN',
        [
         [_,[['CHOICE',_,'SUBRDEF']],[12]],              (2)
         [[],[['FunctionName',_,'ADD']],[14]]            (4) (18)
        ],
        [['var2',[['WIDTH',_,[0,3]]],[23]]],             (10) (16)
        15).
```

Figure 4: Potential order that arcs are processed with commutative ADD.

```
                                                         Arc Processing Order
instruction set fragment:

        instnode('ADD',
                [
                 [_,[['CHOICE',_,'SUBRDEF']],[19]],      (1)
                 ['var2',[],[127,297]],                  (8)
                 ['var3',[],[16]]                        (9)
                ],
                [[[],[],[24]]],                          (4)
                23).

        instnode('ADD',
                [
                 [_,[['CHOICE',_,'SUBRDEF']],[19]],      (2)
                 [[],[['FunctionName',_,'ADD']],[23]],   (5)
                 [1,[],[22]]                             (10)
                ],
                [[[],[],[25]]],                          (6)
                24).

        instnode('ASSIGN',
                [
                 [_,[['CHOICE',_,'SUBRDEF']],[19]],      (3)
                 [[],[['FunctionName',_,'ADD']],[24]]    (7)
                ],
                [['var4',[['WIDTH',_,[0,3]]],[29]]],     (11)
                25).

hardware resource usage stack:

        hwdatapath('ADD',722).
        hwdatapath('ADD',237).
        hwdatapath('ADD',296).

instruction set fragment source code:

        var4 = var2 + var3 + 1;

a sample of nodes in the order they
are found in the global data base:
                                        Node Search Order

        hwnode('ADD',_,_,7).            (5)
        hwnode('ADD',_,_,27).           (6)
        hwnode('ADD',_,_,58).           (7)
        hwnode('ADD',_,_,237).          (3)
        hwnode('ADD',_,_,296).          (4)
        hwnode('ADD',_,_,477).          (8)
        hwnode('ADD',_,_,385).          (9)
        hwnode('ADD',_,_,722).          (2)
        hwnode('ASSIGN',_,_,723).       (1)
        hwnode('ADD',_,_,801).          (10)
```

Figure 5: Node search order for ADD node.