NORMA: A Graph Reduction Processor

Mark Scheevel
Burroughs Corporation
Austin Research Center

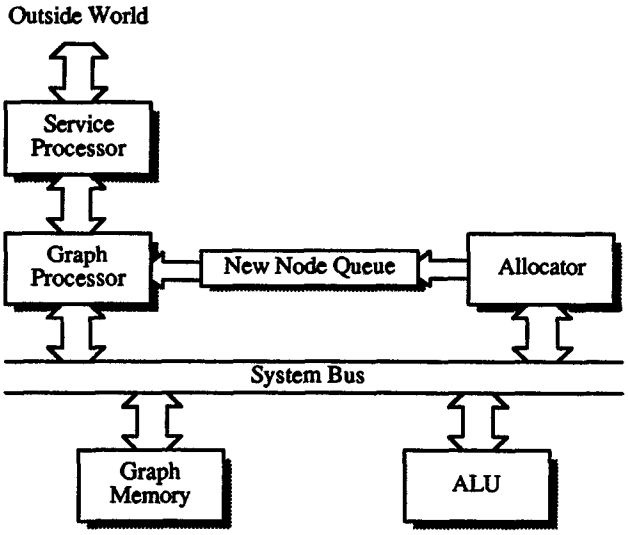
1. Introduction

NORMA (Normal Order Reduction MAchine) is a special-purpose machine designed explicitly for graph reduction, with some optimizations for combinator graph reduction in particular. It is a microprogrammable machine, and as such it could presumably be configured to function as an interpreter for any number of graph reducers; we have used it only to implement a combinator reducer employing Turner's SK combinator set (with a few extensions) as described in [Turner78] and [Turner79]. NORMA consists of five main elements:

1. the service processor,
2. the graph processor,
3. the graph memory,
4. the allocator, and
5. the ALU.

Figure 1-1 shows the relationship of these main elements. This paper describes the first four elements in detail and provides some insight into their use in graph reduction. The final section provides some performance figures.

A number of tradeoffs were made in the design of NORMA. In particular, we decided to accept the expense and trouble of a very wide microword (approximately 370 bits) in order to achieve parallelism within each instruction. Because we did not feel that we currently understand the relationship between graph reduction and virtual memory well enough, we also decided not to build a virtual memory machine; consequently, programs are constrained to the size of the physical memory of the machine. Because cyclic graphs are common in graph reduction (with our SK reducers, at any rate) and because we decided we did not want to introduce the complications of reference-counting (a notorious source of errors in the past), we also decided to use the simple mark-scan garbage collection strategy; an optimization of this scheme (discussed later) has yielded quite acceptable performance.



NORMA Block Diagram
Figure 1-1.

2. The Service Processor

The NORMA service processor is an ordinary microcomputer that serves primarily as an I/O processor for the rest of the machine. It is based on an 8086 with 512K bytes of RAM and 32K bytes of PROM, and connects to a host (currently an Apollo DSP-80 server node) via a Multibus card. The service processor and the graph processor communicate

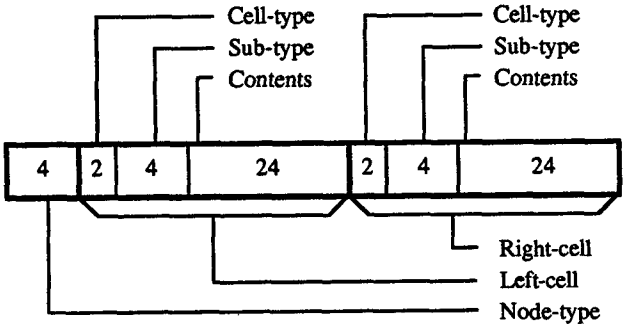
through two 64-bit registers, one for data going to the graph processor and the other for data coming from it. The service processor receives an interrupt whenever the outgoing register is emptied or the incoming register is filled. The output from NORMA, which is generated one character at a time, is buffered in service processor memory and transferred (via DMA) to the Apollo when a complete message is accumulated. Similarly, input messages are DMA'ed into service processor buffers from which characters are supplied one at a time when requested by the graph processor.

