Viewing A Program Transformation System At Work

Robert Paige*
Computer Science Department, New York University/Courant Institute
251 Mercer St., New York, NY 10012, USA
paige@cs.nyu.edu

Abstract

How to decrease labor and improve reliability in the development of efficient implementations of nonnumerical algorithms and labor intensive software is an increasingly important problem as the demand for computer technology shifts from easier applications to more complex algorithmic ones; e.g., optimizing compilers for supercomputers, intricate data structures to implement efficient solutions to operations research problems, search and analysis algorithms in genetic engineering, complex software tools for workstations, design automation, etc. It is also a difficult problem that is not solved by current CASE tools and software management disciplines, which are oriented towards data processing and other applications, where the implementation and a prediction of its resource utilization follow more directly from the specification.

Recently, Cai and Paige reported experiments suggesting a way to implement nonnumerical algorithms in C at a programming rate (i.e., source lines per second) that is at least five times greater than a conventional approach in which C programs are written entirely by hand [12]. The runtime performance of the C programs produced by this new approach was shown to be comparable to good hand code. The proposed software development methodology makes use of fully automatic, generic program transformations that capture algorithm design principles. This paper discusses some of the ideas underlying the transformational methodology, and illustrates these ideas through explanatory examples of the APTS system at work.

1 Introduction

A new program development methodology for implementing nonnumerical algorithms has been developed. The methodology makes use of generic, fully automatic program transformations that can produce high performance code. These program transformations have been tested on paper by using them for algorithm inventions and explanations. The methodology is implemented in the APTS transformational programming system [46]. Feasibility of the approach has been tested by conducting experiments with APTS to compare productivity of the new methodology with conventional hand-crafted programming. This paper explains features of the methodology through an annotated transcript that records how the APTS system automatically translates a 3-line mathematical specification of Propositional Horn Clause Satisfiability into a several-hundred-line C program with worst case running time and space linear in the input space.

*This research is partially supported by Office of Naval Research Grant N00014-93-1-0924, Air Force Office of Scientific Research Grant AFOSR-91-0308, and National Science Foundation Grant MIP-9300210.

1.1 Transformational Methodology

The transformational methodology is based on three generic (as opposed to domain specific) fully automatic transformations. The first transformation makes use of a dominated convergence argument to compute least and greatest fixed point operations efficiently [13]. The second transformation, finite differencing, uses a syntactic characterization of the amortized complexity to maintain and exploit program invariants [43,10] in order to turn costly repeated calculations into more efficient incremental counterparts. One major consequence is a way to implement efficient searching for subsets that satisfy a property (exemplified by a database index used to compute a selection query in relational algebra [55]). The final transformation (data structure selection) simulates associative access on a sequential RAM in real-time [44,11,15]. The result is a way to implement primitive set theoretic operations efficiently. All three of these transformations are combined to form functional specification languages (making use of fixed point operations) that can be compiled into programs with worst case asymptotic time and space bounded by a specific function of the input and output space [48,10,17].

The problems of efficient implementation of associative access, of indexes to support searching a set for a property, and of fixed points have concerned the programming language and database communities for over twenty years (e.g., included within the programming language community are Jay Earley at UC Berkeley [24,25,26], Jim Morris at Los Alamos [41,62], Low and Rovner at the U. of Rochester [38,51], Barstow [4] and Kant [35] at Stanford, Fong and Ullman at Princeton [27], Jack Schwartz at NYU [53,54,49,52,21], and Cousot and Cousot [19]; included within the database community are Naqui and Henschen [42,31], Aho and Ullman [1], Selinger [55], Banchilhon [2], Ramakrishnan [3], and Willard [63,64]).

One key contribution of our work has been to provide *a priori* algebraic and syntactic reasoning about the program speedup that would result from applying transformations (see [10,44,15,17]. One key open problem is to reformulate these transformations for parallel models of computation.

1.2 Algorithm Design

How can we evaluate a transformational methodology aimed at algorithm implementation? Some utility of a transformational methodology might be seen in using it to 'explain' or prove well known algorithms. However, greater viability would be demonstrated if the methodology could help to uncover new deep structures that lead to improved solutions. Then its impact on algorithmics is likely to be greater.

The main point of our algorithm design work is to test the viability and utility of our transformational methodology before launching into an implementation. The methodology has been used to invent new algorithms for propositional Horn Clause Satisfiability on a sequential pointer machine [48], transformation from regular expressions to DFA's [18], and ready simulation [8]. We are currently working on transformational explanations and proofs of parallel algorithms.

Recent algorithm inventions due to Jeuring [33] and Meertens [40] add credibility to the calculational approach to algorithm derivation due to Bird [7] and Meertens [39]. Smith has also used program transformations in algorithm design [57].

A major benefit of algorithm design by transformation is that it produces an actual program along with an algorithm. Consequently, we can conduct experimental benchmarks to obtain computational results that complement the theoretical improvements. This gives our work a potential for pragmatic as well as theoretical impact.

1.3 Transformational System

The APTS meta-transformational environment was built by Cai and Paige in order to implement the preceding transformations [46]. The whole system consists of only 15,000 lines of SETL2 [58] source code (including comments), and it makes use of no foreign tools. APTS has meta-tools for syntax, semantics, conditional rewriting, and built-in finite differencing (an abstract mechanization of top down stepwise refinement [22]) supporting a single integrated transformational language called RSL. This language has been designed to implement very complex transformations together with the program analysis needed to justify when such transformations can be applied. Compiler construction is a major application.

