Creating Efficient Programs By Exchanging Data for Procedures

By

John Franco

and

Daniel P. Friedman

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 245

Creating Efficient Programs
By Exchanging Data for Procedures

by

John Franco and Daniel P. Friedman

March, 1988

This report is based on work supported in part by the Air Force Office of Scientific Research under Grant No. AFOSR 84-0372 and by National Science Foundation Grants DCR 83-03325 and CCR 87-02117.

ABSTRACT

We present a programming style which fosters the rapid development of programs with low asymptotic complexity. The crucial idea behind the new programming style is that mutually recursive procedures, each assigned the task of returning the solution to a subproblem of the given problem, are constructed directly from the problem instance. A side benefit is that these procedures replace array structures, and, thus, there is no need for array subscript arithmetic. Moreover, the new programming style preserves generality and enhances comprehensibility.

Creating Efficient Programs by Exchanging Data for Procedures

John Franco and Daniel P. Friedman, Indiana University

1. Introduction

We present a programming style which fosters the rapid development of programs with low asymptotic complexity. The crucial idea behind the new programming style is that mutually recursive procedures, each assigned the task of returning the solution to a subproblem of the given problem, are constructed directly from the problem instance. A side benefit is that these procedures replace array structures, and, thus, there is no need for array subscript arithmetic. Moreover, the new programming style preserves generality and enhances comprehensibility.

Conventional compilers lack the sophistication needed to optimize complexity. Hence, it is up to the programmer to tune his programs for efficiency. In Lisp-systems, on the other hand, macro-sublanguages provide a facility for extending the compiled language. With a macro-facility it is therefore feasible to design algorithms that translate input data directly into efficient code. Such a solution is particularly preferable when macro definitions are easy to design as with `extend-syntax` ([5,6]).

With a macro-algorithm, the input symbols become program identifiers which are bound to procedures. As an example, consider the problem of testing set membership (without recursion) in a small finite set. In the macro-programming style, the algorithm becomes:

```
(extend-syntax (member-in-set?)
 [(member-in-set? n ...) (let ([n (lambda (x) (eq? x 'n))] ...)
                            (lambda (x) (or (n x) ...)))])
```

This expands to

```
(let ([a (lambda (x) (eq? x 'a))]
      [b (lambda (x) (eq? x 'b))]
      [c (lambda (x) (eq? x 'c))])
 (lambda (x) (or (a x) (b x) (c x))))
```

for the input `(member-in-set? a b c)`, *i.e.*, the input symbols `a`, `b`, `c` become identifiers that denote the procedures `(lambda (x) (eq? x 'a))`, *etc*.

Several problem solving methods can be implemented in the proposed style. Examples of these are Dynamic Programming (*e.g.*, the Knapsack problem), Greedy (*e.g.*, the Minimum Spanning Tree problem), Depth First Search (*e.g.*, finding the cutpoints of a graph), and Breadth First Search. Furthermore, our style can turn a recursive Divide and Conquer solution into an "inverted" Dynamic Programming solution. These implementations are so similar that once the style is understood, new solutions are readily obtained.

The remainder of this paper consists of four sections plus a conclusion. Section 2 introduces our new programming technique. In Section 3 we apply the technique to the problem of determining a topological sort of a directed graph [4]. This example illustrates our style for a data structure more complicated than a finite set. Section 4 contains our solution to the problem of finding the cutpoints of a connected graph [1]. This gives a taste of the sophistication achievable. In the fifth section we discuss how simple input structures are expandable into highly efficient stream-programs. The concrete example for this is the Dynamic Programming solution to the Partition problem [3]. Our programs are theoretically the most efficient known for the problems they are designed to solve. That is, the runtime required is bounded from above by a function of the input size which is within a constant factor of the complexity of the best solution. Ordinarily the compile time is not taken into account when computing complexity since compilation occurs just once. However, in our unusual style compilation occurs every time the input data is changed. Therefore, we must also include compile time in our complexity measure. Unfortunately, compile time depends on specific implementations of extend-syntax and letrec. However, in a future paper we show that extend-syntax and letrec can be implemented so that compile time in our style is linear in the size of the "data-network" constructed by these facilities (this should be intuitively clear). If we assume this to be the case then the total complexity of our programs, including runtime and compile time, is optimal to within a constant factor.

