Transforming Conjunctive Match into RETE: A Call-Graph Caching Approach

Mark W. Perlin

May, 1991
CMU-CS-91-142 2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the National Library of Medicine of the National Institutes of Health under Contract 5R29 LM04707-04, and by the Pittsburgh NMR Institute.

Keywords

Graph Data Structures
Knowledge Representation
Program and Recursion Schemes
Rule-Based Knowledge Representations
Evaluation Strategies

Abstract

Conjunctive match is often used in Artificial Intelligence as the kernel of a pattern-directed inference [37] engine. Conjunctive match entails generating and testing all possible combinations of objects against a pattern of constraints. While simple to program, it is an expensive, exponential cost computation.

To reduce this average match cost in production system engines, the RETE match algorithm [8] was devised. RETE compiles each rule's pattern of constraints into a network, and then incrementally updates partial matches as objects are inserted and deleted. RETE, however, has its own cost: conceptual and implementational complexity.

Call-graph caching (CGC) [20] is a mechanism for transforming recursive specifications into highly optimized networks. In this paper, we describe CGC, and use it to transform a family of recursive conjunctive match formulations into their corresponding RETE networks. Our approach illustrates the ideas behind RETE, and shows their application to other algorithms.

Table of Contents

1. Introduction

We introduce the concept of conjunctive match, and its highly optimized counterpart, the RETE match algorithm. Call-Graph Caching is then introduced as a mechanism for automating the construction of RETE.

1.1 Conjunctive Match

Conjunctive match is a ubiquitous algorithm in Artificial Intelligence (AI), used extensively in pattern directed inference [37]. It is employed, for example, in both forward and backward rule systems [15] to compute all possible combinations of objects that match a specified pattern.

Conjunctive match requires a *pattern*, and a set of *objects*. A pattern is a set of constraints on an n-tuple of objects. When a single candidate n-tuple satisfies *all* of the pattern's constraints, the n-tuple is said to conjunctively *match*, or *instantiate*, the pattern. Generally, the goal is to find all the n-tuple instantiations of a pattern. This has a cost: the set of all n-tuples has size $|\#objects|^n$, so the worst-case computational complexity is exponential in n.

In a forward rule, or *production*, system, each rule has an associated pattern. A rule can be triggered by just those n-tuple instantiations of working memory (WM) objects that match its pattern. A production system repeatedly performs a three phase cycle: (1) *matching* all the rule patterns against all the working memory objects, (2) *selecting* zero or more instantiated rules, and then (3) *executing* the selected rule instantiations. The system terminates when there are no matching instantiations.

In the phase 1 match operation of a production system, all the rule patterns are matched against the WM objects. These exponential cost conjunctive match computations occur within the system's inner cycle. In order to scale up production systems to realistic applications, a more sophisticated approach to conjunctive match, such as RETE, is required.

1.2 RETE Matching

RETE [8] is an efficient incremental algorithm for conjunctive match. It exploits the temporal redundancy often found in rule systems: the WM set of objects changes very little (e.g., less than 1%) between match cycles. Therefore, instead of unnecessarily repeating most of the match computation each cycle, RETE incrementally computes only the requisite *changes* to the match state.

RETE, Latin for network, recasts the constraints of a conjunctive match pattern into a network data structure. The standard RETE network contains alpha and beta nodes, which use these constraints to act on and filter WM data. Alpha nodes filter sets of objects. Beta nodes selectively augment k-tuples (k<n) with WM objects, to eventually construct n-tuple instantiations. The RETE network organizes the control of this constraint testing activity, as well as the memory for the partial k-tuple instantiations.

RETE's incremental operation can reduce conjunctive match's exponential average case behavior down to fast, manageable polynomial complexity. This efficiency, however, has its own price. Developing an optimized RETE matcher is a far more arduous task than writing a simple, inefficient conjunctive match program. For example, changes to the topology of the network [33, 34] or its evaluation mechanism [12, 33] may be of significant research interest, and often require major reimplementation.

1.3 Call-Graph Caching

Where do complex network programs such as RETE come from? Consider a recursive computation. The recursive process proceeds top-down. If a recursive call lacks sufficient information to perform its computation, it recursively initiates further subcalls. Conceptually, in a terminating computation, the subcalls eventually arrive at leaf nodes, which require no further recursion for their computation.

The tree of recursive calls formed by recording this recursive process is termed a *call-tree* [36]. A call-tree is evaluated bottom-up, from the leaves back up to the root. When all of its children have completed, a call node has sufficient information to compute. This partial ordering, requiring children to compute before parents, determines the control structure of the evaluation.

If a particular computation is repeated often enough, the call-tree may be cached and reused as needed. This dispenses with redundant top-down constructions of the call-tree, resulting in purely bottom-up computation, from the leaves up to the root. When sets of call-trees are merged in a consistent way (see below), they form a network termed a *call-graph.*

Call-graph caching (CGC) [20] is the preservation of call-graphs for subsequent reuse. CGC enables us to use our intuitions about simple recursive processes to understand how network programs arise, and thereby effectively implement complex efficient network algorithms. In this paper, we focus on one application of CGC: designing and constructing incremental network algorithms such as RETE.

We begin in Section 2 by presenting Call-Graph Caching via simple arithmetic examples. In Section 3, we provide several recursive formulations of conjunctive match. RETE match is motivated, and then constructed, in Section 4 by applying CGC to conjunctive match. Section 5 describes several styles of programming that interface well with CGC. Preceding and related research is discussed in Section 6.

2. Call-Graph Caching

Recursive computation can often be described with a set of recursive expressions. Each expression specifies a local action that can be performed on (possibly recursive) arguments. Recursive evaluation takes these expressions, together with input data, and unravels them into a call-tree.

Each node in this expanded call-tree specializes some expression with additional information, such as (1) local data or (2) links to other call-nodes (e.g., immediate predecessor and successor nodes). Once binding values for the input leaves are known, evaluation completes by a bottom-up traversal of the call-tree, executing local actions on arguments according to the control sequence implicit in the tree links' partial order.

In this Section, we tailor CGC to the evaluation of recursively defined expressions. We begin by examining the evaluation of simple arithmetic expressions. The techniques we then develop will enable us to transform conjunctive match into RETE.

2.1 Examples

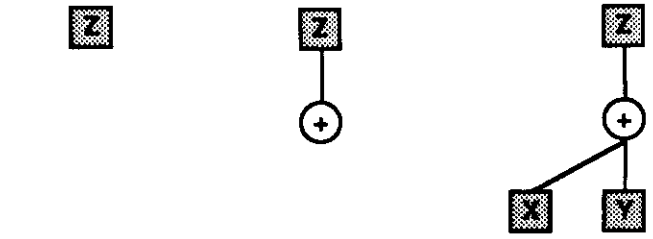
An arithmetic expression can be viewed as the expansion of a set of rewrite rules, or *recursive expressions*. Consider, for example, the arithmetic expression (x+y)*y. (More dynamic expressions and techniques will be introduced in Section 2.5.) Calling the result w, this can be rewritten as the recursive expressions:

w ← *(z,y)

z ← +(x,y).

W can be recursively computed once binding values for variables x and y are known. We illustrate by evaluating the single expression z←+(x,y) using an eager-expansion/lazy-update strategy.

As shown in Figure 1, the evaluation begins by (A) expanding a call for an *output processor* z that will contain the final result. Z computes its value by (B) expanding top-down into a call for +(x,y). This call builds a *processor* that has the local action +, together with calls for its two arguments, the variables x and y. These calls (C) expand into the leaf call-node *input processors* x and y, terminating the expansion.



(A) (B) (C)

Figure 1. A. Expanding the call for output processor z. **B.** Expanding the call for processor + on arguments x and y. **C.** Expanding the call for the input processors for variables x and y.

To complete the top-down expansion, a recursive evaluator binds values to the variables, and updates each call-tree node. This bottom-up update process must respect the call-tree's partial order, updating parent call-nodes only after all children have completed. Any topological ordering [14] will suffice. (Since no scoping or side-effect issues arise with our expressions, we do not need more restrictive topological orderings such as Depth-First Ordering [3].)

Suppose, at base level 1 of our call-tree, we bind the input processors as x=1 and y=2. The resulting bottom-up level-by-level topological traversal [25] is shown in Figure 2. After

completing level 1, the + processor at level 2 is evaluated. Here, the + action is applied to the values of the immediate predecessor nodes x and y, computing and updating the local value to 3 (=1+2). Moving on to level 3, the output processor then accesses this 3 value, and updates the value of z.

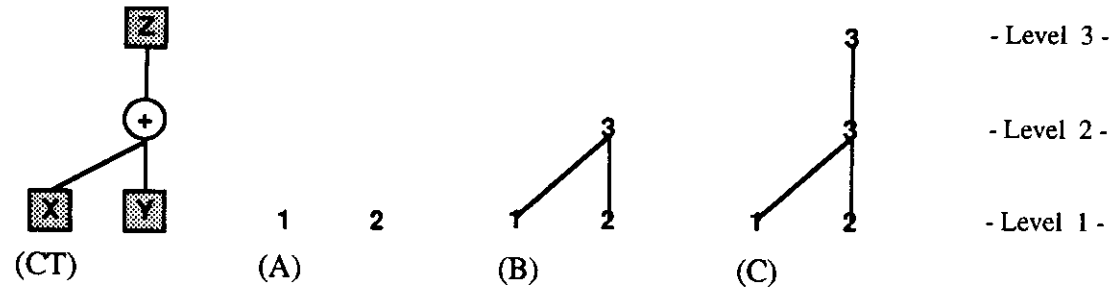


Figure 2. CT. The call-tree. **A.** Binding the variables to x=1 and y=2. **B.** Using the + processor to add 1 and 2. **C.** Returning the output value of z.