Compilers written in RSL make use of the following APTS components. The Rule Database (RDB) contains inference rules that define relations storing program properties, e.g. type or dataflow. These relations may be defined over a variety of domains including conventional domains (booleans, integers, strings), the domain of abstract program points (i.e., simple contexts), program terms (the syntactic values that occur at program points), and lisp-like s-expressions (a general domain used to embed arbitrarily deeply nested types amenable to first order unification). Inference rules are specified in a language similar to Datalog [61] augmented with function symbols and primitive pattern matching predicates. The APTS inference engine analyzes the program (being compiled) for properties specified in the RDB, and it outputs the Program Database (PDB) of finite relations (sets of tuples that represent ground terms) storing the program's properties.

The Transformation Database (TDB) contains conditional meaning preserving program transformations of two kinds - rewriting or finite differencing. 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 the efficient bottom up pattern matching algorithm of Cai, Paige, and Tarjan [16]. The inference engine used to calculate RDB relations [14] combines this bottom up pattern matcher with RETE style pattern matching [28] and seminaive evaluation of Datalog [5,2,61].

Other transformational systems currently in use with capabilities similar to APTS are Refine [50], KIDS [56] (which is built on top of Refine), Centaur [9], and ASF+SDF[36].

The most pressing and perhaps most difficult open problem in APTS system research is in devising a scheme to maintain the consistency of PDB relations incrementally in the presence of conditional rewriting. Current methods work for extensional relations such as *monotone* and *type* (which are invariant with respect to equivalence-preserving expression replacement). Automatic methods fail for intensional relations such as *free* and *bound* variables, *control flow*, and most subtype relations.

Another major problem has to do with ensuring reliability of the program development process. Currently, all transformations used within APTS preserve program semantics, but this is only proved on paper outside the system. Since APTS is capable of unbridled production of dangerously large quantities of code without any human intervention or oversight, the absence in APTS of machine-assisted meta-level support to prove transformations correct is a major deficiency that needs to be overcome.

1.4 Productivity Experiments

We have used APTS to build two compilers for conducting experiments to test the feasibility of the transformational methodology. One of these compilers translates a major fragment of SETL2 into C. The other translates SQ2+ [13] (a strongly typed functional subset of SETL2 augmented with fixed point expressions) into C. Our SQ2+ and SETL2-to-C compilers are written in about 10,000 lines of RSL. Perhaps the most interesting aspect of these compilers is the subtype inference method used to implement data structure selection based on the real-time simulation of an abstract set machine on a RAM [15]. The subtype inference algorithm was shown to be based on a logic in which all type variables represent unique finite minimal domains that could be computed efficiently. These domains are the kernels of data structure aggregates supporting the real-time simulation of set theoretic primitive operations.

The preceding translators were used in productivity studies, and the results were highly encouraging and surprising [12]. These results suggest that our approach can achieve at least a five-fold improvement in programming productivity of nonnumerical algorithm implementation (compared with conventional hand-crafted programming) without sacrificing more than 50% in run-time performance. The experiments tested productivity in C for sequential machines, and the largest C program was 12 pages long. Although this approach conserves manual programming labor, it falls short of a practical technology, because the automated software production costs are too high. That is, the rate at which user-supplied mathematical specifications are automatically translated into efficient C code is too slow - about 3.3 lines of SETL2 translated into 24 lines of C per minute. A solution to this problem is sketched in Section 3.

2 Transcript of an APTS Derivation

It is convenient to discuss aspects of our transformational methodology by tracing through and commenting on an actual transcript of mechanical program development in APTS. The transcript demonstrates how a formal specification of the relational attribute closure problem (which also solves Propositional Horn Clause Satisfiability), written in only a few lines of the SQ2+ language, can be turned automatically into a C program of several hundred lines with worst case running time and space linear in the input space.

Bernstein and Beeri [6] and Dowling and Gallier [23] independently solved this problem with the same asymptotic complexity as our compiled C program, and all three solutions use the power of address arithmetic to directly access arrays. In [48] Paige and Henglein showed on paper how program transformations like the ones demonstrated in the transcript below could be used to obtain a pointer machine implementation [37,60] of attribute closure with linear worst case time and space AFTER the input is converted to a necessary internal form. That result together with the input processing algorithms in [47] provide a full pointer machine solution in worst case time and space linear in the length of the external input string. Consequently, it can be shown that the attribute closure specification given in the transcript resides within a variant of the linear time language described in [10].

The transcript begins just below with a directive to startup the APTS system running under UNIX. The SETL2 runtime interpreter 'stlx' is invoked with the SETL2 program 'apts', which resides in the SETL2 executable module library. APTS also uses a module library, which like its SETL2 counterpart, is declared with a UNIX setenv command. The first thing that APTS does is to recognize its module library, and to bring the module containing the executable form of RSL syntax into the user environment. All commands executed by APTS are written in RSL.

The first command loads an APTS module containing procedures for the SQ2+-to-C translator.

These procedures then become part of the environment, and can be executed. We type a command to print out their names.

```
Script started on Mon Jun 20 16:44:37 1994
himinhrjot 21 > stlx apts
Begin transformational programming
type command
>:loadcm squb;
>:help cmrules;
{"front", "ccload", "sqload", "ubload", "backend", "frontend", "sq2x", "ccgen",
"refcount", "setlx", "nsetlx", "raysetlx", "fdsetlx", "testsetl", "stlcload",
"converge", "fpform", "ubnorm", "testsq2", "stlstlub", "sqgen", "srgen",
"ubgen", "myclean", "stlcgen", "setlc", "gencomps", "gentabs", "commonback"}
```