In addition to its I/O responsibilities, the service processor also helps with the marking phase of garbage collection: it retains pointers to all reachable graphs (of which there are four in the present implementation), and when garbage collection is required it directs the graph processor to mark each graph in sequence.

3. The Graph Memory

The NORMA graph memory is divided into two parts: the node memory, where the program graphs reside, and the mark memory, where the mark bits required for garbage collection reside. Each NORMA memory board consists of 512K words of error-corrected node memory (implemented with dynamic RAMs) and 512K bits of mark memory (implemented with static RAMs). Normal configurations contain two memory boards.

NORMA graph nodes are 64-bit words that are divided into three subparts: a 4-bit node-type field and two 30-bit fields called the left-cell and right-cell field. These cell fields are further subdivided into three parts: a 2-bit cell-type field, a 4-bit sub-type field, and a 24-bit contents field. The subdivisions are illustrated in Figure 3-1.



Node Layout
Figure 3-1.

The node-type field is used to distinguish nodes of different types, e.g., nodes that represent function applications from nodes that represent list pairs.

The cell-type field is used to distinguish pointers from atoms. This is sufficient to distinguish among atoms and three types of pointers, which are required to support a pointer-reversing graph traversal scheme.

The sub-type field is used to further identify a cell. When a cell is an atom, this field identifies its type, e.g., number or combinator. When a cell is a pointer, this field identifies the type of the node pointed to, if known. This provides a convenient way of identifying sub-graphs that have already been evaluated.

The contents field is where actual data resides. For pointer cells, this field is interpreted as a node address. For atomic cells, the interpretation of the field depends upon the value of the sub-type field. For example, if the sub-type field indicates that a cell should be interpreted as an integer, the low-order 16 bits of the contents field are interpreted as a two's-complement integer; if the sub-type field indicates that it should be interpreted as a character, the low-order 8 bits of the contents field are interpreted as an ASCII character.

Each graph node has one mark bit associated with it. Whenever a node is read from graph memory, this bit is also read and latched into a one-bit register in the data section, where it may be examined by subsequent instructions. The mark bit may also be (concurrently) set or reset during one of these accesses; in these cases the data section register receives the old value of the mark bit.

In addition to this one-bit access, another access mode is provided for the allocator. In this mode, the mark bits of eight successive nodes (starting on any eight-node boundary) are returned as an eight-bit mark vector. The benefit of this mode is explained in Section 5, where the allocator is described.

4. The Graph Processor

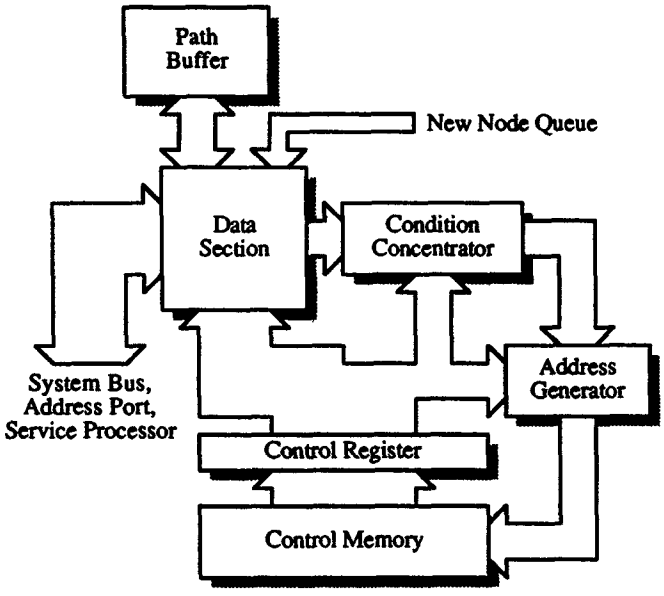
The graph processor, where all graph transformations are actually performed, is the most significant element of NORMA; Figure 4-1 shows its main components. The major cycle time for the graph processor is 200 ns; most instructions require only one major cycle, although some require two. There are two broad classes of instructions: transfer instructions and test instructions. Every instruction carries the base address of its successor (there is no subroutine capability). For transfer instructions this base address is the actual address of the successor instruction. For test instructions, this is the address of a vector of instructions, one