2.2 Caching Call-Trees

In standard recursive evaluation, once the computation terminates, the call-tree is discarded. We can exploit our eager-expansion/lazy-update evaluation strategy for recursion, and separate evaluation into two distinct components:

(1) a top-down *expansion* that constructs the full call-tree, and preserves it for subsequent reuse, and

(2) a later bottom-up *update* that uses the call-node processors to perform computation on particular input data.

This strategy is a use of *call-tree caching* (CTC).

The two algorithms EXPAND and UPDATE, shown in Figure 3, implement CTC for any finite expression. Starting from a root call-node r and an expression S, EXPAND(S,r) will construct the call-tree. Expansion halts on atomic nodes, such as variables and constants, which have no subexpressions. To update, bindings are assigned to the leaf variables, and UPDATE(node) is called on every affected call-node, sequencing the updates in a topological ordering.

```
EXPAND(expression, caller)
   (1) Build the call-node for the expression, and name it Self.
   (2) Examine the expression to identify and store the action,
     local data, and further subexpression arguments.
   (3) For every subexpression,
       LINK(Self, EXPAND(subexpression, Self))
   (4) Return Self.
```

```
UPDATE(node)
   Set the node's value to
      Apply the node's action to
         the values of predecessor nodes and
         any local data.
```

Figure 3. Algorithms EXPAND and UPDATE. EXPAND is called on an input expression and an initial root node. The UPDATEs are sequenced in topological order.

For evaluating expressions, EXPAND/UPDATE is better visualized, hence more intuitive, than the standard interleaved EVAL/APPLY mechanism [1] found in LISP. EVAL/APPLY is somewhat complicated by its effort to reduce space cost to just a single path in the call-tree via an auxiliary stack. Note that EXPAND also requires an auxiliary stack to effect its recursion, and that UPDATE needs an auxiliary agenda of queues indexed by level. However, EXPAND/UPDATE does not use these structures to reduce space cost; rather, it insists on full expansion of the call-tree.

2.3 Incremental Evaluation

Despite the increased space cost, caching call-trees has thus far introduced no efficiency gains. All we have saved is the cost of building the call-tree when repeatedly updating with different variable bindings. But when recursively evaluating an expression, this construction cost is at most proportional to the cost of *using* the call-tree for update computation. So when is CTC useful?

There are a number of situations in AI where a cached call-tree is nontrivially reused. With recursive transition network (RTN) parsing, discussed below, the call-trees are required for proper operation. In linear unification, mentioned below, the call-tree's structure is directly exploited for sharing. And here, with expression evaluation, the call-tree permits incremental recomputation of changing variable values, producing efficiencies unavailable with standard recursive evaluation.

There are two sources of efficiency in incremental operation:

(1) reducing the number of nodes visited during update, and

(2) reducing the local computation at each node.

Source (1) arises when there are many variable leaves in a call-tree. Each variable is a potential source of change and recomputation. Standard recursion is initiated top-down, reexpanding the

entire call-tree with each evaluation. The bottom-up update, however, need only recompute those call-nodes that are affected by (i.e., in the transitive closure of) the changed variable(s).

When variables describe sets of values, situation (2) provides even greater potential for reducing recomputation. Suppose, in Figure 2 above, there were 1,000 bindings of x, and 1,000 bindings of y. Using standard recursion, changing just a single value of y would necessitate recomputation of all 10^6 <x,y> pairs in the cartesian product. With a cached call-tree, though, it is possible to recompute just the 1,000 required values.

One advantage of separating the expansion of expressions from their data-driven value update is modularity. Once CTC has used EXPAND to construct the call-tree, any number of UPDATE implementations may be employed. We give two approaches to updating sets of values: processor-based update and instance-based update.

2.3.1 Processor-based update

When a processor's arguments allow sets, its local update mechanism must construct the cartesian product of argument sets, and apply the action to each tuple in the cartesian product set. The simple, naive approach is to completely reconstruct and recompute all the tuples whenever the processor performs its bottom-up update. To reduce this exponential cost, we present a more selective update mechanism.

To each processor P, we associate a memory M(P) containing all the tuples constructible from the input sets. Associated with each tuple t is its value, computed by P's action f as f(t). M(P) is indexed by the cartesian product $\prod_i M(P_i)$ of immediate predecessor nodes $\{P_i\}$. Standard recursive update computes the entire set

$$\{ f(t) \mid t \in \prod_i M(P_i) \},$$

which has a cost no less than the size of the exponential cartesian product $|\prod_i M(P_i)|$.

When, however, the input sets change little between successive update cycles, we can effectively reuse the M(P) from previous cycles. Specifically, we can write the finite difference equation for a processor:

$$M(P) = M_0(P) - M_-(P) + M_+(P),$$

where $M_0(P)$ is the tuple set from the previous cycle, and $M_-(P)$ and $M_+(P)$ denote the incremental difference (deletes and inserts, respectively) used to form the new memory contents M(P).

Writing

$$M(P) = \prod_i M(P_i),$$

by inductively applying the difference equation to P's predecessors $\{P_i\}$, we obtain

$$M(P) = \prod_i (M_0(P_i) - M_-(P_i) + M_+(P_i)).$$

Expanding the product of the sums into a sum of products, we have the sum over all terms v

$$M(P) = \sum_{\text{all terms } v} (\text{Sign}(v) \prod_i M_{j(v,i)}(P_i)),$$

where

- j takes values from the set {0,-,+},
- j(v,i) selects the j value of the i^{th} component of term v, and
- Sign(v)="-" when any term component has j(v,i)="-".

Since a single delete suffices to delete a tuple, we count multiply deleted terms only once. Further, we can ignore terms that combine a delete with an insert or another delete, since these do not contribute to M(P). Separating the sum into old tuples, newly formed tuples, and deleted tuples, we have

$$M(P) = \prod_i M_0(P_i) + \sum_{v, \text{ at least one } j=+, \text{ and no } j=-} (\prod_i M_{j(v,i)}(P_i)) - \sum_{v, \text{ exactly one } j=-, \text{ and no } j=+} (\prod_i M_{j(v,i)}(P_i)),$$

which, since $M_0(P) = \prod_i M_0(P_i)$, yields

$$M(P) = M_0(P) + \sum_{+ \text{ terms}} - \sum_{- \text{ terms}}. \qquad (*)$$

When the changes between update cycles to variable sets are very small, the first term $M_0(P)$ in (*) overwhelmingly predominates. Further, $M_0(P)$ is available to the processor from the previous update cycle. Therefore, only the small incremental difference sets $\sum_{+ \text{ terms}}$ and $\sum_{- \text{ terms}}$ need be computed; these sets can be used to efficiently construct M(P) from $M_0(P)$.

For example, let processor C receive input from the two preceding processors A and B, as in Figure 4.A. The naive approach would construct anew the full product set $M(A)\times M(B)$, and then apply the action. With an incremental update at the processor, though, only the smaller set

$$\sum_{+ \text{ terms}} - \sum_{- \text{ terms}} = M_0(A)\times M_+(B) + M_+(A)\times M_0(B) + M_+(A)\times M_+(B) - (M_-(A)\times M_0(B) + M_0(A)\times M_-(B))$$

will be computed and evaluated, as in Figure 4.B.

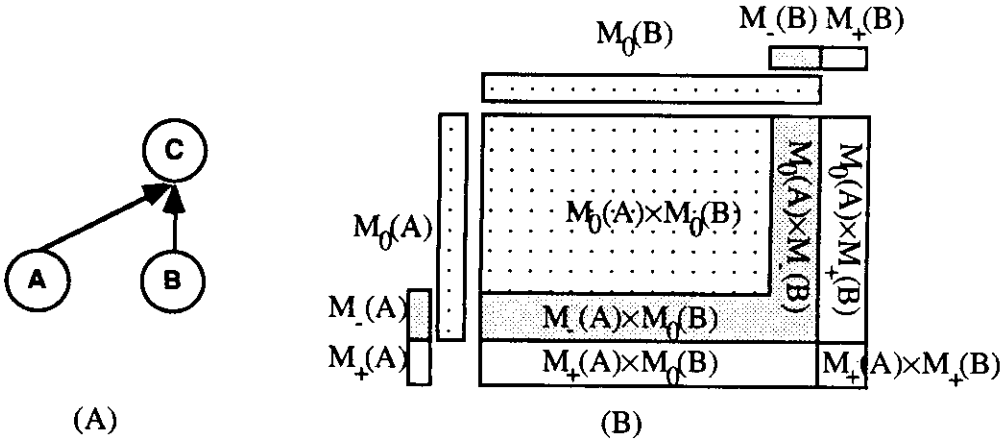


Figure 4. **A.** Processor C and its two predecessors, A and B. **B.** A two dimensional representation of incremental update at processor C. The largest product set, $M_0(A)\times M_0(B)$, is not recomputed.

2.3.2 Instance-based update

A more distributed approach is to let each tuple perform its own synchronization and updating. To do this, it is useful to reconceptualize the tuple as an independent object that specializes its processor node, rather than as an element of a memory set. In fact, in this reformulation, there *are* no memory sets. Rather, each tuple is an object-oriented instance of its processor class.

We present two methods, INSERT and DELETE, that in concert effect incremental bottom-up computation. When a tuple instance is dequeued during a topological traversal, the instance applies one of these methods to itself, initiating local computation and further enqueuing of instances. INSERT is given in Figure 5.

```
INSERT(instance)
{
Compute value of instance.
   ASK instance
      (1) Link up with predecessor instances.
      (2) Install self.
      (3) ENQUEUE successor instances for insertion. }
```

Figure 5. A method that inserts an instance of a processor. INSERT is used when dequeuing a newly created tuple instance.