The RSL source code for procedure 'sq2x', which is the main procedure for the SQ2+-to-C translator, appears below. The body of this procedure contains sequentially executed RSL commands that include calls to other procedures.

```
procedure sq2x ( .input , .output ) ;
  sqload ;           -- load modules for SETL syntax, rewriting, finite
  initdb;            -- differencing, and inference
  parse .input;      -- compile the SQ2+ specification
  unparse .input;
  frontend;          -- apply dominated convergence and finite differencing
  loadrw rwub;       -- load new rewriting and inference modules
  loadrc rcub;
  backend(.output); -- apply real-time simulation, and obtain C program
end procedure ;
```

The transcript continues with the translation of the following SQ2+ specification of attribute closure (stored in file aclose.src) into C,

```
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 ;
```

where x is a set of given attributes, and f is a set of functional dependencies. This specification says that the set of attributes that are functionally determined by x is the smallest superset s of x such that if y $\rightarrow$ z is a functional dependency in f, and if y is a subset of s, then z must be an element of s. The operator # means set cardinality, domain f represents the set $\{u: \exists\ v \mid [u,v] \in f\}$, image set f{y} stands for the set $\{v: [y,v] \in f\}$, and the extended image set f[Q] for set valued argument Q is $\{v: \exists\ y \in Q \mid [y,v] \in f\}$.

Procedure 'sq2x' is called with parameters that refer to the file 'aclose.src' containing the attribute closure specification, and the file 'aclose1000.in' containing its runtime input. (The input instance is of size 1000, and was generated by a random number generator for the productivity experiments mentioned earlier.)

```
>:sq2x(aclose,aclose1000);
The program just below should be in fixed point form.
    program aclose ;
      read ( x , f ) ;
      print ( clfp ( x , s + f [ { y : y in domain f | y subset s } ] , s ) ) ;
    end ;
```

The original specification has now been transformed by straightforward rewriting rules into the fixed point normal form just above, where + is set union, and clfp(x, e(s), s) is the smallest fixed point of set expression e(s) with respect to s satisfying the condition $x \subseteq s$. The SQ2+ language is a strongly typed variant of the Curry/Hindley type discipline for the λ-calculus [20,32]. It is parametric since it contains type variables, but it is not polymorphic. Based on the SQ2+ type inference system described in [15], the APTS inference engine computes types for each program expression. It also computes other program properties, such as monotone expressions, the bound and free variables of expressions, and a kind of abstract syntax. These properties are all stored as relations in the the APTS PDB (Program Database).

The binary 'type' relation is defined over the domain of program terms (the first component) and s-expressions representing types (the second component). The binary relation 'mono' is defined over program terms. In response to the 'help' commands below, APTS prints out the 'type' and 'mono' relation stored in the PDB.

```
>:help type;
     clfp ( x , s + f [ { y : y in domain f | y subset s } ] , s )  :  [ set t5 ]
     { y : y in domain f | y subset s }  :  [ set [ set t5 ] ]
     domain f  :  [ set [ set t5 ] ]
     y  :  [ set t5 ]
     f [ { y : y in domain f | y subset s } ]  :  [ set t5 ]
     y subset s  :  bool
     f  :  [ set [ tuple [ [ set t5 ] t5 ] ] ]
     x  :  [ set t5 ]
     s  :  [ set t5 ]
     s + f [ { y : y in domain f | y subset s } ]  :  [ set t5 ]
>:help mono;
     s is monotone in s
     x is monotone in s
     s + f [ { y : y in domain f | y subset s } ] is monotone in s
     y is monotone in s
     f [ { y : y in domain f | y subset s } ] is monotone in s
     f is monotone in s
     domain f is monotone in s
     { y : y in domain f | y subset s } is monotone in s
     y subset s is monotone in s
```

RSL code for two of the inference rules used to compute types are shown below,

```
match(%expr, .x%) | null(z, type(%expr, .x%, z)) ->
  bind(.t, newatom(t)) and type(%expr, .x%, .t);
match(%expr, .x + .y%) | type(%expr, .x%, .t) ->
  type(%expr, .x + .y%, .t) and type(%expr, .y%, .t);
```

The first rule states that for every program point p in the SQ2+ specification with syntactic category 'expr' (i.e., for every occurrence of an expression) such that 'type' is not defined for the term r stored at p, create a new type variable t, and make t the type for r (i.e., store pair [r,t] in the 'type' relation). The second rule is more complicated. It states that for every program context p1 + p2, where p1 and p2 are program points containing terms t1 and t2 respectively, if the 'type' relation contains pair [t1,t], then perform the following actions. Obtain a most general unifier t' of t and the types of t2 and t1+t2 if these types already exist, and let t' = t otherwise. Then store pairs [t1,t'], [t2,t'], and [t1+t2,t'] in relation 'type' after deleting each of the types for t1, t2, and t1+t2 that may exist.

Inference rules in APTS may be performed in any order. There is a notion of 'safe' rules (under a closed world assumption) to allow for limited forms of negation and built-in predicates, and a semantics similar to the logic databases found in [61]. More details about the inference rules and how they are implemented in APTS can be found in [14].

The conditional rewriting transformation 'nminfp' can turn the functional attribute closure specification above into a simple imperative program that computes the conditional least fixed point by dominated convergence. The transformation is displayed in APTS using the 'help' feature.

```
>:help nminfp;
lhs code to be matched
    print ( clfp ( .w , .s + .k , .s ) ) ;
rhs replacement code
      .x := .w ;
      while exists .z in ( .k - .x ) loop
        .x with := .z ;
      end loop ;
      print ( .x ) ;
pattern expansion code is
    readvar ( .x ) and genvar ( .z ) and subst ( .k , .s , .x )
database action code is
      type ( .x , [ set , .t ] ) and type ( .z , .t ) and type ( %
        expr , .k - .x %, [ set , .t ] )
enabling predicate
    mono ( % expr , .s + .k %, .s ) and type ( .s , [ set , .t ] )
```

This rule applies to any program point that is matched by the left-hand-side pattern and that satisfies the enabling predicate. Nonlinear pattern matching is carried out between the left-hand-side pattern (essentially a sentential form in the SQ2+ grammar) and the program. Variables preceded by a period are pattern variables that match program points. For matching to succeed, all occurrences of the same pattern variable in a pattern must match the same terms.

