Discovering Procedural Executions of Rule-Based Programs

David Gadbois and Daniel Miranker
University of Texas at Austin
Deptartment of Computer Sciences
Taylor Hall 2.124
Austin, TX 78712-1188
gadbois@cs.utexas.edu, miranker@cs.utexas.edu

Abstract

Executing production system programs involves directly or indirectly executing an interpretive match/select/act cycle. An optimal compilation of a production system program would generate code that requires no appeal to an interpreter to mediate control. However, while a great deal of implicit control information is available in rule-based programs, it is generally impossible to avoid deferring some decisions to run-time.

We introduce an approach that may resolve this problem. We propose an execution model that permits the execution of the usual cycle when necessary and otherwise executes procedural functions.

The system is best characterized as a rule rewrite system where rules are replaced with chunks, such that the chunks may have procedural components. Correctness for replacement of rules is derived by a constrained abstract evaluation of the entire program via a general-purpose theorem prover at compile time. The analysis gives a global dependency analysis of the interaction of each of the rules.

For a group of popular benchmark programs, we show that there is ample opportunity to automatically substitute interpretive pattern matching with procedural elements and a concomitant improvement in performance.

1 Introduction

Naive interpretation of production system programs involves a match/select/act cycle. The interpreter checks each rule's antecedent conditions to see if it is satisfied by the database, picks one of the satisfied rules to fire, and performs its consequent actions. The cycle repeats until no more rules are satisfied. Approaches to improving the performance of programs and knowledge bases written in rule based form encompass improved incremental match algorithms, such as TREAT and RETE, local optimizations such as sharing and join-optimization, as well as parallelism and learning (Forgy 1982; Miranker & Lofaso 1991; Ishida 1991; Gupta 1988; Miranker *et al.* 1990; Laird, Newell, & Rosenbloom 1986).

The current implementations are adequate for small-scale main-memory based production systems. However, for very large systems or ones in which access to the working memory is mediated by a database manager, the current compilation technology falls short (Stonebraker 1992). Direct implementation in terms of the match/select/act cycle can lead to grossly inefficient executions. To be specific:

- The execution cycle imposes a considerable overhead in maintaining global run-time data structures that may not be necessary while executing a particular fragment of a program.
- Performing some actions incrementally over a number of cycles, such as maintaining various lists in sorted order, often imposes considerable algorithmic overhead.
- The cycle does not take advantage of statically determinable relations among rules — much control information implicit in a program is constantly recomputed at run-time.
- The execution cycle imposes strict synchronization constraints that inhibit distributed mappings of production system execution.

We claim that an optimal compilation of a production system program would generate code that requires no appeal to the interpreter to mediate control. While this ultimate goal is probably impossible in the general case, we have developed techniques that allow for a substantial reduction in the amount of control mediation required. This approach makes plausible the goal of obtaining execution performance for declarative programs within the range of equivalent procedural ones.

While there has been some work on explicitly representing procedural control knowledge at a linguistic level in production system programs (Georgeff & Bonollo 1983) as well as automatically determining control information (Stolfo & Harrison 1979), we assert that within every declarative program is a procedural one waiting to be discovered, and that it is the job of a sufficiently smart compiler to extract that procedural program.

In this paper we introduce an approach that resolves this problem. We propose an execution model that permits the execution of the usual cycle when necessary, but otherwise executes specific, procedural control. The key is to arrange for the both control styles to leave the system in a consistent state for the other to take over.

Our system is best contrasted with the SOAR system (Laird, Newell, & Rosenbloom 1986). SOAR is a learning system that augments a program with new rules, called *chunks*. Chunks are synthesized from the existing rules of the program and are expressed in the original source language of the rule program. The output of the SOAR system may then be fed back to itself for ongoing improvement. We loosen the requirement of systemic learning: Our system replaces (rewrites) rules drawing from a language that includes procedural constructs. Consequently, our transformations provide a single compile-time performance improvement. It is, nevertheless, the case that our analysis can and should be reapplied to already transformed programs. The procedural constructs have explicit representations for binding points, search state, type information, and conditional and loop constructs as well as various compiler bookkeeping data.