Step (1), linking up with predecessor instances, may be viewed as *bottom-up* Call-Graph Caching [28]. (This perspective is not needed for the development in this article, and we shall not pursue

it further.) Step (2) installs the instance as an object available for communication with other instances. Step (3) constructs the cartesian product for its successor class(es). This is done by having the class ask its successor class who all its sibling classes are; a cartesian product is then formed between the current instance and all installed instances of sibling classes. Since instances contribute to cartesian products only after they are installed, the incremental difference set corresponding to $M(P)$-$M_0(P)$ is properly constructed. Figure 6 shows an example of INSERT.

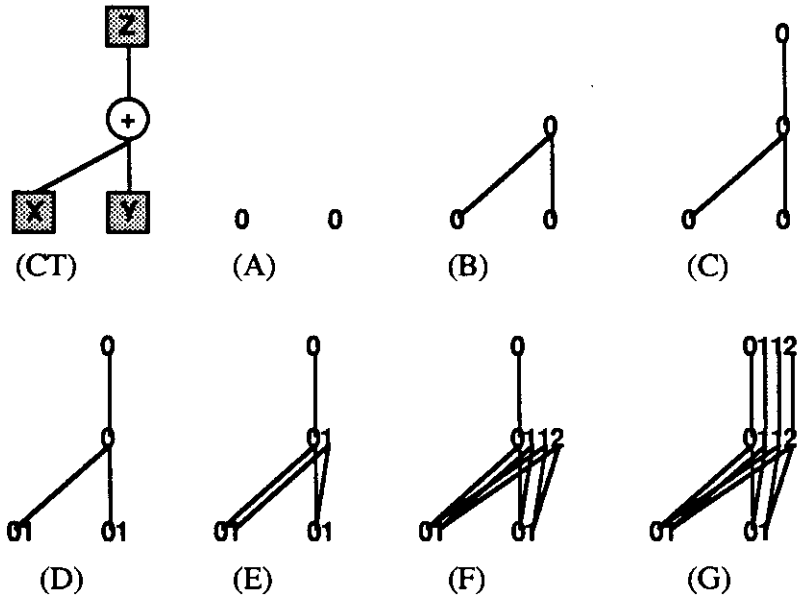


Figure 6. Inserting four instances of input processors. **CT.** The call-tree. INSERT x=0, INSERT y=0. **A.** Install x=0. **B.** Install y=0, and propagate. **C.** Complete. INSERT x=1, INSERT y=1. **D.** Install x=1. **E.** Propagate. **F.** Install y=1.**G.** Complete. Note the level-by-level topological traversal following the INSERTs.

The DELETE method, given in Figure 7, generates no new cartesian product tuples. Rather, it uses links between instance tuples to directly communicate deletion. An example is shown in Figure 8. With proper data structures, such as doubly-linked lists, a instance can be locally deleted in constant time [23]. Any bottom-up topological traversal will correctly interleave DELETEs with INSERTS.

```
DELETE(instance)
{
   ASK instance
      (1) Unlink from predecessor instances.
      (2) Deinstall self.
      (3) ENQUEUE successor instances for deletion. }
```

Figure 7. The DELETE method.

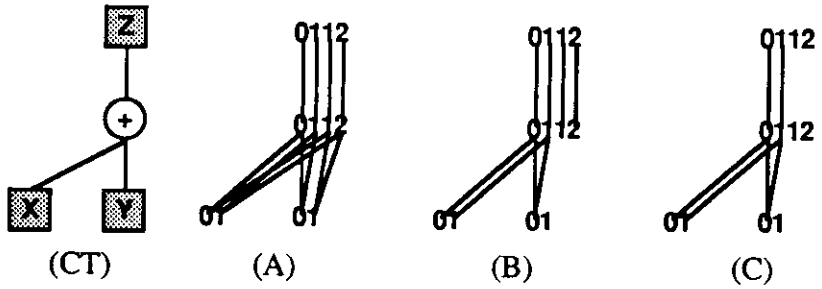


Figure 8. CT. The call-tree. **A.** DELETE y=1. **B.** DELETEs at level 2. **C.** DELETEs at level 3.

2.4 Merging Trees into a Graph

Our general mechanism is Call-Graph Caching, not Call-Tree Caching. When two expanded call-trees produce identical bottom-up computation, they can be partially merged into a single call-graph network. The determination of equivalence and the merging process can be done node-by-node during the bottom-up return of EXPAND.

We modify EXPAND as shown in Figure 9. Here, EXPAND first looks for an equivalent node in the network. If none is found, EXPAND operates identically to the old version of Figure 3. When an equivalent Self' is found, however, no linking is done, and Self' is returned to the caller. Step 3 is modified to (3b) look for an equivalent node, and (3c) delay the conditional linking until after this search.

```
EXPAND (expression, caller)
   (1) Build the call-node for the expression, and name it Self.
   (2) Examine the expression to identify and store the action,
     local data, and further subexpression arguments.
   (3a) For every subexpression i,
      Predecessor_i ← EXPAND(subexpression, Self)
   (3b) Let Self' ← EQUIVALENT(Self, {Predecessor_i})
   (3c) WHEN Self = Self'
     For every Predecessor_i,
        LINK(Self, Predecessor_i)
   (4) Return Self'.
```

Figure 9. EXPAND modified to perform bottom-up merging. EQUIVALENT examines the predecessor nodes to find a node equivalent to Self. If none is found, Self is returned.

EQUIVALENT(Node, {Predecessor$_i$}) determines whether there is a node *Node* having predecessors {Predecessor$_i$} and an action identical to Node's. If so, by induction, *Node* performs a bottom-up computation equivalent to Node's, and may be used in its place. One algorithm that implements EQUIVALENT is given in Figure 10. It exploits the fact that a node can merge only if all its predecessors have merged. Any equivalent *Node* must lie in the intersection of all these predecessor's successors. These candidates can then be checked for identical action and predecessors. (Some update actions require strict ordering of arguments, which can be checked for as well.)

```
EQUIVALENT(Node, {Predecessor_i})
(1) UNLESS all Predecessor_i have merged, RETURN Node.
(2) Let S = ∩_i Successors(Predecessor_i)
(3) RETURN
  OR   1. Find a node in S having an action identical to that
       of Node, whose predecessors are exactly {Predecessor_i},
       possibly in the same order.
       2. Node.
```

Figure 10. One algorithm for finding an equivalent node.

For example, we can merge the call-tree of expression w←*(+(x,y),y) together with the previously expanded z←+(x,y). Since the expanded first argument of w's action shares the bottom-up computation of z's expression, when expanded, their tree structures can be partially merged into a graph, as shown in Figure 11. The bottom-up computation is identical, with the corresponding merging of structure, until the processor node "*" is reached. This new node corresponds to no node in the old call-graph, so new structure is added to the network.

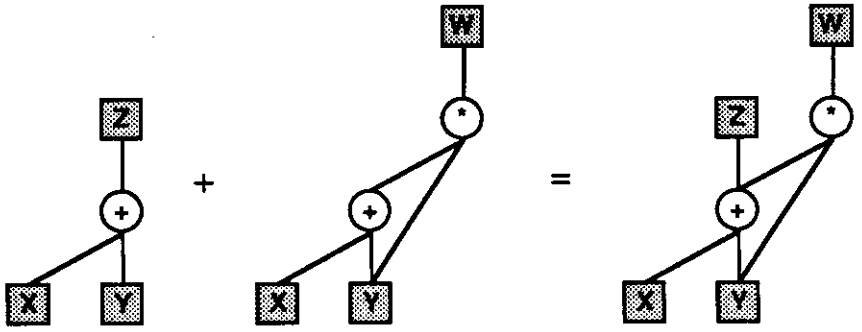


Figure 11. The network z←+(x,y) merging with w←*(+(x,y),y). Here, actions such as "+" are equivalent if they share the same name. Since w's expansion uses variable y twice, y has merged and has two parents.

2.5 Utility

Until this point, our examples have used expressions having (1) a simply described action and (2) fixed, predetermined arguments. Our algorithms and methods, however, can dynamically reconfigure an expression to determine more complex actions, along with appropriate subcalls. Our techniques are therefore applicable to more general recursive expressions.

Consider the function POWER(x,n). POWER is defined by the recursive expressions:

POWER(x,k) ← x * POWER(x,k-1)

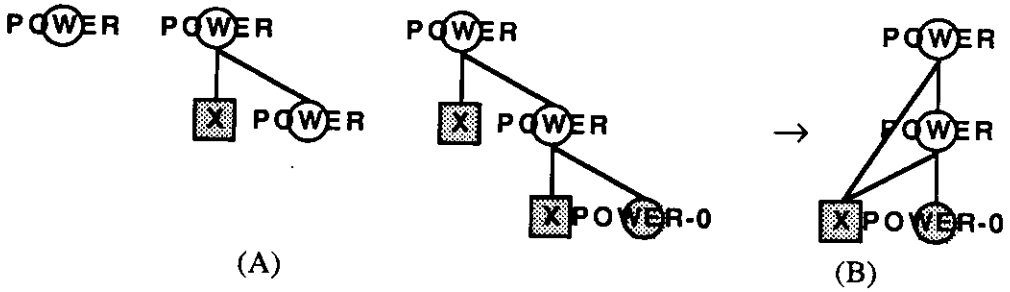
POWER(x,0) ← 1.

