User-Defined Predicates in OPS5: A Needed Language Extension for Financial Expert Systems

CUCS-496-89

Alexander J. Pasik
Department of Computer and Information Science
New Jersey Institute of Technology
Newark, New Jersey

Daniel P. Miranker
Department of Computer Science
University of Texas at Austin
Austin, Texas

Salvatore J. Stolfo
Department of Computer Science
Columbia University
New York, New York

Thomas Kresnicka
School of Business
Columbia University
New York, New York

Abstract

OPS5 is widely used for expert system development in industry as well as for academic research. Its limited expressive power, however, can lead to cumbersome and inefficient code. Often a single domain rule must be encoded as a series of OPS5 rules requiring extensive performance overhead and resulting in an awkward representation of the knowledge. In the financial expert system ALEXSYS, which performs mortgage pool allocation, the lack of user-defined predicates proved to be a major obstacle, prohibiting real time performance.

This work describes the addition of user-defined predicates in OPS5, supported by a patch to Carnegie-Mellon University's Common Lisp OPS5 implementation. Also, the necessity of this extension is demonstrated in the context of the ALEXSYS mortgage pool allocation expert system, both in terms of increased efficiency and improved knowledge representation.

Introduction

Since its introduction in 1981, OPS5 has become a popular language for building expert systems. OPS5 and its derivatives (such as c5 [Vesonder 1988] and OPS83 [Forgy 1985]) have been used both in academic production system research [Allen 1982, Barachini 1988, Laird *et al.* 1986, Miranker 1986, Pasik 1989, Scales 1986, Schor *et al.* 1986] and commercial expert system development [Gordin *et al.* 1988, Millikin *et al.* 1988, Vesonder *et al.* 1983]. This extensive use can be compared to the proliferation of FORTRAN programs: the language being the first to provide a specific functionality with adequate performance, OPS5 became widely used. However, like FORTRAN, OPS5 suffers because of its originating status; later production system research revealed the need for more powerful language constructs [van Biema 1986].

As rule-based expert systems are used more frequently in industry, certain domains reveal the specific needs for additional, more powerful language constructs. While building the financial expert system ALEXSYS for mortgage pool allocation, aspects about financial expert systems in general were revealed to require more complex numeric operations than available in OPS5. Particularly, OPS5 does not allow for arbitrary, user-defined tests on values in the left-hand-side of rules. Several derivatives of OPS5 have incorporated this language feature [Allen 1982, Forgy 1985, Giarratano 1988], attesting to the need of this facility in building rule-based programs. The work described herein serves two purposes:

1. to demonstrate the importance of user-defined predicates by showing the effects of their presence or absence on a commercial, financial expert system. These effects include performance and knowledge representational issues.

2. to provide a portable Common Lisp patch to Carnegie-Mellon University's Common Lisp OPS5 interpreter which extends OPS5 to include the facility of user-defined predicates.

The ALEXSYS Problem

The mortgage pool allocation problem is faced by financial companies which trade in mortgage pools. Each month, a set of transactions must be processed so as to provide

a maximum profit potential for the company. The decision making process determines the allocation of available mortgage pools to the contracts made in the previous month. The decisions, however, must be made within the constraints imposed by a set of federal regulations.

The volume and profitability of a trading floor is limited by the capacity of the allocators to advantageously fill sell orders during the final rush of the settlement days. An allocator traditionally operates with a calculator in one hand and the telephone in the other. As institutions handle ever-increasing volume, the allocators become hard-pressed to support the activity during the contract settlement hours, much in the same way as activity comes to a head during the closing minutes in the trading pits. Allocators carry the additional responsibilities of ensuring that inventories are delivered into contracts in legal amounts subject to complex rules set by a federal agency and ensuring that profit is made from the small variance allowed in how contracts are filled. Toward the end of a settlement day an allocator's primary concern is to deliver correct settlement information by telephoning counter-parties on overloaded telephone lines. There often is insufficient time to determine an optimal allocation for each contract.

Much of an allocators' expertise can be encoded into an expert system. As an interactive allocator's assistant, the expert system ALEXSYS can rapidly recalculate allocations as quickly as market conditions and inventory information is updated. ALEXSYS can enhance an allocators' performance by optimally allocating contracts, rapidly adjusting the allocations according to dynamic conditions and freeing the allocator to handle telephones and stipulated trades. ALEXSYS optimizes allocations for maximum profitability, maintains inventory under quality constraints, and reduces *fails*. Fails are the primary source of lost revenue in the allocation process. Fails can occur because of technical or clerical violations of the federal regulations. Fails also occur as a result of insufficient inventory due to short positions or as a domino effect of a fail by a counter party. ALEXSYS eliminates technical errors and promptly warns the allocators of uncovered positions. As a hedge against counterparty pool changes, allocators do not always use the entire variance on delivery. The rapidity with which the computer may reallocate contracts allows the allocators to exploit the full variance and capture this lost source of profit.

