Information and Computer Science

THE IRVINE PROGRAM TRANSFORMATION CATALOGUE

A Stock of Ideas for Improving Programs
Using Source-to-Source Transformations

TR#161

T. A. Standish, D. C. Harriman, D. F. Kibler, and J. M. Neighbors

UNIVERSITY OF CALIFORNIA
IRVINE

Version 0.2

Caution: This preliminary version of the Catalogue has not been debugged thoroughly.

THE IRVINE PROGRAM TRANSFORMATION CATALOGUE

A Stock of Ideas for Improving Programs Using Source-to-Source Transformations

TR#161

T.A.Standish, D.C.Harriman, D.F.Kibler, and J.M.Neighbors
Department of Information and Computer Science
University of California at Irvine
Irvine, California 92717

Research Supported by
THE NATIONAL SCIENCE FOUNDATION
under Grant DCR75-13875

January 7, 1976

TABLE OF CONTENTS AND

TRANSFORMATIONS BY NUMBER AND TITLE

Introduction

The Irvine Program Transformation Catalogue is basically a source book of ideas for improving programs. By casual browsing, the reader may discover a number of useful techniques for transforming programs into better ones. When the moment comes to apply a particular technique, the Catalogue can be referenced for details.

The ideas are presented in the form of source-to-source transformations. These transformations show how to change source language programs in a given form into improved source language programs. Many of the transformations are given as rules of exchange of the form P = Q. This is intended to follow conventional usage in mathematics. For example, in algebra, we might find the distributive law given in a form such as $x\ (\ y + z\ ) = xy + xz$, in logic, we might find a simplification law given as $b \wedge (b \vee c) = b$, and in a table of integrals, we might find a rule such as $\int u\ dv = uv - \int v\ du$. Each of these exchange laws can be used to replace a given expression with an equivalent one.

In a similar vein, we can specify exchange rules for expressions used in computer programs. For instance, we

might give a distributive rule involving conditional expressions, such as

$$x + (\underline{if}\ b\ \underline{then}\ y\ \underline{else}\ z) = (\underline{if}\ b\ \underline{then}\ x+y\ \underline{else}\ x+z)$$

To specify that a given exchange rule is intended for use in a preferred direction, we write P => Q. For example, writing $b \wedge (b \vee c) \Rightarrow b$ signifies our preference for replacing the more complex expression $b \wedge (b \vee c)$ with the simpler expression b, where possible.

Under some circumstances, we might give separate names to the same exchange applied in different directions. For instance, $xy + xz \Rightarrow x(y+z)$ might be called "factoring out the mononomial x", whereas $x(y+z) \Rightarrow xy + xz$ might be called "multiplying through by x".

The use of the double shafted arrow (=>) in place of the equal sign (=) can also help relieve possible ambiguities in the transformation of expressions containing the equal sign itself. For example, writing

$$b = c = d \quad \Rightarrow \quad (b = c)\ and\ (c = d)$$

is less ambiguous than writing

$$b = c = d \quad = \quad (b = c)\ and\ (c = d) \quad .$$

For this reason, we prefer to indicate bi-directional exchange rules using a double-headed arrow (<=>) as in

$$x(y + z) <=> xy + xz \quad .$$

Some transformations can be applied only under special conditions. For example, provided $a \neq 0$ we can write $a^0 => 1$. Sometimes, we state such conditions formally as part of a transformation rule, as in writing

provided $a \neq 0$:

$$a^0 => 1 \quad .$$

In this case the condition ($a \neq 0$) is called an enabling condition.

Not all source-to-source transformations can be conveniently stated as exchange rules. Sometimes it is preferable to give a procedure for transforming a program into another. This condition usually applies when we cannot easily devise a simple syntactic "pattern" to match situations in which a transformation applies.

For example, one source-to-source transformation of interest consists of removing "useless" assignment statements, which assign values to variables which are

never subsequently used in a program. This process is better expressed as a procedure with several steps than as an exchange rule on various forms of programs.