POWER(x,k) expands into the action "*", and the two arguments x and POWER(x,k-1). The second argument, POWER(x,k-1), is not immediately expanded. Instead, it is first partially evaluated using the top-down argument k, and only then expanded as POWER(x,*evaluate*(k-1)). While we have only considered bottom-up information flow until now, Step 2 of EXPAND can use top-down information in extracting local data and constructing appropriate recursive subcalls.

To illustrate, we set n=2, and present POWER(x,2) to EXPAND. Together with the recursive expressions for POWER(x,n), this completely expands POWER(x,2) as:

POWER(x,2) = x * POWER(x,1)
= x * (x * POWER(x,0))
= x * (x * 1).

In each step, the second subcall POWER(x,k-1) is dynamically constructed. The call-tree expansion is shown in Figure 12, along with the merged call-graph. POWER's second argument k provides top-down information that dynamically structures the call-tree topology, in this case, expanding to a linear depth of 3.



(A) (B)

Figure 12. **A.** The dynamic call-tree expansion of POWER(x,2). **B.** The call-graph of POWER(x,2), after merging.

As a second example, consider the function POLY(x,c_n), $c_n=\{a_i, 1\leq i\leq n\}$, which computes the polynomial $f(x) = \sum_i a_i x^i$. POLY can be described by extending the recursive expressions for POWER to:

POLY(x, c_k) ← (a_k * POWER(x,k)) + POLY(x,c_{k-1})

POLY(x, ∅) ← 0

POWER(x, k) ← x * POWER(x, k-1)

POWER(x, 0) ← 1.

Here, expansion of POLY(x,c_k), k>0, recasts the top-down information c_k into the action $\lambda.z_1.z_2.+(*(a_k,z_1),z_2)$, where:

- a_k is a local constant, computed as *k^{th}-element*(c_k),
- z_1 is the subcall POWER(x,k), where k is computed as *size*(c_k)-1, and
- z_2 is the subcall POLY(x,c_{k-1}), where c_{k-1} is computed as *set-difference*(c_k,a_k).

Once the action, local data, and subcalls are determined in Step 2 of EXPAND, expansion proceeds as described before.

This more dynamic usage of EXPAND is employed in our transformation of recursive conjunctive match into RETE. We continue, then, with conjunctive match.

3. Conjunctive Match

Conjunctive match requires a *pattern*, and a set of *objects*. A pattern is a set of constraints on an n-tuple of objects. When a single candidate n-tuple satisfies *all* of the pattern's constraints, the n-tuple is said to conjunctively *match*, or *instantiate*, the pattern. Generally, the goal is to find all the n-tuple instantiations of a pattern. This has a cost: the set of all n-tuples has size $|\#objects|^n$, so the worst-case computational complexity is exponential in n.

In the Introduction, we defined the conjunctive matching of an n-tuple against a pattern's constraints. For AI systems, our objective is to efficiently generate all the n-tuple instantiations satisfying a set of constraints [21]. In this Section, we describe several recursive (nonincremental) approaches to generating all such valid tuples.

3.1 Constraints

A *constraint* is a predicate that must hold on one or more objects. A constraint is operationalized by computing the predicate on its arguments, returning T or F. We write constraints in standard function notation as "predicate(arg_1,arg_2,...,arg_n)".

For example, we might want to form a 3-tuple of objects comprised of

(1) a Farmer object,

(2) a Thing object, and

(3) an Information object,

subject to the constraints that

(a) the Thing has the name "Goat",

(b) the Farmer and the Thing are on the same side of a river, and

(c) the Information holds that the lastmoved object is not named "Goat".

(These constraints are used in Rule-I of the toy Farmer puzzle [25]. In this puzzle, a Farmer must get his Wolf, Goat, and Cabbage across a river intact. His boat can hold him and at most one other possession. However, unattended, the Wolf will eat the Goat, and the Goat will eat the Cabbage.)

Rule-I's pattern can be written with constraints in standard function notation as:

Form a 3-tuple of objects comprised of (1) Farmer, (2) Thing, and (3) Information objects, subject to the constraints:

(a) =(Value(tuple,2,name), "Goat"),

(b) =(Value(tuple,1,side), Value(tuple,2,side)), and

(c) ≠(Value(tuple,3,lastmoved), "Goat").

Value(tuple,number,attribute) references the attribute of the numbered object in the tuple.

3.2 Constrained Tuple Generation

To test a single n-tuple against a constraint set is straightforward. It is implemented, for example, by the function CHECK in Figure 13. For a given pattern, testing constraints on a single tuple has a fixed, bounded cost.

```
CHECK(tuple, Constraints)
    For every predicate in Constraints:
        Let truth value ← Apply predicate to the tuple.
        If the truth value is F,
            return from CHECK with value F.
            Otherwise, continue.
    Return T.
```

Figure 13. CHECK(tuple, Constraints) tests whether a tuple satisfies all the constraints in Constraints.

More interesting is the generation of a *cartesian product* of n-tuples, each of which is then tested against the constraints. For example, we could have two farmers, three Things, and one Information object. Then Rule-I's pattern for 3-tuples would generate and test 6 (= 2×3×1) 3-tuples. Generally, the cost of pattern matching is proportional to the size of the cartesian product of data.

In our previous arithmetic examples, the computed values ranged over the set of natural numbers. With constraint testing, the computed values are limited to the smaller two-point set {T,F}. Rather than generating all n-tuples, and associating with them a value of T or F, we can instead constrain the generation of tuples: only those tuples having a value of T are preserved. The remainder, tuples having a value of F that do not satisfy some constraint, are discarded. This approach *filters* the cartesian product, instead of retaining all tuples.

3.3 Recursive Formulations

The generation and filtering of all possible n-tuples for a pattern can be described and computed using recursion. In fact, the different approaches to reducing average cost are well represented by their corresponding recursive expressions.

3.3.1 Single Node Topology

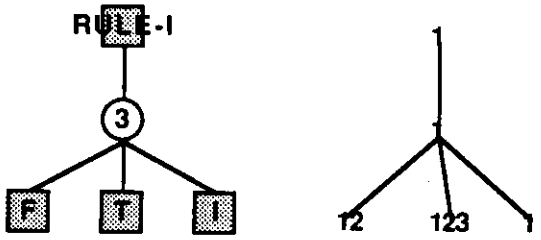
The simplest, and least efficient, formulation is brute force generation of every possible n-tuple, followed by constraint testing. This can be written with the single expression:

$T \leftarrow C\,(\prod_i D_i)$.

T is the tuple set of instantiations formed by applying the entire constraint set C to the cartesian product of data sets D_i. Computing T in this way has cost proportional to the product size $|\prod_i D_i|$.

The recursion's call-tree has a single processor node for C that performs the testing of every n-tuple.

For our example rule's constraints, EXPAND constructs the call-tree shown in Figure 14.A. Bottom-up evaluation generates all possible 3-tuples in the cartesian product, and then filters them to the one instantiation shown in Figure 14.B.



(A) (B)

Figure 14. A. A call-tree having a single inner processor node. The number "3" indicates the number of constraints at the processor node. At the leaves, F, T, and I designate Farmer, Thing, and Information, respectively. **B.** The shape of the one valid 3-tuple. The class instances are sequentially numbered, as shown.

3.3.2 Unary Topology

Computationally, we filter out the F-valued tuples in order to reduce the cost of cartesian product set formation. If products can be constructed from smaller, filtered sets, then the worst-case exponential size of the complete cartesian product may be reduced, on average. We show this for preliminary unary filtering of objects.

There is no overhead for cartesian product formation when generating 1-tuples from objects. A good strategy, then, is to test constraints that apply to just a single object, before testing multary constraints on multiple objects. We do this by *partitioning* the constraints into the sets C' and A_k, where:

C' = the set of multary constraints, affecting more than one object, and

A_k = the set of constraints affecting just object k, $1 \leq k \leq n$.

With this partition, we can write the recursive expressions:

$T \leftarrow C' (\prod_k U_k)$,

$U_k \leftarrow A_k (D_k)$.

This changes the pattern matching cost from $\prod_k |D_k|$ to $\prod_k |U_k| + \sum_k |D_k|$, which has a potentially smaller product in the $|U_k|$ terms. The call-graph of this recursion has a multary testing node, preceded by n unary processors.

Using our domain example, we can partition the constraints into the sets

$C' = \{ =(\text{Value(tuple,1,side)}, \text{Value(tuple,2,side)}) \}$

$A_1 = \{\}$

$A_2 = \{ =(\text{Value(tuple,2,name)}, \text{"Goat"}) \}$

$A_3 = \{ \neq(\text{Value(tuple,3,lastmoved)}, \text{"Goat"}) \}$.

This partition results in the call-graph of Figure 15; the processor labels indicate the number of constraints at the node.

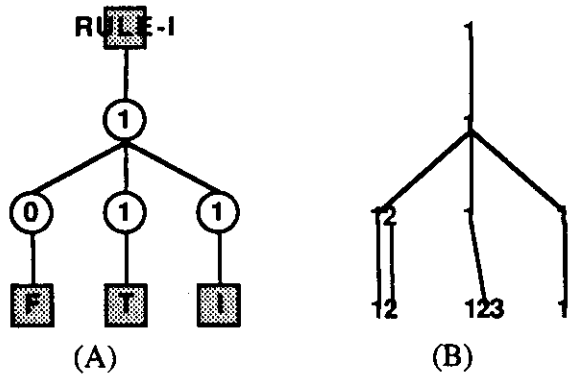


Figure 15. A. A call-tree with the unary filtering topology. **B.** The resulting partial tuple matches, leading to one complete 3-tuple.

