Towards Increased Productivity of Algorithm Implementation

Jiazhen Cai and Robert Paige

Dept. of Computer Science
New York University/ Courant Institute
251 Mercer St.
New York, NY 10012

December 3, 1993

Towards Increased Productivity of Algorithm Implementation

Jiazhen Cai and Robert Paige

Dept. of Computer Science
New York University/ Courant Institute
251 Mercer St.
New York, NY 10012

ABSTRACT

This paper reports experimental results that support the feasibility of a new transformational approach developed by the authors for implementing complex algorithms correctly and efficiently. The class of algorithms amenable to our methods includes non-numerical graph algorithms. Experiments were performed to measure how this approach affects productivity (in terms of the number of source lines in the implementation divided by manual programming time) and running times. Comparative benchmarks showed that productivity can be increased over a conventional ad hoc approach by factors ranging from 5.1 to 9.9. Preliminary results also showed that the running time of C code produced by this new approach can be as fast as 1.5 times that of tightly coded high quality Fortran.

1. Introduction

Since 1985 the authors have developed a transformational methodology aimed at making it easier to produce reliable and efficient implementations of complex algorithms. In 1991 we completed a transformational programming environment called APTS at the University of Wisconsin in order to implement our new transformational techniques. We used APTS to build two compilers for conducting experiments to test the feasibility of these techniques. One of these compilers translates a major fragment of SETL2 [37] into C [20]. The other translates SQ2+ [5] (a strongly typed functional subset of SETL2 augmented with fixed point expressions) into C.

Experiments were conducted to measure,

1. the programming productivity in SETL2 and SQ2+ compared with C;
2. the running times of C produced from SETL2 and SQ2+ compared with SETL2 running times;
3. the running time of C produced from SETL2 and SQ2+ compared with a commercial package;
4. the translation time needed to produce C from SETL2 or from SQ2+.

In formulating these experiments, programming productivity in a language L is defined operationally (in a crude approximation of programming effort) to be the rate at which source code in L can be produced per unit of manual programming time. For simplicity only program construction is considered. The important costs due to maintenance are excluded.

Based on informal observation we assume that productivity is the same in C and SETL2 (i.e., the same number of lines takes the same time to write in either SETL2 or C); program size is measured in terms of pretty-printed source program lines using the UNIX *wc* function. Based on informal observation, we also assume that the C program produced automatically from SETL2 is roughly the same size as a C program produced manually to implement the same algorithm. Consequently, the productivity improvement obtained from our methods can be determined by the number of lines in a manually produced SETL2 program P divided into the number of lines in the C program produced automatically from P.

Running times are measured using the UNIX *user_time* function (which excludes costs due to input and output) with 20ms accuracy. Translation times are measured in terms of the UNIX *real_time* function (i.e. elapsed time) with 1-second accuracy.

Our definition of programming productivity in terms of source code lines produced per unit time of manual labor needs some justification. It is well known that short but obscure programs can be written in APL [19] (with its overabundance of overloaded operators) or in languages that make use of higher order functions. Complex abstraction mechanisms can also serve to compress programs, sometimes at the expense of clarity. Certainly programs rampant with pointers or other dictions realizing indirection can be obscure, if they are not brief. However, SETL2's compactness has to do with its small but extremely rich repertoire of finite set theoretic dictions. These dictions provide compactness without sacrificing clarity. SETL2 itself lacks pointers and arrays; we further restrict the language to be strongly typed, and to have no higher order functions. We foster reliability by ensuring the absence of many of the major sources of error in the low level C code generated by our translators. This code is guaranteed to have (1) all variables initialized before they are used, (2) all array accesses in bounds, (3) no dangling references, (4) no dereferences of *nil* pointers, (5) no type errors occurring at runtime, and (6) no unintended side-effects due to pointer aliasing.

The preceding arguments suggest that it is not so far fetched to regard the labor involved in producing one line of our low level restricted form of SETL2 to be on a par with a single line of C. However, the authors have no convincing arguments supporting such a claim for the very high level SQ2+ language. To the contrary, it is safe to assume that one line of SQ2+ specification takes longer to produce than one line of C. Consequently, we believe that the number of lines in a manually produced SQ2+ specification P divided into the number of lines in the C code produced automatically from P indicates a general level of productivity improvement that our methods cannot generally exceed.

2. Approach

Alternative transformational approaches for solving the productivity problem differ according to whether the software development process is interactive or fully automatic, whether it is specific to a problem domain or generic, and whether the software that is developed is a prototype or high performance code. Our 'pragmatic compiler' approach is a fully automatic, generic approach that aims to produce code with performance competitive with the best ad hoc methods.

Underlying our transformational methodology is the hypothesis that much of the difficulty involved in implementing complex nonnumerical algorithms is due to human factors that are unrelated to any inherent problem complexity and can be solved by specific transformations implemented in APTS. These factors stem largely from the extensive level of precise detail needed (1) to craft efficient program loops (2) to write the essential imperative bookkeeping operations that serve to maintain program invariants, and (3) to implement tedious but necessary physical data structures that implement conceptually simpler set and map operations. Although this crafting is error prone if done by hand, our transformations allow us to avoid having to specify implementation details without sacrificing the benefits of high performance, because these details can be recovered automatically.