2. A Style for Efficient Programs

Given a decomposable problem P, we construct a network of nodes where each node is a procedure for solving a subproblem of P. Each node has one or more of two types of sections called *complex* and *simple*. A complex section, unlike a simple section, initiates communication with other nodes; a simple section returns final and intermediate answers. When evaluated, a node changes its state by switching from a complex section to a simple section. In addition, a node may change its behavior through communication with another node.

Typically, a node has three states: never communicated with, trying to figure out the solution to the subproblem it represents, and has found the solution to its corresponding subproblem. In the first state a node is complex while awaiting an "order" to compute its solution, in the second state it is simple and reports partial results, and in the third state it is simple and reports its final result. The first action of a complex section is to switch to a simple section or a less complicated complex section that is guaranteed to perform a small number of communications with other nodes. The collection of all sections that a node can assume and the conditions under which each section is assumed is called a node's *functionality.*

We consider combinatorial problems which allow the functionality of each node to be the same. Then the number of nodes needed to solve a problem may be huge and may be different from instance to instance, but the code for each node is identical with the exception of the neighborhood relationship. When the given problem is such that nodes must communicate with any other nodes we make use of `letrec` to define them. Identifier lookup with `letrec` is assumed to be random access so there is no complexity penalty in its use.

The power of our technique lies in the design of each node so that its complex section(s) gets computed only once. Each section (or state) corresponds to a procedure. Switching between sections, or changing state, is accomplished through assignments to the node identifier. In particular, assignments prevent recomputations of complex sections. Put abstractly, our programming style generalizes Felleisen and Friedman's [2] re-definition technique for implementing `import-by-need` and `c-letrec` in module specifications.

3. Topological Sort Using Mutual Recursion

The problem of Topological Sort (hereafter referred to as sort) is stated as follows: given a directed graph $G = (V, E)$, find a total ordering of the vertices of V if one exists, otherwise return an appropriate error message.

Input and output representation are chosen for convenience. Thus, an input is assumed to be a list of lists, one sub-list for each vertex, where each sub-list contains the precedence information for that vertex (a list of vertices called precedence neighbors). A successful output is a display of vertices in a total ordering.

For any graph, if there exists a path from vertex v to vertex w we say that w is *reachable* from v. We call a vertex *free* if its precedence list is null or contains only vertices that have already been displayed. Observe the following:

1. A sort of V is obtained if, for each $v \in V$, all vertices reachable from v can be sorted.

2. All vertices reachable from v can be sorted if all vertices reachable from each precedence neighbor of v can be sorted, regardless of the order in which these sub-sorts are carried out.

3. A vertex is *free* when its reachable vertices have been displayed.

4. A vertex that becomes *free* may be displayed immediately.

These observations suggest a one-to-one correspondence between vertices and nodes. We extend reachability in the obvious way to nodes. We develop node functionality as follows. From (2) the complex section of node n should sort all nodes reachable from n. From (3) and (4), the symbol n should be displayed right after the complex section is completed. Failure occurs if a node which is executing its complex section receives a message from some node since this implies the node is reachable from itself (a cycle). One way to detect failure, then, is to assign the node an error function before the complex section is entered. The error function prints an error message and stops computation if invoked. Putting it all together we have the following program for finding a topological sort of a directed graph.

```
(extend-syntax (topo)
  [(topo ([n (m ...)] ...))
   (letrec
     ([n (lambda ()
            (set! n (lambda () (error 'n "is part of a cycle")))
            (m) ...
            (set! n (lambda () "don't care"))
            (writeln 'n))]
      ...)
     (n) ...)])
```

The code (see the appendix for its expansion) may be tested with

    (topo ([a (b c e)] [b (e f)] [f (c)] [c ()] [e (f c)])).

Note the following: First, *none* of the input symbols are quoted because each is regarded as a procedure. The data is a collection of procedure identifiers and these identifiers may clash with those given in the code. We have avoided such clashes when choosing identifiers for our data. Avoiding clashes is rendered unnecessary by using hygienic expansion [5]. Second, the complexity of topo is linear in the number of input symbols since each complex section is executed once and is responsible for communicating to a number of nodes that is equal to the number of its neighbors. Third, no tests are used in topo. This is a result of the first assignment in each node. Fourth, the code for topologically sorting a partially ordered set found in [4] uses a sophisticated data structure in order to achieve linear complexity, but here data structure maintenance is hidden.