Our technique represents a two-fold expansion of earlier work on the decomposition of rule programs into collections of smaller concurrently executable rule programs (Kuo, Miranker, & Browne 1991; Schmolze 1991; Ishida 1991; Kuo, Moldovan, & Cha 1990). First, we go beyond identifying weak mutual exclusion relationships by formalizing and identifying control flow precedence relationships. This is necessary to determine when rules may be replaced instead of simply adding new rules. We determine the necessary data relationships using an automatic theorem prover to determine the strongest possible conditions on data values. Second, the resulting dataflow graph is expanded to include additional relationships and to used to determine possibilities of transformation.

Our results to date include:

- identifying opportunities for exiting the match/-select/act cycle and replacing portions of the execution with procedural code.
- finding opportunities for "RETE style sharing" that go beyond simple common subexpression elimination.
- detecting, at compile time, rule instantiations that can be fired concurrently, but in previous work eluded parallel execution.
- using the above techniques to find and obtain significant performance improvements, both in terms of decreased cycle count and lower running time, in the execution of a suite of benchmark programs.

The remainder of this report is organized as follows. In Section 2, we describe the architecture of the STAR (STatic Analysis of Rules) system and the global analysis it performs. Section 3 covers three optimizing transformations for production system languages. Empirical evidence of the opportunities for transformation in standard OPS5 benchmark programs and the results of applying them are presented in Section 4. Finally, in Section 5 we summaries our results and suggest directions for further research.

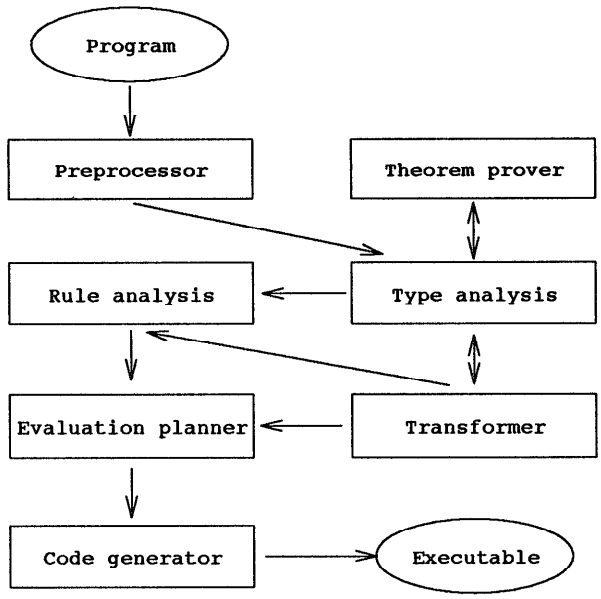

Figure 1: STAR system architecture

2 System Architecture

The global analysis approach presented here generalizes the one presented in (Ishida 1991). The STAR system parses a complete OPS5 (Forgy 1981) source program into an internal representation called the Abstract Rule Language (ARL.) For each condition element and working memory modifying action pair in the program, the system computes the set theoretic relationships between them. Given this information, the system can compute control relations between the program's rules. Using this global information, the system finds opportunities to replace groups of rules with procedural chunks. The process iterates until no more transformations apply. Figure 1 shows a block diagram of the system architecture.

Type relations

We consider the condition elements of the LHS of rules and the working memory modifying actions of their RHSs as first-order formulae, which we will call *types*, and define relations between them that can be seen as the set-theoretic relations of subsumption, intersection, and disjointness on their extensions. A working memory element belongs to a type (i.e., is a member of its extension) if the element satisfies the type formula.