Such procedures are program manipulating programs. In this situation, programs constitute dual entities that do the manipulating and are themselves manipulated. This requires introducing appropriate notational conventions to distinguish between agents and objects of actions. The notions are introduced in the Catalogue as required, and a full explanation is given in an appendix.

Sometimes an idea behind a general principle of source-to-source transformations is given by example. Once the example is properly understood, the general principle can be applied in a large variety of circumstances.

For instance, (following [2]) consider searching a table T to see if it contains an item X. Suppose the table has entries numbered from 1 to N, signified by $T[1]$, $T[2]$, ... , $T[N]$. A procedure to determine whether T contains X is as follows:

```
1   procedure Search(T,X); local i;
2       begin
3       i←N;
4       while i>0 do
5           if T[i]=X then Return(true) else i←i-1;
6       Return(false)
7       end procedure;
```

The "inner loop" of this procedure tests both whether $i>0$ and whether $T[i]=X$ for each distinct value of i tried.

Suppose we extend the table T to contain a new 0-th entry $T[0]$, and we initialize this entry to contain X. Whether or not X was in the original table T, we can now be confident that X is in the extended table. The way the above procedure determines that X is not in T is is by determining that the table has been exhaustively searched without having found any entry $T[i]$ containing X. However, using the extended version of T enables us to eliminate the test for exhaustion, as seen in the following procedure:

```
1   procedure Search(T,X); local i;
2       begin T[0]←X; i←N;
3       while T[i]≠X do i←i-1;
4       if i=0 then Return(false) else Return(true)
5       end procedure;
```

This procedure runs faster and is syntactically simpler than the previcus one because the test ($i>0$) has been eliminated from the inner loop.

This specific program improvement is an example of a general principle, which can be stated as follows. Suppose you are searching to find whether a search space S contains an element X by systematically enumerating elements of S in some prescribed order and comparing them to X. If you extend S to include one more element containing X, which is guaranteed to be enumerated last in the particular order of search used, then there is no need to test explicitly for exhaustion of the search space in your algorithm. Instead, you can generate elements in the extended version of S until finding X, and then you can conclude X was in the original unextended space S, if and only if it was not found in the extension.

Once this general principle has been understood it can be applied not only to tables, but to search spaces of many different shapes and organizations, such as binary trees, list structures, and indexed files. Such a general principle cannot be easily expressed as an exchange rule P => Q,

because the many forms of programs P to which it applies are not readily characterized as instances of a simple syntactic pattern P. Nor can we conveniently encode the general principle as a program manipulating procedure because the variety of programs and data structures to which it applies is too large to yield easily comprehensible code. But we can express such an idea by giving examples, and stating the general principle of which the examples are instances. Given the nature of our current knowledge of computer science, the latter is the best choice available to us for explaining such general program transformation ideas to people (as opposed to machines).

Thus, the transformations in the Catalogue are given in three forms: (1) pattern-directed exchange rules, (2) program manipulating programs, and (3) examples plus discussion.

Using the Catalogue to improve a given program may involve transformations at all three of these levels. For example, after producing the improved search program above, line 4 (at the bottom of page 5) reads:

4 *if i=0 then Return(false) else Return(true)* .

There are two pattern-directed exchange rules that can be applied to simplify this statement, namely:

factoring conditionals (see §3.2.c)

if b then F(c) else F(d) => F(if b then c else d)

and

simplifying conditionals (see §3.1.b)

if b then false else true => ∿b

Applying these transformations to line 4, yields the following sequence of transformations:

4 *if i=0 then Return(false) else Return(true)*

⇓

4 *Return (if i=0 then false else true)*

⇓

4 *Return(∿(i=0))*

This last result can be rewritten as *Return(i≠0)* by applying the synonym transformation *∿(x=y) <=> (x≠y)*.

Facility with the application of transformations in the Catalogue may be gained through practice.

The transformations in the Catalogue are intended to preserve program equivalence. In particular, this means we can apply certain transformations only under particular enabling conditions that guarantee equivalence. Two such enabling conditions are of such general applicability, however, that they deserve special mention in the introduction. These are called commutativity and freedom from side-effects.

