A SYSTEM WHICH AUTOMATICALLY IMPROVES PROGRAMS

J. Darlington and A.M. Buxatall

Department of Machine Intelligence
University of Edinburgh

Abstract

We give methods of mechanically converting programs that are easy to understand into more efficient ones, converting recursion equations using high level operations into lower level flowchart programs.

The main transformations involved are (i) recursion removal (ii) eliminating common sub-expressions and combining loops (iii) replacing procedure calls by their bodies (iv) introducing assignments which overwrite list cells no longer in use {compile-time garbage collection).

t Introduction

This paper ia an introduction to an automatic program improving system that we have implemented and are developing further.

A programmer is able to present his algorithms to the system in a clear and abstract language. The system converts them to efficient but probably not transparent versions.

For example, here are two versions of one program which reverses lists.

```
(i)  reverse(xl) = if null(xl) then nil
                   else concat(reverse(tl(xl)),
                               cons(hd(xl),nil))*

(ii) reverse(xl) = result:=nil;
                   while not null(xl)
                     do begin
                        temp:=tl(xl); tl(xl):=result;
                        result:=xl; xl:=temp
                        end
```

One is clear and abstract, the other more tortuous but efficient. Siven the first as a definition, a competent programmer should be able to produce the second. Our system can do this for him.

The system is built around the concept of abstract programming, and we hope to encourage a user to formulate his algorithms in abstract terms appropriate to hia problem domain and leave to the system the task of implementing them efficiently.

Our work was partly inspired by Minsky's homily on form versus content in computer science in which he recommended programming as a good application area for Artificial Intelligence work. It was also influeneed by Dijkstra^ ideas on structured programming, differing in that we start from a functional LISP-like language.

Our investigation took as its starting point a collection of prooedures written by Ambler and Burstall which aimed to provide transparent, but quite efficient operations on finite sets. We used this example to study the transformations which are needed to implement a collection of high level procedures as efficient code. To enable one to write programs about finite sets some operations on sets, for example, UNION, INTERSECTION, StBTBACT.NULLSET, NILSET, CONSSET, CHOOSE and MINUS are defined. These basic set operations must be implemented as structured definitions in terms of the array or list primitives available in the programming language. The user of the set system can define and run new functions, such as powerset, using these operations. However, when he looks at his definition. he notices that he could have produced a much more efficient program by writing a special procedure for powerset directly in terms of the array or list primitives. This is what our system attempts to do. A well written program in a LISP-like language expresses its structure as a hierarchy of functions. Our system eliminates higher level function calls to gain efficiency, flattening this hierarchy. Four distinct improvement processes seemed to be indicated.

1. Recursion removal,
2. Eliminating redundant computation, by merging common subexpressions and combining loops.
3. Replacing procedure calls by their bodies.
4. Causing the program to reuse data cells which are no longer needed, in order to reduce storage allocation and garbage collection at run time.

We have devised algorithms to perform these processes on programs and implemented these algorithms as POP-2 programs (section 7 gives an example of their use). These algorithms are applicable to a variety of domains and require only a collection of rules specifying potentially useful equivalences for a particular domain. We have tried our programs on our original example (programs about finite sets), and we can reproduce automatically a lot of the trioks incorporated in the original handwritten programs for the sets domain.

The overall system using processes 1 to 4 takes as input programs in a non-imperative language of recursive definitions and converts them, via intermediate stages, into an imperative language. This imperative language uses while statements as well as recursive definitions and permits assignment to components of data structures.

In order to produce efficient programs the system must use properties such as associativity or commutativity for the operations to be performed.

The principle techniques used are (i) matching, involving functional abstraction to detect the form of a recursive definition, (ii) matching to detect common subexpressions and compound operations involving the occurrence of several functional symbols, using algebraic equivalences, (iii) symbolic running to extract meanings from programs and check that a tentatively constructed sequence of instructions produces the required result. No elaborate theorem proving techniques are used, and the programs run quite faat, even though we have not xrked to code them efficiently.