An environment *env* that maps pattern variables to program points is created as a side effect of matching. Matching the left-hand-side of 'nminfp' with the SQ2+ specification results in the following environment:

env(.w) = x
env(.s) = s
env(.k) = f [{ y : y in domain f | y subset s }]

Next, each pattern variable .x occurring in the enabling predicate is replaced by *env*(.x). The enabling predicate

```
  mono(s + f [ { y : y in domain f | y subset s } ], s) and type(s, [ set, .t] )
```

that results from this substitution must then be matched against the mono and type relations stored in the PDB. That is, the ground predicate mono(s + f [{ y : y in domain f | y subset s }], s) must be stored in the PDB (i.e., the pair [s + f [{ y : y in domain f | y subset s }], s] must be stored in relation mono, and relation type must store a pair [s, [set, τ]] for some s-expression τ that represents a type. Referring to the earlier display of relations mono and type, we see that the mono predicate is satisfied. Since variable s has type [set t5], the type predicate is also satisfied. As a side effect, the environment is extended and *env*(.t) = t5.

At this point the environment is extended further by executing the pattern expansion commands. Command readvar (.x) requires the user to supply an identifier from the terminal. In response we will supply the string a, after which *env*(.x) = a. Command genvar (.z) creates a new identifier automatically. In this case the system supplies x1, after which *env*(.z) = x1. Command subst (.k , .s , .x) is a general substitution mechanism that replaces all occurrences of *env*(.s) in *env*(.k) by *env*(.x). In this case, substitution will result in *env*(.k) = f [{ y : y in domain f | y subset a }]

The right-hand-side replacement code in the 'nminfp' transformation can now be instantiated by substituting the environment value for each occurrence of a pattern variable. The instantiated right-hand-side can now replace the SQ2+ subtree matched by the left-hand-side. Finally, the database action code indicates how to update the type relation in the PDB for the new terms that are introduced by tree replacement. The transcript below shows what happens.

```
replacement code is
      .x := x ;
      while exists .z in ( f [ { y in domain f | y subset s } ] - .x )
        loop
        .x with := .z ;
      end loop ;
      print ( .x ) ;
supply identifier for .x
>:a
The program just below results from dominated convergence
    program aclose ;
1     read ( x , f ) ;
2     a := x ;
3     while exists x1 in f [ { y in domain f | y subset a } ] - a loop
4       a with := x1 ;
      end loop ;
5     print ( a ) ;
    end ;
```

The SQ2+-to-C translator proceeds to the next phase of compilation, which applies finite differencing, named after the 16th century method attributed to Henry Briggs for computing tables of successive polynomial values by difference polynomials [29]. Using the 'help' feature of APTS, a finite difference rule called m2 is printed below.

```
>:help m2;
initialization code
      .e := { } ;
      for [ .x , .y ] in .f loop
        .e ( .x ) + := 1 ;
      end loop ;
pattern expansion code is
    genvar ( .y ) and genvar ( .x )
database action code is
      type ( .y , .t2 ) and type ( .x , .t1 ) and type ( .z , .t1 )
expression to be replaced
    # .f { .q }
enabling predicate
    type ( .f , [ set , [ tuple , [ .t1 , .t2 ] ] ] )
replacing expression
    .e ( .q )
pattern expansion code is
    readvar ( .e )
database action code is
      type ( .e , [ set , [ tuple , [ .t1 , int ] ] ] ) and ref_count (
        .e , .f ) and ismap ( .e )
modification
```

```
    .f { .q } with := .n ;
prederivative
    .e ( .q ) + := 1 ;
modification
    .f { .q } less := .n ;
prederivative
    .e ( .q ) - := 1 ;
modification
    .f := { } ;
prederivative
    .e := { } ;
```

This rule applies to cardinality expressions # b{e} (for arbitrary program expressions e) that match pattern # .f { .q }, that satisfy the enabling predicate, and that occur in a program loop L in which all modifications to b within L must match the modification patterns indicated within the rule. In this case the rule inserts code on entry to L that establishes the invariant c{e} = # b{e} for all value of e (where identifier c is supplied by the user in response to the 'readvar' command). Within L it inserts appropriate 'prederivative' code P just prior to each modification M to map variable b. Assuming that the invariant holds just prior to P, the execution of P reestablishes the invariant, which may be falsified by M, just after M is executed. Finally, all occurrences of expression # b{e} that are made redundant by the preceding steps can be replaced by occurrences of c{e}. When the cumulative expense of differentially maintaining reference counts c is less than the repeated calculation of the cardinality expression (which would certainly be the case if variable b becomes useless as a result of finite differencing), then this transformation is profitable. The semantics of finite differencing is described in [43,45].

In order to expose opportunities for finite differencing and also to regularize the program into a simplified form for which a limited number of finite differencing rules can have wide utility, the translator turns the program into a normal form. This is done by applying a group of conditional rewriting rules exhaustively bottom-up until there is no further change. The group transformation 'prepfd' is defined below in RSL by listing the individual rewriting rules in an arbitrary order. Several of these groups can be defined in the same module, all of whose individual rewriting rules must belong to a single rewriting module.

```
-- turn program into normal form prior to finite differencing
        prepfd : primd , rcc1 , rcd1 , rfu1 , prima , intelim , invelim
          , pelim , difelim , subselim , rcd2 , rfu2 , extosf , rcd3 , seqc ;
```

The algorithm used to implement group transformations is based on the incremental linear pattern matching preprocessor found in [16]. Regardless of the number of rewriting transformations belonging to a group, matching can proceed bottom-up so that the exact subset of individual rules whose left-hand-sides match a given program subtree can be computed in unit time, and presented in linear time in the subset size. Efficient incremental preprocessing of groups with respect to addition and deletion of individual rewriting rules is also supported.