The federal regulations which control mortgage pool allocation take the form of rules which indicate what sorts of pool combinations are legal. For example,

If a $1,000,000 contract for mortgage pools of a coupon rate of less than 12% is to be filled, then no more than 3 pools can be used to fill the contract. Also, no two of the three pools can account for more than $975,000 of the contract.

During the development of ALEXSYS, the necessity of user-defined predicates was revealed. For example, the rule mentioned above cannot be expressed in a single production in standard OPS5. Rather, the rule should be able to be written encoding the information as follows:

```
(p fill-contract-with-three-pools
    (contract ^value 1000000 ^coupon < 0.12)
    (pool ^id <p1> ^amount <x1>)
    (pool ^id <p2> ^amount <x2>)
    (pool ^id <p3> ^amount <x3>)
    SUCH THAT
    (and (= (+ <x1> <x2> <x3>) 1000000)
         (< (+ <x1> <x2>) 975000)
         (< (+ <x1> <x3>) 975000)
         (< (+ <x2> <x3>) 975000))
    -->
    ...)
```

Because arbitrary tests cannot be performed on OPS5 left-hand-sides, the above process can only be performed by, in a first set of rule firings, computing the four sums in temporary working memory elements, one corresponding to each three pool combination in working memory. Then a second set of rule firings would select one of the combinations, and a third set of firings would remove the remaining combinations from memory. This three rule encoding artificially distributes the knowledge that is being encoded. Also, there is a large efficiency cost to this approach. It is hypothesized that this scenario is common among financial systems.

User-Defined Predicates

Based upon the ALEXSYS experience, the necessity of user-defined predicates in OPS5 was demonstrated. The lack of such functionality leads to not only awkward knowledge representation, but also unacceptable performance costs. Referring to the sample rule provided in the previous section, a valid vanilla OPS5 encoding follows:

```
(p fill-contract-with-three-pools-MAKE-SUMS
    (contract ^value 1000000 ^coupon < 0.12)
    (pool ^id <p1> ^amount <x1>)
    (pool ^id <p2> ^amount <x2>)
    (pool ^id <p3> ^amount <x3>)
    - (sums ^ids <p1> <p2> <p3>)
    -->
    (make sums ^ids <p1> <p2> <p3>    ; ids is a vector attr
         ^sum (compute <x1> + <x2> + <x3>)
         ^sum12 (compute <x1> + <x2>)
         ^sum13 (compute <x1> + <x3>)
         ^sum23 (compute <x2> + <x3>)))


(p fill-contract-with-three-pools-CHOOSE
    (contract ^value 1000000 ^coupon < 0.12)
    { (sums ^ids <p1> <p2> <p3>
         ^sum 1000000
         ^sum12 < 975000
         ^sum13 < 975000
         ^sum23 < 975000) <sums> }
    (pool ^id <p1> ^amount <x1>)
    (pool ^id <p2> ^amount <x2>)
    (pool ^id <p3> ^amount <x3>)
    -->
    (modify <sums> ^chosen t)
    ...)


(p fill-contract-with-three-pools-REMOVE
    { (sums ^chosen <> t) <sums> }
    -->
    (remove <sums>))
```

The rules are not complete, but the necessary aspects are represented. When there are a large number of pools (n), the first rule will match successfully against all combinations of pools (n^3) and generate as many working memory elements in successive firings. The second rule will match only against the few combinations which generated valid sums (potentially much less than n^3). Finally, the last rule serves to remove unwanted working memory elements.

By allowing user-defined predicates in OPS5, the rule can be written as follows:

```
(p fill-contract-with-three-pools
    (contract ^value 1000000 ^coupon < 0.12)
    (pool ^id <p1> ^amount <x1>)
    (pool ^id <p2> ^amount { <x2>
          (sum< 975000 <x1>) })
    (pool ^id <p3> ^amount { <x3>
          (sum< 975000 <x1>)
          (sum< 975000 <x2>)
          (sum= 1000000 <x1> <x2>) })
    -->
    ...)
```

In this rule, the user-defined predicates sum< and sum= are used. They are defined in Common Lisp as follows:

```
(defun sum< (wm-value amount &rest args)
    (and (numberp wm-value)
         (numberp amount)
         (every #'numberp args)
         (< (apply #'+ (cons wm-value args)) amount)))


(defun sum= (wm-value amount &rest args)
    (and (numberp wm-value)
         (numberp amount)
         (every #'numberp args)
         (= (apply #'+ (cons wm-value args)) amount)))
```