Let F be a well-formed program fragment (i.e. a phrase in the grammar of the programming language at hand). Let $R(F)$ be the set of variables non-local to F that F either reads but never writes or reads before writing, and let $W(F)$ be the set of non-local variables that F writes (whether or not it also reads them). Here a local variable v in F is a variable used only within F and not elsewhere in the program.

Given two program fragments A and B, we wish to know when it is permissable to exchange their order of execution from $A;B$ to $B;A$ while preserving program equivalence. It is permissable for A and B to read the same read-only variables, but neither A nor B may write into variables the other reads before writing, nor may they write into common variables. In symbols, A and B are commutative provided that $W(A)\cap(W(B)\cup R(B))=\emptyset$ and $W(B)\cap(W(A)\cup R(A))=\emptyset$.

Whenever we are given a transformation that implicitly changes the order of execution of some of its constituent program fragments, the fragments whose execution order is changed must be commutative. This requirement is so pervasive that it is scarcely worth mentioning it explicitly every time it applies. For example, in the transformation:

$$a\ ;\ (\underline{if}\ b\ \underline{then}\ c\ \underline{else}\ d) \Rightarrow (\underline{if}\ b\ \underline{then}(a;c)\underline{else}(a;d))$$

the order of execution of a and b is implicitly changed, so a and b must be commutative. This would exclude transforming

$$x\leftarrow 3\ ;\ (\underline{if}\ x>w\ \underline{then}\ y\ \underline{else}\ z)$$

into

$$(\underline{if}\ x>w\ \underline{then}\ (x\leftarrow 3;y)\ \underline{else}\ (x\leftarrow 3;z))$$

for instance.

If the program fragment F does not write into any non-local variables (i.e. if $W(F)=\emptyset$), then we say that F is side-effect-free. Some transformations preserve program equivalence only if one or more of the constituent program fragments is side-effect-free. For example, the McCarthy conditional transformation:

$$a \wedge b \Rightarrow \underline{if}\ a\ \underline{then}\ b\ \underline{else}\ \underline{false}$$

requires b to be side-effect-free in order to preserve program equivalence, in general.

The Irvine Program Transformation Catalogue originates from a current NSF research project at U.C. Irvine on interactive program manipulation. We are trying to understand how to program a computer to look at a given program and to refine or improve it stepwise using source-to-source transformations.

When we speak of stepwise refinement of a program, we mean the following. Suppose P is a high-level program, written without certain commitments to underlying data representations. Suppose we apply transformations to P that "fill in details" mechanically by supplying data representations and generating associated lower-level program text so as to provide an implementation of P at a more concrete level. This concrete version of P is said to be a refinement.

Experience shows that the text of programs generated by mechanical refinement often requires improvement. Program improvement consists of transforming a program to achieve a better appearance or better performance properties, perhaps with the objective of meeting required operational constraints.

The Catalogue arose out of our attempts to describe and classify a spectrum of source-to-source transformation ideas we hope eventually to mechanize. We discovered,

coincidentally, that the Catalogue was a rich source of ideas for people to use to improve programs manually. It is also a beneficial pedagogical device to use to provide beginning students with a stock of ideas and techniques helpful for improving their programs.

While our basic research quest is aimed at mechanizing program refinements and improvements, one of its early by-products seems already to be potentially useful to students, educators, and, perhaps, practicing professionals alike. This may represent an instance where basic research spins off immediately usable results, and we intend to pursue further its potential benefits in these contexts.