The transcript just below prettyprints the result of the normalization process just mentioned. Next, the finite differencing transformation automatically detects invariants of the form x = e (where e is a program expression and x is a new variable uniquely associated with e) that should be maintained and exploited in order to avoid the costly repeated calculation of the truth set at the top of the while-loop at line 3. These invariants are detected by bottom-up analysis of the truth set expression, and their left-hand-side variables are all supplied by the user.

```
The normal form for finite differencing appears just below.
    program aclose ;
1     read ( x , f ) ;
```

```
2     a := x ;
3     while exists x1 in { x3 in f [ { y in domain f | # { x2 in y | x2
        notin a } = 0 } ] | x3 notin a } loop
4       a with := x1 ;
      end loop ;
5     print ( a ) ;
    end ;
invariant for { .x in .q | .x notin a }
<=
replacement code is  .e { .q }
supply identifier for .e
>:b
invariant for # b { .q }
<=
replacement code is .e ( .q )
supply identifier for .e
>:c
invariant for  { .x in domain f | c ( .x ) = 0 }
<=
replacement code is .e
supply identifier for .e
>:d
invariant for f [ d ]
<=
replacement code is .e
supply identifier for .e
>:e
invariant for  { .x in e | .x notin a }
<=
replacement code is .e
supply identifier for .e
>:g
invariant for { .x in domain f | .q in .x }
<=
replacement code is .e { .q }
supply identifier for .e
>:h
```

After the six invariants are detected, a stream processing transformation based on Goldberg and Paige [30] inserts code that aims to establish the invariants on entry to the while loop using a minimal number of loops. In this case stream processing generates the code appearing on lines 3 through just before line 24 below. A chain rule is used to perform the necessary bookkeeping operations needed to reestablish these invariants just before they are falsified by the modification to a at line 38. Consequently, the costly truth set expression at the top of the while loop is made redundant, and is replaced by the variable g.

Going back to the transcript, we can see the results of finite differencing.

```
    program aclose ;
1     read ( x , f ) ;
2     a := x ;
3     h := { } ;
4     for x8 in domain f loop
5       for x9 in x8 loop
6         h { x9 } with := x8 ;
```

```
          end loop ;
        end loop ;
7       c := { } ;
8       b := { } ;
9       for x4 in domain f loop
10        for x5 in x4 loop
11          if x5 notin a then
12            c ( x4 ) + := 1 ;
13            b { x4 } with := x5 ;
          end if ;
        end loop ;
      end loop ;
14    g := { } ;
15    e := { } ;
16    d := { } ;
17    for x13 in domain f loop
18      if c ( x13 ) = 0 then
19        for x16 in f { x13 } | x16 notin e loop
20          if x16 notin a then
21            g with := x16 ;
            end if ;
22          e with := x16 ;
          end loop ;
23        d with := x13 ;
        end if ;
      end loop ;
24    while exists x1 in g loop
25      for x6 in h { x1 } loop
26        if x6 in domain f then
27          if c ( x6 ) = 0 then
28            d less := x6 ;
            elseif c ( x6 ) = 0 + 1 then
29            for x16 in f { x6 } | x16 notin e loop
30              if x16 notin a then
31                g with := x16 ;
                end if ;
32              e with := x16 ;
              end loop ;
33            d with := x6 ;
            end if ;
          end if ;
34        c ( x6 ) - := 1 ;
35        b { x6 } less := x1 ;
        end loop ;
36      if x1 in e then
37        g less := x1 ;
        end if ;
38      a with := x1 ;
      end loop ;
39    print ( a ) ;
    end ;
```

The final phase of compilation turns the low level SETL2 code just above into C code that implements each primitive set operation in unit time. However, at the beginning of this phase an important cleanup transformation is performed.

As was suggested earlier, finite differencing often causes code in the untransformed program to become useless. It also causes some of the intermediate invariants themselves to become useless. The next step of the SQ2+-to-C translator performs a dead code analysis and elimination transformation. First, new rewriting modules are loaded from the APTS library. Then inference rules are used to compute control flow, data flow, types, and an abstraction of the program that allows live code to be calculated. Next, the inference rules are used to determine those program statements that contribute either directly or indirectly (via dataflow and control flow) to the print statement at line 39. This step is carried out in logic programming fashion by calculations involving only relations in the PDB and without reference to the program syntax tree. The results of this analysis are printed out in the transcript.

```
analyze for bases and strongly based sets and maps.
supply type for
b27
:int
dead code elimination
>:help dead;
23  d with := x13 ;  is dead
35  b { x6 } less := x1 ;  is dead
28  d less := x6 ;  is dead
13  b { x4 } with := x5 ;  is dead
16  d := { } ;  is dead
8   b := { } ;  is dead
33  d with := x6 ;  is dead
```

The preceding 8 statements are removed from the program, and control structures are subsequently simplified.

It is worth mentioning that the transcript above includes the only true manual step needed by the SQ2+-to-C translator. The type system for SQ2+ allows for input polymorphism, which for this case means that the types for the elements of the attribute set x, the elements of the range of f, and the elements belonging to each set-valued element of the domain of f can be determined at input time. Since C does not support input polymorphism, we have to supply the specific type of each type variable that is uninstantiated in the program. In this case, there is only one - b27 - which the system asks us to instantiate manually. Type 'int' is supplied, and compilation continues.