of which will be selected based on the results of the tests performed. Both types of instructions will be explained more fully below. The control memory holds 2K instructions.

4.1. The Data Section

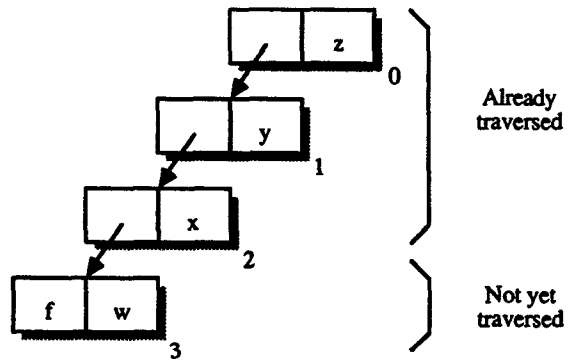
The data section consists of five node registers, two cell registers, two node ports, and two address ports. Of the five node registers, four (named B0, B1, B2, and B3) are intended to hold the current redex, while the fifth (named T) is intended as a spare register for intermediate results. The two cell registers (named R and F for "reverse" and "forward") are intended to be used as "place holders" in a pointer-reversing graph traversal. One node port (named DP) provides access to the system bus (and hence, the graph memory and the ALU), while the other (named BP) provides access to the path buffer. One address register (named AP) is used to supply node addresses for graph memory accesses, while the other (named NNA), is used to obtain addresses from the new node queue (see Section 5). Each of these registers is subdivided into appropriate sub-registers in the same fashion as graph memory. For example, B0 ultimately comprises seven sub-registers named B0.NT, B0.LC.CT, B0.LC.ST, B0.LC.C, B0.RC.CT, B0.RC.ST, and B0.RC.C.

The power of this data section derives from its ability to perform transfers involving all of the ports and registers simultaneously. All like sub-registers are connected by a crossbar, so that during any transfer instruction any sub-register or port may be transferred to (nearly) any other like sub-register or port (e.g., R.CT to DP.CT, or B1.RC.C to F.C), and any number of these transfers may take place in parallel. The only restrictions involve transfers to and from the path buffer port. This power is not without a price, however; many of the bits in each microword are used to provide the necessary crossbar configuration control.



Graph Processor Block Diagram
Figure 4-1.

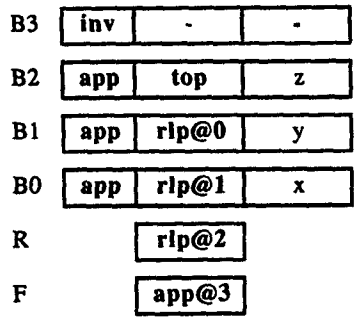
This flexibility permits, for example, all registers to be modified at the same time that a graph node is read into the data section, making it possible to traverse a graph pointer in a single instruction. Similarly, the rewrite rules for many of the common combinators can be performed in one or two instructions. As an example, consider the common operation of traversing a graph pointer in search of the next redex. As an optimization we will keep the most recently traversed nodes in the registers B0-B3, so that when we eventually encounter a combinator the nodes of the redex will already be in the data section. Assume that the graph we are traversing is as shown in Figure 4-2, and that we have traversed the indicated portion.



Example Graph
Figure 4-2.