We consider three main transformations that allow prototype programs to be written at three distinct levels of abstraction (illustrated in detail later) and compiled into C. The highest level of abstraction is SQ2+, where programs are written succinctly in terms of least and greatest fixed point expressions. Our dominated convergence transformation [5] can compute these fixed points in terms of high level SETL2 loops that generate 'chaotic' sequences that are less expensive than Tarski sequences [39,40,11]. The next highest level of abstraction is high level SETL2, which allows programs to be written in terms of quantification, set comprehension, nondeterministic search, and other high level set theoretic notations. Our finite differencing transformation [30,24] is a way of implementing costly repeated set theoretic expressions by more efficient incremental lower level SETL2 primitives. Low level SETL2 is our lowest level of prototyping, and allows programs to be written in terms of set theoretic primitives such as membership testing, element addition and deletion, single or multi-valued map retrievals and so forth. Our real-time simulation transformation [25,26,6] selects a conventional physical structure written in C for implementing each such primitive set operations in unit worst case time. The experiments reported here are limited to testing programming productivity improvement at the highest and lowest levels.

Although we are convinced that programming at any of these three levels of abstraction is easier than programming in C, it is important for a programmer (e.g. an algorithm implementor in the experimental operations research community, where run-time performance is vital) to be guided by complexity considerations and to conveniently produce highly efficient code. In order to explain how this is achieved, we define a somewhat vague notion derived from an idea due to Wirth. Let IL be an implementation language (for us it will be C) with a given compiler and target machine, and let PL be a prototyping language with a transformation T mapping programs in PL into equivalent programs in IL. Let the performance of a PL program P mean the run-time resource utilization of T(P) relative to the compiler for IL and its target machine. Then we say that PL has *computational transparency* relative to T if the following conditions hold.

1. (algorithmic analyzability) The performance of every PL program P is as easily predictable by inspection as is T(P);
2. (algorithmic expressiveness) For every PL computable function f, there is a PL program whose performance matches the performance of the best IL program that computes f;
3. (algorithmic writablility) Given an IL program I (i.e., an algorithm denotation) that computes a PL computable function f, it is straightforward to find a PL program that computes f with performance no worse than I.

The design of polynomial time subsets of SQ2+ [7,8] is an attempt to strengthen the computational transparency at the highest level. Integrating our syntactic formulation of amortized complexity with the finite differencing transformation [25,7] is a similar attempt at the second level. However, our most convincing experiments are with prototypes written in the lowest level of abstraction, where they are turned into C by the real-time simulation transformation alone [26,6]. This transformation lies closest to the machine level and contributes most strongly to the subtle constant factors that are so important to high performance code. It is the most generally applicable of our transformations, and accomodates the level of abstract programming with the greatest degree of computational transparency.

3. Transformational Methodology

It is convenient to explain our transformational methodology (and the corresponding three levels of prototyping) by illustrating how it can be used to automatically generate C code implementing Beeri and Bernstein's linear time attribute closure algorithm [3] (which can also solve propositional horn clause satisfiability) from a perspicuous mathematical specification. The code fragments shown below form part of a 'photo' generated APTS derivation performed by our SQ2+-to-C compiler. This derivation resembles a partially automated derivation of the same algorithm that was sketched previously in [25].

The attribute closure problem inputs a finite set x of attributes and a finite set f of functional dependencies, and outputs the set of attributes logically implied by f and x. This set is the smallest superset s of x such that if y -> z is a functional dependency in f, and if y is a subset of s, then z must be an element of s.

We can specify the attribute closure problem in SQ2+ (our highest level of prototyping) as follows:

```
(1)  program aclose ;
     read ( x , f ) ;
     print ( the s : x subset s | ( forall y in domain f |
        ( y subset s impl f { y } subset s )) minimizing # s ) ;
    end program ;
```

where the functional dependencies are represented by a set f of pairs [y,w], where y is a set of attributes and w is a single attribute implied by y. Set cardinality is denoted by # s; domain f stands for the set { y : ∃ y, w | [y,w] ∈ f } of first component values of all pairs in f; expression f { y } represents the image set { w: ∃ w | [y,w] ∈ f }; implication is denoted by impl.

Specification (1) is first transformed into the fixed point normal form:

```
(2)  program aclose ;
     read ( x , f ) ;
     print ( clfp ( x , s + f [ { y : y in domain f | y subset s } ]
          , s ) ) ;
    end program ;
```

where + is set union, and clfp(x, e(s), s) is the smallest fixed point of e(s) with respect to s satisfying the condition that x is a subset of s. If w is a set, then expression f [w] represents the extended image set { u : ∃ y, u | [y,u] ∈ in f and y ∈ w }.

SQ2+ is a high level declaration-free language, where semantic details are isolated in the syntax. In our methodology such details are extracted by Hindley [18] and Milner-style [23] inference, and propagated by transformation down to lower levels of abstraction, where these details are used to justify further transformations. Within program (2) semantic analysis determines the monotonicity of the expression s + f [{ y : y in domain f | y subset s }] relative to parameter s, and finds the types of the sub-expressions (as described in [6]). This information is used to prove that the fixed point in (2) can be computed by the following iterative procedure according to the dominated convergence argument developed in [5]:

```
(3)   a := x ;
      ( while exists x1 in { x3 in f [ { y in domain f | # { x2 in y
         | x2 notin a } = 0 } ] | x3 notin a } )
         a with := x1 ;
      end while ;
```

Program fragment (3) is written in the second level of abstraction, which we call high level SETL2. The finite differencing transformation [30] automatically detects invariants (shown below as named equalities) that should be maintained in order to avoid the costly repeated calculation of the truth set at the top of the while-loop of (3).

```
b{y} = { x2 in y | x2 notin a }
c(y) =  # b { y }
d = { y in domain f | c ( y ) = 0 }
e = f [ d ]
g =  { x3 in e | x3 notin a }
h{z} =  { w in domain f | z in w }
```

