Transforming Programs into Networks: Call-Graph Caching, Applications, and Examples

Mark W. Perlin
December 1988
CMU-CS-88-202 2

Abstract

There are computer programs that use the same flow of control when run on different inputs. This redundancy in their program execution traces can be exploited by preserving suitably abstracted call-graphs for subsequent reuse. We introduce a new programming transformation *Call-Graph Caching* (CGC) which partially evaluates the control flow of sets of such programs into a network formed from their call-graphs. CGC can automatically generate efficient state-saving structure-sharing incremental algorithms from simple program specifications. As an example, we show how a straightforward, inefficient LISP program for conjunctive match is automatically transformed into the RETE network algorithm. Simple and understandable changes to elegant functional (and other) programs are automatically translated by CGC into new efficient incremental network algorithms; this abstraction mechanism is shown for a class of conjunctive matching algorithms. We establish criteria for the appropriate application of CGC to other AI methods, such as chart parsing, consistency maintenance, and analogical reasoning. Detailed pedagogical examples are also presented.

This publication was supported in part by grant R29 LM 04707 from the National Library of Medicine.

Table of Contents

List of Figures

1. Introduction

Caching is a general efficiency mechanism for exploiting redundancy in computation by reusing previously computed information [3, 21]. For example, AI systems often cache problem solving experience learned over time, or *knowledge*, as a set of programs. Intelligent *interactive* systems apply this program set over repeated cycles of interaction with external data input. We define as *persistent* those intelligent interactive systems that apply all knowledge to all data on every cycle. Straightforward implementations of persistent systems are often inefficient, since only some programs in the knowledge base are relevant each cycle, and potentially reusable intermediate computations are discarded.

We have identified a key source of computational redundancy: a program's flow of control, or "call-graph"[1] structure. This control-flow redundancy can be exploited by caching call-graphs.

In this paper, we shall:

1. Present a new program transformation *Call-Graph Caching* (CGC) that partially evaluates and then reuses a program's control structure. We
    a. Provide applicability conditions for the use of CGC.
    b. Present a working prototype EVAL' which performs partial evaluation of LISP programs into call-graphs.
    c. Show how CGC helps in the conceptual and implementational derivation of efficient state-saving structure-sharing incremental network algorithms.
    d. Illustrate how CGC exploits the fixed call-graph structure to reverse the flow of computation: data can flow up the call-graph, instead of always moving top-down from the program. Thus program execution can be driven from incremental changes in input data, instead of being rigidly preset.
2. We demonstrate the utility of CGC by:
    a. Using CGC to transform a simple, inefficient LISP program for conjunctive matching into the classic RETE match algorithm [8]. This also demonstrates the use of our automatic transformation prototype EVAL'.
    b. Showing other applications of CGC in AI, such as indicating how small changes in our conjunctive match LISP program can mechanically generate alternative network join topologies. Such transformations enable researchers to spend minutes modifying short functional programs, instead of months engaged in low-level network programming.
    c. Suggesting that CGC is uniquely suited to AI, in that it potentially increases the efficiency of persistent knowledge-based systems.

After introducing the CGC transformation (Section 2), we show how to transform matching programs into RETE networks (Section 3). We then present a number of other uses of CGC in AI (Section 4). An extended pedagogical development of CGC on several very simple programs is given in Appendix A.

---

[1] A call-graph [25] is a trace of an executing program's flow of procedural control. With recursive languages like LISP, this is the explicit tree of a program's dynamic procedure calls.

2. Call-Graph Caching

Call-graph caching is a program transformation from arbitrary programs[2] into network programs. After defining our notion of "program" (Section 2.1), and establishing what "network programs" are (Section 2.2), we motivate how the control flow of a program can be cached into a network (Section 2.3). We then present the mechanics of the transformation (Section 2.4), and give a sample implementation (Section 2.5).

2.1. Programs

Programming languages provide *control* and *data* constructs for writing programs [20]. For example, LISP's control mechanism is depth-first recursion, while its principal data (and code[3]) constructs are symbols and lists. At a finer grain, there is also the evaluator which glues the control and data together (e.g., LISP's EVAL program), and the distinctions between data/code organization, local data, and input data. To execute a program in a language, the control mechanism traverses the program's data organization, applying the evaluator to local and input data. For example, LISP recursively traverses a program's code tree, applying EVAL to the tree node's lambda expressions and arguments. In presenting Call-Graph Caching, we shall largely focus on the coarser-grained control and data language features.

2.2. Network Programs

Network-based programs may be considered to be members of a simple programming language. The common *data* construct is a network, or directed acyclic[4] graph (DAG). The *control* mechanism for these networks is Partial Order enumeration, usually implemented with some version of topological sorting [13]. Partial Order enumeration of a DAG assures that every node is visited exactly once: after each of its predecessors have been visited. We term this class of programs NETWORK.

Within the overall DAG-structured data organization of NETWORK programs, there is the finer level of local node and input data. DAG nodes may contain memory for both active data (executable *code*) and passive information. The passive memory has two parts, a short-term *buffer* and a longer-term *store*, the use of which is shown below. Input data is usually acted on by a node's code, in the context of the local memory state; the result of these computations on input data is often stored in the node's memory.

NETWORK programs are very efficient, with overhead at most linear in |DAG|, the number of nodes and edges [12]. Propagating from all the leaves, and keeping newly computed values in nodes' buffer memory, topological sorting assures that each node is recomputed exactly once [19]. If changes are

---

[2]Our initial focus in Sections 2 and 3 is on functional-style LISP programs. In Section 4 we look at other programming language styles, such as rules.

[3]In this paper, we shall consider a program's code to be part of its language's *data* organization, rather than part of any control construct.

[4]Graphs with cycles can also form the data component, as long as these cycles are broken at run time; i.e., the graph *acts* like a DAG.

made to only a subset of DAG leaves, using the store memory (which persists between change cycles) the computation need only be propagated to the transitive closure DAG subset AFFECTED, |AFFECTED| ≤ |DAG|. This produces *state-saving incremental* algorithms that perform minimal recomputation, directed from just the *changes* to their input.[5]

The price for this efficiency is inflexible control structure: NETWORK program DAGs have the restricted form of basic blocks [2]. Despite this limitation, NETWORK finds extensive application. Efficient incremental spreadsheets use the algorithm sketched above. Conjunctive matching programs are often cast in network form for efficiency. Other potential applications include the representation of dependency relations, multiple inheritance, consistency maintenance reasoning [6], and incremental attribute grammar evaluation [22].

2.3. Caching Control Flow

Let P(t,u) be a program in some language, having arguments t and u. Suppose that P's control structure is independent of u. Then partial evaluation [10] of P(t,u) with respect to some fixed t_0 will yield a new program P'(u) of one argument having a *unique* call-graph.

An interactive program's input either depends on external factors, or is independent of external interactions. Now suppose that program P(t,u) is interactive, and its argument t is independent of external factors. Then P'(u)

1. completely characterizes P(t,u)'s interactions with the external environment, and
2. has a unique call-graph.

We define an interactive program P(t,u) to be **basic** when t is independent of external factors, and P's control structure is independent of u.

P'(u) has a unique call-graph which can be cached as a NETWORK program for later reuse. This is accomplished by providing the call-graph nodes with memory (completing the data component), and then inverting the flow of control from top-down to bottom-up. Partial Order traversal supplies the new control component. The new NETWORK program is operationally equivalent to P. As illustrated in Figure 2-1, the call-graph network has captured all the control and much of the data structuring of the original program.

2.4. The Transformation

We describe the Call-Graph Caching (CGC) transformation in several loosely coupled steps. The first step assembles the call-graph from its subgraph components (Section 2.4.1). The second collects a set of call-graphs into a network (Section 2.4.2). The resulting cached call-graph network structure is used (and reused) as a data cache (Section 2.4.3). Appendix A presents simple examples of these steps.

---

[5]Further efficiency gains are possible by restricting the DAG traversal to the subgraph influencing only select node computations.