In defining user-defined predicates, the first argument is assumed to come from the working memory element being matched. Thus in calling the function from a rule, there is an implicit first argument coming from working memory. For example, in the previous rule, the call (sum< 975000 <x2>) in the third pool condition element would result in a call to the lisp function sum< with *wm-value* bound to the data being matched (that is, the value of the third pool's amount attribute), *amount* bound to 975000, and *args* bound to a list containing the remaining arguments (that is, a list containing the value that OPS5 bound to <x2>). Similarly, the call (sum= 1000000 <x1> <x2>) results in the lisp function call:

(sum= *value-of-amount-attribute* 1000000 *OPS5-value-of-<x1>* *OPS5-value-of-<x2>*)

This rule will fire once for each valid combination, selecting an appropriate set of pools which conform to the federal regulation encoded. The restrictions imposed by the additional predicates limit the amount of matching greatly, and thus result in improved efficiency.

The ability to use user-defined predicates is accomplished via a patch to the Common Lisp OPS5 interpreter. Calls to the user-defined predicates are compiled into the existing Rete pattern-matching network [Forgy 1982]. Thus, user-defined predicates fit within the algorithmic framework of the Rete algorithm. A similar approach can be used in other interpreters using alternative algorithms such as TREAT [Miranker 1986]. These algorithms share a similar mechanism for pattern matching based on the combination of database operations of selects and joins. The addition of user-defined predicates is accomplished at the level of allowing arbitrary tests for the selects and joins without modifying the underlying mechanism.

The performance improvements provided by user-defined predicates can be illustrated using the same example. As shown in figures 1, 2, and 3, the vanilla OPS5 version running with n pools executes in $O(n^3)$ number of cycles generating a maximum of $O(n^3)$ working memory elements. The run time of the system grows exponentially. However, the version with user-defined predicates executes in O(1) cycles, independent of the size of working memory. It uses working memory only to represent the actual pools, that is $O(n)$, and executes in $O(n)$ time.

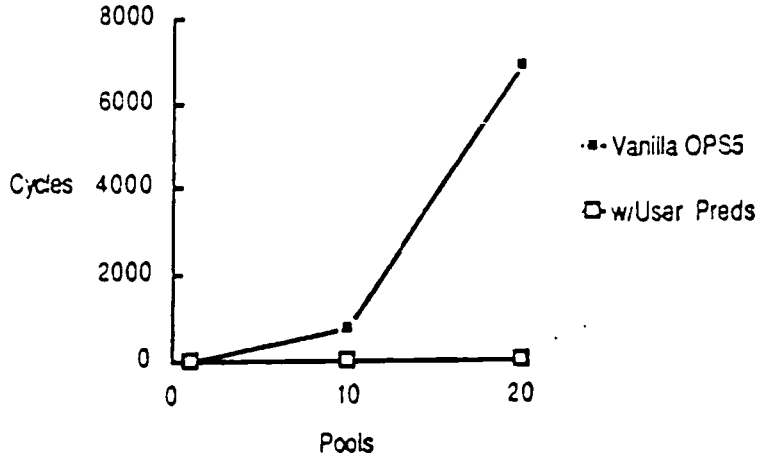

Figure 1: As the number of pools in working memory increases, number of cycles increases $O(n^3)$ in vanilla OPS5, but remains constant (1 cycle) with user-defined predicates.

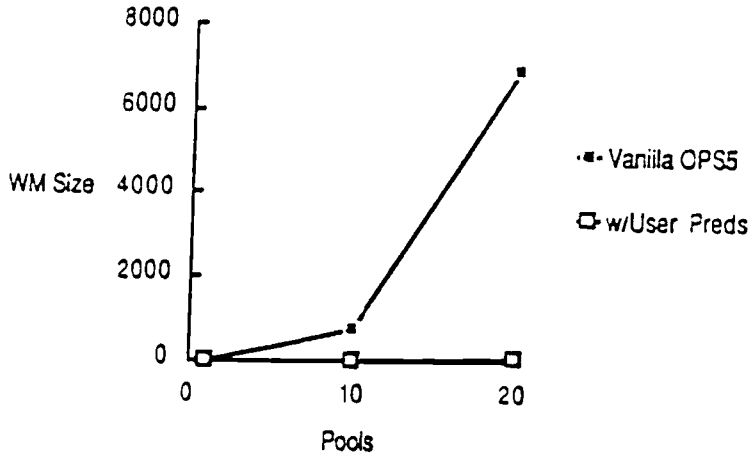

Figure 2: As the number of pools in working memory increases, the maximum working memory size increases $O(n^3)$ in vanilla OPS5, but remains linear in the number of pools with user-defined predicates.