3.3.3 Linear Topology

The n-tuples have thus far been formed by simultaneous cartesian product formation from all n predecessors. A more gradual approach would augment 1-tuples into 2-tuples, 2-tuples into 3-tuples, and so on. Thus, a k-tuple t that fails some constraint would be filtered out immediately, precluding the generation of any (k+1)-tuple extension of t. This approach to constrained generation can be described by a linear recursion.

We partition the constraints into the sets

$B_k = \{ c \in C \mid k \text{ is the maximum tuple index referenced by } c \}$.

With this partitioning into n sets, we can write the linear recursion:

$T \leftarrow T_n$

$T_k \leftarrow B_k (T_{k-1} \times D_k)$

$T_0 \leftarrow \emptyset$

Note that the cartesian product is now not done all at once, but instead gradually augments (k-1)-tuples into k-tuples. A k-tuple can be generated only if its j-tuple prefixes, $\forall j<k$, have not failed any test. The match cost becomes the sum $\sum_k |T_k|$, where, by filtering, $|T_k| \leq |T_{k-1}| \times |D_k|$. The recursion from T expands into a linear call-graph.

Continuing with our example, we partition our constraints into the sets $\{B_k\}$ according the maximum tuple index k that a constraint references:

$B_1 = \{\}$

$B_2 = \{$ =(Value(tuple,2,name), "Goat"), =(Value(tuple,1,side), Value(tuple,2,side)) $\}$

$B_3 = \{ \neq$(Value(tuple,3,lastmoved), "Goat") $\}$.

Figure 16 shows these B_k generating corresponding a call-tree from a linear recursion.

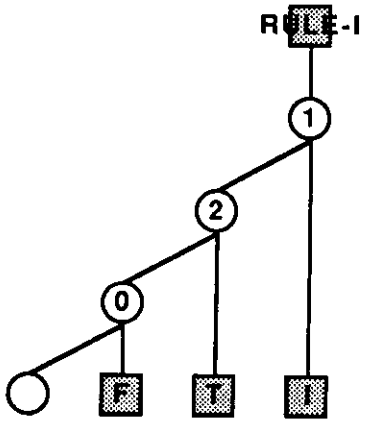


Figure 16. Expanding the linear recursion into a call-tree with linear topology.

3.3.4 Unary & Linear Topology

The unary and linear filtering strategies are readily combined. We partition each constraint set B_k into the unary constraint set A_k, and the multary constraint set B_k':

$A_k = \{ c \in B_k \mid c$ is an unary constraint $\}$.

$B_k' = \{ c \in B_k \mid c$ is a multary constraint $\}$.

This finer partition suggests the combined recursive expressions:

$T \leftarrow T_n$

$T_k \leftarrow B_k' (T_{k-1} \times U_k)$

$T_0 \leftarrow \emptyset$

$U_k \leftarrow A_k (D_k)$

Thus, objects are filtered by unary constraints before they enter into the cartesian product augmentation. The match cost is then $\Sigma_k|T_k|$, where, by combined filtering, $|T_k| \leq |T_{k-1}| \times |U_k|$.

Since B_1' is always $\emptyset$, the first nontrivial set product in the linear recursion is formed for T_2 from $D_1 \times D_2$. It is therefore conventional to use a reduced linear recursion. We write this by replacing the expression for T_0 with one for T_2:

$T \leftarrow T_n$

$T_k \leftarrow B_k' (T_{k-1} \times U_k)$

$T_2 \leftarrow B_2' (U_1 \times U_2)$

$U_k \leftarrow A_k (D_k)$

This reduced linear recursion performs equivalent computation, but eliminates unnecessary call-nodes. Call-trees are shown in Figure 17 for (A) the combined unary/linear recursion and (B) its reduced form.

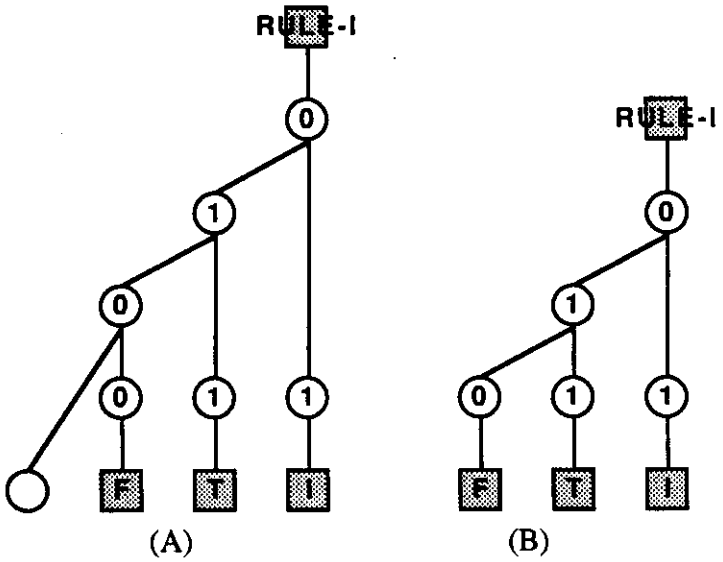


Figure 17. A. A call-tree having a unary/linear topology. **B.** A call-tree with the reduced unary/linear topology.

Returning to our example domain, we impose this finer partition on the constraint set:

$A_1 = \{\}$

$A_2 = \{$ =(Value(tuple,2,name), "Goat") $\}$

$A_3 = \{ \neq$(Value(tuple,3,lastmoved), "Goat") $\}$

$B_1' = \{\}$

$B_2' = \{$ =(Value(tuple,1,side), Value(tuple,2,side)) $\}$

$B_3' = \{\}$.

These expand into the call-trees shown in Figure 17.

3.3.5 Other Topologies

There are other useful constraint partitions and corresponding recursive expressions for constrained generation of tuples. We briefly describe several approaches.

The strategy used in *OPS-5* [9] pattern matching is an even finer partitioning of the unary/linear partition. It further subdivides the unary constraint sets A_k into singleton sets, as:

$A_{k,j} = \{ c \mid c \text{ is the } j^{th} \text{ member of } A_k \}$,

$B_k' = \{ c \in B_k \mid c \text{ is a multary constraint} \}$.

This partitioning is used in the recursive expressions:

$T \leftarrow T_n$

$T_k \leftarrow B_k' (T_{k-1} \times U_{k,m(k)})$

$T_2 \leftarrow B_2' (U_{1,m(1)} \times U_{2,m(2)})$

$U_{k,j} \leftarrow A_{k,j} (U_{k,j-1})$

$U_{k,0} \leftarrow A_{k,0} (D_k)$

The resulting cartesian product computation is equivalent to the unary/linear strategy's.

When there are no constraints at a processor node, it is false economy to use that node in constrained tuple generation. We can extend the insight of the reduced linear strategy, and eliminate *all* nodes that have no constraints. With the unary/linear topology, this would result in a *minimal* unary/linear topology, as shown in Figure 18. This modification produces filtering computation equivalent to the unmodified topology.

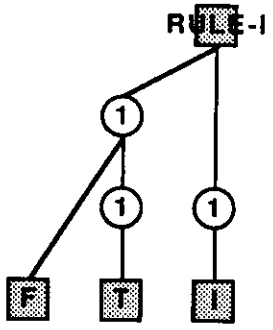


Figure 18. A call-tree with the minimal unary/linear topology.

A *binary divide-and-conquer* is also effective, replacing our linear recursions with tree-structured ones. The constraints can be partitioned into the inordered [2] subsets $\{B_m''\}$, where $0<m\leq n$. Define the height j(m) as the greatest j such that 2^j divides m (e.g., $j(6) = j(110_2) = 1$, since 2^1 divides 6, but 2^2 does not). Then the partitioned subsets are specified by how member constraints affect objects indexed in an n-tuple:

B_m'' = { c ∈ C | c is a unary constraint on object (m+1)/2 }, m odd.

B_m'' = { c ∈ C | c is a multary constraint on
at least one object in INTERVAL($m-2^{j(m)-1}$), and
at least one object in INTERVAL($m+2^{j(m)-1}$)
and no other INTERVALs }, m even,

where

INTERVAL(m) = INTERVAL($m-2^{j(m)-1}$) ∪ INTERVAL($m+2^{j(m)-1}$), m even,
{ (m+1)/2 }, m odd.

With the inorder partition, the binary divide-and-conquer is described by the recursive expressions, where n' is the least $2^k \geq n$,

$T \leftarrow V_{\log n', n'}$

$V_{j,m} \leftarrow B_m'' (V_{j-1,m-2^{j-1}} \times V_{j-1,m+2^{j-1}})$

$V_{0,m} \leftarrow B_m'' (D_{(m+1)/2})$.

Expansion constructs a binary call-tree, with each constraint set B_m'' located at the inordered processor node m. If n is not a power of 2, a smaller binary tree can be built using the expression

$V_{j,m} \leftarrow B_m'' (V_{j-1,m-2^{j-1}})$

when $n < m+2^{j-1}$.

For our example constraints, imposing the inorder partition produces the nonempty sets:

B_2'' = { =(Value(tuple,1,side), Value(tuple,2,side)) }

B_3'' = { =(Value(tuple,2,name), "Goat") }

B_5'' = { ≠(Value(tuple,3,lastmoved), "Goat") }

The expanded binary call-tree is shown in Figure 19.

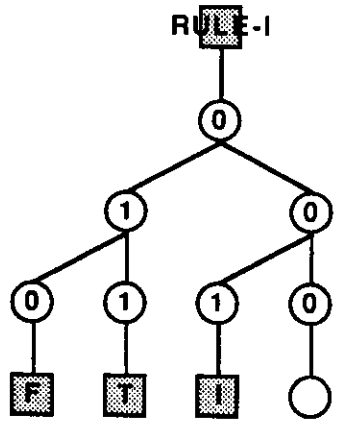


Figure 19. A call-tree with the binary topology.