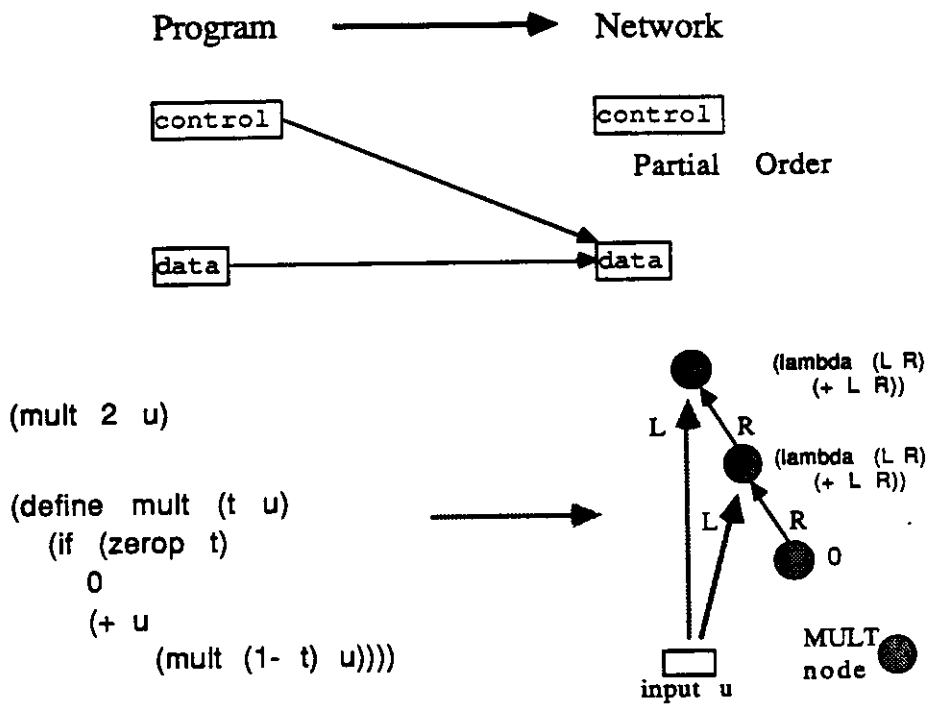

Figure 2-1: Compiling control into a network.

2.4.1. Building the Call-Graph

Control-Flow Caching is an algorithm which builds the call-graph of a program on some input. The construction takes as

- **Input** either
  1. the compile-time text of a program, together with its partial input, or
  2. the run-time program executing on its complete input, and
- **outputs** the call-graph of that program.

The procedure employs an auxilliary data structure, the *Control-Flow Cache*, which is used in the assembly of the final call-graph structure. There is also an optional argument specifying the key execution steps to cast into graph nodes.

Control-flow caching proceeds as three separate steps. **First**, with a program, (partial) input, and a set of the key steps to abstract[6], a trace is formed of the program's execution. Each node in the resulting call-graph represents one (key) step in the program's trace. The actual formation of the call-graph is facilitated by specific control-flow cache management strategies. One such strategy is the above abstraction mechanism of recording only the "key" steps as nodes. Another strategy, used in chart parsing [27], is exploiting the constraints posted in the control-flow cache to help reduce the executing

[6]This user-definable set ranges from the empty set to all possible steps.

program's computation, i.e., dynamic programming. Yet another, say for a LISP program, would be to passively cache the succession of execution branches into a full call tree. Regardless of the specific strategies, the resulting call-graph captures (in space) the program's execution over time, as shown in Figure 2-2, step 1.

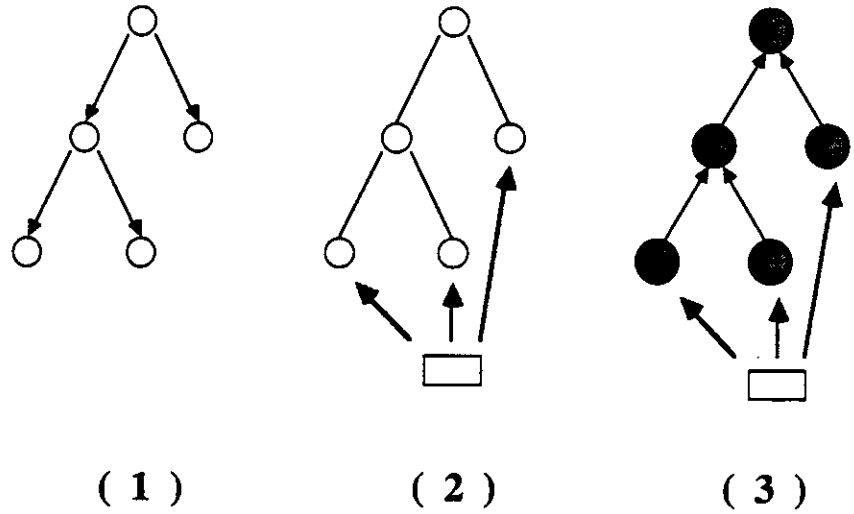

Figure 2-2: Control-Flow caching: assembling the call-graph.

Program P(t,u) has thus far been evaluated on input t_0, with the partially evaluated P'(u) preserved as a call-graph. The **second** step of Control-Flow Caching connects the external input u to the program's graph. As shown in Figure 2-2, step 2, the resulting call-graph of the complete program is a DAG[7].

The **third** step *operationalizes* the call-graph into a usable data structure. For example, graph nodes can be augmented with the requisite buffer and store memory with a system-specific representation.

At this point, saving and reusing *just this single* call-graph provides a fully functional state-saving incremental NETWORK program. Directing input u through the graph in a Partial Order enumeration, and using the node memories as a data cache, will efficiently perform the computations of $P(t_0,u)$.

2.4.2. Collecting Call-Graphs

The Call-Graph Caching transformation is completed by collecting a *set* of $P(t_0,u)$ call-graphs, for a variety of P's and t_0's. This set is called the *Call-Graph Cache*, and, like other caches, usually employs efficient cache management strategies. For example, spreadsheets and conjunctive matchers exploit common shared structure, and merge the call-graphs into a single connected network. In allocating the often limited resource of space, another common strategy is to perform a cost/benefit analysis, determining which call-graphs stay in the cache, and which are removed.

---

[7]This is because the caching of the program's execution over time breaks (i.e., unravels) any cycles present in the flow of control.

2.4.3. Using the Call-Graph Cache

After building the call-graphs of $\{P(t_i,u) \mid i \in I\}$ and collecting them into a call-graph cache, the resulting network is used (and reused) as a data cache. Values or sets of values of u propagate bottom-up through the network, employing the "buffer" memory within each propagation cycle, and "store" memory between cycles. For efficiency and correctness, the network traversal control mechanism is Partial Order enumeration (implemented with a topological sort). This completes the transformation of a finite set of basic programs (in any programming language) into an efficient state-saving NETWORK program.

2.5. An Implementation

To demonstrate the workability of Call-Graph Caching, we implemented in Common LISP a simple partial evaluator EVAL', which transforms a large class of LISP programs into their corresponding call-graphs. The input to EVAL' is the symbolic LISP expression representing "$P(t_0,u)$", for some P and t_0, and a set of labels denoting the key execution steps to cache. The output is a call-graph, where each node specifies

- the label of the node type;
- a lambda expression containing all the information required to execute the node's computation when applied to the values of its immediate predecessor nodes;
- recursively, the nodes of its immediate predecessors.

EVAL' performs the following computations:

1. Arrive at (the label of) a key node (i.e., LISP function or symbol) to be abstracted.
2. Perform EVAL' on the unevaluated arguments to the function.
3. Substitute these values into the function, and then execute EVAL' on the LISP function's code tree.

This delayed evaluation is done recursively, caching the control structure into a call-graph. The call-graph's nodes abstract out the set of labelled functions, preserving the local code and data required for later execution.

We have also developed a variety of network structure-sharing programs for the merging of call-graphs. To execute these Cached Call-Graph networks as programs, we have a toolkit for general efficient Partial Order network traversal.

3. RETE Networks: An Example of Call-Graph Caching

RETE matching [9] is a state-saving structure-sharing incremental algorithm used in OPS-5 [7] and other production systems [14] for conjunctively matching many patterns against many objects. Because it provides excellent average-case behavior for an important NP-hard AI problem, it has been extensively studied and varied. It is also a good example of the Call-Graph Caching program transformation, illustrating nontrivial usage of the Control-Flow Cache and the Call-Graph Cache.

3.1. Rule Matching

Forward-chaining Rule Systems (or "production systems", such as OPS-5) are programming languages with *match* as their control element. Program data is organized into a set of rules, having left-hand-side (LHS) *tests* and RHS actions. External input (often called "working memory") comes from a slowly varying set D of data *objects*. If a rule's tests match objects, the rule becomes a candidate for firing; when executed, its actions serve to modify D.

Following common practice, we fix the form of the rules' LHSs to be a set of conjunctive *patterns*, each pattern containing tests restricting the set of matchable objects. An *instantiation* of a rule having n LHS patterns is an n-tuple of objects satisfying the rule patterns' tests. On every interaction cycle, the rule evaluator must try to match each rule against all possible combinations of objects in D, forming its set of instantiations

$$\{inst \in D^n \mid \forall test \in Rule,\ test(inst)\},$$

where D^n is DxDx...xD, n times. Letting TEST be the set of allowable tests, and TESTS be its power set 2^{TEST}, *Match* may be described as a function which filters the set of possible instantiations D^n into those satisfying some tests in TESTS,

$$Match'(TESTS, D \times D \times D..D) \rightarrow D \times D \times D...D.$$

Since all patterns are matched against all objects, we may rewrite *Match* in the more convenient form

$$Match(TESTS, D) \rightarrow D \times D \times D..D,$$

which applies a set of tests to a set of objects, producing a filtered set of n-tuple instantiations.