A chain rule is applied to maintain the six invariants above within program fragment (3). The main while-loop of the resulting program is shown below.

```
(4)   ....  -- initialize the invariants
      ( while exists x1 in g )
        ( forall x6 in h { x1 } )
          if x6 in domain f then
            if not c ( x6 ) = 0 then
              if c ( x6 ) = 0 + 1 then
                ( forall x16 in f { x6 } | x16 notin e )
                  if x16 notin a then
                    g with := x16 ;
                  end if ;
                  e with := x16 ;
                end forall ;
              end if ;
            end if ;
          end if ;
          c ( x6 ) - := 1 ;
        end forall ;
        if x1 in e then
          g less := x1 ;
        end if ;
        a with := x1 ;
      end while ;
```

Program (4) is written in our lowest level prototyping language, which we call low level SETL2. As a notational convention, assignments of the form

```
a op:= b ;
```

are abbreviations for

```
a := a op b;
```

Also, two forms of procedural quantifiers are used. The first form

```
forall u in s | k(u) loop
  block(u)
end loop;
```

searches through the elements of set s in an arbitrary order (but without repetition). For each element u ∈ s that satisfies loop predicate k(u), the sequence of statements in block(u) is executed. The second form

```
forall u in s loop
  block(u)
end loop;
```

is the same as the first form, but a loop predicate *true* is implicitly assumed. Finally, note that whenever set s is nonempty, then the existential quantifier

$\exists u \in s$

is true and has the side affect of assigning an arbitrary element of s to bound variable u. Thus, the scope of variable x1, which is bound to the existential quantifier at the top of the while-loop of (4), is the entire block inside the while-loop.

Type information, which is obtained earlier by analyzing program (2) and then updated during program transformation, needs to be augmented with subtype information in order to select efficient data structures for implementing data types such as sets and maps occurring in (4). Although it would be highly efficient to obtain this subtype information (as we do for type information) by analysis of the brief text (2) and by efficient transformational propagation (as was proposed in [6])), we did not do so for the following reason. Because subtyping relations depend on the way programs are written, they need to be updated when programs are transformed. To recompute these relations from scratch after each such transformation would take too much compute time. To encode how to update the subtype relation efficiently within each transformation would require a lot of programming effort. Our compromise was to perform subtype inference directly on program (4).

Unlike subtyping relations our typing relation depends on the values of program expressions. Thus, the type relation is preserved by meaning preserving transformations that replace one expression by another. The problem of updating so-called 'intentional' relations in the presence of transformations that preserve meaning is a very important and interesting research problem for the future.

Because our prototyping languages allow input polymorphism, which is lacking in C, we need to replace all type variables with concrete types. In this case, an integer representation for a single uninstantiated type variable denoting attributes (i.e., elements of x, elements of the range of f, and elements of the elements of the domain of f) will be supplied interactively.

Our final transformation implements all set and map datatypes using array and list data structures. This transformation, which is related to Schwartz [34], rests on the discovery of finite universal sets, called *bases*, to be used for avoiding data replication and for creating aggregate data structures that implement logical associative access operations using simpler cursor or pointer access. The approach is to simulate each associative access (e.g. x ∈ s) in real-time; i.e., by a unit time array or pointer retrieval [33,27].

This final transformation first implements program (4) in a lowest level form of SETL2 (not shown here) in which each set can be efficiently implemented as a linked list. The final C code, part of which is shown below, has 435 lines, and runs in time linear in the size of the input and output.

```
(5)   main ( ) {
      struct {     /*   type declarations */
```

```
  int size ;
   ...
} b10;
...
readkey ( b10 ) ;  /* macro calls for reading inputs */
 ...               /* and initializing invariants  */
while ( exists ( & x1 , g ) != NULL ) {
 { cell * x33 ;
  x33 = b6 . recs [ x1 -> data . val ] . h . image -> next ;
  while ( x33 != NULL ) {
   x6 = x33 ;
   if ( b10 . recs [ x6 -> data . val ] . f . mem )
    if ( ! ( b10 . recs [ x6 -> data . val ] . c . image )
       -> data . val == 0 )
     if ( ( b10 . recs [ x6 -> data . val ] . c . image )
        -> data . val == 0 + 1 ) {
      cell * x32 ;
      x32 = b10 . recs [ x6 -> data . val ] . f . image
          -> next ;
      while ( x32 != NULL ) {
       x16 = x32 ;
       if ( ! b6 . recs [ x16 -> data . val ] . e . mem )
        { if ( ! b6 . recs [ x16 -> data . val ] . a .
         mem ) {      /* begin: g with:= x16 */
         cell * x25 , * x26 ;
         x25 = alloccell ;
         x26 = g -> next ;
         x25 -> prev = NULL ;
         x25 -> next = x26 ;
         x25 -> data . val = x16 -> data . val ;
         if ( x26 != NULL )
           x26 -> prev = x25 ;
         else g -> prev = x25 ;
         g -> next = x25 ;
        }                 /* end: g with:= x16 */
        { int x40 , x41 ;  /* begin: e with:= x16 */
         x41 = x16 -> data . val ;
         x40 = b6 . firsts . e ;
         b6 . recs [ x41 ] . e . mem = true ;
         b6 . recs [ x41 ] . e . prev = 0 ;
         b6 . recs [ x41 ] . e . next = x40 ;
         b6 . firsts . e = x41 ;
         if ( x40 != 0 )
          b6 . recs [ x40 ] . e . prev = x41 ;
         else b6 . lasts . e = x41 ;
        }                /* end: e with:= x16 */
       }
      x32 = x32 -> next ;
     }
    }
   (b10.recs[x6->data.val].c.image)->data.val -= 1 ;
   x33 = x33 -> next ;
  }
 }
 if ( b6 . recs [ x1 -> data . val ] . e . mem ) {
  cell * x27 , * x28 ;          /* begin: g less:= x1*/
  x27 = x1 -> prev ;
  x28 = x1 -> next ;
  free ( x1 ) ;
  if ( x27 != NULL )
    x27 -> next = x28 ;
  else g -> next = x28 ;
  if ( x28 != 0 )
   x28 -> prev = x27 ;
  else g -> prev = x27 ;       /* end: g less:= x1*/
 }
 { int x42 , x43 ;
  x43 = x1 -> data . val ;     /* begin: a with:= x1*/
  x42 = b6 . firsts . a ;
  b6 . recs [ x43 ] . a . mem = true ;
  b6 . recs [ x43 ] . a . prev = 0 ;
  b6 . recs [ x43 ] . a . next = x42 ;
  b6 . firsts . a = x43 ;
  if ( x42 != 0 )
   b6 . recs [ x42 ] . a . prev = x43 ;
  else b6 . lasts . a = x43 ;    /* end: a with:= x1*/
 }
}                    /* macro call for printing output */
PRINT_SBSET ( b6 , a ) ;
}
```