2.. Recursion Removal

In this stage the system attempts to oonvert the set of recursion equations to a single iterative program in the same operations. During this process particular attention is paid to the semantics of the operations making up the recursion equations. This

* The operations we use are based on the POP-2 language, Burstall, Collins and Popplestone . The main features to note are that hd is the LISP car, t1 the. LISP cdr and concat joins two lists (the LISP append;.

process is generally applicable; the only input required is the set of recursion equations and a table of rules giving algebraic laws for the operations used.

There has been considerable theoretical investigation into how to translate recursive achemas into equivalent iterative achemas e.g. Strong , Garland and Luckham , although as far as we know only the BBN LISP compiler makes use of any of these ideas, and this only for simple recursions* These studies apply to translations of a schema which preserves its effect for all interpretations of the primitives. Our program uses translations which preserve the effect of a schema only for a class of interpretations in which the primitives obey a given set of algebraic laws; we follow Cooper who gave examples of such translations. We are only interested in translations which will improve efficiency. The results for translations of schemas to maintain equivalence under all interpretations seem to be too weak for practical purposes.

The translations that we achieve are of two types

(i) where the computation sequence of the resulting iterative program is a rearrangement of the computation sequence of the recursive program but contains the same number of steps. In these cases we save time and storage overheads associated with the stacking mechanisms (of. factorial function below)

Cii) where the tree grown by the recursive calls contains redundancies because the same values are calculated at separate nodes. Our system may produce an iterative program whose computation sequence is shorte* as well as having fewer overheads (cf. the Fibonacci function below).

Our system for recursion removal consists of four parts:

(i) A set of translation rules. Each rule has (a) a recursive schema over certain primitives (b) an iterative schema over these primitives and (c) a set of equations over the primitives {and possibly some extra restrictions) which, if satisfied, ensure that the iterative schema produces the same result as the recursive one.

(ii) A matching algorithm. This determines whether a set of equations is an instance of the recursive schema in one of the rules, and if so finds the substitution,

(iii) A simple equality-based theorem prover. This seeks to prove that a substitution is legitimate, i.e. that the equations associated with the rule are satisfied.

(iv) A control program. This first partitions the input equations into the smallest disjoint subsets such that if the equation for f involves a call of g and vice versa then they are in the same subset. Then, for each subset separately, it tries to find a translation rule which applies to that subset, using the matohing algorithm and the theorem prover, and effects the translation to iterative form if it finds one.

The matching algorithm is a second order one, in that it finds a substitution which takes primitive constants to expressions and also primitive functions to functions gr lambda expressions. It is described in Darlington . It was coded for lucidity, not speed. Consider for example this translation rule

Recursion schema

```
f(x) = / a -> b
       1 no_t a -> h(d,f(e))
```

Iterative schema

```
f(x) = if a then result:=b
       else begin result:=d; x:=e;
       while not a do
         begin result:=h(result,d);
               x:=e;
         end;
         result:=h(result,b);
         end
```

Equations

$h(h(\alpha,\beta),\gamma) = h(\alpha,h(\beta,\gamma))$ (associativity)

Restriction: x does not occur free in h.

The factorial function defined by the recursion equation

$$f(x) = \begin{cases} x=0 \rightarrow 1 \\ x\neq 0 \rightarrow \text{mult}(x,\text{fact}(x-1)) \end{cases}$$

is an instance of this recursive schema, with substitution a=(x=0), b=1, d=x, e=(x-1), h=mult. This is legitimate since 'mult' satisfies the equation for h in the rule.
The translation is

```
fact(x) =
   if x=0 then result:=1;
          else begin
               result:=x; x:=x-1;
               while x≠0 do
                 begin result:=mult(result,x);
                       x:=x-1;
                 end;
               result:=mult(result,1);
               end
```