At that point the data section will contain the data shown in Figure 4-3. To take the next step down the graph, we will need to make room for the node to be read in by "pushing" the nodes that are already in the data section up one location, reading in the new node image, and reversing the appropriate pointers. All of this can be accomplished with one instruction that specifies the following transfers (along with some other appropriate control information):

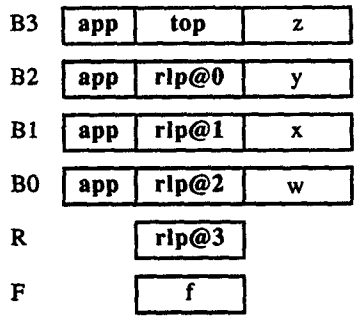
B2 to B3
B1 to B2
B0 to B1
F.C to AP
DP.NT to B0.NT
R to B0.LC
DP.RC to B0.RC
constant **ptr** to R.CT
constant **rlp** to R.ST
F.C to R.C
DP.LC to F



inv	indicates an invalid node
top	indicates the top of the graph
app	indicates an application node
rlp@N	indicates a reverse pointer to node N

Data Section before Traversal Step
Figure 4-3.

The effect of these transfers is to read the node whose address is in F.C into F and B0 and to rearrange the rest of the data section to accomodate the new information. Notice that the left side of the newly read node is placed in F, so that we are ready to repeat the process until F no longer contains a pointer. Figure 4-4 shows the contents of the data section after the instruction completes.



Data Section after Traversal Step
Figure 4-4.

Transfer instructions that involve node memory accesses, such as the preceding example, complete in two major cycles (400 ns), although instructions that follow node memory writes and do not use the system bus will be overlapped with the second phase of the write, yielding an effective time of 200 ns for the write. Transfer instructions that do not involve node memory accesses require only one major cycle (200 ns).

4.2. The Condition Concentrator

Graph transformation is only part of the story in a graph reducer. Because reduction is a highly interpretive activity, a great deal of run-time condition testing is necessary, so that as graph transformation time is reduced, the time spent testing conditions begins to dominate.

The condition concentrator is a response to this problem. It is to condition testing what the data section is to graph transformation -- a means of performing a number of tasks (in this case, condition tests) concurrently. The usual tests involve comparing tag fields against specific literal values, e.g., to see whether a cell is a combinator or whether a node is an application. In a conventional machine, these sorts of tests must be programmed as a series of transfers to an ALU followed by conditional jumps to other pieces of code. In NORMA, up to fifteen different comparisons of cell-type/sub-type (or node-type) registers with literals can be performed in parallel. Limited combinations of these comparisons may be OR'ed with one another and assigned a unique priority from zero (highest) to fourteen (lowest); the highest-priority OR term that is true is used as an offset to determine the next instruction. This scheme permits fast multiway branching on fairly complex tests. All test instructions complete in one major cycle (200 ns).

To illustrate the advantages of this approach, consider again the example of traversing a graph in search of the next redex. After performing the transfer instruction that actually moves the new node into the data section, we have practically no idea what to do next; everything depends upon the nature of the values in F and B0.NT. The various possibilities are:

1. F contains a reversed pointer. This indicates that we have just traversed a cycle in the graph and that the graph being reduced represents an undefined value (semantic bottom). In this case we back up and recover.
2. B0.NT indicates that the node just read in does not represent an application node. (This happens when the pointer to the node has not yet been updated with the appropriate sub-type information.) In this case we must update the type information in the pointer and back up. The remaining tests assume that B0.NT indicates that the node in B0 is an application node.
3. F contains a forward pointer to an application node. In this case we need to repeat the traversal step.

4. F contains a forward pointer to a list node. This indicates that a list is being indexed, so we must branch to the list indexing code.
5. F contains a forward pointer to some other type of node. This is an error condition.
6. F contains a combinator. In this case we branch to the setup code for the combinator.
7. F contains some other atom. This is another error condition.