The actual real-time simulation of certain set element additions and deletions occurring in program (4) is indicated by comments occurring in program (5). The underlying idea has been described before, for which see [26,6].

One additional comment should be made about the input. Data for the original program specification (1) is presented in an appropriately very high level constant denotation language. This input language differs substantially from the constant denotations of C, and cannot be directly readable by the low level C program (5). APTS has a general purpose procedure to convert SETL data into a form that can be accepted by a C input macro and stored within the initial data structures. Because of some encouraging new algorithmic results [28] we believe that this transformation can actually be performed efficiently at runtime as part of a generic C procedure to read high level input directly.

4. Implementation

All of our experiments have been conducted using the APTS system, whose source code is written in about 15,000 lines of SETL2, a nice manageable size encouraging further development. SETL2 is written in C, which makes APTS portable to more machines than other competing systems such as Centaur [4] and Refine [31]. APTS is seamless in the sense of having no foreign tools and uniform data structures throughout. Hence, we are able to consider every aspect of system functioning for algorithmic improvement, which is also the focus of our research. It is a meta-transformational environment with convenient meta-tools for syntax, semantics, conditional rewriting, and built-in finite differencing supporting a single integrated transformational language called RSL [29]. This language has been designed to support very complex transformations based on algorithm design principles. The language also supports complex program analysis needed to justify such transformations. Our SQ2+ and SETL2-to-C compilers are written in about 9,000 lines of RSL.

RSL compilers make use of the following APTS components. The RULE DATABASE (RDB) contains inference rules specifying program properties, e.g. type or dataflow. Inference rules are specified in a language similar to Datalog [41] augmented with function symbols and primitive pattern matching predicates. An inference engine analyzes the program (being compiled) for properties specified in the RDB, and it outputs the PROGRAM DATABASE (PDB) of finite relations storing the program's properties. The TRANSFORMATION DATA BASE (TDB) contains conditional meaning preserving program transformations of two kinds - rewriting or finite differencing [24]. A transformation engine selects a transformation T by matching T with a portion of the program and by ensuring that the applicability condition for T, when instantiated with the current PDB, is satisfied. It then applies transformation T to the program to obtained a new transformed program. Consequently, the PDB must be updated to be consistent with the new program and the RDB.

The inference and transformation engines make use of our highly efficient bottom up pattern matching algorithm [9]. The inference engine used to calculate RDB relations combines our fast bottom up pattern matching [9] with RETE style pattern matching [16] and seminaive evaluation of Datalog [2,1,41]. Like the RETE algorithm, the APTS inference engine avoids redundant search through the PDB. However, its bottom up tree multi-pattern matching algorithm is faster then the RETE algorithm. Like seminaive evaluation, the APTS inference engine reduces the number of redundant modifications (i.e., actions) to the PDB. However, its differential techniques generate fewer intermediate relations [10].

In the attribute grammar approach to semantic analysis [21] (which is used, for example, by the Synthesizer Generator [32]) the program syntax tree is the main data structure on top of which the attributed dag is laid. Since semantic processing proceeds along navigational paths in the dag, which is limited by the tree, attribute grammars tend to be too inefficient (and too awkward to program) for building program optimizers that depend on global analysis in which widely dispersed program facts must be gathered (by say dataflow chains). For such applications attribute grammar-based systems allow an escape so that efficient semantic analysis can be implemented more naturally and efficiently by a low level C program. In the logic-based approach to semantic analysis used by APTS (and also Centaur), the program syntax tree is regarded merely as one more data structure along with a collection of relations (defined over a variety of domains, only one of which is the set of nodes in the syntax tree). Semantic inference is achieved in a way largely independent of the tree and limited only by the efficiency of logic programming.