This translation took 3.5 seconds on an ICL 4130 where the time for a CONS is 400 microseconds, and for RD and TL 50 microseconds. The same system took approximately 0.6 seconds to calculate factorial(30) using the recursive definition and 0.06 seconds using the iterative.
The reverse function defined by the recursion equation

$$\text{reverse}(x) = \begin{cases} \text{null}(x) \rightarrow \text{nil} \\ \underline{\text{not}}\ \text{null}(x) \rightarrow \text{concat}(\text{reverse}(\text{tl}(x)), \text{cons}(\text{hd}(x),\text{nil})) \end{cases}$$

is also an instance of this recursive schema, with substitution a=null(x), b=nil, d=cons(hd(x),nil), e=tl(x), h=λx1 x2; concat(x2,x1). This is a legitimate substitution. Notice the abstracted function that matches with h. The translation is

```
reverse(x) =
   if null(x) then result:=nil;
              else begin
                   result:=cons(hd(x),nil); x:=tl(x);
                   while not null(x) do
                     begin result:=concat(
                                cons(hd(x),nil),result);
                           x:=tl(x);
                     end;
                   result:=concat(nil,result)
                   end
```

This translation took 52.7 seconds. To reverse a list of length 40 took approximately 1.8 seconds using the recursive definition and .06 seconds using the iterative.
Another translation rule:-

Recursive schema

$$f(x) = \begin{cases} a \rightarrow b \\ \underline{\text{not}}\ a \rightarrow h(f(d1(x)),f(d2(x))) \end{cases}$$

Iterative schema

```
f(x) =
    y1:=b; y2:=b; result:=b;
        while not a do
            begin result:=h(y1,y2); y1:=y2
                  y2:=result; x:=d1(x)
            end
```

Equation

$d2(d1(x))=d1(d1(x)),\ h(\alpha,h(\beta,\gamma))=h(\beta,h(\alpha,\gamma))$

Restriction: x does not occur free in h or b.

This applies to the Fibonacci function

$$Fib(n) = \begin{cases} n=0 \vee n=1 \rightarrow 1 \\ \underline{not}(n=0 \vee n=1) \rightarrow Fib(n-1)+Fib(n-2) \end{cases}$$

giving the translation

```
y1:=1; y2:=1; result:=1
while not(n=0 ∨ n=1) do
  begin result:=y1+y2; y1:=y2; y2:=result; n:=n-1
  end
```

This translation took 48.25 seconds. To calculate Fib(20) took 12.8 seconds using the recursive definition and .125 using the iterative definition.

This section of our system shows the influence of the MIT philosophy 'of embedding knowledge in programs'. New knowledge, in the form of translation rules can easily be added to the system. We currently have six schemes, mostly with several translations.

3. Eliminating Redundant Computation

3.1. Compound operations

After removing as much recursion as possible the flattening is continued by removing procedure calls. This proceeds top down, a layer at a time, rewriting each program into another one. However, we do not proceed by the normal method of first replacing the calls of each procedure by its definition (in-line code introduction) and then optimising. Before replacing procedure calls we manipulate the program to eliminate as much redundant computation as possible and make the program more amenable to efficient implementation (see section 4). This we do by examining the semantic content of the given program. At present we are unable to do this for whole programs and we modify them in portions consisting of sequences of tests and assignments within a loop. Consider the following sequence of assignments

```
S:=union(intersection(T,U),V);
W:=union(intersection(T,U),W);
```

Clearly we can eliminate repeated computation of intersection(T,U), rewriting the sequence as

```
X:=intersection(T,U);
S:=union(X,V);
W:=union(X,W);
```

Suppose now that intersection and union are to be implemented in terms of list processing. The procedure body for union(X,Y) might be

```
result:=Y;
while not null(X) do begin
                    if not member(hd(X),Y)
                     then
                      result:=cons(hd(X),result);
                    X:=tl(X)
                   end
```

By replacing each call of union by this code we would obtain a program which has two loops on X. But this is unnecessary; the two can be combined into a single loop on X which builds up two result lists simultaneously. We could let the system replace each call of union separately and then try to combine the loops, but this would be quite difficult. Instead we let the system do as much manipulation as possible at the higher level, before replacing procedure calls by their bodies. Thus it would first synthesise a compound operation, which we will call doubleunion, with definition

```
doubleunion(X,Y,Z) = <union(X,Y),union(X,Z)>
```