4. RETE Matching

In Section 2 we developed CGC, which can construct incremental network processes from recursive descriptions. And, in Section 3, we gave a number of recursive formulations for conjunctive match. This Section describes RETE. In Section 4.3 we apply CGC to conjunctive match, and construct the RETE match algorithm.

4.1 Incremental Conjunctive Match

In pattern-directed inference engines, such as production systems, rule conditions are matched against a current WM state in order to determine the next rule action to execute. Generally, a rule action changes only a very small fraction of the WM state. Incremental match computation can therefore provide tremendous speedup.

A number of algorithms have been developed for incremental conjunctive match. The first one in common use was RETE match, the efficient match engine underlying OPS-5 and many other production system languages.

4.2 Standard RETE

Standard RETE is often described as a "data-driven bottom-up state-saving" algorithm. That is, RETE is

- *data-driven*: its control is initiated by changes to data;

- *bottom-up*: computation proceeds from the input data up to output rule instantiations;
- *state-saving*: partial matching computation is preserved and reused from one cycle to the next.

The key *data structure* in standard RETE is a data-flow *RETE network* that contains the tests and memories used for tuple augmentation. The network is conceptualized and implemented on a node-to-node basis, linking testing and memory nodes along their proper data-flow paths. The node types include:

(α) *Alpha test nodes*: These have one input, and perform a single test on an object.

(α-mem) *Alpha memory nodes*: These have one input, and store the objects that are not filtered out by alpha test nodes. The memory is used as input to the beta test nodes.

(β) *Beta test nodes*: These have two inputs, a left beta memory and a right alpha memory. They construct the incremental cartesian product, and perform tuple filtering.

(β-mem) *Beta memory nodes*: These memories store tuples formed at beta test nodes.

Our CGC techniques do not build networks from such single nodes. Rather, CGC builds networks out of entire tree expansions, and requires only a single type of processor.

With a set of rules, nodes that perform identical testing can be shared. This is done by a bottom-up merging of the nodes into a graph. In standard RETE, the alpha test nodes are are connected as a linear chain of tests. When these are merged, a discrimination tree is formed, routing data from input nodes up to appropriate alpha memory nodes.

The key *control structure* in standard RETE is a *Depth-First* traversal that propagates the insertion and deletion of tuples up through the RETE network. Unlike our topological traversal, multiple messages received at a node are not well synchronized, and extra work may be generated.

4.3 Transforming Conjunctive Match into RETE

Using any recursive conjunctive match formulation from Section 3.3, and a set of n-tuple rule patterns, the EXPAND algorithm of Section 2.2 will build a RETE network. For each rule, the pattern constraints are partitioned into different processor nodes in the rule's call-tree.

The call-trees of multiple rules are merged, as in Section 2.4, to build a network data structure. If the finely partitioned recursive formulation for OPS-5 described in Section 3.3.5 is used, the

linear chains of unary constraints will be merged into a single trie (i.e., discrimination tree) data structure.

When presented with input sets of WM objects, the expanded networks may be updated, using one of the UPDATE mechanisms in Section 2.3. A one-shot update will initialize the tuples; topological traversal assures efficient generation of all T-valued n-tuples. Subsequent updates incrementally insert and delete tuples.

4.4 Generality

There are many ways to increase the expressibility of RETE. Our modular development of the algorithm allows us to briefly survey several approaches.

4.4.1 Generalized Constraints

Any computable predicate on a k-tuple of objects can be used as a constraint in RETE's conjunctive match. Predicate testing may explicitly use the attribute values of objects, as in the equality constraint

=(Value(tuple,1,name), Value(tuple,2,owner)),

or it may entail more complex tests on the objects themselves, as with the *owns* predicate's unspecified computation in

owns(Object(tuple,1), Object(tuple,2)).

For rapid predicate computation or uniformity of syntax, some production system languages restrict the allowable predicates. For example, OPS-5 only tests equality, inequality, ordinality, and type on at most two attributes.

Since a predicate is any computable function mapping into {T,F}, it follows that a predicate may:

(1) have any number of arguments; (An argument may be a WM object, an attribute of a WM object, or a value not in WM.)

(2) be a disjunction of other predicates;

(3) employ arbitrary recursive computation in evaluating its truth value.

4.4.2 Absence Tests

Just as conjunctive match can test for the existence of tuples of objects, so can it test for their absence. Until now, we have viewed matching as finding the set of all tuples t, such that C(t), for some constraints C. Another view is that we want all solutions t to the formula:

$\exists t, C(t)$.

A condition for the existence of the objects in tuple t could be the nonexistence of other objects u_i. That is, we could check for the *absence* of the u_i. We write this as:

$\exists t, C(t)$ and $\neg\exists u_i, C^-_i(t,u_i), \forall_i$,

where each C^-_i is a set of absence constraints on t and u_i. More generally, absence testing applies to tuples. To check for the absence of tuples v_i, we can write

$\exists t, C(t)$ and $\neg\exists v_i, C^-_i(t,v_i), \forall_i$.

To implement absence testing, we extend our processor graphs to permit *absence links*, as well as the usual presence links. For a k-tuple instance t of a processor, the presence links work just as before: t is linked to an instance of each predecessor class, and tested as C(t). Each absence link i, however, links t to the *set* of instances $\{v_i\}$ that satisfy $C^-_i(t,v_i)$. When this set is not empty, then $\exists v_i, C^-_i(t,v_i)$, and t is blocked by (at least) that v_i. Therefore, tuple t is returned exactly when C(t) is true and $\neg\exists v_i, C^-_i(t,v_i), \forall_i$.

We give an example in Figure 20. Here we draw the call-tree of the conjunctive expression for recognizing all men without dogs:

Form a 1-tuple comprised of (1) a Man, and NOT (2) a Dog, subject to the constraint:

owns(Object(tuple,1), Object(tuple,2)),

Even with absence tests, the *type* of a node is still the product of the types of just its presence-linked predecessors. We therefore label the product type of each node in the Figure.

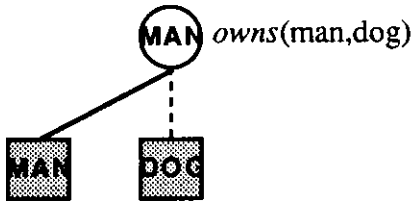


Figure 20. The call-tree for recognizing dogless men. The dashed line indicates an absence link.

4.4.3 Nested Conjuncts

Observe that the statement

$\exists t, C(t)$ and $\neg\exists v_i, C^-_i(t,v_i), \forall_i,$

is equivalent to

$\exists t, C(t)$ and $\forall v_i, \neg C^-_i(t,v_i), \forall_i.$

This introduces the universal quantifier $\forall$, and the capability of conjunctive match to recognize expressions in first-order predicate logic (FOPL) [15]. For full generality, we abandon the flat sequences of linear tuples, and move to tree-structured nested conjuncts. We illustrate with an example.

Suppose we want to state that there is a man who owns no flealess dog. In FOPL, we could write

$\exists x, man(x), \forall y, dog(y), owns(x,y) \rightarrow \exists z, flea(z), owns(y,z).$

Note that the $\exists z$ quantifier is scoped inside the logical implication. This will result in nesting the conjunct "$\exists z, flea(z)$" inside the scope of y. Removing the implication, we rewrite

$\exists x, man(x), \forall y, dog(y), \neg owns(x,y) \vee \exists z, flea(z), owns(y,z).$

To rewrite this expression in conjunctive form, we note that $\forall$ is just $\neg\exists\neg$, and write

$\exists x, man(x), \neg\exists y, dog(y), \neg(\neg owns(x,y) \vee \exists z, flea(z), owns(y,z))$, or,

distributing the "$\neg$" and rewriting,

$\exists x, man(x),$

$\neg\exists y, dog(y), owns(x,y) \wedge$

$\neg\exists z, flea(z), owns(y,z).$

This final expression is cast into the call-tree shown in Figure 21. The type predicates such as "man(x)" form the leaf classes, and the constraints are positioned by their processor nodes. We label the product type of each node.

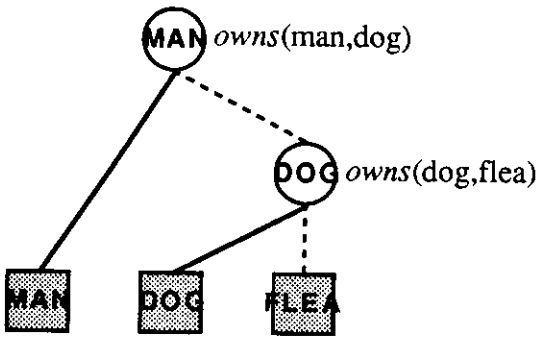


Figure 21. The call-tree for recognizing men without flealess dogs. Dashed lines indicate absence links., and the labels are the tuple types.

4.4.4 Partial Conjunctive Match

Conjunctive match finds the set of all tuples of objects that conjunctively satisfy a set of constraints. To relax this strict conjunctive requirement, solutions may include (1) tuples not containing all objects, or (2) tuples not satisfying all constraints. Since (1) is readily implemented with multiple partial call-trees, we explore the more interesting (2).

The single node topology (in Section 3.3.1) represents the simplest form of conjunctive match: all the constraints form a single predicate. To decrease match cost, we used more selective tuple generation strategies, such as the unary/linear topology (in Section 3.3.4). These efficiently filtered out large sets of n-tuples by eliminating all tuples having a failed k-tuple prefix, $k \leq n$. This approach worked because the conjunctive constraints are *decomposable*. That is, the single predicate of the single node topology can be decomposed into a set of smaller predicates, each of which may return F and independently eliminate the k-tuple.