With the condition concentrator, all of these tests may be performed simultaneously, and the branch to the appropriate successor instruction is accomplished in one major cycle. In fact, in our interpreter this particular instruction may be reached from several other instructions, so that we can assert even less about the state of the data section when this instruction is performed. Accordingly, several other tests are also performed at the same time, but there is no impact on performance.

4.3. The Path Buffer

The path buffer is a cache for the normal-order path of the graph being reduced. It is a stack-like entity with a capacity of 2K 64-bit entries. Nodes may be pushed onto it (through the path buffer port in the data section) from B3, and nodes may be popped from it (either the youngest or the oldest) into B3 or T (again via the path buffer port). These path buffer accesses may take place in parallel with other data section register transfers, including memory accesses.

The purpose of the path buffer is to avoid the cost of writing out the "reversed" node images of the normal-order path (the path from the current redex back to the root of the graph); the reversed images are stored only in the data section and the path buffer. Notice that in the graph traversal example above, the modified image of the node that was put into B0 was not also written back into memory. We also avoided another problem by assuming that B3 was empty before we performed the step, so that there was no problem in moving the contents of B2 into B3 to make room for the new node. In the usual case, however, B3 will contain something significant. To make room in this case, we transfer B3 into the path buffer at the same time that we perform the other transfers.

On rare occasions, the path buffer will become completely full. When we need to push a node into a full path buffer, we first spill the oldest node out of the path buffer and into memory, and then push the new node into the path buffer. Because of this, some portion of the reversed normal-order path will actually reside in graph memory. These nodes will be unreversed and pulled back into the data section during later reductions.

5. The Allocator

The NORMA garbage collector is a variant of a standard mark-scan garbage collector inspired by [Hughes82b]. When garbage collection is needed, reduction is stopped and all relevant graphs are marked from their roots. However, instead of then performing the scan phase at once and building up a linked list of free nodes, reduction is resumed. The task of finding and reclaiming unmarked nodes is left to another processor, called the allocator, which operates in parallel with the graph processor.

The allocator's job is to scan mark memory, placing the addresses of unmarked nodes in the new node queue and resetting the mark bits of marked nodes. The new node queue consists of 256 24-bit entries from which the graph processor retrieves node addresses when it needs to allocate a new node. When the queue is sufficiently full, the allocator suspends its operation and waits for the queue to be emptied somewhat. When that point is reached, the allocator resumes its activity. When the allocator reaches the end of memory and the new node queue falls below a certain minimum, the graph processor signals the service processor that another garbage collection is necessary.

In order that the graph processor receive as much memory bandwidth as possible, the allocator uses a special mark memory access mode that minimizes contention. In this mode, the allocator supplies a 21-bit address to the graph memory and receives an eight-bit vector of mark bits in return. The address is that of a block of eight nodes, and the vector consists of a mark bit for each node in the block (the mark bits are reset concurrently, so that they will be ready when the next mark phase begins). The allocator places this vector in local memory, and then manufactures the address associated with each reset mark bit, placing each such address in the new node queue as it is manufactured. When the vector has been completely scanned, another is fetched, and so on, until all of memory has been scanned.

The addresses in the new node queue are available for immediate consumption by the graph processor. To allocate a new node during reduction, the graph processor need do nothing more than designate the new node port as the source for the address port when it writes the image of the new node. Addresses can be supplied one per instruction as long as there are addresses in the new node queue.

6. Performance

NORMA's performance has been very encouraging: it is capable of sustaining a reduction rate of over 250,000 reductions per second (including garbage collection time), which compares quite favorably with other reducers with which we are familiar. The interpreter that was previously in use at the Austin Research Center was written in Burroughs B1900 microcode and was capable of approximately 5,000 reductions per second on a B1955. Another interpreter, written in C, is capable of approximately 10,000 reductions per second on a VAX 11/780. Another hardware effort, the SKIM project at Cambridge [Stoye84], was able to achieve reduction rates of approximately 100,000 reductions per second.