While it is certainly possible to compute the type relations in an ad hoc manner, doing so is tricky, error-prone, and difficult to extend to new kinds of literals (for example, to those whose terms may contain references to function symbols, or even to completely different rule languages.) We have instead implemented the type analysis computation using an automated theorem prover. Propositions corresponding to assertions of the above relations may be constructed directly out of condition element and action pairs and can be passed to the prover for validation. The STAR system uses the Otter resolution-based prover (McCune 1990).

The precise type relations so obtained are of general utility. Computing the static clustering algorithm for parallel rule execution given in (Kuo, Miranker, & Browne 1991) using the subsumption relation instead of the syntactic relations used in the report yields as good, or, in some cases, much better results.

Rule relations

We wish to characterize the possible sequences of rule firings. Given that an instantiation for a particular rule has fired, we characterize the rules that may fire next with three overlapping relations. The relations are similar to those defined in (Kuo, Moldovan, & Cha 1990).

Given the relations between the types of the working memory modifying actions of one rule and those in the LHS of another, we can define useful control relationships between the two rules. We are chiefly interested in whether the results of firing one rule may or may not lead to new instantiation for other rules. There are three possibilities:

- Enables

  One rule enables another if the type of its head literal is subsumed by that of a literal in the body of the other rule. Results from evaluating the first rule will definitely feed into an evaluation of the second.

- Maybe enables

  One rule might enable another if the type of its head literal intersects with that of a literal in the body of another rule. Results from evaluating the first rule may lead the second rule to produce new results.

- Doesn't enable

  If the type of the head literal of one rule is disjoint from that of the body of another rule, then evaluating the first rule will have no direct effect on the evaluation of the second.

In general, it is not possible to determine exactly which rule may follow another without particular knowledge of the specific instantiation and the contents of the database. In some cases, though, the set of possibilities can be sufficiently narrowed so that specific code may be generated for each case such that, with succinct dynamic condition analysis, we may decide at run-time which code segment to execute without having to incur the overhead of executing the full match/select/act cycle.

3 Optimizing Transformations

To date we have defined three transformations for production system programs that can be applied to reduce the overall number of cycles required to execute the program. These optimizations are far from exhaustive and are less general than they can be. Our goals include the development of a large number of special and general purpose transformations.

The three, rule jamming, loop rolling, and branch factoring, capture several major idioms used to express procedural control in production system language programs. Each define conditions sufficient to determine when a group of rules may be replaced with a procedural chunk.

In order to replace a set of rules with a single, procedural rule, it is necessary to determine that no execution path other than the intentionally transformed one can "pass through" the chunk of rules. More formally, for any initial execution state, a transformation must preserve a non-empty subset of the final states of the original program (if it had any) and as well as preserve termination.

To satisfy these correctness requirements, the conditions for determining whether a particular transformation applies requires information about all the rules in the system. It does not suffice to determine, say, that execution of one rule is always followed by another in order to replace the two with a single rule; it is also necessary to determine that no other rule can ever lead to the firing of the second.

The three optimizations mentioned here feed into each other. The most important one in terms of performance, loop rolling, is often enabled by performing the others.

For each transformation, we give an example of the applied transformation in an OPS5-like syntax. We must stress here, though, that the ARL target language into which the source rules are transformed is not OPS5. In contrast to chunking in SOAR, which transforms rules from the source language back into the same source language, ARL has explicit representations for binding points, search state, type information, and conditional and loop constructs as well as various compiler bookkeeping data. As such, it analogous to a the register transfers language common in conventional compilers.

Using a different language in the compiler has several advantages. The chief among them is that it obviates the need to re-recover control information after each round of transformation. Having explicit representations for the procedural constructs makes it easier to generate code for them as well as to specify transformations that build on others. As an internal language, ARL can have more special-purpose rough edges and

```
(p make-big-block
    (goal ↑state make-big-block)
    →
    (make block ↑size 10))

(p note-big-block
    (block ↑size >= 10 ↑qual-size nil)
    →
    (modify 1 ↑qual-size big))
```

Figure 2: Rules that may be jammed together.

```
(p make-big-block
    (goal ↑state make-big-block)
    →
    (make block ↑size 10 ↑qual-size big))
```

Figure 3: A jammed rule.

is more flexible than a general-purpose programming language.