It would then rewrite the assignment as

```
X:=intersection(T,U);
<S,W>:=doubleunion(X,V,W);
```

The system is able to produce a body of code for doubleunion(X,Y,Z), viz.

```
result1:=Y; result2:=Z;
while not null(X) do
  begin if not member(hd(X),Y)
          then result1:=cons(hd(X),result1);
        if not member(hd(X),Z)
          then result2:=cons(hd(X),result2);
        X:=tl(X)
  end;
```

and this body will be used to replace the call of doubleunion in the next stage (see section 4).

Notice that if we had started instead with the program

```
S:=union(intersection(U,T),V);
W:=union(W,intersection(T,U));
```

we could have made the same economies, given the fact that intersection and union are both commutative.

3.2. Origin of compound operations

To be able to spot opportunities for forming compound operations and produce the appropriate code bodies, the system preprocesses the definitions to produce

(i) A list showing which combinations of higher level operations can be formed into useful compound operations. This is used during the manipulation of the higher level program.

(ii) A schematic table that enables any synthesised compound operation to be expanded into code which is already in an optimised form. This table is used in the next stage to replace procedure calls by their bodies (see section 4).

This list and table are then used in the transformations of all programs written using these operations. To produce them the program uses a rewrite rule showing how certain combinations of loops can be condensed.

Suppose for example that it is given these definitions:

```
union(X,Y) <= result:=Y;
              while not null(X) do
                begin if not member(hd(X),Y) then
                  result:=cons(hd(X),result);
                  X:=tl(X)
                end
subtract(X,Y) <= result:=nil;
                 while not null(X) do begin
                   if not member(hd(X),Y) then
                     result:=cons(hd(X),result)
                     X:=tl(X)
                                        end
```

```
intersection(X,Y) <= result:=nil;
                     while not null(X) do
                     begin
                      if member(hd(X),Y) then
                        result:=cons(hd(X),result);
                        X:=tl(X)
                     end
```

All these definitions can be condensed using a rewrite rule

```
f1(x,y) <= y:=a(x);
           while p(x) do
             begin y:=g(x,y); x:=f(x)
             end;
```

and

```
f2(x,y) <= z:=b(x);
           while p(x) do
             begin z:=h(x,z); x:=f(x);
             end
```

rewrites to

```
f12(x,y,z) <= y:=a(x); z:=b(x);
              while p(x) do
                begin z:=h(x,z); y:=g(x,y);
                      x:=f(x)
                end
```

so that f12(x,y,z) = <f1(x,y),f2(x,z)>

Here x,y,z may each represent several variables, not just one, and a,b,f,g,h are arbitrary functions.

The system uses this to preprocess the definitions and produces

(i) a list ((union,subtract,intersection)) showing that all these operations can be combined into compound operations, provided that when they occur in programs the variables they iterate on (in this case the first variable) have the same value,

and (ii) a table from which code can be produced to realise any synthesised compound operator of this type, namely

```
<body>::=<part 1>;
         while not null(var1) do
                        begin
                          <part 2>;
                          var1:=tl(var1)
                        end

<part 1> for union::=        result1:=var2
         for subtract::=     result1:=nil
         for intersection::=result1:=nil
<part 2> for union::=        if not member(hd(var1),var2)
                             then
                              result1:=
                               cons(hd(var1),result1)
         for subtract::=     if not member(hd(var1),var2)
                             then
                              result1:=
                               cons(hd(var1),result1)
         for intersection::=if member(hd(var1),var2)
                             then
                              result1:=
                               cons(hd(var1),result1)
```