The final transformation, which makes use of concepts found in [53], but relies mainly on the type/subtype inference mechanism of [15], implements all set and map datatypes using array and list data structures. This transformation 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 in s) in real-time; i.e., by a unit time array or pointer retrieval. The final C code with 250 labeled statements runs in time linear in the size of the input and output. In prettyprinted form the C program would take over 8 pages to print, so only a fragment of the C translation of the while loop of the SETL2 program at line 24 is included here. It is annotated by hand to indicate the corresponding SETL2 code that is translated. A copy of the full transcript will be passed out separately at the conference.

The transcript ends with the compilation and execution of the C program. As was reported in [12], it runs 30 times faster than the SETL2 program running under Snyder's 'stlx' interpreter. Based on Snyder's and our experiments, this indicates performance comparable to good hand coded C.

```
The final C program appears just below.
```

```
...elided text...
153   while ( exists ( & x26 , g ) != NULL )  /* SETL2 while loop at line 24 */
154   {
...elided text...
              /* Translation of SETL2 lines 30 - 32 */
178                        if ( ! b27 . recs [ x16 ] . a . mem )
179                        {
180                          {                 /* g with:=x16, line 31
                               cell * x29 , * x30 ;
181                            if ( g == NULL )
182                            {
183                              g = alloccell ;
                               }
184                            x29 = alloccell ;
185                            x30 = g -> next ;
186                            x29 -> next = x30 ;
187                            x29 -> data . intv = x16 ;
188                            if ( x30 != NULL )
189                            {
190                              x30 -> prev = x29 ;
                               } else
191                            {
192                              g -> prev = x29 ;
                               }
193                            g -> next = x29 ;
                             }                   /*end g with:=x16
                           }
194                        {                     /*e with:=x16, line 32
                             int x41 , x42 ;
195                          x42 = x16 ;
196                          x41 = b27 . firsts . e ;
197                          if ( b27 . recs [ x42 ] . e . mem == false
                               )
198                          {
199                            b27 . recs [ x42 ] . e . mem = true ;
200                            b27 . recs [ x42 ] . e . prev = 0 ;
201                            b27 . recs [ x42 ] . e . next = x41 ;
202                            b27 . firsts . e = x42 ;
203                            if ( x41 != 0 )
204                            {
205                              b27 . recs [ x41 ] . e . prev = x42 ;
                               } else
206                            {
207                              b27 . lasts . e = x42 ;
                               }
                             }
                           }                       /*end e with:=x16
...elided text...
      }          /* end of SETL2 while loop from line 24 */
...elided text...
>:stop;
end transformational programming
himinhrjot 22 > cc c.c
himinhrjot 23 > a.out < aclose1000.out
```

```
{19 39 50 32 9 45 33 11 17 1 30 7 2 36 18 28 27 26 13 49 23 10 41 12 46 6 25 20 31 14 29 37 }
himinhrjot 24 > exit
exit
script done on Mon Jun 20 17:09:07 1994
```

3 Towards A Practical Software Development Technology

A serious shortcoming of the program development technology just described is the slow rate at which high level specifications are compiled into C. An ambitious plan to improve the translation rate of the SETL2-to-C compiler by a factor of 6000 is sketched below.

First, we plan to reduce the labor involved in writing and maintaining compilers within APTS. The most difficult and time-consuming part of writing compilers in RSL is in specifying how the PDB (Program Database) is updated to make it consistent with the RDB (Rule Database) after a program is changed by transformation. Currently, this is partly done automatically and partly achieved by specific instructions manually supplied within each transformation. We want to find a way for the APTS system to generate the instructions to update the PDB automatically for each transformation; that is, to mechanically generate an Incremental Rule Database (IRDB) from the batch RDB and a program transformation. This would allow the inference engine to recalculate the PDB efficiently by executing the IRDB each time a program transformation is applied. Some combination of differential techniques and partial evaluation is needed to solve this crucial but difficult research problem.

Second, we plan to speed up compiler execution in APTS. Currently, the SETL-to-C compiler translates 4.5 SETL lines into 22 C lines per minute on a SPARCstation 2. At this rate two days would be needed to translate APTS. We plan to improve this performance in several ways including application of partial evaluation techniques developed by Neil Jones's group [34].

Three sources of inefficiency will be examined to improve the performance of SETL2-to-C translation using APTS: the SETL2-to-C translator in the APTS meta-language, the APTS interpreters, and the SETL2 system in which APTS is implemented. Both Snyder's [59] and our experiments suggest that at least a factor of 30 improvement can be obtained in SETL2-to-C translation by bootstrapping the entire APTS system into C. We believe that another factor of at least 10 can be obtained by using partial evaluation to turn the APTS interpreters for logical inference and conditional rewriting into compilers.

The latter estimate is based on an experiment with a 14 page SETL2 module in APTS that performed useless code elimination specific to SETL programs parsed according to a particular SETL2 grammar. We replaced it with a 5 page implementation using APTS inference and rewriting rules. The specification in the APTS meta-language is much more readable and modifiable than the SETL2 program and is independent of the specific SETL2 grammar. A portion of it is even syntax-independent. Unfortunately, the new APTS-interpreted useless code elimination transformation runs about 10 times slower than the original 14 page SETL2 program. It will be interesting to see if partial evaluation can eliminate the interpretive overhead and recover the original performance.

We believe that the performance of the APTS inference and transformation engines can be improved by modifying the bottom-up pattern matching and semi-naive evaluation algorithms. A speedup by a factor of 2 is a coarse conservative estimate.