To generalize to partial constraint satisfaction, we allow our predicates to return values outside the set {T,F}. Instead of eliminating a k-tuple because of a single violated constraint, we can delay elimination until more information is gathered. We look at the value sets **N** and **R**, and associated decomposition strategies.

The function CHECK(constraints,tuple) in Figure 13 is a predicate returning T or F. We can modify this function to compute the number of new constraint violations, and return the running total. That is, our conjunctive match now associates with each tuple a failure count *integer* in **N**, rather than a boolean value in {T,F}.

The UPDATE mechanism can then employ some threshold θ to decide when to eliminate a tuple. When the failure count reaches θ, the k-tuple is pruned. This scheme can be extended to weight each constraint c_i by its relative importance w_i, $0 \leq w_i \leq 1$. Each tuple t is then associated with the real-valued sum $\sum_i w_i \chi(c_i,t)$ (χ=1 if true, 0 otherwise), which must be kept less than θ.

Associating certainty factors [4] with tuples is another approach that uses a real-valued update function. For example, if a man instance is detected with certainty p, and a dog instance is known with certainty q, then a certainty function f may compute f(p,q) on the tuple

t=<man,dog>, returning the value r∈ **R**, and associating r with t. A certainty threshold θ can be set, so that tuple values beneath this threshold are filtered out.

5. Programming Styles

CGC computation can be initiated by a variety of computer programming styles. We made extensive use of rule programming throughout our presentation, using recursive rewrite expressions. Two other styles have also figured prominently in our implementations: object-oriented and functional specification.

5.1 Object-Oriented Programs

Step 2 of the EXPAND algorithm (in Section 2.2) examines an expression, and extracts the action, local data, and recursive subcalls. For any class of call-node, this is readily programmed using object-oriented methodology.

When a call-node is expanded, the expression and the calling node are known to EXPAND. The expression's first argument specifies the node's class. Methods can then be invoked to build the call-node structure, and initialize its components. These methods are inherited from the class, and include:

- Building the call-node structure, by instantiating the class.
- Setting the action, which is largely inherited from the class. The action may be specialized by additional information in the expression.
- Recording any local data, which can be extracted from the expression.
- Constructing the subcalls. This may entail some nontrivial computation to continue the recursion and ensure proper termination.

For example, we could program the class POLY(x,c_k), described in Section 2.5. POLY inherits the generic call-node constructor. Its action for a nonleaf node is the class function

```
(lambda (z1 z2)
     (+ (* coefficient z1) z2)).
```

The local variable *coefficient* is computed by a method that takes the k^{th} element of the set c_k. The first subcall is constructed by a method that calls POWER(x,*size*(c_k)-*1*). The second by a method that calls POLY(x,*set-difference*(c_k,a_k)).

When the class POLY is instantiated from an expression describing c_k, these methods are invoked to build and initialize the instance call-node. The instance can then recursively EXPAND. All information required for bottom-up computation is then available at the call-node instance.

We built an object-oriented CGC system, CACHE™ [29], in the Object LISP extension of Macintosh Allegro Common LISP. CACHE™, a Color Animated Call-grapH Environment, constructs call-graphs by preprocessing expressions, and then applying the EXPAND algorithm. CACHE™ animates the construction of these call-graphs (and their bottom-up evaluation) by having each call-node object position and draw itself. For example, the call-graph diagrams in this article were all drawn by CACHE™. The system includes CGC implementations of several classical AI algorithms, including RETE, linear unification, chart and Tomita parsing, as well as new AI algorithms under development.

5.2 Function Programs

Another approach is to simply write a recursive LISP function program, and use it as a specification for the Call-Graph Caching process. This was our original approach in [20]. By identifying which functions to highlight in the CGC trace, and supplying partial input data, a call-graph can be constructed. The nodes in this call-graph correspond to the highlighted functions, and the call-graph topology (and actions) is partially determined by the input data.

The technique can be understood as a folding and unfolding [5] of lambda expressions, partially evaluating the recursive specifications into a bottom-up program. A derivation can be found in [22], and detailed examples using POWER and POLY are presented in the Appendix of [19]. Here, we give a short example using POLY.

Suppose we want to construct a network for evaluating the polynomial "x+2", where the network nodes compute POLY and POWER. We can highlight the (italicized) functions POWER and POLY in the function

```
(defun poly (x C)
  (if (null C)
      0
      (+ (* (first C)
            (power x (1- (length C))))
         (poly x (rest C)))))),
```

and then expand the expression (POLY x '(1 2)).

Since C is not empty, and POWER and POLY must be recursively evaluated, we partially evaluate the expressions containing the top-down argument C='(1 2) as

- `(first '(1 2)) => 1`
- `(1- (length '(1 2))) => 1`
- `(rest '(1 2)) => (2)`

This leads to the call-node's instantiated action

```
(lambda (z1 z2)
     (+ (* 1 z1) z2)),
```

which can be applied to the results of the two subcalls

`(power x 1)`, and

`(poly x '(2))`.

This transformation of a program into a CGC process has been used in implementing the RETE engine of a visual production system [30], and for the network in the CONTRAST expert system [31]. The RETE engine is built by CGC transformation of simple linear recursive conjunctive match programs such as

```
(defun match (tests data)
  (if (null tests)
      (empty-tuple)
      (cartesian-product-filter
         (first tests)
         data
         (match (rest tests) data))))
```

into networks. `Cartesian-product-filter`(test,set1,set2) filters SET1×SET2 using the tests. Additional topologies and mechanisms for RETE are detailed in [26]. This CGC transformation can be generalized, and incorporated into the object-oriented class approach above.

6. Related Work

Our Call-Graph Caching network expansion and updating has ties to other techniques in AI and Computer Science. We describe these connections.

6.1 AI Algorithms

The top-down EXPAND routine fully expands an expression's call-tree prior to any bottom-up data evaluation. EXPAND has applications beyond the incremental recomputation we used in developing RETE.

In context-free *parsing*, EXPAND can be used to construct recursive transition networks (RTNs) directly from a grammar [27]. The RTN is simply a call-graph. Parsing may then be done, for example, by bottom-up completion from the input sentence, using INSERT [27]. This can straightforwardly implement the chart parsing [7, 39] and Tomita's algorithm [35] used in natural language processing.

EXPAND's merging strategy combines trees into networks. The algorithm can be used in the merging of simple linear chains into trees. This constructs the trie data structures [2] used in AI, such as *discrimination trees* [6], and *inheritance* hierarchies [24].

By exploiting the graph structure of expressions, EXPAND can implement efficient (almost) *linear unification* [17]. Two expressions unify exactly when their call-graphs completely merge. As in [17], variables are equivalence classes whose bindings may not conflict.

UPDATE uses a bottom-up topological traversal to complete EXPAND's recursive expansion. This traversal properly synchronizes the bottom-up operation, assuring that each node is visited at most once, with no redundant updates [18]. If we view the RETE network construction as a caching of top-down recursive subcalls [20], then topological traversal is RETE's correct control organization for bottom-up updating [25]. In particular, it properly synchronizes the data-flow for all RETE-like topologies [33].

Our INSERT/DELETE methods for instance-based update draw on ideas from object-oriented programming. A processor call-node in a RETE network is seen as an instance of its processor class, specialized with its local constraint set and links to other call-nodes. A tuple (sometimes termed a *partial instantiation*) is an instance of a processor call-node, specialized with particular data objects. That is, a tuple is an instance of a processor node, which in turn is an instance of a processor class.

In addition to facilitating intuition and programming, this object-oriented perspective also suggests parallel hardware implementations. If every tuple instance is allocated its own processor, and class information is copied rather than inherited, than INSERT/DELETE provides a general mechanism for distributed parallel update. This has empirical support in DRETE [12], which, properly reinterpreted, applies this paradigm to RETE. By distributing the tuple instances, rather than their processor classes, RETE's processor bottlenecks described in [11] may be overcome.

6.2 General Computation

Our methods overlap with many ideas from general Computer Science. Our bottom-up merging is a generalization of the directed acyclic graph (DAG) construction for expressions found in compilers [3]. Topological sorting finds many applications in computation [14], and has been used for incremental update in non-production system programming environments [32]. Top-down and bottom-up information corresponds, respectively, to the inherited and synthesized values used in attribute grammars [13].

6.3 Programming Transformations

CGC is not a programming transformation. Rather than performing a source-to-source program text compilation, CGC takes tree descriptions and builds graph processes from them in a purely run-time environment. Nonetheless, CGC exploits certain ideas that have been widely promoted by the programming transformation community.

Partial evaluation [10] is the evaluation of a program on partial input, producing a new (hopefully optimized) program. A compiler is one example. CGC views the construction of a bottom-up network program as the top-down half of a recursive evaluation, which is certainly a partial evaluation.

Finite differencing incrementally recomputes sets, as in Section 2.3.1, and has a long history in Computer Science. It can be used as a program transformation, automatically transforming a nonincremental program into an incremental one [16]. CGC uses the same principle, but changes the evaluator of an executing process, rather than transforming a program. Interestingly, there is a formal construction of RETE based on combining the partial evaluation and finite differencing program transformations [38].

7. Conclusion

We have detailed conjunctive match, the Call-Graph Caching process, and how CGC transforms conjunctive match into RETE. We conclude with some more general observations.

7.1 Caching Recursion into Networks

Recursive evaluation can be unraveled into networks. By recording the control decisions at each recursive step, the entire trace is cached and preserved for subsequent reuse; this process is Call-Graph Caching. The resulting networks can support processes far more efficient than the original recursive description.