3.3. Program manipulation

If we work on assignment sequences directly it is not easy to recognise common subexpressions and to detect opportunities for introducing compound operations. So the system executes the given *sequence of assignments* symbolically and finds what state transformation it produces, that is, it computes the final value for each identifier as an expression in terms of the initial values. In the above example, given initial symbolic values S_0,T_0,U_0, V_0 and W_0, for variables S,T,U,V and W the final state vector is

<S=union(intersection(T_0,U_0),V_0),T=T_0,U=U_0,V=V_0,
W=union(intersection(T_0,U_0),W_0)>

The *search for common subexpressions* and compound operations is performed on the 5-tuple of expressions constituting the values of S,T,U,V and W. This state 5-tuple is rewritten, introducing subsidiary variables in where clauses (in fact the internal representation uses pointers to list structures), thus

```
<S,T,U,V,W>=<y,T0,U0,V0,z>
            where <y,z>=doubleunion(x,V0,W0)
            where x=intersection(T0,U0)
```

This new 5-tuple of expressions must now be converted back to assignments.

To summarise, there are three steps (1) convert from assignments to state transformation (2) rewrite the state transformation, and (3) convert back to assignments. We will make a few more remarks about each.

Step 1. From the sequence of commands the transformation induced is extracted by running the sequence *on symbolic data. From this state transformation* a set of equivalent transformations are produced by applying all the appropriate algebraic equations (commutativity etc.), to a given degree of effort.

Step 2. In each of these state transformations the program seeks common subexpressions and compound operations.

The improvements made in this step can be represented as rewriting rules on n-tuples of expressions. They introduce 'where clauses' using new identifiers, to show the sharing. Let E, F and G be meta-variables denoting expressions and v be a meta-variable denoting a (suitably distinct) identifier. We use $E(E_1,\ldots,E_n)$ to denote an expression with subexpressions $E_1,\ldots,E_n$.

The rewriting rule for extracting common subexpressions is:-

Rewrite '$\langle E_1(E),\ldots,E_n(E)\rangle$' as '$\langle E_1(v),\ldots,E_n(v)\rangle$ where $v=E$'

The rewriting rule for introducing a compound operation f, defined by $f(x_1,\ldots,x_k) = \langle F_1(x_1,\ldots,x_k),\ldots,F_m(x_1,\ldots,x_k)\rangle$ is:-

Rewrite '$\langle E_1(F_1(G_1,\ldots,G_k),\ldots,F_m(G_1,\ldots,G_k)),\ldots,$
$E_n(F_1(G_1,\ldots,G_k),\ldots,F_m(G_1,\ldots,G_k))\rangle$'
as '$\langle E_1(v_1,\ldots,v_k),\ldots,E_n(v_1,\ldots,v_k)\rangle$
where $\langle v_1,\ldots,v_k\rangle = f(G_1,\ldots,G_k)$'

A quite elaborate matching process is used to determine where these rules are applicable, (Darlington[8] gives details). For each of the variant state transformations produced by using the equations, the system finds all subexpressions occurring in it and matches the F_i against them for each compound f. At this stage only the most promising one, as judged by a rough efficiency estimate is retained. This is rewritten in terms of the compound operations.

Step 3. The system converts the rewritten state transformation to a sequence of assignments. It calculates a set of differences between the final expressions for the identifiers and their initial values, and then performs a G.P.S.-like search (Ernst and Newell[9], Simon[10]). For example, trying to achieve the transformation <X=consset(X,Y),Y=X,Z=Y> the system notices three differences between initial and final values. It tries to remove these in all possible sequences until it succeeds. It would first try 'X:=consset(X,Y);', but Y still differs

from its final value and X_0 is lost. So it tries successively 'Y:=X; Z:=X;', which loses Y too soon, and 'Z:=X; X:=conset(X,Y); Y:=Z'. If necessary it introduces extra identifiers.

It seems crucial that the manipulation is done on the higher level program, before replacing procedure calls, for two reasons.

(a) The higher level programs are often much simpler and easier to understand, since usually a single operation expands into a body of code.