5. Experimental Results

Our experimental results are highly encouraging. The latest experiments compared benchmarks for implementations of 10 different algorithms (Figs. 1 and 2). Within Figs. 1 and 2, these algorithms are denoted by reach (use an arbitrary search to compute all vertices reachable along paths in a directed graph from a given subset of vertices), maxind (compute a maximal independent set in an undirected graph), dijkstra1 (Dijkstra's single source shortest path algorithm for dense directed graphs with non-negative edge weights [13]), cycle (cycle decision procedure for undirected graphs), center (find the center of a free tree), aclose (compute the relational attribute closure), topsort (Knuth's topological sorting algorithm [22]), dijkstra2 (same as dijkstra1, but for nondense graphs using 2-heaps), orbit (compute the orbit of a finite function [24]). For nine of these algorithms benchmarks were made comparing executable low level SETL2 specifications with C codes automatically generated from them (see Fig. 1). The sizes of the SETL2 specifications ranged from 25 lines for a maximal independent set algorithm to 103 lines for an implementation of Dijkstra's single source shortest path algorithm [13] with 2-heaps (see dijkstra2 in Fig. 2). The Generated C programs ranged from 133 lines for graph reachability to 766 lines for Dijkstra's algorithm with 2-heaps. All operations were explicitly written into the source code (e.g. heap operations were written explicitly into the SETL2 specification of dijkstra2); there were no hidden library routines. Source lines were measured from pretty-printed text (free from comments) using the UNIX *wc* function.

	SETL2-to-C		SQ2+-to-C	
	$\frac{\text{C lines}}{\text{SETL2 lines}}$	$\frac{\text{SETL2 time}}{\text{C time}}$	$\frac{\text{C lines}}{\text{SQ2+ lines}}$	$\frac{\text{SETL2 time}}{\text{C time}}$
reach	5.1	$13 \to 57$	66	$23 \to 38$
maxind	7.6	$3700 \to 24914$		
dijkstra1	9.9	$43 \to 32$		
cycle	6.6	$926 \to 6550$	80	$124 \to 2183$
center	5.6	$17 \to 27$		
aclose	7.6	$15 \to 38$	87	$15 \to 38$
topsort	6.4	$600 \to 4533$		
dijkstra2	7.4	$36 \to 95$		
orbit	9.2			

Figure 1. Ratio Summary

	SETL2	C	SQ2+	C
reach	26	133	4	265
maxind	25	189		
dijkstra1	42	418		
cycle	36	237	5	402
center	42	234		
aclose	54	409	5	435
topsort	46	295		
dijkstra2	103	766		
orbit	26	238		

Figure 2. lines of Source

Based on assumptions mentioned above, the increase in productivity can be measured by the number of source lines of the compiled C program divided by the number of source lines of SETL2 (cf. Fig. 1). For the nine algorithms mentioned the productivity increase ranged from 5.1 for graph reachability to 9.9 for Dijkstra's single source shortest path algorithm without heaps (see dijkstra1 in Figs. 1 and 2). If, instead of using source lines, program size were measured in terms of words (i.e., strings of characters delimited by a space, tab, or newline) calculated by the UNIX *wc* function, then the productivity increase would be even better, ranging from 8.6 for an algorithm to compute the center of a free tree to 16.8 for dijkstra1. When program size is measured in terms of words, our pretty-printed source programs had spaces delimiting every token.

All running times were measured on a SPARC 2 with 32 megabytes of physical memory. The numbers resulted from averages of over 3 or more trial runs. Times were measured by the UNIX *user_time* function. Inputs were produced by a random graph generator paramaterized to control graph density. Graph sizes ranging from 1000 to 10,000 edges were used for 8 of these algorithms (cf. Fig. 1). Speedups tended to improve as the problem size increased, but were uniformly impressive. For small inputs (graphs with 1000 edges) speedups (SETL2 running time divided by C running time) varied from 13 for graph reachability to 3700 for a maximal independent set algorithm. For large inputs (graphs with 10,000 edges) speedups ranged from 27 for the tree center algorithm to 24,914 for maximal independent sets. The enormous speedups registered for three of the eight algorithms are due to a hidden copy problem in the SETL2 compiler implementation. It is fair to say that our approach seems to yield a 30-fold improvement over the best SETL2 performance. This also corroborates independent benchmarks conducted by Kirk Snyder, the implementor of the SETL2 compiler.

The C generated from the dijkstra2 specification was also compared to a highly crafted and efficient Fortran implementation due to Prof. Michael Grigoriadis. His code was written years ago and without anticipation of this project. The C code generated from a SETL2 specification performed no worse than 50% slower than the Fortran code for all inputs of 10,000 edges.

The translation rate from SETL2 to C was the weakest result, and indicates the need for further improvements to the APTS system. Total translation time ranged from 4 minutes for graph reachability to 49.5 minutes for dijkstra2. The average translation rate was 3.3 lines of SETL2 translated into 24 lines of C per minute. Clearly, the translation rate needs to be improved to facilitate scaled-up applications.

Three of the algorithms implemented by the SETL2-to-C translator were also implemented by the SQ2+-to-C translator. These were graph reachability, cycle testing, and attribute closure [3,25]. For these implementations the SQ2+ specifications were only 4 and 5 lines long (cf. Fig. 2). The ratio of C lines divided by SQ2+ lines ranged from 66 to 87, but one line of SQ2+ should be expected to take somewhat longer to produce than one line of C (cf. Fig. 1). Therefore, no claim of a specific productivity improvement can be made. However, such an assertion might still be possible, because 75 to 80 percent of the lines in our SQ2+ specifications are filler (e.g., enders) as compared with about 33 percent of low level SETL2 programs. If a formal correctness proof were considered a programming obligation, then SQ2+ programming might even be easier per line than C. Nevertheless, we can use these numbers to draw a conservative negative expectation - namely, that our generic methods are not likely to achieve productivity improvements beyond factors of 66 to 87. To exceed these bounds, we believe that a domain specific or interactive approach would be necessary.