The APTS specification of the SETL2-to-C translator can also be greatly improved. There are two major sources of inefficiency. The first involves a rather straightforward modification to the way language translation is handled (which currently forces the transformation engine to execute many redundant bottom-up passes through the program being compiled), and should result in at least a 5-fold speedup (by eliminating these redundant passes).

The second more interesting source of inefficiency has to do with an unnecessary degree of global semantic analysis of low level programs. For example, in order to apply useless code elimination, we currently perform data flow analysis on a low level program compiled from a briefer higher level specification. Far more efficient would be to perform this analysis on the higher level specification and propagate the information using local analysis during rewriting. Unfortunately, as was mentioned earlier, in the current version of APTS it is difficult for the compiler writer to indicate explicitly the precise way in which dataflow is propagated for each transformation. This problem would be solved by automating the generation of these PDB updating instructions, as mentioned above. We don't yet have an accurate estimate for the speedup, but a factor of 2 would be conservative.

Combining all of the preceding improvements would allow SETL2-to-C translation to be performed at a rate of 450 SETL lines per second. At this rate APTS could be bootstrapped into high performance C in less than half a minute, which is over 10 times faster then it takes the current SETL2 system to compile APTS into intermediate form for interpretive execution.

4 Conclusion

The key open problem for the future is to build a new practical software development technology in which large-scale reliable and efficient software can be produced with (1) translation rates two orders of magnitude better (than the current 24 lines of C per minute), (2) a convincing five to fifteen-fold improvement in productivity over the best hand-crafted programming, (3) no sacrifice in performance over the best hand-crafted code, and (4) concurrent as well as sequential target architectures. A grand challenge would be to use this technology to translate all of APTS into high performance C.

References

[1] A. Aho and J. Ullman. Universality Of Data Retrieval Languages. In *Proc. ACM 6th POPL*, pages 110-117, Jan, 1979.

[2] F. Bancilhon. Naive Evaluation of Recursively defined Relations. *On Knowledge-Base Management Systems*, N, Brodie and H, Mylopoulos, editors, pages 165-178, 1986.

[3] F. Bancilhon, and R. Ramakrishnan. An amateur's introduction to recursive query-processing strategies. In *ACM SIGMOD*, pages 16-52, 1986.

[4] D. Barstow. Knowledge-Based Program Construction, Elsevier North-Holland, 1979.

[5] R. Bayer. Query evaluation and recursion in deductive database system. unpublished manuscript, 1985.

[6] C. Beeri and P. Bernstein. Computational Problems Related to the Design of Normal Form Relation Schemes. *ACM TODS*, 4:(1):30-59, 1979.

[7] R. Bird. An introduction to the theory of lists. in *Logic Programming and Calculi of Discrete Design*, M. Broy ed., pp. 5-42, NATO ASI Series F, Vol 3, 6, Springer-Verlag, 1987.

[8] B. Bloom and R. Paige. Computing Ready Simulations Efficiently. In *Proc. First North American Process Algebra Workshop*, A. Zwarico and S. Purushothaman, editors, Workshops in Computer Science Series, Springer-Verlag, pages 119 - 134, 1992.

[9] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. Centaur: the system. *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Pracical Software Dev. Environments*, pp. 14-24, 1989.

[10] J. Cai and R. Paige. Binding Performance at Language Design Time. In *Proc. ACM POPL*, pages 85 - 97, Jan, 1987.

[11] J. Cai and R. Paige. Look Ma, No Hashing, And No Arrays Neither. In *Proc. ACM POPL*, pages 143 - 154, Jan, 1991.

[12] J. Cai and R. Paige. Towards Increased Productivity of Algorithm Implementation. In *Proc. ACM SIGSOFT*, pages 71 - 78, Dec. 1993.

[13] J. Cai and R. Paige. Program Derivation by Fixed Point Computation. *Science of Computer Programming*, 11:197-261, 1988/89.

[14] J. Cai. A Language for Semantic Analysis. Technical Report 635, Courant Institute/New York University, 1993.

[15] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type Transformation and Data Structure Choice. In *Constructing Programs From Specifications*, B. Moeller editor, North-Holland, pages 126 - 124, 1991.

[16] J. Cai, R. Paige, and R. Tarjan. More Efficient Bottom-Up Multi-Pattern Matching In Trees. *Theoretical Computer Science*, 106:(1):21 - 60, Nov. 1992.

[17] J. Cai and R. Paige. Languages Polynomial in the Input Plus Output. In *Algebraic Methodology and Software Technology*, M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, Workshops in Computing Series, Springer-Verlag, pages 287 - 302, 1992.

[18] C.-H. Chang and R. Paige. From Regular Expressions to DFA's Using Compressed NFA's. In *Proc. CPM '92*, A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, Springer-Verlag, Lecture Notes in Computer Science number 644, pages 88 - 108, 1992.

[19] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific J. Math.*, 82:(1):43-57, 1979.

[20] H. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83-92, 1969.

[21] R. Dewar, A. Grand, S. C. Liu, J. Schwartz, and E. Schonberg. Program by Refinement as Exemplified by the SETL Representation Sublanguage. *TOPLAS*, 1:(1):27-49, July, 1979.

[22] E. W. Dijkstra. A Discipline of Programming, Prentice-Hall, 1976.

[23] W. Dowling and J. Gallier. Linear-Time Algorithms For Testing The Satisfiability of Propositional Horn Formulae. *J. Logic Programming*, 3:267-284, 1984.

[24] J. Earley. Toward an Understanding of Data Structures. *CACM*, 14:(10):617-627, Oct. 1971.

[25] J. Earley. High Level Operations in Automatic Programming. In *Proc. Symp. on Very High Level Langs.*, Sigplan Notices, 9:(4), Apr, 1974.