Production systems are *persistent*, in that they

1. maintain their knowledge in a finite set of experientially derived programs (the rule set), and
2. apply all programs to all available data on each interactive cycle.

They are also *inefficient*. Consider just one rule having n patterns matching against only two data objects- the set of candidate instantiations D^n grows exponentially in n. In fact, conjunctive rule matching is NP-hard [15]. Generally, however, only a small fraction of the object set changes each cycle. So instead of rigidly applying *all* rule programs to the data, perhaps the incremental *changes* to *data* should drive the matching computation.

For each rule program in the rule set specifying some tests in TESTS, *Match*(tests,D) is the computationally expensive subprogram. Observe that with

P = *Match*, t = tests, and u = D,

1. t is independent of the external data input D, and
2. the conjunctive matcher P(t,u) is programmable so that P's control is independent of u (e.g., Section 3.2).

Therefore P is *basic*, and Call-Graph Caching will generate an efficient state-saving structure-sharing incremental *Matching* algorithm.

3.2. Transforming Rule Matching into RETE Networks

We now illustrate the use of Call-Graph Caching by generating the RETE network from a simple functional programming specification of the matching function. We first write down the functional program (Section 3.2.1), then transform each match program into its unique call-graph (Section 3.2.2), and finally merge these call-graph units into a RETE network (Section 3.2.3). Section 3.2.4 summarizes the RETE construction, and the network's processing of data objects.

3.2.1. Rule Matching as LISP Programs

A rule specifies a fixed set of tests T_0 for its match component. The conjunctive match program *Match*(T_0,D) can be formulated so that its control is independent of working memory D. We now write such a filter *Match* as a functional-style LISP program.

For efficiency on a serial processor, we first impose a fixed ordering on the rules' patterns. Each test examines one or more objects in a candidate n-tuple $\in D^n$; these objects are now ordered by the pattern ordering. We now order the test set: associate to each test the number of the *last* object it examines, and arrange the tests with respect to this index. For efficiency, the match is performed by a conditional AND, testing a candidate n-tuple against the first test subset, then the second, and so on through the n^{th}. Within the k^{th} test subset, $k \leq n$, the tests may be further grouped into two classes:

A. *alpha* tests on a single object $\in$ D, and

B. *beta* tests on more than one object, i.e., k-tuples $\in D^k$.

There are many ways to write the LISP code for this simple filter (e.g., as one function, iteratively, recursively, etc.). While the CGC transformation is independent of programming style, for clarity, we present *Match* using linear recursion.

```
;;; Match the rule's tests against the data.
(define match (tests data)
  (beta-join
      (first tests)
      (second tests)
      data))
;;; Joins together the cross product of preceding
;;; sift and join sets with a filtering beta test.
(define beta-join (A B D)
  (if (null A)
    '(())
    (filter (first B)
            (cross-product
                (alpha-sift (first A) D)
                (beta-join (rest A) (rest B) D)))))
;;; Sifts out objects with a filtering alpha test.
(define alpha-sift (A D)
    (filter A D))
```

Match takes a preordered set of tests, and a set of data objects as its arguments. The key interesting function is *beta-join*, which merges the simple *alpha-sift* filter with further recursive calls to *beta-join*; this produces a linearly recursive call-graph. Note how the tests in *match*'s *tests* argument are deposited

locally at each level of filtering, and that no control decisions are made using the *data* argument.

The auxilliary functions *filter* and *cross-product* may be defined, for instance, as:

```
;;; FILTER removes elements of the SET not satisfying TEST.
(define filter (test set)
    (remove-if-not test set))
;;; CROSS-PRODUCT augments the (k-1)-tuples of set Y with the
;;; elements of set X, forming the cross product k-tuples.
(define cross-product (set-X set-Y)
    (mapcan #'(lambda (y)
                (mapcar #'(lambda (x)
                            (cons x y))
                        set-X))
            set-Y))
```

3.2.2. Building the Call-Graph

STEP 1. For any rule r, calling EVAL' on *match*(tests$_r$,D) with the set of labels {match, alpha-sift, beta-join} will save the calling structure of the rule's tests. The control-flow cache is used with the functional program *match* to store to the growing call tree. As shown in Figure 3-1 A, the call-graph has a linear spine, with the appropriate tests localized at each node.

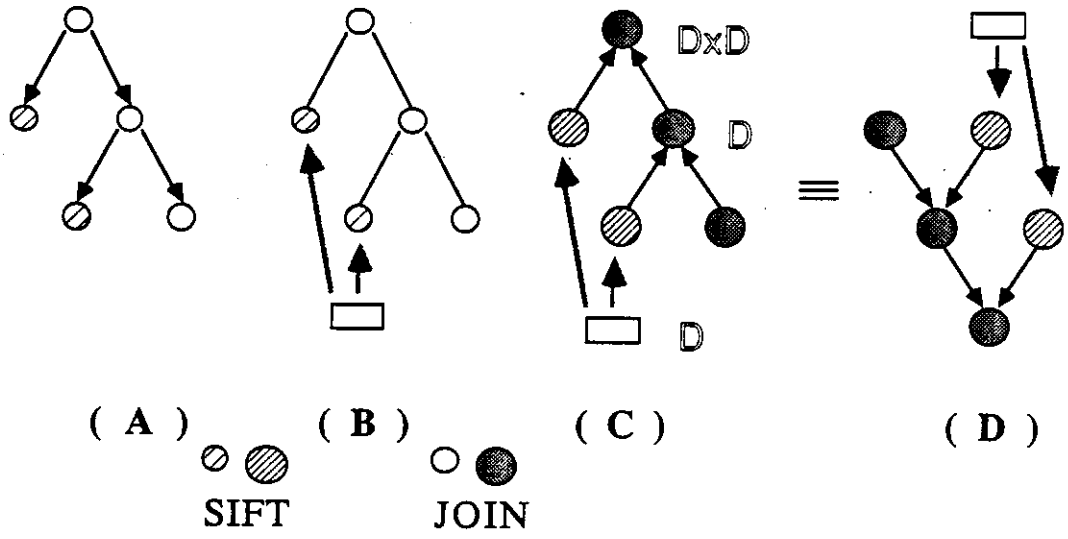

Figure 3-1: Call-Graph Caching a linearly recursive conjunctive matcher.

STEP 2. In Figure 3-1 B, the free input variable *data* is attached to the call-graph as an input source, turning the call tree into a DAG.

STEP 3. The graph structure of this single rule's match component can now be completed. Memory for the intra-cycle buffer and the inter-cycle store (and other information) can be specialized into a specific graph representation. This call-graph can be reused as a bottom-up NETWORK filtering program. In Figure 3-1 C, the domains of the filtered objects are shown.

Using the buffer memory, Partial Order traversal of the call-graph from the data computes the filtered instantiation subset of D^n. If the nodes' longer-term store memories are initially loaded with D (and then continually updated), only *changes* to the object set ΔD are needed for computing further instantiations. That is, the call-graph is a *state-saving* data-driven incremental NETWORK program for computing the state of a single rule's match.

This is not surprising: Figure 3-1 D shows that the RETE network beta join topology is isomorphic to the *Match* call-graph. The (empirically less important) RETE alpha discrimination network is easily added to this picture by substituting tail recursive *alpha-test* and *alpha-memory* functions for for the *alpha-sift* filter. We have generated these (and other) call-graphs by applying our prototype EVAL' (see Section 2.5) to various functional match programs; the resulting call-graphs are readily inserted into preexisting OPS-style production systems.

3.2.3. Collecting Call-Graphs

A rule system is comprised of a finite *set* of rules; we therefore form the corresponding set of call-graphs, one for each rule's match component. This set is the Call-Graph Cache. One cache management strategy for conserving cache space (with some associated speedup) is to *merge* the call-graph DAGs into one connected network. The matcher's behavior is unchanged if, proceeding from the data input source, nodes are merged based on prefix *sharing of tests*. A succession of call-graphs merging into a common beta-join node trie [1] is depicted in Figure 3-2.