Rule Jamming

A common idiom in production system programming is to divide a series of operations upon memory tuples across several rules so that the result of firing one rule enables another to do its job. Rule jamming compresses the operations of the multiple rules into a single one.

In Figure 2, the `note-size` rule triggers off the `make-big-block` rule to (eventually) modify every block. If there are no other dependencies on use of the `qual-size` of the block or modification of its numerical size, then the computation of the `qual-size` may be added on to block creation and done in one fell swoop as in Figure 3. If, after jamming the two rules together, there are no other dependencies that could lead to the `note-big-block` rules firing, then it may be deleted entirely.

Rule jamming is a form of static chunking (Laird, Newell, & Rosenbloom 1986). As such, the transformation is vulnerable to the problem of expensive chunks (Tambe & Newell 1988). A naive implementation may compute large cartesian products where a search involving multiple rules would have cut off before getting too deep. If the evaluation technique includes an intelligent backtracking scheme and forward information about possible join targets, however, a jammed rule can avoid this problem. In any case, the size of the products are the same, and so asymptotically no more work will be done.

Branch Factoring

The only form of conditional execution available to production system programs is through the use of multiple rules that are otherwise distinguished only by the

```
(p note-large-blocks
    (block ↑size > 10 ↑qual-size nil)
    →
    (modify 2 ↑qual-size big))

(p note-small-blocks
    (block ↑size <= 10 ↑qual-size nil)
    →
    (modify 2 ↑qual-size small))
```

Figure 4: Unfactored rules.

```
(p note-block-size
    (goal ↑state note-size)
    (block ↑size <n> ↑qual-size nil)
    →
    (if <n> <= 10
      then (modify 2 ↑qual-size small)
      else (modify 2 ↑qual-size big)))
```

Figure 5: A factored rule.

branch conditions. Multiple rules, with the same join conditions but differing in tests upon constant values, can be merged into a single rule that tests for the constant in the execution phase of rule firing.

In Figure 4, the two rules `note-large-blocks` and `note-small-blocks` differ by only a constant test on the size of a block. Since, in this case, all blocks must be examined, it is desirable to match against all the blocks and decide which size category a block belongs to on the right-hand side of a single rule as in Figure 5.

For branch factoring, the goal is to find types in the antecedent conditions of each of several rules that can be merged into a single more general type such that the pairwise differences between the original types are unique. The rules can then be merged into a single rule that replaces the specific types with the general one and tests for exactly which one is satisfied in the action part of the rule.

Branch factoring allows sharing of code for rules beyond simple common subexpressions. It differs from RETE-style sharing in that it effectively provides a two-level network where the output of the first network is fed back into the second.

This kind of sharing is particularly important in active database settings, where a search for instantiations of a factored rule may require only a single scan of the database, whereas searching for instantiations for distinct rules may require two.

Loop Rolling

Production system languages typically have no iteration construct. Loop rolling involves detecting situations where multiple tuples are operated upon in a

```
(p note-large-blocks
    (goal ↑state note-large-blocks)
    (block ↑size > 10 ↑qual-size nil)
    →
    (modify 2 ↑qual-size big))

(p done-noting-large-blocks
    (goal ↑state note-large-blocks)
    - (block ↑size > 10 ↑qual-size nil)
    →
    (modify 1 ↑state do-something-else))
```

Figure 6: An unrolled loop.

```
(p note-large-blocks
    (goal ↑state note-large-blocks)
    all (block ↑size > 10 ↑qual-size nil)
    →
    (modify 2 ↑qual-size big)
    (modify 1 ↑state do-something-else))
```

Figure 7: A rolled loop.

similar manner and arranging for the operations to occur outside the system cycle.