(b) We are able to make full use of the algebraic laws appropriate to this higher level. For example, once calls to set operations have been replaced by their list processing bodies many possibilities for rearrangement and optimisation will have been lost. Thus union(X,Y)=union(Y,X) but if union is represented by list concatenation, $\otimes$, the results $X \otimes Y$ and $Y \otimes X$ are two different lists although they represent the same set. So no list optimiser would commute the arguments however great the gains in efficiency.

4. Replacing Procedure Calls by their Bodies

Here the system replaces calls of basic operations by their procedure bodies and replaces compound operations by code bodies produced using the compound operator table described earlier.

In our example above the program

```
X:=intersection(T,U);
<S,W>:= doubleunion(X,V,W);
```

expands into a list program

```
result:=nil;
while not null(T)
  do begin if member(hd(T),U)
             then result:=cons(hd(T),result);
          T:=tl(T);
     end;
X:=result; result1:=V; result2:=W;
while not null(X)
  do begin if not member(hd(X),V)
             then result1:=cons(hd(X),result1);
           if not member(hd(X),W)
             then result2:=cons(hd(X),result2);
          X:=tl(X);
     end;
S:=result1; W:=result2;
```

At present the system has definitions to provide a choice of two representations for sets: lists or bit strings. These involve different sets of compound operations. The user chooses one of the representations.

5. Reusing Discarded List Cells

In any program that contains lists as data structures our system is able to take the improvement further. It attempts to transform constructive list programs, which allow assignment only to identifiers, into destructive ones, which allow assignment to parts of structures, for example 'hd(Y):=X'. In LISP terms it eliminates CONS in favour of REPLACA and REPLACD. Programs written 'destructively' are efficient in store usage, but they are recognised as being difficult to understand or debug since side effects of destructive assignments are often far from obvious.

The process involved in this transformation follows the same pattern as described in section 3. It is regarded as a rewriting between two levels of language, a 'constructive' list language and a 'destructive' list language. Again the system operates by finding the state transformation and reinterpreting it as efficiently as possible.

The optimisation attempted is to avoid store usage by reusing any list cells that will have been discarded. This process can be thought of as a compile time garbage collection.

The system takes a sequence of assignments and works out the symbolic state transformation they induce. Usually information is available to make the starting state more detailed than before. Thus before X:=tl(X) we know that X should have as value a list of at least one element. To examine store usage we explicitly name list cells and define a starting state in terms of identifiers, I, Atoms, A, list cells, C, and variables, V. Thus a state is a triple of finite functions,

$$\begin{aligned} \text{val}&: I \to A \cup C \cup V \\ \text{hd}&: C \to A \cup C \cup V \\ \text{tl}&: C \to A \cup C \cup V \end{aligned}$$

The variables, V, stand for unspecified list cells or atoms (compare the variables X_0, Y_0, Z_0 used previously).

For example before the program

```
Y:=cons(hd(X),Y);
X:=tl(X)
```

the system creates a symbolic state

$$\text{val}(X)=c_1, \text{val}(Y)=c_3, \text{hd}(c_1)=a_1, \text{tl}(c_1)=c_2, \text{hd}(c_3)=v_1, \text{tl}(c_3)=v_2, \text{hd}(c_2)=v_3, \text{tl}(c_2)=v_4$$

and after executing the above program on this symbolic state it would have a final state

$$\text{val}(X)=c_2, \text{val}(Y)=c_4, \text{hd}(c_1)=a_1, \text{tl}(c_1)=c_2, \text{hd}(c_4)=a_1, \text{tl}(c_4)=c_3, \text{hd}(c_3)=v_1, \text{tl}(c_3)=v_2, \text{hd}(c_2)=v_3, \text{tl}(c_2)=v_4$$

The system now performs a 'symbolic garbage collection' on this state description to see if any cells have been discarded, and attempts to reuse these to avoid the introduction of new cells. In the example it finds that it can use the discarded cell c_1 instead of introducing the new cell c_4, and it rewrites the final state thus

$$\text{val}(X)=c_2, \text{val}(Y)=c_1, \text{hd}(c_1)=a_1, \text{tl}(c_1)=c_3, \text{hd}(c_2)=v_3, \text{tl}(c_2)=v_4, \text{hd}(c_3)=v_1, \text{tl}(c_3)=v_2$$