These are all measures of (more or less) directly comparable machines: all compile programs to combinator expressions, and all use nearly the same fixed set of combinators. Measuring the performance of NORMA relative to other implementation techniques, such as a supercombinator and G-machine approach ([Hughes82a], [Augustsson84], [Johnsson84], [Kieburtz85]), is more difficult, in large part because of the lack of a standard set of benchmarks. Augustsson and Johnsson reported the performance of their LML compiler for some small programs in [Augustsson84] and [Johnsson84]; those times are included in Table 6-1 below (the SASL source for the benchmarks is included in Figures 6-1 through 6-5 at the end of the paper). We have also included the performance of the B1955 interpreter and the standard Pascal compiler for an Apollo DN300 diskless workstation for some of the programs.

As is evident in Table 6-1, the implementations of SASL on NORMA and LML on a VAX appear to be comparable; whether this would continue to be true for realistically-sized programs is an interesting question that has not yet been answered. None of these programs makes very heavy use of higher-order functions or lazy evaluation, features that seem more and more necessary as programs grow larger. It is too early to say whether one implementation or the other provides better support for those features.

The comparison with Pascal on the DN300 (a 68010-based machine) indicates that, with a little effort, the performance of applicative languages can be brought near that of conventional imperative languages. In the two benchmarks that are lop-sidedly in favor of Pascal, the difference can be attributed directly to the use of arrays in the Pascal solutions, while the SASL versions were restricted to lists. This issue seems orthogonal to the question of applicative-vs-imperative languages; with some work on a decent implementation of arrays, the SASL (and LML) times could likely be brought into line with the Pascal times.

7. Acknowledgements

The NORMA hardware was designed and built by Gary Logsdon, Brent Bolton, Frank Williams, and Mike Winchell. The NORMA software was designed and written by a team consisting of Bob Bethke, Tom Crockett, Curt Hern, David Dow and the author. We have also benefitted tremendously from discussions with the rest of the staff at the Austin Research Center and David Turner.

8. References

Augustsson84. Augustsson, L. "A Compiler for Lazy ML." Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming. Austin, Texas. August 1984.

Benchmark	NORMA/SASL	VAX/LML	B1955/SASL	DN300/Pascal
7 queens	2.0	3.2	-	-
8 queens	8.5	-	422.2	0.5
300 primes	0.7	0.5	-	-
1000 primes	3.7	-	479.9	0.2
fibonacci 20	0.7	0.8	-	-
fibonacci 23	3.1	-	104.0	1.7
tak 18 12 6	4.0	10.0	174.3	2.1
symbolic deriv	6.6	-	374.0	7.7

All times are for one iteration of the benchmark,
except for the symbolic derivative time, which is for 1000 iterations.

Execution times of some sample programs (in seconds).

Table 6-1.

Hughes82a. Hughes, R.J.M. "Super-Combinators: A New Implementation Method for Applicative Languages." Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming. Pittsburgh, Pennsylvania. August 1982.

Hughes82b. Hughes, R.J.M. "A Semi-Incremental Garbage Collection Algorithm." Software - Practice and Experience 12, 11 (November 1982).

Johnsson84. Johnsson, T. "Efficient Compilation of Lazy Evaluation." Proceedings of the 1984 Symposium on Compiler Construction. SIGPLAN Notices 19, 6 (June 1984).

Kieburtz85. Kieburtz, R.B. "The G-machine: A fast, graph-reduction evaluator." Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture. Nancy, France. September 1985.

Stoye84. Stoye, W.R., Clarke, T.J.W. and Norman, A.C. "Some Practical Methods for Rapid Combinator Reduction." Proceedings of the 1984 ACM Symposium on LISP and Functional Programming. Austin, Texas. August 1984.