[26] J. Earley. High Level Iterators and a Method for Automatically Designing Data Structure Representation. *J of Computer Languages*, 1:(4):321-342, 1976.

[27] A. Fong and J. Ullman. Induction Variables in Very High Level Languages. In *Proc. Third ACM POPL*, pages 104-112, Jan, 1976.

[28] C. Forgy. RETE, a fast algorithm for the many patterns many objects Match problem. *Aritifical Intelligence*, 19:17-37, 1982.

[29] H. Goldstine. A History of Numerical Analysis, Springer-Verlag, 1977.

[30] A. Goldberg and R. Paige. Stream Processing *ACM Symposium on LISP and Functional Programming*, pp. 53-62, Aug, 1984.

[31] L. Henschen and S. Naqvi. On Compiling Queries in Recursive First-order Database. *CACM*, 31:47-85, 1984.

[32] J. Hindley. The principal type-scheme of an object in combinatory logic. *J Trans. AMS*, 146:29-60, 1969.

[33] J. Jeuring. The Derivation of On-Line Algorithms, with an Application to Finding Palindromes. *Algorithmica*, 11:(2):146-184, Feb., 1994.

[34] N. Jones, C. Gomard, and P. Sestoft. Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993.

[35] E. Kant. Efficiency in Program Synthesis, UMI Research Press, 1981.

[36] P. Klint, ed. The ASF+SDF Meta-environment User's Guide, version 26, 1993.

[37] D. Knuth. The Art of Computer Programming, Vol 1: Fundamental Algorithms, Addison-Wesley, 1973.

[38] J. Low and P. Rovner. Techniques for the automatic selection of data structures. In *Proc Third POPL*, pages 58 - 67, 1976.

[39] L. Meertens. Algorithmics - towards programming as a mathematical activity. *Proc. CWI Symp. on Mathematics and Computer Science*, J. W. de Bakker, M. Hazewinkel and J.K. Lenstra eds., CWI Monograph Vol 1, North-Holland, 1987.

[40] L. Meertens. Incremental Optimum-Fit Line Breaking *WG2.1 TR 720 REN-5*, Jan., 1994.

[41] J. Morris and M. Wells. The Specification of program flow in Madcap VI. In *Proc. ACM Natl. Conf.*, pages 755 - 762, 1972.

[42] S. Naqvi and L. Henschen. Synthesizing Least Fixed Point Queries into Non-recursive Iterative Programs. In *Proc. IJCAI 83*, pages 25-28, Aug, 1983.

[43] R. Paige. Programming With Invariants. *IEEE Software*, 3:(1):56-69, Jan, 1986.

[44] R. Paige. Real-time Simulation of a Set Machine on a RAM. In *Proc. ICCI '89*, Computing and Information, Vol II, R. Janicki and W. Koczkodaj, editors, pages 69-73, 1989.

[45] R. Paige. Symbolic Finite Differencing - Part I. *Proc. ESOP 90*, N. Jones ed., Springer-Verlag, Lecture Notes in Computer Science 432, 1990.

[46] R. Paige. APTS Reference Manual. unpublished manuscript, 1993.

[47] R. Paige. Efficient Translation of External Input in a Dynamically Typed Language. *Proc. IFIP Congress 94*, 1994.

[48] R. Paige and F. Henglein. Mechanical Translation of Set Theoretic Problem Specifications Into Efficient RAM Code - A Case Study. *Journal of Symbolic Computation*, 4:(2):207-232, Aug., 1987.

[49] R. Paige and J. Schwartz. Expression Continuity and the Formal Differentiation of Algorithms. In *Proc. 4th ACM POPL*, pages 58-71, Jan, 1977.

[50] Refine User's Guide Version 3.0, Reasoning Systems, May 1990.

[51] P. Rovner. Automatic Representation Selection for Associative Data Structures. Technical Report 10, Dept. of Computer Science, University of Rochester, 1977.

[52] E. Schonberg, J. Schwartz, and M. Sharir. An Automatic Technique for Selection of Data Representations in in SETL Programs. *ACM TOPLAS*, 3:(2):126-143, Apr, 1981.

[53] J. Schwartz. Automatic Data Structure Choice in a Language of Very High Level. *CACM*, 18:(12):722-728, Dec, 1975.

[54] J. Schwartz. Optimization of Very High Level Languages, Parts I, II. *J. of Computer Languages*, 1:(2,3):161-218, 1975.

[55] P. Selinger, et.al. Access path selection in a relational database management system. In *Proc. ACM SIGMOD*, pages 23-34, May, 1979.

[56] D. Smith. KIDS - A Knowledge-Based Software Development System. *Proc. Workshop on Automating Software Design, AAAI-88*, pp. 129-136, Sept. 1988.

[57] D. Smith. private communication, 1991.

[58] K. Snyder. The SETL2 Programming Language. Technical Report 490, Courant Institute/ New York University, 1990.

[59] K. Snyder private communication, 1993.

[60] R. Tarjan. A Class Of Algorithms Which Require Nonlinear Time To Maintain Disjoint Sets. *J. Comput. Sys. Sci.*, 18:110-127, 1979.

[61] J. Ullman. Principles of Database and Knowledge-Base Systems, I, Computer Science Press, 1988.

[62] M. Wells and J. Morris. The unified data structure capability in Madcap VI. *Intl. Journal of Comp. and Info. Sci.*, 1:(3):193 - 208, Sept. 1972.

[63] D. Willard. Abstract Predicate Retrieval Theory. Technical Report 83-3, State University of New York at Albany, Aug, 1983.

[64] D. Willard. Quasilinear Algorithms for Processing Relational Calculus Expressions. In *Proc. ACM PODS*, pages 243-257, 1990.