The derivation of attribute closure sketched earlier takes about 20 minutes. During execution the maximum size of the APTS knowledge base (i.e. the RDB, TDB, and PDB) is about 5 megabytes. The C program generated from SQ2+ specification (1) runs about 30 times faster than the SETL2 program shown in (4), and has 87 times more source lines than (1). It is interesting to note from Fig. 1 that the attribute closure C program generated by our SETL2-to-C translator from a much more detailed low level SETL specification than (1) runs no faster than the C program shown in (5). It is also interesting to note that if Kirk Snyder's and our estimate that his SETL2 is about 30 times slower than hand coded C is accurate, then the performance of program (5) would be comparable to good hand code.

Finally, an experiment was performed to test the potential for scaled up applications. Raytheon, an industrial affiliate of the DOD-sponsored prototech projects, provided the investigators with a moderate sized prototype software package to generate tracks from input radar plot data using an alpha-beta filter with constant scalar coefficients. "The program processes a file of radar plot data and produces a Postscript output file which shows the original plots and program generated radar tracks superimposed on a plan of the radar site." The Raytheon code, which comprises 9 packages and classes and over 1000 lines of source, makes use of every aspect of SETL2. The experiment involved translating the whole main SETL2 module into C, which would interoperate with the remaining SETL2 modules. The productivity improvement (i.e., 477 lines of generated C plus SETL2 interoperability code divided by 64 lines in the SETL2 main module) was 7.7. The overall speedup was 2.6, which was surprisingly good, considering the tremendous overhead in the way that SETL2 currently implements interoperability with C. Overhead is due to added levels of procedure calls, and massive datatype conversion, because only strings are currently allowed to pass between C and SETL2. Recall also, that only one out of the 9 modules was translated into C.

6. Previous Results

The SETL research program directed by Jack Schwartz from about 1971 until 1989 [35] has had the most positive influence on our work. Schwartz effectively demonstrated the conceptual simplicity of the SETL language with its rich repertoire of universal set theoretic dictions, and introduced many important concepts in an attempt to solve the formidable task of implementation and optimization. SETL's mathematical value semantics made costly hidden copy operations a problem. Dynamically allocated aggregate datatypes with arbitrary levels of nesting made memory management difficult. Dynamic weak typing, massive overloading, and the prevalence of operations that relied on associative access compounded the runtime costs and the complexity of an efficient implementation.

The SETL optimizer dealt with the hidden copy problem by making a conservative estimate of when a set could be updated destructively or not, and by using a run-time bit (that could be turned on but not off) to indicate this fact. Snyder's SETL2 implementation uses reference counts for the same purpose. The SETL system relied on compaction garbage collection for dynamic allocation, whereas SETL2 uses collections of stacks. SETL used an elaborate hashing mechanism as a default implemention of associative access for arbitrary datatypes. It's optimized data structures combined hashing with bit vectors and plex structures to reduce the amount of replicated data and the expense of hashing [12,33,17]. SETL2 uses tries, and attempts no compile-time optimization. SETL attempted to generate more efficient code using type analysis [40] with an accuracy up to three levels of nesting for sets, maps, and tuples (in order to guarantee termination of type inference). SETL2 has only run-time type checking. The best implementations of both SETL and SETL2 are interpreted systems.

Framed in the context of previous results coming out of the SETL project [35], our results are surprising. The SETL project concentrated on methods to improve the performance of a default implementation that had hashed data structures for sets and maps. An optimizer of over 20,000 lines of SETL was produced, but was never fully operational [33,17,35]. Two dissertations evaluating the SETL optimizer and data elaboration language failed to observe any speedup [36,38]. Straub [38] reported that annotating SETL programs to indicate how to implement sets and maps rarely improved performance and often slowed it down considerably. He attributed much of the slow speed to the inefficiency of hashing that could not be eliminated in the optimized SETL data structures. A recent informal set of comparative benchmarks by Prof. Fritz Henglein showed a reduction in performance using the optimizer applied to declaration-free SETL programs, even for toy examples.

Informal benchmarks conducted by Kirk Snyder show that Snyder's SETL2 implementation is about 4 times faster than SETL, and about 30 times slower than hand coded C, which is consistent with our benchmarks. However, our benchmarks indicate that SETL2 suffers from hidden copy operations that can degrade performance by arbitrary orders of magnitude. We believe that much of this could be fixed by a better utilization of reference counts and improved code generation.

It is also interesting to note Doberkat's related effort to mechanically translate SETL into Ada [14] within the Mentor environment [15]. Comparative benchmarks indicate that the Ada code generated from SETL ran between two and three times slower than the SETL runtime, a result of simple default data structures for sets and no data structure optimization [6].

There are several factors that can explain our benchmarks. Certainly, our SETL2-to-C translator eliminates interpretive overhead that exists in both SETL and SETL2. However, we do not believe that this is a major factor. In Dave Shields's empirical study [36] the interpretive overhead in the SETL system [35] was reported to contribute between 50% and 80% of the runtime performance. Assuming this is also true for SETL2, then a factor of between six and fifteen from the thirty-fold improvement in SETL2-to-C translation over the SETL2 interpreted system results from real-time simulation.