Turner78. Turner, D.A. "Another Algorithm for Bracket Abstraction." Journal of Symbolic Logic 44, 3 (December 1978).

Turner79. Turner, D.A. "A New Implementation Technique for Applicative Languages." Software - Practice and Experience 9, 1 (January 1979).

```
# (queens 8)
WHERE
queens N    = solve N
              WHERE
              solve 0 = [ [] ]
              solve n = [ q:b; b <- solve (n-1); q <- [1 .. N]; safe q b ]
safe q b    = all (not.check q b) [1 .. #b]
check q b i = q = b i | abs (q - b i) = i

|| (all p L) is true if every element of L satisfies p.
```

Queens Benchmark
Figure 6-1.

```
primes 300
WHERE
    primes = 2 : filter (not.factorable) [3,5 ..]
    factorable n = any (divides n) (upto (not.candidatefor n) factors)
    divides n p = n REM p = 0
    candidatefor n p = p*p <= n
    factors = upto (greaterthan limit) primes
    greaterthan x y = x<y
    limit = trunc (sqrt maxint)

    || (any p L) is true if some element of L satisfies p.
    || (upto p L) is the longest prefix of L that contains
    || no element that satisfies p.
```

Primes Benchmark
Figure 6-2.

```
fib 23
WHERE
fib n = n<3 -> 1; fib (n-1) + fib (n-2)
```

Fibonacci Benchmark
Figure 6-3.

```
tak 18 12 6
WHERE
tak x y z = ~(y < x) -> z;
            tak (tak (x-1) y z) (tak (y-1) z x) (tak (z-1) x y)
```

Tak Benchmark
Figure 6-4.

```
deriv example

WHERE

example = sum [4, prod [7,'x','y'], exp 'x' 2, exp 'x' 3]

deriv ('+':S)   = sum (map deriv S)
deriv ['-',x,y] = diff (deriv x) (deriv y)
deriv ['~',x]   = neg (deriv x)
deriv ('*':S)   = sum (map prod (distribute deriv S))
deriv ['/',x,y] = div (diff (prod [y,deriv x]) (prod [x, deriv y])) (exp y 2)
deriv ['^',x,y] = prod [y, exp x (diff y 1)]
deriv 'x'       = 1
deriv y         = 0

sum S    = #vars=0 -> k;
           k=0 -> (#vars=1 -> hd vars; ('+':vars));
           ('+':k:vars)
           WHERE
           k = prelude_sum consts
           [consts,vars] = split number S
diff x y = number x & number y -> x-y;
           x=0 -> neg y;
           y=0 -> x;
           ['-',x,y]
neg x    = number x -> -x;
           ['~',x]
prod S   = #vars=0 -> k;
           k=0 -> 0;
           k=1 -> (#vars=1 -> hd vars; ('*':vars));
           ('*':k:vars)
           WHERE
               k = product consts
               [consts,vars] = split number S
quot x y = number x & number y -> x DIV y;
           x=0 -> 0;
           y=1 -> x;
           cancelpowers x y
exp x y  = number x & number y -> x**y;
           x=1 | y=0 -> 1;
           x=0 -> 0;
           y=1 -> x;
           ['^',x,y]

cancelpowers ['^',x,y1] ['^',x,y2] = exp x (diff y1 y2)
cancelpowers x          ['^',x,y]  = exp x (diff 1 y)
cancelpowers ['^',x,y]  x          = exp x (diff y 1)
cancelpowers x          y          = ['/',x,y]

distribute f (a:x) = (f a:x) : map (cons a) (distribute f x)
distribute f []    = []

|| The value (split p L) is a pair of lists [L1,L2].
|| L1 consists of all elements of L that satisfy p and
|| L2 consists of all elements of L that do not satisfy p.
|| Relative ordering is preserved in both lists.
|| (prelude_sum L) is the ordinary sum of all elements of L.
```

Symbolic Derivative Benchmark
Figure 6-5.