There has been some work on adding set-oriented constructs to rule system languages (Delcambre & Etheredge 1988; Widom & Finkelstein 1989). Our approach avoids the tricky problem of specifying a precise and general semantics for set-oriented constructs by leaving their definition within the compiler.

We describe an optimization for a simple case of loop rolling that captures a fairly common iteration idiom. The conditions for this optimization are somewhat complex in presentation but fairly simple in concept. The trick is to find a type in a pair of rules that serves as the invariant condition for the loop. One of the rules is called the "body" of the loop, and the other is the "guard." It must be the case that no other rules reference the invariant condition, so that the rules are active exclusively. The body and guard rules must not interfere with each other. The body rule must "count down" the type being operated upon, and the guard rule must check to see when there are no more data elements left. The transformed rule performs all the actions of the body rule that would have been spread out over a number of cycles in a single one.

In Figure 6, the rule `note-large-blocks` serves as the body rule, and `done-noting-large-blocks` as the guard. The body repeated removes blocks with a null `qual-size` attribute, and the guard checks to see if there are any such blocks left. The type of the goal element (assuming there are no other references to it in the program) serves as the invariant. The resulting rolled loop appears in Figure 7.

In main memory implementations, loop rolling can significantly reduce the overhead required to maintain run-time data structures. For example, most systems maintain lists of instantiations in sorted order. When one instantiation is produced at a time, maintaining the lists is an $O(n^2)$ insertion sort. If a number of instantiations are produced at once, a much more efficient $O(n \log n)$ sorting algorithm may be used.

Program	Rules	Rule Jamming	Branch Factoring	Loop Rolling
Manners	8	0	0	2
Waltz	33	0	2	2
ARP	111	4	9	6
Weaver	637	18	56	11

Figure 8: Occurrences of transformations in benchmark programs.

In the contexts of database integration and parallel execution, loop rolling passes more information to the query processor at once, allowing it to do a better jobs of optimizing selection and join sequences. Set-oriented constructs may appear explicitly in the generated code for the rolled loops, and so fewer queries can be issued, and the ones remaining are more easily parallelized.

In both cases, loop rolling reduces the activity of the interpreter by eliminating an invocation of it upon each loop iteration. In parallel implementations the interpreter calls can require a considerable amount of overhead.

Loop rolling can reduce the asymptoptic complexity of program execution. Using conventional matching algorithms, worst case matching complexity is W^n, where W is the size of the working memory and n is largest number of condition elements in a single rule. If it is possible to roll all firings of the bounding rule into a single cycle, matching complexity is reduced by an exponential factor.

4 Experimental Results

We have examined a number of commonly-used OPS5 benchmark programs for the applicability of the transformations given in Section 3. The programs are extensively studied in (Brant *et al.* 1991). Figure 4 summarizes the programs and opportunities for transformations. Some small programs, such as Manners, do not exhibit many possibilities for the three transformations described in this report.

Figure 4 shows the number of interpreter cycles needed to execture the benchmarks with and without applying the transformations. Figure 4 shows the wall-clock execution times of the programs.

5 Conclusions

We have described an architecture for an optimizing compiler for production system languages and several

Program	Raw	Transformed	Ratio
Manners	183	47	3.9
Waltz	39	18	2.2
ARP	2885	831	3.5
Weaver	1718	204	8.4

Figure 9: Interpreter cycles needed to execute untransformed and transformed programs.

Program	Raw	Transformed	Ratio
Manners	48.4	37.5	1.3
Waltz	4.4	3.0	1.5
ARP	581	94	6.2
Weaver	1654	195	8.5

Figure 10: Run-time (seconds) performance of untransformed and transformed programs.

optimizations it can perform. Besides the three mentioned in this report, there are a number of other possible transformations that can be developed using this framework. Due to the generality of using first-order representations and using a general-purpose theorem prover to compute their relationship, the framework can be tailored to deal with the peculiarities of particular production system languages and exploit additional opportunities for optimizations.