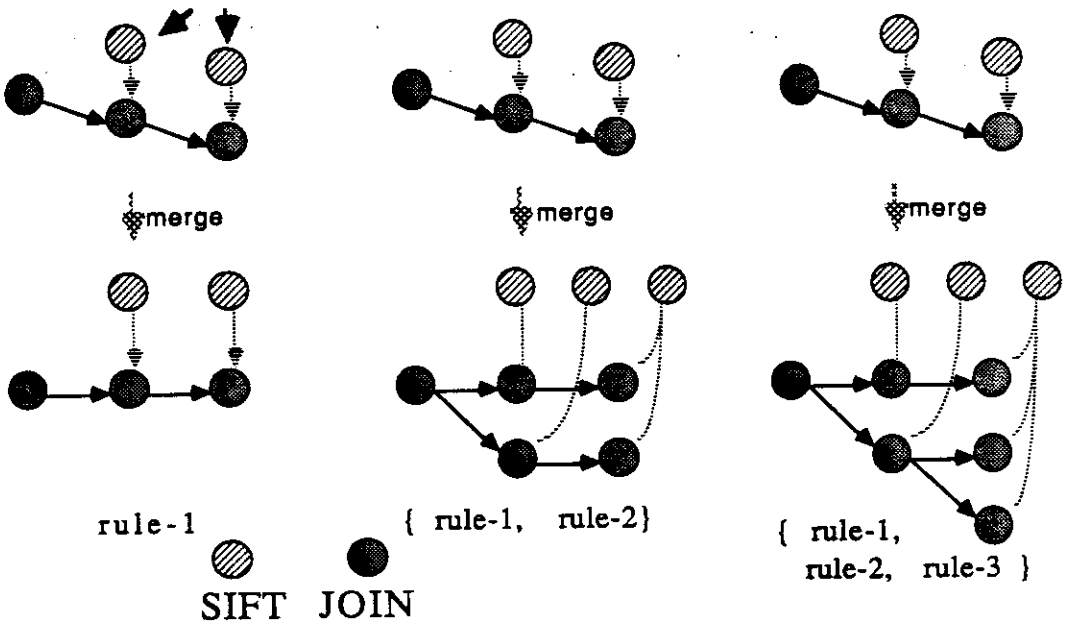

Figure 3-2: Trie-based merging of call-graphs in the Call-Graph Cache.

The alpha sift trie, and the full RETE network having both alpha and beta tries, are shown in Figure 3-3.

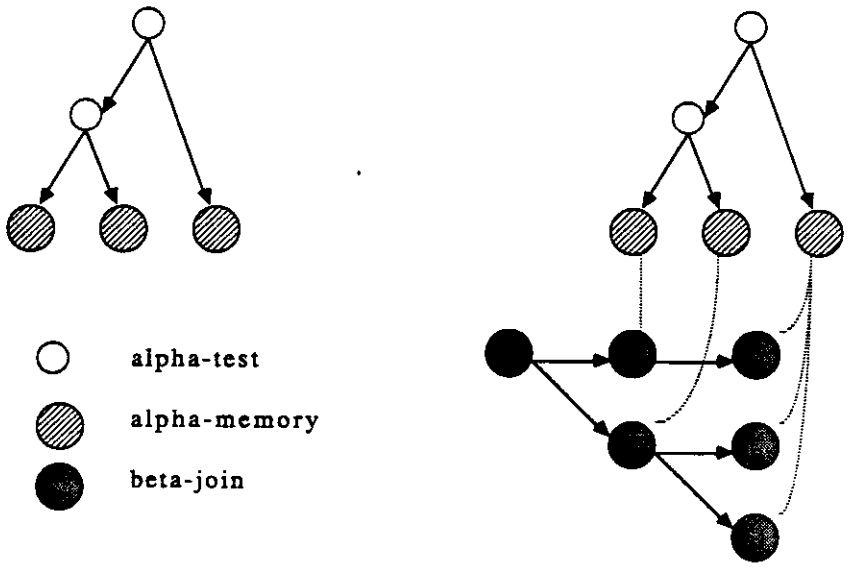

Figure 3-3: The alpha trie; the full RETE network.

3.2.4. Using the Call-Graph Cache

Partial Order traversal of the RETE network will perform the match of the rule set (cached as a shared set of call-graphs) against working memory input D. The intracycle "buffer" and intercycle "store" memories at each node are used as a *data cache* to preserve the partial match computations (within and between) each cycle.

This completes the construction of the RETE network algorithm.

1. We started from an easily specified, though inefficient, set of functional programs.
2. Using an auxilliary control-flow cache, partial evaluation of the basic program *match*(T_0,D) produced a call-graph which cached *match*'(D)'s control flow. This call-graph was usable as an incremental data-driven *state-saving* NETWORK program which could cache the input data as intermediate matching results.
3. The Call-Graph Cache merged the individual call-graphs in order to conserve space, and achieve some speedup. This was done by *test sharing*: nodes with common test prefixes were combined to form a single trie data structure.

3.3. Alternate Join Topologies

Call-Graph Caching generates more than RETE networks: one application is the generation of families of efficient conjunctive matchers. By making simple variations in *match*'s LISP specification, and changing which key function names are cached into call-graph nodes, many different join topologies can be designed, easily specified, and automatically constructed. Also, there are other Call-Graph Cache merging strategies besides trie-based prefix sharing.

The RETE example was described above: a linearly structured call graph. One known alternative approach is to *not* cache the beta join nodes [16]. Another is to structure the call graph as binary

tree [24, 18], reducing the long linear chains problematic in RETE. We are currently exploring and assessing a variety of new join topologies using CGC as a rapid prototyping tool. These topologies can be custom tailored to task-specific requirements, such as learning or parallelism [11].

4. Other Uses

We are investigating other uses of CGC in AI. We describe the application of Control-Flow Caching (Section 4.1) and the complete Call-Graph Caching transformation (Section 4.2) to several problems. We then examine (Section 4.3) ways of weakening the definition of *basic* programs to increase the applicability of CGC. Section 4.4 concludes with some observations on future research in AI and software engineering using the CGC transformation.

4.1. Control-Flow Caching

The control-flow cache is an auxilliary data structure that assists in the construction of a program's explicit call-graph. It may simply record the unraveling of an execution tree over time, as in the RETE example, or take a more active role, such as enabling constraint-directed dynamic programming. For example, in efficient context-free parsing [27], each graph node represents an individual firing of a grammar production; the control-flow cache (or "chart") records past firings to constrain future ones.

Letting the graph nodes represent a rule firing, the call-graph resulting from a rule system's problem solving instance forms a trace of the rules' executions. This record (DAG) of the rule and data dependencies may then undergo Consistency Maintenance analysis [6] to explore various alternative reasoning scenarios. Here, the call-graph forms a NETWORK program exploiting its data cache; ground instance (data) changes can be incrementally propagated in Partial Order DAG traversals.

4.2. Call-Graph Caching

Persistent processes maintain their knowledge in a program cache; this cache is augmented or modified as the knowledge changes over time. When the cached programs have call-graphs with sufficient redundancy (e.g., are *basic*), the knowledge may be compiled into an efficient Call-Graph Cache network. For example, consider the rule trace discussed above: DAGs representing arbitrary rule execution are not likely to share similar morphology, and, therefore, they are not usually cached as networks. Rather, such call-graphs are abstracted into networks having a *single* inner node (or *chunk*) using EBG [17, 26] or some other execution trace generalization method. These reformed networks have sufficient operationality to then be reused in the program cache, resulting in potential efficiency improvements. (In some systems [14], they may be compiled into RETE networks at a lower level of abstraction for further efficiency gains.)

There are many common persistent processes comprised of basic programs. For example, a (multiple) inheritance network will be automatically generated as the cached call-graph set from the process of successive subset classification on some input set. As another example, window systems may be thought of as caching an inefficient "painter's algorithm" redisplay execution into a call-graph based on

the *in-front-of* relation. Efficiencies accrue since, in general, the data inside the windows is independent of window redisplay.

4.3. Extensions

CGC is applicable when *persistent* interactive intelligent systems are comprised of *basic* programs. Since not all programs are basic, we consider how to use CGC under weaker assumptions.

As in Section 2.3, let P'(u) be a partially evaluated program. P'(u) need not have a unique call-graph for CGC to be useful. As long as its set of call-graphs is manageable, some caching strategy could succeed. For instance, P'(u) might have only a small finite number of call-graphs. Alternatively, a skewed distribution of P'(u)'s call-graphs could probabilistically ensure manageability. Extending the RETE match example, when disjunction (i.e., choice) is introduced into primarily conjunctive rules, there is still much redundancy in control flow. Though weaker, this redundancy is effectively exploited in RETE-based OPS-5 via copying and conflict resolution.

Another approach is to only use part of the CGC method. For example, with script-based plans [23] and case-based reasoning, the graph is used for recognition and generation, but not as a data cache. Similarly, derivational analogy [5] preserves a call-graph to assist with future problem solving; transformational analogy [4] goes further in abstracting this call-graph into a more useful form. While both methods exploit control-flow redundancy in problem solving to cache control, neither currently caches the data state.