The real-time simulation transformation fares better than the SETL optimizer for several reasons. We restrict SETL2 in two major ways. First we require SETL2 to be strongly typed. This greatly simplifies the allowable types (e.g. in the absence of recursive types, we disallow hereditarily finite types). It also makes our Hindley/Milner style type inference mechanism much sharper than the inferencing due to Tenenbaum, because we can detect types of sets, maps, and tuples of arbitrary depth of nesting. Consequently, our analysis for inclusion and membership relations is sharper. Such information allows us to statically allocate storage for data that would otherwise be allocated dynamically.

The second reason is in our restricting SETL2 to have pointer and value semantics simultaneously (analogous to the notion of an Ada program being 'erroneous' when it has a different meaning under call by reference and call by value). Consequently, we don't use reference counts and don't suffer hidden copy operations.

Finally, there were conceptual as well as engineering problems with the SETL data structure optimization. The fundamental data structure that formed the spine of a data structure aggregate was the hash table. Hashing was used heavily in inputting data, in forming the initial optimized data structures, and was used to implement any set that was iterated over. The heuristic nature of the SETL optimizations and the unpredictability of expensive hidden operations was observed by both Shields and Straub.

Our real-time simulation provides new theoretically sound conceptual underpinnings to data structure selection with precise guarantees of speedup. Our transformation is grounded in basic algorithm theory by forming data structure aggregates from arrays and linked lists instead of hash tables. Iteration over a SETL2 set does not force us to resort to hashing. Our proposal to input data and to form initial data structures is highly efficient without making use of hashing [28].

7. Conclusion

Algorithms (like those used in Operations Research) are often complex; their actual performance on different data sets are often unpredictable without extensive testing. There is a continuous search for hybrid algorithms that perform well on a wide variety of input instances (which is why an approach based solely on fixed libraries is not likely to work). The ad hoc implementations of these algorithms are rendered in a low level language (like Fortran or C) with notations that stress high performance over clarity. Consequently, implementation design is labor intensive, codes are unreliable, and modification is difficult.

The alternative transformational method proposed here involves construction of an executable prototype implementation (in low level SETL2) close to the text book form of an algorithm. Such a prototype is brief, understandable, and easier to verify empirically or analytically. It is also easier to maintain, modify, and document. It is transformed mechanically into a more complex but correct low level C program (assuming the prototype is correct) with certain theoretical performance guarantees.

Our future research plans are to use APTS to generate more complex and larger implementations, to improve on the 5-fold productivity increase observed in the preliminary experiments reported here, and to produce C programs with run-time performance comparable or exceeding high quality hand code. As a challenging case study, we plan to translate the entire APTS system into C. Secondly, we plan to reduce the labor involved in writing and maintaining compilers within APTS. Finally, we plan to speedup compiler execution in APTS.

A convincing experiment demonstrating a 5-fold productivity improvement on scaled-up, efficient algorithm implementations would be a breakthrough in software engineering. In addition, the speedup in the APTS translation rate would make this new software development technology practical. It would help to bridge a widening gap between theory and practice by making it easier to test the practical potential of a number of complex 'theoretically fast' algorithms. It should also facilitate the construction of compilers for parallel architectures.

Acknowledgements

The research of both authors was partially supported by Office of Naval Research Grant No. N00014-90-J-1890 and Air Force Office of Scientific Research Grant No. AFOSR-91-0308. Part of this research was done while both authors were visiting Purdue University. The research of J. Cai was partially supported by National Science Foundation Grant CCR-9002428. The research of R. Paige was partially supported by National Science Foundation Grant MIP-9300210.

References

1. Bancilhon, F., "Naive Evaluation of Recursively defined Relations," in *On Knowledge-Base Management Systems*, ed. Brodie, M and Mylopoulos, J, pp. 165-178, 1986.

2. Bayer, R., *Query evaluation and recursion in deductive database system*, 1985. unpublished manuscript

3. Beeri, C. and Bernstein, P., "Computational Problems Related to the Design of Normal Form Relation Schemes," *ACM TODS*, vol. 4, no. 1, pp. 30-59, 1979.

4. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., and Pascual, V., *Centaur: the system,* INRIA, 1987. Rapports de Recherche

5. Cai, J. and Paige, R., "Program Derivation by Fixed Point Computation," *Science of Computer Programming*, vol. 11, pp. 197-261, 1988/89.

6. Cai, J., Facon, P., Henglein, F., Paige, R., and Schonberg, E., "Type Transformation and Data Structure Choice," in *Constructing Programs From Specifications*, ed. B. Moeller, pp. 126 - 124, North-Holland, 1991.

7. Cai, J. and Paige, R., "Binding Performance at Language Design Time," in *ACM POPL*, pp. 85 - 97, Jan, 1987.

8. Cai, J. and Paige, R., "Languages Polynomial in the Input Plus Output," in *Algebraic Methodology and Software Technology*, ed. M. Nivat, C. Rattray, T. Rus, and G. Scollo, Workshops in Computing Series, pp. 287 - 302, Springer-Verlag, 1992.

9. Cai, J., Paige, R., and Tarjan, R., "More Efficient Bottom-Up Multi-Pattern Matching In Trees," *Theoretical Computer Science*, vol. 106, no. 1, pp. 21 - 60, Nov. 1992.

10. Cai, J., *A Language for Semantic Analysis,* New York University, 1993. Technical Report

11. Cousot, P. and Cousot, R., "Constructive versions of Tarski's fixed point theorems," *Pacific J. Math.*, vol. 82, no. 1, pp. 43-57, 1979.