The system now tries various sequences of assignments (exhaustively, but with a G.P.S.-like difference-operator table) to achieve a sequence of instructions to implement the new transformation but avoiding cons's. In our example the sequence produced is

```
NEWVAR1:=tl(X); tl(X):=Y; Y:=X; X:=NEWVAR1
```

Thus the list reverse program

```
result:=nil;
while not null(X) do
  begin result:=cons(hd(X),result); X:=tl(X);
  end;
```

is automatically converted to a reverse program which uses no new store:-

```
result:=nil;
while not null(X) do
  begin NEWVAR1:=tl(X);
    tl(X):=result; result:=X; X:=NEWVAR1;
  end;
```

This translation took 2.125 seconds.

6. Use of the System

One can experiment with the system using simple interactive facilities; a sample dialogue is given below. The user can invoke any of the three processes: recursion removal (Stage 1 above),

eliminating redundant computation with code introduction (Stages 2 and 3), and destructive list proceasing (Stage 4).

The system can be applied to a new domain of diacourse just by giving it new rules and definitions. The table of recursive schemas and iterative equivalents and the table of iterative compounds are independent of domain; they may of course be extended To apply the system to the finite sets domain, so that it translates recursion equations in basic set operations (conseet, choose, nullget etc.) to destructive list processing programs or to bit string processing programs, we have provided the following.

(i) Equations about the basic set operations, conmutativity etc. (used in Stages 1 and 2)

(ii) Procedure bodies to implement the basic set operations by list operations (used in Stage ?)

(ii*) as (ii) for bit strings instead of lists.

For the finite set application the system has a fetf extra tricks built into it as program. We would like to express these by rules or tables.

The idea of symbolic execution of parts of a program has proved fruitful and we would like to explore further applications for it. It has also been used at Edinburgh by Burstall and Topor and Boyer and Moore . Our expression optimising techniques seem to go beyond those used in current 14 compilers (see for example Sheridan , Roh1 and Lin ; Hopgood , page 74 gives further references).

7. Example of Use

```
: STARTSYSTEM();              (user commences dialogue)
  START OF OPTIMISATION.INPUT PROGRAM
  TO BE OPTIMISED             (systems response)
: union(x,y)=                 (user inputs program)
  nullset(x)->y,
  not nullset(x) ->consset(choose(x),
                   union(minus(choose(x),x),y))
  WHICH OPTIMISATION PROCESS WOULD YOU LIKE PERFORMED
: REC           (user requests recursion removal stage)
  RESULTING PROGRAM IS
  result:=y;                  (system outputs result)
  while not nullset(x)  do
  begin result:=consset(choose(x),result);
  xsave:=minus(choose(x),x);
  y:=y;
  x:=xsave;
  end
  WHICH OPTIMISATION PROCESS WOULD YOU LIKE PERFORMED
     (user wishes to continue optimising this program)
: CODEINTRO
  DO YOU WISH TO IMPLEMENT IN LISTS OR BIT STRINGS
: BIT STRINGS
  RESULTING PROGRAM IS
  l:=length(x); n:=1; result:=y;
  while n≤l do
  begin if x[n]=1 then
                  begin result[n]:=1; x[n]:=0;
                        xsave:=x;
                  end;
      n:=n-1
  end
  WHICH OPTIMISATION PROCESS WOULD YOU LIKE PERFORMED
  FINISHED     (user has finished with this program)
  DO YOU WISH TO PERFORM ANOTHER OPTIMISATION
  YES          (user continues with another program)
  INPUT PROGRAM TO BE OPTIMISED
           doneshuffle:=true;  (user inputs sorting
           while doneshuffle do              program)
            begin doneshuffle:=false;
                  result:=nil; last:=hd(xl);
              while not null(tl(xl))  do
                  begin if last≤hd(tl(xl))
                        then
                         begin result:=cons(last,result);
                               xl:=tl(xl); last:=hd(xl);
                         end
                        else
                         begin xl:=tl(xl);
                               result:=
                                     cons(hd(xl),result);
                               doneshuffle:=true
                         end
                  end;
                 result:=cons(last,result);
                 xl:=reverse(result)
            end;
           result:=xl;
  WHICH OPTIMISATION PROCESS WOULD YOU LIKE PERFORMED
  MAKEDEST          (stage 4 store overwriting)
  RESULTING PROGRAM IS
       doneshuffle:=true;
       while doneshuffle do
        begin doneshuffle:=false;
              result:=nil; last:=hd(xl);
          while not null(tl(xl))  do
            begin if last≤hd(tl(xl))
              then begin
                     hd(xl):=last; last:=hd(tl(xl));
                     newvar1:=tl(xl); tl(xl):=result;
                     result:=xl; xl:=newvar1
                   end
              else begin
                     newvar1:=tl(xl);
                     hd(xl):=hd(tl(xl));
                     tl(xl):=result; result:=xl;
                     xl:=newvar1; doneshuffle:=true;
                   end
            end
          result:=cons(last,result);
          xl:=reverse(result)
        end;
       result:=xl;
```