The key to the approach is the wholesale replacement of rules whenever global static analysis can determine that the structure of the rules and the semantics of the program allow a single execution path. The resulting program then avoids the overhead of determining that path at run-time.

We have demonstrated good execution improvements using only a few transformations. Armed with a sufficient number of these transformations, a production system compiler can arrange to significantly reduce the number of match/select/act cycles needed to execute a program.

6 Acknowledgments

This research was supported by the State of Texas Advanced Technology Program, the University of Texas Applied Research Laboratories Internal Research and Development Program, and ARPA, grant number DABT63-92-0042.

References

Brant, D.; Lofaso, B.; Gross, T.; and Miranker, D. P. 1991. Effects of Database Size on Rule system Performance: Five Case Studies. In *Proc. 17th VLDB Conf.*

Delcambre, L. M. L., and Etheredge, J. N. 1988. The Relational Production Language: A Production Language for Relational Databases. In *Proc. 2nd Int'l Conf. on Expert Database Systems*, 153–162.

Forgy, C. L. 1981. OPS5 User's Manual. Technical report, Department of Computer Science, Carnegie Mellon University.

Forgy, C. L. 1982. RETE: A Fast Match Algorithm for the Many Pattern/Many Object Pattern Match Probem. *Artificial Intelligence* 19:17–37.

Georgeff, M., and Bonollo, U. 1983. Procedural Expert Systems. In *Proc. Int'l Joint Conf. on AI*, 151–157. William Kaufmann, Inc.

Gupta, A. 1988. *Parallelism in Production Systems.* Pittman/Morgan-Kaufman.

Ishida, T. 1991. Parallel Rule Firings in Production Systems. *IEEE Trans. on Knowledge and Data Engineering* 3(1).

Kuo, C.-M.; Miranker, D. P.; and Browne, J. C. 1991. On the Performance of the CREL System. *Journal of Parallel and Distributed Computing* 13(4):424–441.

Kuo, S.; Moldovan, D.; and Cha, S. 1990. Control in Production Systems with Multiple Rule Firings. In *Proc. IEEE Int'l Conf. on Parallel Processing*, volume II, 243–2246. IEEE.

Laird, J.; Newell, A.; and Rosenbloom, P. 1986. Soar: An Architecture for General Intelligence. *Artificial Intelligence* 33(1):1–64.

McCune, W. W. 1990. OTTER 2.0 Users Guide. Technical Report ANL-90/9, Argonne National Laboratory.

Miranker, D., and Lofaso, B. J. 1991. The Organization and Performance of a TREAT Based Production System Compiler. *IEEE Trans. on Knowledge and Data Engineering* 3–10.

Miranker, D. P.; Brant, D.; Lofaso, B.; and Gadbois, D. 1990. On the Performance of Lazy Matching in Production Systems. In *Proc. Nat. Conf. on Artificial Intelligence*, 685–692. AAAI Press.

Piatetsky-Shapiro, G., and Frawley, W., eds. 1991. *Knowlegde Discover in Databases.* AAAI Press and MIT Press.

Schmolze, J. 1991. Guaranteeing Serializable Results in Synchronous Parallel Production Systems. *J. of Parallel and Dist. Com.* 13(4):348–364.

Stolfo, S. J., and Harrison, M. C. 1979. Automatic Discovery of Heuristics for Nondeterministic Programs. In *6th Int'l Joint Conf. on AI.*

Stonebraker, M. 1992. The Integration of Rule Systems and Database Systems. *IEEE Tran. on Knowledge and Data Engineering* 415–423.

Tambe, M., and Newell, A. 1988. Some Chunks Are Expensive. In *Proc. Int'l Conf. on Machine Learning.*

Widom, J., and Finkelstein, S. 1989. A Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems. Technical Report RJ 6880 (65706), IBM Almaden Research Center.