12. Dewar, R., Grand, A., Liu S. C., Schwartz, J. T., and Schonberg, E., ‘‘Program by Refinement, as Exemplified by the SETL Representation Sublanguage,’’ *TOPLAS*, vol. 1, no. 1, pp. 27-49, July, 1979.

13. Dijkstra, E. W., ‘‘A note on two problems in connexion with graphs,’’ *Numer. Math.*, vol. 1, no. 5, pp. 269-271, 1959.

14. Doberkat, E. and Gutenbeil, U., ‘‘SETL to Ada - tree transformations applied,’’ *Information and Software Technology*, vol. 29, no. 10, pp. 548-557, Dec. 1987.

15. Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B., ‘‘Programming environments based on structured Editors: the Mentor Experience,’’ in *Interactive Programming Environments*, ed. D. Barstow, H. Shrobe, and E. Sandewall, McGraw-Hill, 1984.

16. Forgy, C., ‘‘RETE, a fast algorithm for the many patterns many objects Match problem,’’ *Aritifical Intelligence*, vol. 19, pp. 17-37, 1982.

17. Freudenberger, S., Schwartz, J. T., and Sharir, M., ‘‘Experience with the SETL Optimizer,’’ *ACM TOPLAS*, vol. 5, no. 1, pp. 26-45, 1983.

18. Hindley, J., ‘‘The principal type-scheme of an object in combinatory logic,’’ *Trans. AMS*, no. 146, pp. 29 - 60., 1969.

19. Iverson, K., *A Programming Language,* Wiley, New York, 1962.

20. Kernighan, B. and Ritchie, D., *The C Programming Language,* Prentice-Hall, 1978.

21. Knuth, D. E., ‘‘Semantics of Context-free Languages,’’ *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127-145, 1968.

22. Knuth, D. E., *The Art of Computer Programming, Vol 1: Fundamental Algorithms,* Addison-Wesley, 1973.

23. Milner, R., ‘‘A theory of type polymorphism in programming,’’ *J. Computer and System Sciences*, vol. 17, no. 3, pp. 348 - 375, 1978.

24. Paige, R., ‘‘Programming With Invariants,’’ *IEEE Software*, vol. 3, no. 1, pp. 56-69, Jan, 1986.

25. Paige, R. and Henglein, F., ‘‘Mechanical Translation of Set Theoretic Problem Specifications Into Efficient RAM Code - A Case Study,’’ *Journal of Symbolic Computation*, vol. 4, no. 2, pp. 207-232, Aug., 1987.

26. Paige, R., ‘‘Real-time Simulation of a Set Machine on a RAM,’’ in *ICCI ’89*, ed. R. Janicki and W. Koczkodaj, Computing and Information, Vol II, pp. 69-73, 1989.

27. Paige, R. and Henglein, F., ‘‘Mechanical Translation of Set Theoretic Problem Specifications Into Efficient RAM Code - A Case Study,’’ in *EUROCAL 85*, ed. B. Caviness, Lecture Notes in Computer Science 204, pp. 554-567, Springer Verlag, 1985.

28. Paige, R., ‘‘The High Cost of Reading,’’ in *Liber Amicorum - Lambert Meertens 1966 - 1991*, ed. Johan Jeuring, pp. 55 - 65, CWI, 1991.

29. Paige, R., ‘‘APTS Reference Manual,’’ unpublished manuscript, 1993.

30. Paige, R. and Koenig, S., ‘‘Finite Differencing of Computable Expressions,’’ *ACM TOPLAS*, vol. 4, no. 3, pp. 402-454, July, 1982.

31. *Refine User’s Guide Version 3.0,* Reasoning Systems, May 1990.

32. Reps, T. and Teitelbaum, T., *The Synthesizer Generator,* Springer-Verlag, 1988.

33. Schonberg, E., Schwartz, J. T., and Sharir, M., ‘‘An Automatic Technique for Selection of Data Representations in in SETL Programs,’’ *ACM TOPLAS*, vol. 3, no. 2, pp. 126-143, Apr, 1981.

34. Schwartz, J.T., ‘‘Automatic Data Structure Choice in a Language of Very High Level,’’ *CACM*, vol. 18, no. 12, pp. 722-728, Dec, 1975.

35. Schwartz, J., Dewar, R., Dubinsky, D., and Schonberg, E., *Programming with Sets: An introduction to SETL,* Springer-Verlag, 1986.

36. Shields, R., ‘‘Measuring SETL Performance,’’ Ph.D. Dissertation, Computer Sciences Dept., NYU, 1983.

37. Snyder, K., *The SETL2 Programming Language,* Technical Report 490, Courant Institute/ New York University, New York, NY 10012, 1990.

38. Straub, R., ‘‘Taliere: An Interactive System for Data Structuring SETL Programs,’’ Ph.D. Dissertation, Dept. of Computer Science, New York University, New York, NY, May 1988.

39. Tarski, A., ‘‘A Lattice-Theoretical Fixpoint Theorem and its Application,’’ *Pacific J. of Mathematics*, vol. 5, pp. 285-309, 1955.

40. Tenenbaum, A., *Type Determination for Very High Level Languages*, New York University, Dept. of Computer Science, Oct, 1974. appears in Courant Computer Science Report 3

41. Ullman, J., *Principles of Database and Knowledge-Base Systems, I,* Computer Science Press, 1988.