4. Cutpoints of a Graph, A More Complicated Example

The next problem we consider is the problem of finding the cutpoints of a connected graph $G = (V, E)$ with no self-loops. A cutpoint is a vertex which, when removed from G, causes G to be disconnected. If there are two vertices reachable from a vertex v such that every path connecting those vertices contains v, then v is a cutpoint of G. Let us create one node for each vertex. If n is a node then we let $\text{var}(n)$ denote the corresponding vertex. A node must communicate with, or *ping*, its reachable nodes to determine whether it is a cutpoint. If each node is allowed to do so individually, an $O(|E||V|)$ algorithm will result. We save a factor of $O(|V|)$ by redefining the node as a simple procedure once it has been pinged by another node; doing so prevents a node from asking the same reachability questions repeatedly.

If a node n attempts to ping a previously pinged node m then call the "edge" $\langle n, m\rangle$ a backedge. Let node m be an unpinged neighbor of pinged node n. If all currently unpinged nodes reached from m, except through n, do not reach any nodes that are pinged, then $\text{var}(n)$ is a cutpoint which separates the subgraph corresponding to the nodes visited "below" node m from the rest of G. Therefore, an easy test for discovering whether $\text{var}(n)$ is a cutpoint is to determine whether any backedges from nodes "below" m

to nodes "above" n exist. This is accomplished by counting the number of backedges. The global variable TB (for Total Backedges) keeps count of the total number of backedges encountered. When the complex section of a node is completed, TB is decremented for each backedge terminating at that node. The local variable LB (for Local Backedges) keeps count of the number of backedges terminating at a node. The complex function of node n saves TB, sets LB to 0, and invokes a neighbor node. If the neighbor node has not already been pinged, the value of LB is subtracted from the current value of TB and the saved value is compared with the result. If they are the same, then node n is a cutpoint and is displayed. The node n gets displayed each time it is found to be a cutpoint unless n is the node from which computation commences. Then node n gets displayed one more time than it is found to be a cutpoint (this minor annoyance can be taken care of by removing the last symbol that is output). Here is the code.

```
(extend-syntax (cutpoints)
 [(cutpoints ([n (m ... )] ...))
  (let ([TB 0])
   (letrec
    ([n (let*
         ([LB -1] [ping-neighbors
                   (lambda (m^)
                    (let ([old-TB TB])
                     (set! LB 0)
                     (when (not (negative? (m^)))
                           (set! TB (- TB LB))
                           (when (= old-TB TB) (writeln 'n)))))])
         (lambda ()
          (set! n (lambda ()
                   (set! LB (+ LB 1))
                   (set! TB (+ TB 1))
                   LB))
          (ping-neighbors m) ...
          (set! n (lambda () -1))
          LB))]
       ...)
     (n) ...))])
```

A sample test of cutpoints is

```
(cutpoints ([a (b c d)] [b (a d)] [c (a d)] [d (a e h b c)]
            [e (d f g)] [f (e g h)] [g (e f h)] [h (d f i g j k)]
            [i (h l p)] [j (h k)] [k (h j)] [l (i p)] [p (i l)]))
```

The code performs essentially the same as the Depth First Search solution found in [1] but without the need for Depth-First-Numbers labeling the nodes. Thus our code is less cluttered with data organizational details that are irrelevant to the specification of the solution.

5. Partition Problem, Example of an Acyclic Node Structure

The preceding examples use extend-syntax primarily because the communication structure between nodes is not known before runtime, the communication structure between nodes is complex, and there is a one-to-one correspondence between nodes and the elements of each input. In applications of Backtracking, Divide-and-Conquer, Branch-and-Bound and Dynamic Programming, the communication structure is a tree with a number of nodes that can be exponential in the number of input elements. Frequently, however, only a small subset of nodes needs to be visited in order to obtain a solution to the given problem. If efficient implementations are to be obtained in these cases the unnecessary nodes must not be created. It follows that nodes must be constructed on-the-fly. In this mode extend-syntax and letrec reduce to lazy cons. The result is efficient code for a variety of problems. Below is cons$, our name for lazy cons. We leave car$ and cdr$ to the reader.

```
(extend-syntax (cons$)
 [(cons$ a d) (letrec
                ([n (lambda ()
                      (set! n (let ([v a]) (lambda () v)))
                      (n))]
                 [m (lambda ()
                      (set! m (let ([v d]) (lambda () v)))
                      (m))])
                (cons (lambda () (n)) (lambda () (m))))])
```