Acknowledgements

We would like to thank our colleagues in the School of Artificial Intelligence for many helpful conversations, Brian Randell for valuable criticism, Eleanor Kerse for speedy and accurate typing, and the Science Research Council for financial support.

References

(1) Burstall, R.M., Collins, J.S. and Popplestone, R.J. Programming in POP-2. Edinburgh: Edinburgh University Press. (1971).

(2) Minsky, M. Form and content in computer science. 1970 A.C.M. Turing Lecture. Ass. comput. Mach., Vol. 17, No. 2, 197-215. (1970).

(3) Dijkstra, E.W. Notes on structured programming. TH Report 70-WSK-03, 2nd edition. Technische Hogeschool, Eindhoven, The Netherlands. (1970).

(4) Ambler, A.P. and Burstall, R.M. LIB POLYSETS. POP-2 Program Library Specification. Department of Machine Intelligence and Perception, University of Edinburgh. (1971).

(5) Strong, H.R. Jr. Translating recursion equations into flow charts. Proc. 2nd Annual A.C.M. Symposium on Theory of Computing, A.C.M., New York, pp. 184-197. (1970).

(6) Garland, S.J. and Luckham, D.C. Program schemes, recursion schemes and formal languages. UCLA-ENG-7154. School of Engineering and Applied Science, University of California, Los Angeles. (1971).

(7) Cooper, D.C. The equivalence of certain computations. Computer Journal. Vol. 9. No. 1. May, 45-52. (1966).

(a) Darlington, J. A semantic approach to automatic program improvement* Ph.D. thesis. Department of Machine Intelligence, University of Edinburgh. (1972).

(9) Ernst, G.V. and Newell, A. Generality and SPS. Carnegie Institute of Technology, Pittsburgh, Pennsylvania. (1967).

(10) Simon, H.A. The heuristic compiler, in Representation and Meaning (eds. H.A. Simon and L. Siklossy) Prentioe Hall, New Jersey. (1972).

(11) Burstall, R.M. and Topor, R.W. Private communication. (1972)♦

(12) Boyer, R.S. and Moore, J.S. Proving theorems about LISP functions. Memo 60, Department of Computational Logic, University of Edinburgh. (1973).

(15) Sheridan, P.B. The arffiimetic translator-compiler of the IBM Fortran automatic coding system. C.A.C.M.. Vol. 2. No. 2. 9. (1959).

(14) Rohl, J.S. and Lin, J.A. A note or. compiling arithmetic expressions. Computer Journal. Vol. 15. No. ±, February, 15-14. (1972).

(15) Hopgood, F.R.A, Compiling techniques. Computer Mnnographe. Macdonald: London and American Elsevier Inc: New York. (1969).