4.4. Future Work

The CGC transformation lets us reexamine many network algorithms as *partially evaluated programs* which have their call-graphs preserved. Further, as with RETE, it may be the case that reformulation of the network into a new program in some appropriate language leads to clearer specification of the algorithm. Since NETWORK-like efficiency is guaranteed by the CGC transformation, improvements can then be effected in the abstracted programming language, rather than in the low-level NETWORK language.

Conversely, given a clear specification of a set of programs in some language, when control-flow redundancy is present (whether guaranteed by the "basic" property, or simply empirically observed) CGC becomes another route for improving performance. Possibilities include:

1. Refining classic state-saving incremental network algorithms where call-graph redundancy has already been observed.
2. Reexamining inefficient AI architectures for reusable redundancy in control-flow.
3. Developing new and efficient persistent interactive processes by starting from precise, inefficient programs that have sufficient control-flow redundancy for the CGC transformation to succeed.

5. Conclusion

We have observed that repeated executions of programs may exhibit redundancy in their flow of control, and that program call-graphs provide an operational representation of that control. We therefore proposed that the call-graphs of certain programs executing on (partial) input be preserved for later reuse. When cached, control flow can be reversed on the fixed call-graph structure; this leads to efficient incremental algorithms responding to changes in input data.

As an example, we applied the method to a simple, but inefficient LISP program for pattern/object matching; the transformation produced the RETE match algorithm. Strong and weak criteria were provided for when other algorithms could be transformed into their efficient incremental network counterparts.

APPENDIX

A. Motivating Examples

This appendix illustrates successive examples of evaluating polynomials in a spreadsheet-style network. These examples are selected for their extreme simplicity in demonstrating the transformation of programs into networks, and are presented solely for pedagogical purposes.

CGC transforms the control and data of a program P in some language into a new NETWORK program P'. The control of P' is Partial Order enumeration, and its data component is a network built from the control and data execution of P. The particular programming language L we use in these examples is functional-style LISP specification. The control mechanism of L is recursive function application, and L's data is comprised of symbols, lists, lambda expressions, and the like. The CGC transformation can be effected at

- **run time** by EBG-like capture and generalization of program execution traces, or
- **compile time** by partial evaluation of P on some input.

Our examples demonstrate the compile-time transformation.

The specific task we will develop by the end of the fifth example is evaluating a non-changing set of multivariate polynomials over slowly varying sets of variable values. This could be used, for example, in repeatedly computing some property (say, the minimum value) of the polynomial set over iterative cycles of minor changes to the set of possible variable values.

The CGC transformation is possible for this specific polynomial evaluation task because:

1. Polynomials (and other formulas) are *basic* programs. That is, they can be evaluated as basic blocks of expressions, hence have a natural DAG representation.
2. Repeatedly applying a non-changing set of polynomial programs to sets of values is a *persistent* process.

This redundancy of program control flow (*basic* property), coupled with redundancy in which programs are selected for execution (*persistent* property), suffices for the workability in this case of the program-to-network CGC transformation. Since the value sets will be slowly varying, there will be enough redundancy in program evaluation for noticeable efficiency gains.

The mechanism of this program-to-network transformation is to build, collect, and use cached call-graphs.

Build With aid of an auxilliary Control-Flow Cache, the developing call-graph of the executing program is saved as it processes its (partial) input. The nodes of the call-graph are formed from the key function names designated for being abstracted out; other computation is abstracted away by lambda abstraction. This call-graph may then undergo further processing.

Collect The call-graphs are collected into a Call-Graph Cache network. Some sharing or merging of call-graphs may be performed.

Use The Call-Graph Cache network is then used as a data cache for bottom-up data-driven evaluation of the programs.

Section A.1 presents the call-graph **building** process in great detail, while Section A.2 shows how call-graph **collections** can be merged into tries. Similarly, Section A.3 stresses **building** call-graphs with the control-flow cache, while Section A.4 emphasizes structure sharing in call-graph *collections.* Section A.5 elaborates the **use** of the network of partially evaluated programs as a data cache.

A.1. Power Functions - x^k

Our first function computes a number x to some power, i.e., x^k. The definition of *power* is:

```
power(k,x) =    1, k=0,
                x * power(k-1,x), otherwise.
```

Power can be programmed iteratively, recursively, as an in-line program (for fixed k), with rules, with frame-based demons, etc.: the CGC tranformation does not depend on programming style. For elegance, we use a simple tail recursive LISP function[8]:

```
(define power (k x)
    (if (zerop k)
        1
        (* x
           (power (1- k) x))))
```

Abstraction is a key idea in CGC. The programs that are selected for caching are abstracted out (highlighted) into explicit graph nodes. The remaining programs are abstracted away (hidden) into procedures attached to the graph nodes. At execution time, this lambda abstraction at each node binds values for its arguments from the node's immediate local predecessors.

Suppose we wanted to transform *power(2,x)* (the squaring function) into a network, where the featured nodes were of type *power.* Figure A.1-1 traces through the transformation of the partial program call `(power 2 x)`, highlighting *power*, into an equivalent node of type *power.* We first evaluate the control flow one step: since k=2 is greater than 0, we reach the else-clause of *power*'s **if** statement, `(* x (power 1 x))`. As illustrated, this expression has a natural representation as an expression tree rooted at "*". Now all data (program code is considered to be data here), except the input *x* and the featured program *power*, are abstracted into the lambda expression

`(lambda (L R) (* L R))`.

Here "L" denotes the left argument receiving data from the node's left predecessor, the input node *x*, while "R" denotes the right argument, with data from the right (as yet unevaluated) predecessor `(power 1 x)`. The highlighted program *power* becomes an explicit node to which this lambda abstraction is attached. Evaluating this node's code, with its arguments bound to the values of its predecessor nodes (see below), will effect a bottom-up computation of *power* equivalent to the original top-down LISP specification.

---

[8]The expression `(1- k)` *decrements* k, performing the subtraction `(- k 1)`.

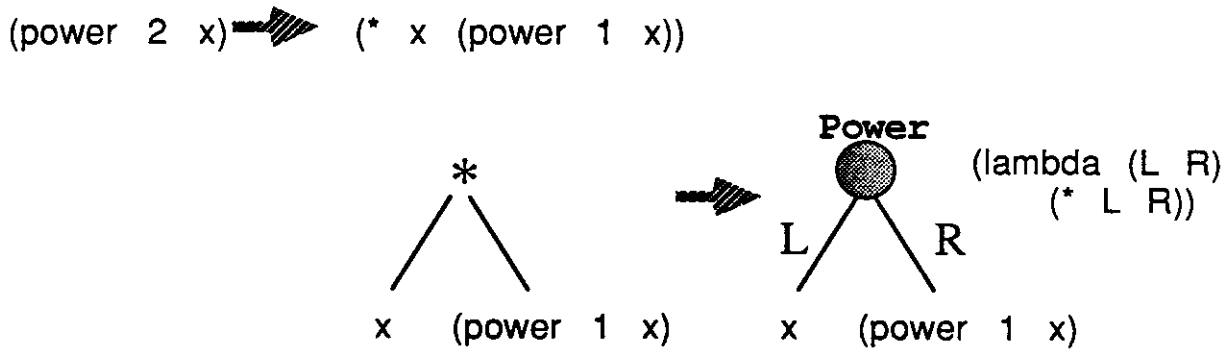

Figure A.1-1: Transforming one step of a program call into a graph node.

Continuing this recording of program control execution decisions, step-by-step we build up a call-graph (Figure A.1-2). The graph is finite exactly when a program terminates; here, when k attains the value 0, the then-clause returns the constant 1. This terminating then-clause produces the second node type in *power(2,x)*'s call-graph. We term the auxilliary storage used in caching the developing call-graph's control flow the *Control-Flow Cache*.