Consider, for example, the Partition problem (see [3]). An instance of the Partition problem is a positive integer B and a set E of elements, each having a positive integer value. The problem is to determine whether there exists a subset of E with values that sum to B. We represent E as a stream of integers. The program slide-merge below takes a stream S of Partition subproblems and a value n as its arguments. We regard the subproblems in S in the same way as nodes above. As long as an unsolved subproblem exists in S, no solution to the given instance has been found, and there is another unconsidered element e in the stream representing E, then another stream of subproblems is created by merging S with the stream of positive valued subproblems obtained by subtracting the value of e from each subproblem in S. If some subproblem turns out to have 0 value then processing is terminated with the result that a partition exists, and if S becomes empty then no solution exists.

```
(define slide-merge
 (lambda (n S)
  (letrec
   ([next-node
     (lambda (S1 S2)
      (cond
       [(null? S2) S1]
       [else (let ([x (car$ S1)] [y (- (car$ S2) n)])
               (cond
                [(< y 0) (next-node S1 (cdr$ S2))]
                [(< x y) (cons$ x (next-node (cdr$ S1) S2))]
                [(= x y) (cons$ x (next-node (cdr$ S1) (cdr$ S2)))]
                [else (cons$ y (next-node S1 (cdr$ S2)))]))]))])
   (next-node S S))))

(define partition
 (letrec
  ([partition-stream
    (lambda (E^ S)
     (let ([NEXT (slide-merge (car$ E^) S)])
       (or (zero? (car$ NEXT))
           (partition-stream (cdr$ E^) NEXT))))])
  (lambda (E B)
   (partition-stream E (cons$ B '())))))
```

A sample test for `partition` is

```
  (partition (cons$ 7 (cons$ 6 (cons$ 4 (cons$ 2 '())))) 15).
```

Although this test uses only a finite stream, the procedure `partition` works for infinite streams.

6. Conclusions

We have presented a style of programming using `extend-syntax` and `letrec` which may be applied to a wide variety of combinatorial problems. Writing programs in this style encourages low asymptotic complexity without sacrificing elegance. Moreover, determining the complexity of programs written in this style seems straightforward partially because the overhead of data structure maintenance is eliminated. In our examples we have substituted node structures for arrays thus the arithmetic of array subscripts is eliminated.

7. References

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

[2] Felleisen, M., and Friedman, D. P., "A closer look at export and import statements," *J. Comp. Lang.* **11** (1986), pp. 29-37.

[3] Garey, M., and Johnson, D. S., *Computers and Intractibility: A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco (1979).

[4] Knuth, D., *Fundamental Algorithms: The Art of Computer Programming*, Addison-Wesley (1968), pp. 259-265.

[5] Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B., "Hygienic macro expansion," *Proc. of the 1986 ACM Conf. on Lisp and Functional Programming,* (January, 1986), pp. 151-161.

[6] Kohlbecker, E., and Wand, M., "Macro-by-example: deriving syntactic transformations from their specifications," *Conf. Rec. 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages,* (Munich, January, 1987), pp. 77-84.

Appendix

Below is the compile-time expansion of the test program for topo.

```
(letrec
  ([a (lambda ()
        (set! a (lambda () (error 'a "is part of a cycle")))
        (b) (c) (e)
        (set! a (lambda () "don't care"))
        (writeln 'a))]
   [b (lambda ()
        (set! b (lambda () (error 'b "is part of a cycle")))
        (e) (f)
        (set! b (lambda () "don't care"))
        (writeln 'b))]
   [f (lambda ()
        (set! f (lambda () (error 'f "is part of a cycle")))
        (c)
        (set! f (lambda () "don't care"))
        (writeln 'f))]
   [c (lambda ()
        (set! c (lambda () (error 'c "is part of a cycle")))
        (set! c (lambda () "don't care"))
        (writeln 'c))]
   [e (lambda ()
        (set! e (lambda () (error 'e "is part of a cycle")))
        (f) (c)
        (set! e (lambda () "don't care"))
        (writeln 'e))])
  (a) (b) (f) (c) (e))
```