7.2 Generality

The CGC process is ubiquitous in AI. Problem solving derivations are cached into graphs for subsequent analysis, processing, and reuse in:

- *Chunking* and *explanation-based learning*. Here, backward rule computation is cached into a graph structure that can be generalized and operationalized for more efficient rule execution.
- *Derivational analogy*. Entire problem solving derivations can be used as a template to direct further problem solving.
- *Planning*. Goal protection, hierarchical planning, and nonlinear planning can directly manipulate the recursion's call-tree in efficiently searching for a plan.
- *Scripts* and *case-based reasoning*. Sequences of high-level plans can be cached and retrieved for use as a template in further planning and understanding.
- *Production systems*. Control knowledge is acquired from human expertise and cached in the form of production rules, which can effectively substitute knowledge for search.
- *Truth maintenance*. Forward rule execution can be cached into network form, permitting subsequent reactivations and deactivations, rather than redundant search.

In general, CGC provides a mechanism for transforming search into knowledge.

At the algorithmic level, as discussed in Section 6.1, AI also employs CGC for efficient parsing, unification, and pattern matching. This paper focused on the use of CGC in constructing incremental algorithms, specifically, RETE matching.

7.3 Utility

How is CGC useful to AI researchers and programmers? CGC can offer considerable simplification for both the conceptualization and implementation of AI techniques.

A straightforward recursive process can be fairly easy to conceptualize. Top-down, a recursive call is constructed, identifying actions and data needed for later evaluation, and preparing any recursive subcalls. Bottom-up, the actions are executed. The conceptual focus is on just the *local* activity at the call-node: how top-down and bottom-up control and data reach the call-node is abstracted away by the recursion.

With this recursive framework, programs can be written as recursive rules, recursive objects, or as recursive functions. All three of these programming styles were used in this paper. These recursive specifications are easy to write, read, modify, test, and maintain. With naive evaluation, however, execution may be highly inefficient.

For efficient execution, a CGC interpreter can be developed. Then the efficiencies of highly optimized graph data structures can be exploited during recursive evaluation. Importantly, the programmer can retain his simple specifications: the CGC implementation automates the graph-structured optimizations during execution.

7.4 Constructing RETE

Conjunctive match, an important algorithm in AI, can be easily understood and implemented with recursion. As detailed in this paper, recursive specification allows intuitive development of a wide variety of recursive formulations. Extensions to RETE, such as partial matching or nested conjuncts, are also readily specified. And simple programs can be easily written in any recursive style to implement these specifications.

Call-graph caching can transform recursion into networks, separating the network construction from its bottom-up evaluation. With this modularity, one mechanism for bottom-up evaluation

can be incremental set update. Such an incremental update module can be incorporated into a general CGC interpreter.

Executing the recursive conjunctive match specifications with an incremental CGC interpreter implements the RETE match algorithm. Any topology for RETE can be built just by modifying the specification. Alternative update mechanisms can be developed for the general CGC interpreter, and tested and shared with many incremental algorithms besides RETE. CGC, therefore, enables rapid development and exploration of the entire family of RETE algorithms for conjunctive match.

RETE is just one family of algorithms that employs incremental network update. Interestingly, incremental network algorithms are just one application of Call-Graph Caching. Perhaps, in the long run, CGC can bring conceptual clarity and implementational efficacy to many other AI methods and algorithms.

Acknowledgment

Conversations with Jaime Carbonell were useful in elaborating these ideas. Caroline Hayes, Milind Tambe, and David Steier provided valuable feedback on earlier drafts. Using our CGC processor for LISP function programs, T. Paul McCartney reimplemented the CONTRAST engine, and Peter Gaertner built the visual rule programming system.

References

[1] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press, 1985.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.

[3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1986.

[4] B. G. Buchanan and E. H. Shortliffe, ed., *Rule-Based Expert Systems*. Reading, MA: Addison-Wesley, 1984.

[5] R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," *J. of the ACM*, vol. 24, no. 1, pp. 44-67, 1977.

[6] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence*. Reading, MA: Addison-Wesley, 1985.

[7] J. Earley, "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM*, vol. 13, no. 2, pp. 94-102, 1970.

[8] C. L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intell.*, vol. 19, no. 1, pp. 17-37, 1982.

[9] C. L. Forgy and J. McDermott, "OPS: A domain-independent production system language," in *Proc. of the Fifth Int. Joint Conf. on Artificial Intelligence*, IJCAI, 1977, pp. 933-939.

[10] Y. Futamura, "Partial evaluation of computation process - an approach to a compiler-compiler," *Comp. Sys. Cont.*, vol. 2, no. 5, pp. 45-50, 1971.

[11] A. Gupta, "Parallelism in Production Systems," Ph.D. dissertation, Department of Computer Science, Carnegie Mellon University, 1986.

[12] M. A. Kelly and R. E. Seviora, "An Evaluation of DRETE on CUPID for OPS5 Matching," in *Proc. of the Eleventh Int. Joint Conf. on Artificial Intelligence*, Detroit, Michigan, Morgan Kaufmann, August, 1989, pp. 84-90.

[13] D. E. Knuth, "Semantics of Context-Free Languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127-145, 1968.

[14] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1973.

[15] N. J. Nilsson, *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga Publishing Co., 1980.

[16] R. Paige and S. Koenig, "Finite Differencing of Computable Expressions," *Trans. Prog. Lang. Sys.*, vol. 4, no. 3, pp. 402-454, 1982.

[17] M. S. Paterson and M. N. Wegman, "Linear unification," *J. Comput. Syst. Sci.*, vol. 16, pp. 158-167, 1978.

[18] M. W. Perlin, "Reducing Computation by Unifying Inference with User Interface," Carnegie Mellon University, Tech Report CMU-CS-88-150, June, 1988.

[19] M. W. Perlin, "Transforming Programs into Networks: Call-Graph Caching, Applications and Examples," Carnegie Mellon University, Tech Report CMU-CS-88-202, December, 1988.

[20] M. W. Perlin, "Call-Graph Caching: Transforming Programs into Networks," in *Proc. of the Eleventh Int. Joint Conf. on Artificial Intelligence*, Detroit, Michigan, Morgan Kaufmann, August, 1989, pp. 122-128.

[21] M. W. Perlin, "Constraint-Based Specification of Production Rules," in *IEEE International Workshop on Tools for Artificial Intelligence*, Fairfax, VA, IEEE Computer Society Press, October, 1989, pp. 332-338.

[22] M. W. Perlin, "A Call-Graph Caching Evaluator for Functional Programs," School of Computer Science, Carnegie Mellon University, Chapter 2 in Tech Report CMU-CS-90-132, May, 1990.

[23] M. W. Perlin, "Scaffolding the RETE Network," in *Second Int. Conf. on Tools for Artificial Intelligence*, Washington, D.C., IEEE Computer Society, November, 1990.

[24] M. W. Perlin, "Scoping, Inheritance, and Frames," School of Computer Science, Carnegie Mellon University, Tech Report CMU-CS-90-114, February, 1990.

[25] M. W. Perlin, "Topologically Traversing the RETE Network," *Applied Artificial Intelligence*, vol. 4, pp. 155-177, 1990.

[26] M. W. Perlin, "Transforming Conjunctive Match into RETE," School of Computer Science, Carnegie Mellon University, Chapter 1 in Tech Report CMU-CS-90-132, May, 1990.

[27] M. W. Perlin, "LR Recursive Transition Networks for Earley and Tomita Parsing," in *Proceedings of the 29th Association for Computational Linguistics Meeting*, Berkeley, CA, June, 1991.

[28] M. W. Perlin, "RETE and Chart Parsing from Bottom-Up Call-Graph Caching," Carnegie Mellon University, submitted to conference, April, 1991.

[29] M. W. Perlin, "CACHE™: a Color Animated Call-grapH Environment," ver. 1.3, Common LISP MACINTOSH Program, Pittsburgh, PA, © 1990.

[30] M. W. Perlin and P. Gaertner, "A Graphical Constraint-Based Production System Environment," in *Second Int. Conf. on Tools for Artificial Intelligence*, Washington, D.C., IEEE Computer Society, November, 1990.

[31] M. W. Perlin and E. Kanal, "Goal Directed Magnetic Resonance Imaging: Clinical Practice and Computer Implementation," in *Advances in Magnetic Resonance Imaging, vol. 1,* E. Feig, ed. Norwood, NJ: Ablex Publishing, 1989, pp. 183-224.

[32] T. W. Reps, *Generating Language-Based Environments*. Cambridge, MA: The MIT Press, 1984.

[33] M. I. Schor, T. P. Daly, H. S. Lee, and B. R. Tibbitts, "Advances in RETE Pattern Matching," in *Proceedings of AAAI-86*, Philadelphia, AAAI, 1986, pp. 226-232.

[34] S. J. Stolfo and D. E. Shaw, "DADO: A Tree-structured Machine Architecture for Production Systems," in *Proc. Nat. Conf. on Artificial Intelligence*, AAAI, August, 1982, pp. 369-388.

[35] M. Tomita, *Efficient Parsing for Natural Language*. Kluwar Publishing, 1986.

[36] J. P. Tremblay and P. G. Sorenson, *The Theory and Practice of Compiler Writing*. New York, NY: McGraw-Hill, 1985.

[37] D. A. Waterman and F. Hayes-Roth, ed., *Pattern Directed Inference Systems*. New York: Academic Press, 1978.

[38] J. M. Wertheimer, "Derivation of an Efficient Rule System Pattern Matcher," Masters thesis, M.I.T., June, 1989.

[39] T. Winograd, *Language as a Cognitive Process, Volume I: Syntax*. Reading, MA: Addison-Wesley, 1983.