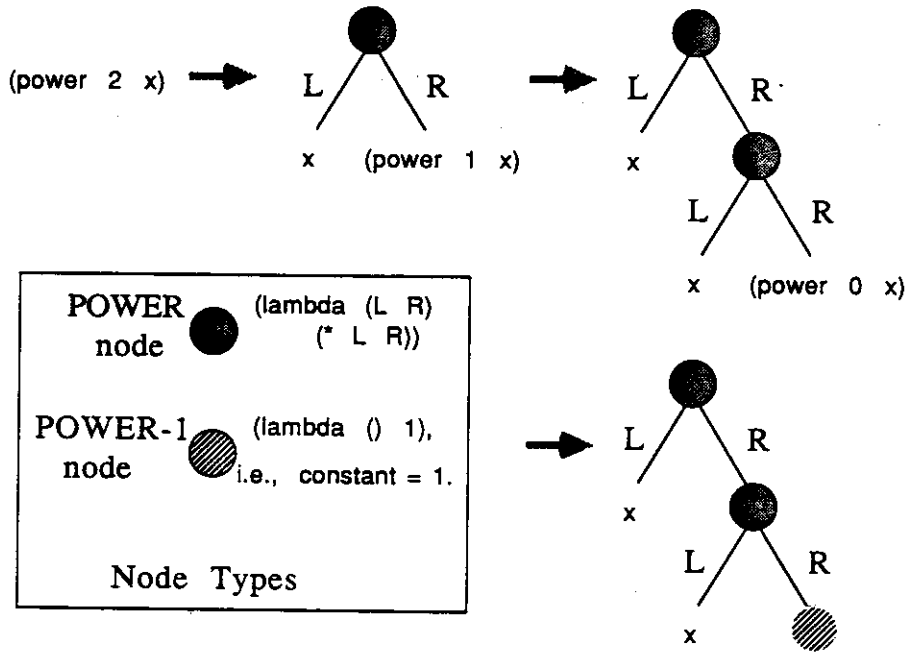

Figure A.1-2: Building a call-graph by caching control flow.

Figure A.1-3 depicts the reversal of control flow.

1. The call-graph is created top-down by caching the control flow of the program executing on partial input.
2. Sharing of the common, control-independent input is exploited, and the functional program's call-tree is converted into a DAG. This provides a common entry point for input data.
3. The control-flow direction is reversed to bottom-up data-driven computation.

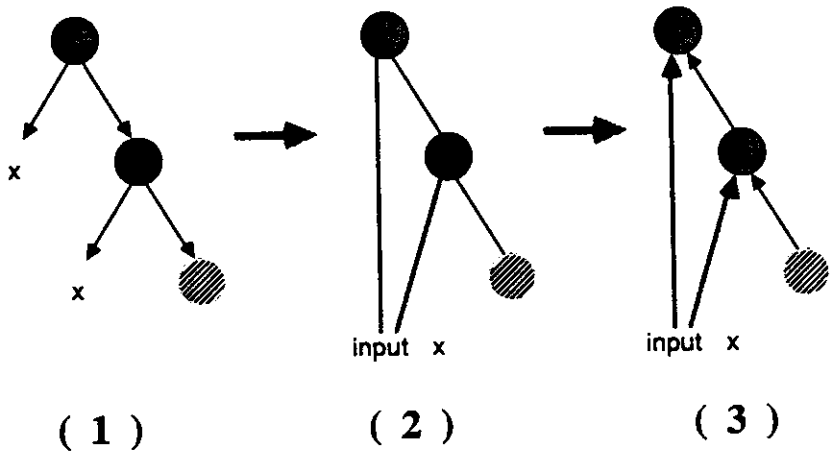

Figure A.1-3: Reversing the control-flow direction.

The completed call-graph can now be used as a NETWORK program to compute x^2. Figure A.1-4 shows the partial order control structure used to evaluate the network data structure starting from the input node value x=3. By NETWORK's partial order semantics, a node cannot be evaluated until all its immediate predecessors have been. Therefore, the top node (x^2) waits for the middle node (x^1) to evaluate first. With the middle node's left input x equal to 3, and its right *power-1* node returning the constant 1, the environment is {L=3, R=1}. Applying the lambda-expression code `(lambda (L R) (* L R))` in this environment computes the value 3. In the second evaluation step, the input value of 3 is multiplied with the middle node's computed value 3 to yield 9, the value of 3^2.

A.2. Sets of Power Functions

The preceding Section showed how redundancy in control-flow could be cached. In this Section, we examine how redundancy in which programs are selected for execution can be exploited by caching.

Suppose we wanted to repeatedly compute the functions {x^2, x^3} for different values of x. Collecting the two call-graphs of the corresponding programs into a set, we could evaluate the functions using two separate networks (Figure A.2-1).

Thus far, our *Call-Graph Cache* has helped in potentially reducing the space and time requirements of the *power* computation. With some additional processing, however, the cache can effect even greater efficiencies. By allowing the two call-graphs to share common structure, they can be merged into a single network (Figure A.2-2). This reduces the network size in the cache, and eliminates the redundant computation of x^2.

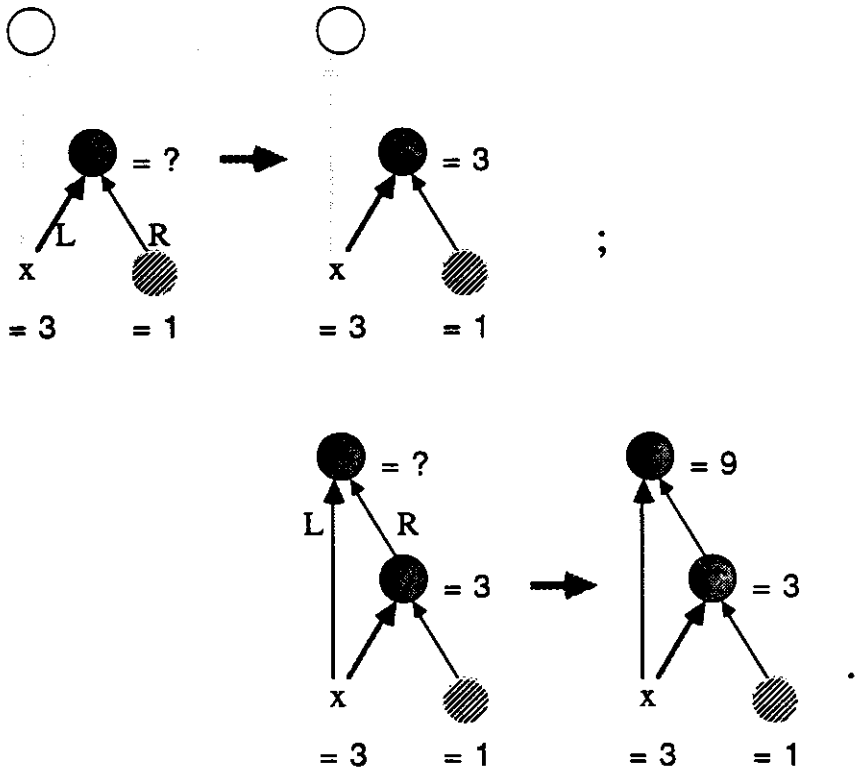

Figure A.1-4: Using the call-graph NETWORK program to compute 3^2.

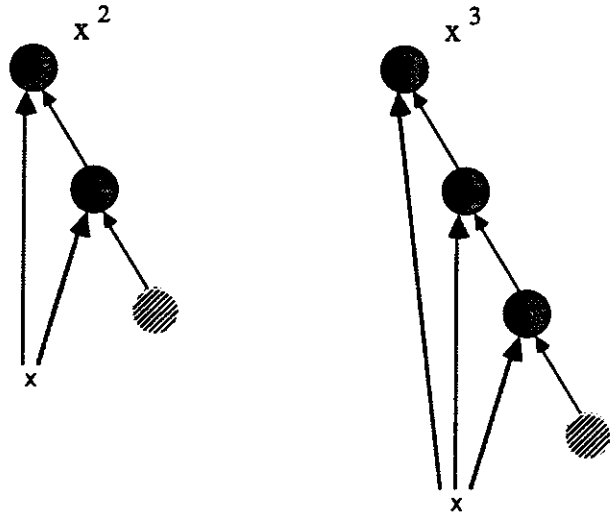

Figure A.2-1: Networks for x^2 and x^3.

The merge strategy employed in this example may be viewed as the trie-based sharing of common prefixes. Since the entire x^2 network is a prefix subgraph of x^3, the final computation for x^3 may simply be

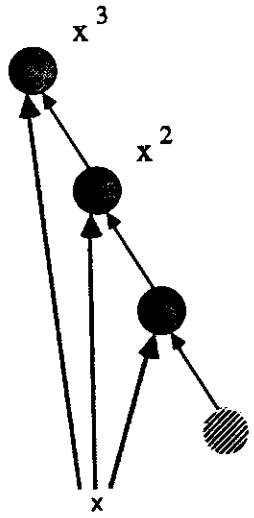

Figure A.2-2: Merging the call-graphs into a single network.

appended to x^2's network. The example readily generalizes to arbitrary sets of programs $\{x^k \mid k \in I\}$, where I is a finite index set of natural numbers.