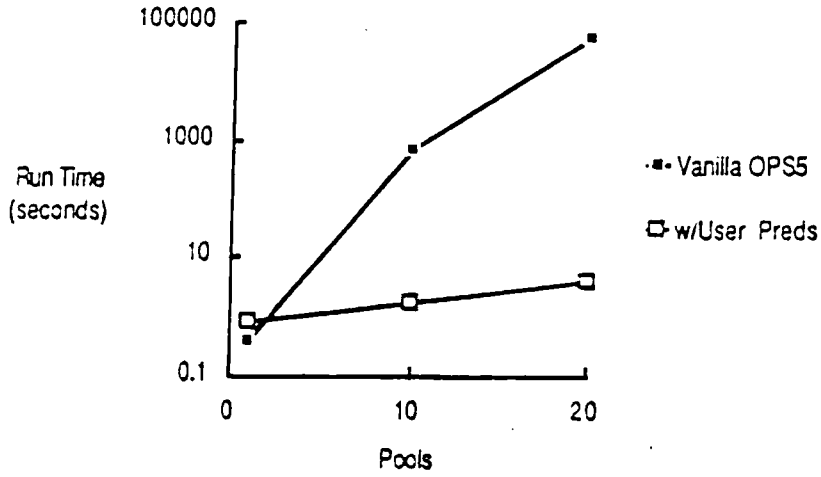

Figure 3: As the number of pools in working memory increases, the run time increases exponentially in vanilla OPS5, but remains linear in the number of pools with user-defined predicates. (Note: The plot is semilogarithmic.)

Conclusion

In writing expert systems which encode decision-making processes based on complex numerical data, user-defined predicates are an essential language feature for production systems. Financial expert systems such as ALEXSYS are such systems, and

the necessity of user-defined predicates in this system is demonstrated both from knowledge representational and performance standpoints.

User-defined predicates have been available in commercial expert system tools, but not in the freely available Common Lisp OPS5 which is used extensively throughout the expert system industry. Also, the effects on performance have not previously been demonstrated.

As a result of this work, a Common Lisp patch to the OPS5 interpreter, available through the Department of Computer Science at Columbia University, extends OPS5 with user-defined predicates. In addition, the enormous benefit in performance demonstrated herein provides incentive for the widespread use of the language extension in financial, real-time expert systems.

References

1. Allen E. (1982) YAPS: Yet Another Production System. Technical Report 1146, Department of Computer Science, University of Maryland.

2. Barachini F. (1988) *PAMELA: A Rule-Based AI Language for Process-Control Applications.* The First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, pages 860-867.

3. Forgy C.L. (1982) Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence 19(1)*: 17-37.

4. Forgy C.L. (1985) *OPS83 User's Manual and Report.* Production Systems Technologies.

5. Giarratano J.C. (1988) *CLIPS User's Guide.* NASA Cosmic Program Documents MSC-21208, MSC-21467, and MSC-21475.

6. Gordin D., Foxvog D., Rowland J., Surko P., and Vesonder G. (1988) *OKIES: A Troubleshooter in the Factory.* The First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, pages 24-28.

7. Laird J., Rosenbloom P., and Newell A. (1986) *Universal Subgoaling and Chunking.* Boston, Massachusetts: Kluwer Academic Publishers.

8. Milliken K.R., Finkel A.J., Klein D.A., and Waite N.B. (1988) *Adding Rule-based Techniques to Procedural Languages.* The First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, pages 185-195.

9. Miranker D.P. (1986) *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* Ph.D. Thesis, Department of Computer Science, Columbia University.

10. Pasik A.J. (1989) *A Methodology for Programming Production Systems and its Implications on Parallelism.* Ph.D. Thesis, Department of Computer Science, Columbia University.

11. Scales D. (1986) Efficient Matching Algorithms for the SOAR/OPS5 Production System. Technical Report, Knowledge Systems Laboratory, Computer Science Department, Stanford University.

12. Schor M.I., Daly T.P., Lee H.S., and Tibbitts B.R. (1986) *Advances in Rete Pattern Matching.* AAAI-86, pages 226-232.

13. van Biema M., Miranker D.P., and Stolfo S.J. (1986) *The Do-loop Considered Harmful in Production System Programming.* First International Conference on Expert Database Systems, pages 88-97.

14. Vesonder G. (1988) Rule-based Programming in the Unix System. *AT&T Technical Journal 67(1):* 69-80.

15. Vesonder G., Stolfo S.J., Zielinski J.E., Miller F.D., and Copp D.H. (1983) *ACE: An Expert System for Telephone Cable Maintenance.* IJCAI-83, pages 116-121.