A.3. Polynomial Functions

Consider the evaluation of a polynomial in x on some value. It may be computed by taking the inner product of a set of coefficients with a set of powers of x, e.g.,

```
(a  b  c)  · (x^2  x^1  x^0).
```

Since the set of powers depends solely on the variable x, a polynomial can be computed from a list of coefficients C and the value of x:

```
poly(C,x) = 0, if C is empty;
            otherwise:
            [ head(C) * power(length(C)-1,x) ] + poly(tail(C),x).
```

For example, $ax^2 + bx + c$ is recursively computed by:

```
poly(<a,b,c>,x)
             = a*power(2,x) + poly(<b c>,x)
             = a*power(2,x) + b*power(1,x) + poly(<c>,x)
             = a*power(2,x) + b*power(1,x) + c*power(0,x) + poly(<>,x)
             = a*power(2,x) + b*power(1,x) + c*power(0,x) + 0.
```

A functional-style LISP program computing *poly* with any set of coefficients C is:

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

Suppose, however, that we wanted to compute only a *particular* polynomial. Then we could pre-compile the coefficients into a new polynomial evaluation program. One way of accomplishing this automatically

is by using CGC to transform the above LISP program into a partially evaluated NETWORK program.

Control-flow caching of *poly(<a,b,c>,x)* produces the call-graph of Figure A.3-1. The highlighted programs *poly* and *power* are represented by explicit nodes. The other functions are abstracted away into lambda expressions such as `(λ (L R) (+ (* L a) R))`, corresponding to the expression call-tree in Figure A.3-1. These lambda abstractions are procedurally attached to the different node types, also shown in Figure A.3-1.

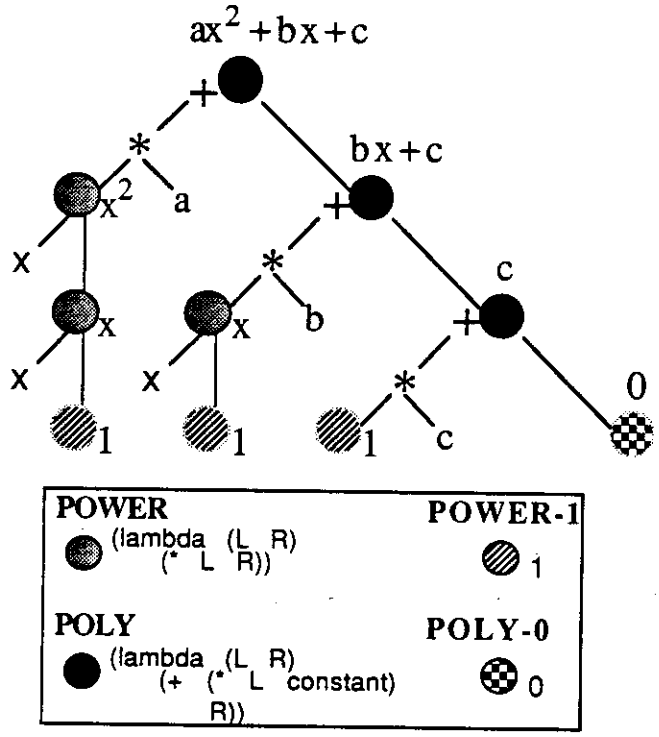

Figure A.3-1: The cached control-flow of *poly*(<a,b,c>,x), highlighting *poly* and *power*.

Intelligent management of the Control-Flow Cache can now process the call-graph of Figure A.3-1. Note that each node in the main spine of *poly* type nodes depends on a separate chain computing x^k. As in Section A.2, these individual chains can be merged into a common shared structure. The resulting (more space and time efficient) processed call-graph network is shown in Figure A.3-2.

A.4. Sets of Polynomial Functions

The previous Section focused on building and processing call-graphs with the Control-Flow Cache. We now discuss strategies for collecting sets of call-graphs for efficient later reuse, i.e., the Call-Graph Cache.

Suppose we wanted to repeatedly compute sets of polynomials on different values of x. We could simply compute each polynomial as a separate network, capitalizing only on the efficiency gains of the preceding Sections. However, if there is redundancy in the call-graphs, structure sharing may yield further time and

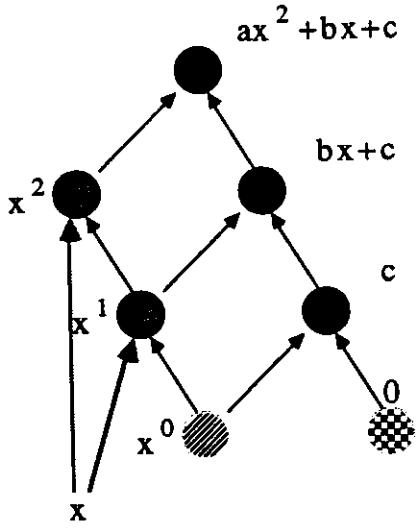

Figure A.3-2: Control-Flow Cache processing to improve the call-graph's efficiency.

space improvements. For example, suppose that starting with the call-graph of ax^2+bx+c (Figure A.3-2), we added the call-graph of another polynomial: dx+c. Since the two call-graphs share a common prefix subgraph, they can be merged into the more compact graph representation of Figure A.4-1. Adding the call-graph of $ex^2 + bx + c$ to the Call-Graph Cache, and again using trie-like sharing of common prefix structure, adds the single *poly* node (and two edges) to the network (Figure A.4-2).

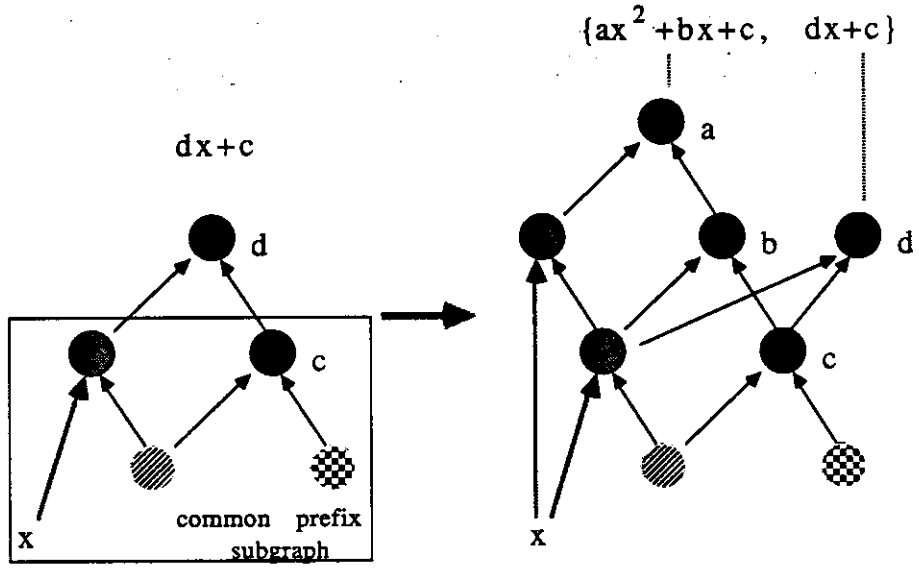

Figure A.4-1: Exploiting common prefix subgraph structure: adding dx+c.

A.5. Multivariate Polynomials

The preceding four Sections gave examples of exploiting redundancy in program control-flow and use to *build* (Control-Flow Cache) and *collect* (Call-Graph Cache) call-graphs. This Section gives examples of how the resulting network can be *used* as a Data Cache to perform incremental computation. The

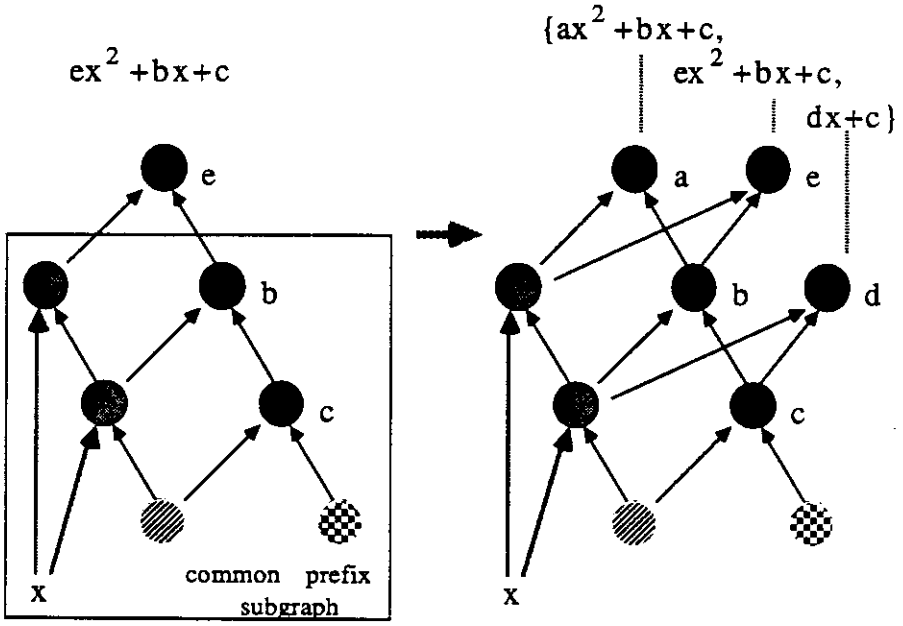

Figure A.4-2: Further trie-like merging in the Call-Graph Cache network.

efficiency gains are greatest when only a small fraction of input data (hence data and partial computations cached in the network) changes each cycle.

Suppose we wanted to compute polynomials in more than one variable. Consider polynomials of the two variables[9] x and y. Previously we had partially evaluated the polynomial program P(C,x) into $P'_C(x)$. Now we shall partially evaluate the multivariate polynomial program P(C,x,D,y) into $P'_{C,D}(x,y)$ For simplicity, consider the case when P' has the form L(x) + R(y); Figure A.5-1 shows a typical resulting call-graph. We now focus on the input values of x and y that may propagate through this network.

If x and y are ordinary values (say, numbers), then the network's data memory (buffer and store) will contain such values at every node. What if the value of x changes, but the value of y does not? Then propagation (and associated computation) will only traverse the left subgraph L(x); R(y) will not be recomputed. That is, the call-graph saves state between the cycles of input value changes. The network is thus an implementation of an *incremental* algorithm that responds to the changes in data, and avoids unnecessary recomputation.

An interesting case is when x and y each denote *sets* of values. For example, the set S_x might equal {3,4,7}, while S_y contained different numbers. Computing L(x)+R(y) then entails calculating the sum of every pair in the cartesian product $S_x x S_y$. For example, if S_x and S_y each had 1,000 members, then 1,000,000 additions would be performed.

Changing S_x, with S_y remaining constant, does not change the overall computational complexity

[9]While CGC works with arbitrarily complex multivariate polynomials, for pedagogy we consider only simpler forms.

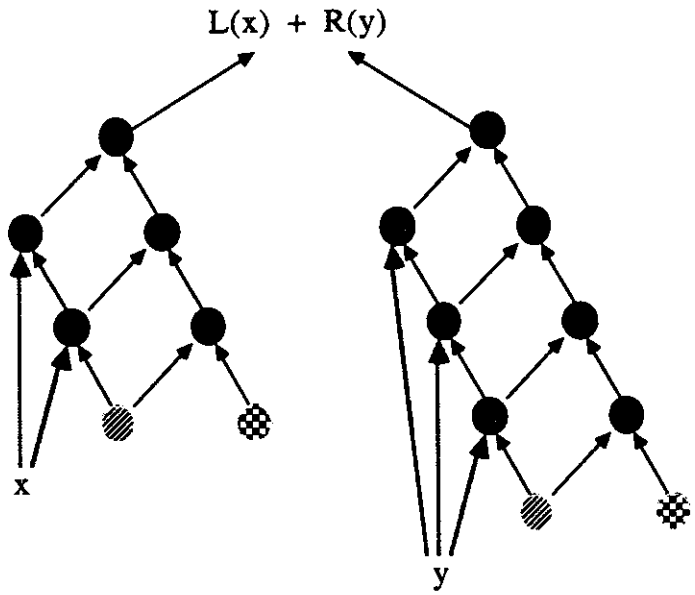

Figure A.5-1: The call-graph of simple multivariate polynomial program.

$O(|S_x x S_y|)$, since each of the new x's must be added to each of old y's. However, when only a small fraction of S_x or S_y change each cycle, the stored state of the network's data cache can lead to great efficiency gains. The amount of required incremental recomputation is then just $O(|\Delta S_x x S_y| + |S_x x \Delta S_y|)$. For example, if $|\Delta S_x| = |\Delta S_y| = 10$, and $|S_x| = |S_y| = 1000$, then (10x1000)x2, or 20,000, recomputations are required, only 2% of the full 1,000,000.

ACKNOWLEDGEMENTS

Jaime Carbonell provided much assistance in the initial formulation of these ideas. David Steier, Peter Lee, and Milind Tambe also contributed to the ideas' evolution. The clarity of the presentation was improved by Yolanda Gil, Ria David, Scott Safier, Caroline Hayes, Gregg Lebovitz, and Carol Scheftic. Steve Minton, Sean Engelson, Haym Hirsh, and the CMU ALIT (Advanced Language Implementation Techniques) seminar (Fall 1988) were helpful sounding boards at critical junctures. Special thanks goes to the CMU Fundamental Structures of Computer Science II class (Fall 1988) for their vigilence in extracting ever-clearer explanations of the RETE match algorithm from me. This work was supported in part by the Pittsburgh NMR Institute.

REFERENCES

[1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D.
Data Structures and Algorithms.
Addison-Wesley, Reading, Massachusetts, 1983.

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

[3] Baer, J.
Computer Systems Architecture.
Computer Science Press, Rockville, Maryland, 1980.

[4] Carbonell, J. G.
Learning by Analogy: Formulating and Generalizing Plans from Past Experience.
In R. S. Michalski, J. G. Carbonell and T. M. Mitchell (editors), *Machine Learning, An Artificial Intelligence Approach.* Tioga Press, Palo Alto, CA, 1983.

[5] Carbonell, J. G.
Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition.
In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (editors), *Machine Learning, An Artificial Intelligence Approach, Volume II.* Morgan Kaufmann, 1986.

[6] Doyle, J.
A Truth Maintenance System.
Artificial Intelligence 12:231-272, 1979.

[7] Forgy, C., and McDermott, J.
OPS: A domain-independent production system language.
In *IJCAI 5*, pages 933-939. 1977.

[8] Forgy, C.L.
On the Efficient Implementation of Production Systems.
PhD thesis, Department of Computer Science, Carnegie Mellon University, February, 1979.

[9] Forgy, C.L.
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
Artificial Intelligence 19(1):17-37, September, 1982.

[10] Futamura, Y.
Partial evaluation of computation process -- an approach to a compiler-compiler.
Computer Systems Controls 2(5):45-50, 1971.

[11] Gupta, A.
Parallelism in Production Systems.
PhD thesis, Department of Computer Science, Carnegie Mellon University, March, 1986.

[12] Hoover, R.
Incremental Graph Evaluation.
PhD thesis, Cornell University, May, 1987.

[13] Knuth, D.E.
The Art of Computer Programming. Volume I: *Fundamental Algorithms.*
Addison-Wesley, Reading, Massachusetts, 1973.

[14] Laird, J. E., Rosenbloom, P. S. and Newell, A.
Chunking in SOAR: The Anatomy of a General Learning Mechanism.
Machine Learning 1, 1986.

[15] Minton, Steven M.
Learning effective search control knowledge: An explanation-based approach.
PhD thesis, Carnegie Mellon University, February, 1988.

[16] Miranker, D.P.
TREAT: A New and Efficient Match Algorithm.
PhD thesis, Columbia University, January, 1987.

[17] Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T.
Explanation-Based Generalization: A Unifying View.
Machine Learning 1:47-80, 1986.

[18] Oflazer, K.
Partitioning in Parallel Processing of Production Systems.
PhD thesis, Department of Computer Science, Carnegie Mellon University, March, 1987.

[19] Perlin, M.W.
Reducing Computation by Integrating Inference and User Interface.
Technical Report CMU-CS-88-150, Carnegie Mellon University, Pittsburgh, PA, June, 1988.

[20] Pratt, T.W.
Programming Languages, Design and Implementation.
Prentice-Hall, Engelwood Cliffs, New Jersey, 1984.

[21] Pugh, W.W.
Incremental Computation and the Incremental Evaluation of Function Programs.
PhD thesis, Cornell University, August, 1988.

[22] Reps, T., Teitelbaum, T., and Demers, A.
Incremental context-dependent analysis for language-based editors.
ACM Trans. Prog. Lang. Sys. 5(3):449-477, July, 1983.

[23] Schank, R. C. and Abelson, R. P.
Scripts, Goals, Plans and Understanding.
Hillside, NJ: Lawrence Erlbaum, 1977.

[24] Stolfo, S.J., and Shaw, D.E.
DADO: A Tree-structured Machine Architecture for Production Systems.
In *Proceedings of National Conference on Artificial Intelligence*, pages 369-388. AAAI, August, 1982.

[25] Tremblay, J.P., and Sorenson, P.G.
The Theory and Practice of Compiler Writing.
McGraw-Hill Book Company, New York, 1985.

[26] van Harmelen, F., and Bundy, A.
Explanation-Based Generalization = Partial Evaluation.
Artificial Intelligence 36(3):401-412, October, 1988.

[27] Winograd, T.
Language as a Cognitive Process, Volume I: Syntax.
Addison-Wesley, 1983.