1. Assignment Forms

1.1 Nested Assignment Introduction (=>) & Elimination (<=)

1.1.1 examples

1.1.1.a $x \leftarrow 0; y \leftarrow 0; z \leftarrow 0 \iff x \leftarrow y \leftarrow z \leftarrow 0$

1.1.1.b $x \leftarrow 0; y \leftarrow x; z \leftarrow y \iff x \leftarrow y \leftarrow z \leftarrow 0$

1.1.1.c $x \leftarrow y * z + 5;\ w \leftarrow x \iff w \leftarrow (x \leftarrow y * z + 5)$

1.1.1.d

$$x \leftarrow P(x);\ \underline{while}\ B(x)\ \underline{do}\ [f(x); x \leftarrow P(x)] \iff \underline{while}\ B(x \leftarrow P(x))\ \underline{do}\ [f(x)]$$

1.2 Transformations on Straight Line Sequences of Assignments

The transformations in this section (§1.2) deal with sequences of assignments of the form

$$x_1 \leftarrow E_1;\ \ x_2 \leftarrow E_2;\ \ . \ . \ . \ ;\ x_n \leftarrow E_n\ ;$$

where the x_i are variables and the E_i are expressions which are either variables or constants, or are formed by applying operators to subexpressions formed from operators, variables, and constants.

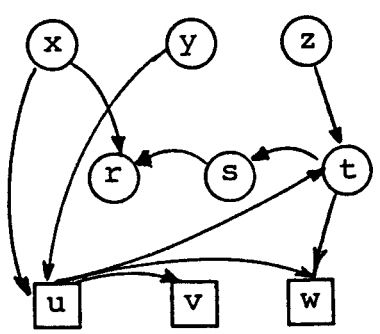
1.2.1 Useless Assignment Elimination

Repeatedly delete assignments of the form $x \leftarrow E$ where x is not used subsequently in the program and where E is side-effect-free.

1.2.1.a <u>example</u> Let x, y, z, be input variables and u, v, w be output variables. We assume that the input variables have initial values, and that only the values of the output variables are needed. (In particular, r is not an output variable.)

```
u←x+2*y;        u←x+2*y;        u←x+2*y;
t←3*u+z;        t←3*u+z;        t←3*u+z;
s←t-1;          s←t-1;          empty;
r←s+x;    =>    empty;    =>    empty;
v←u/6;          v←u/6;          v←u/6;
w←t+u;          w←t+u;          w←t+u;
```

1.2.1.b <u>algorithm</u> - Construct a directed graph with a node for each variable. Construct for each assignment an arc from node v_1 to v_2 whenever $v_2 \leftarrow E(\ldots v_1 \ldots)$ where $E(\ldots v_1 \ldots)$ is any expression containing an occurrence of v_1. Now put squares around output variables. For instance, example §5.2.1.a leads to the following graph:

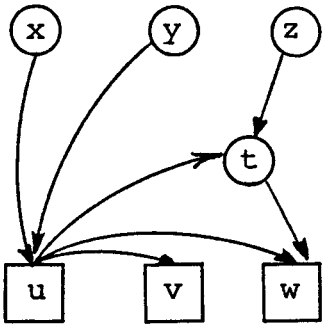


input variables

temporary variables

output variables

Retain that portion of the graph connected by at least one path to some square (i.e. to some output variable),and discard nodes not connected to some square by at least one directed path. Now eliminate assignments corresponding to deleted nodes. For instance, the final form of 1.1.2.a comes from the following graph:



input variables

temporary variables

output variables

1.2.2 Redundant Assignment Elimination

In a straight line sequence of assignments, if a variable is assigned twice in succession without being referenced after the first assignment but before the second, then the first assignment may be eliminated. (Aho and Ullman give an algorithm covering the functions of §1.2.1.b and §1.2.2 in [9] pp.850-851.)

1.2.2.a example

```
x←2*a+b;                    empty;
y←3*b-c;          =>        y←3*b-c;
z←d/4+e;                    z←d/4+e;
x←y+2*z;                    z←y+2*z;
```

1.2.3 Eliminating Assignments by Equality

1.2.3.a If the value of the variable a always equals the value of the variable b then $a \leftarrow b$ can be eliminated.

1.2.3.b special case:

$a \leftarrow a$ => *empty*

1.2.4 Eliminating Assignments by Constant Propagation and Substitution

In a straight line sequence of assignments, an assignment of the form $x \leftarrow E$ can be eliminated by substituting E for x everywhere x is referenced subsequently to the occurrence of $x \leftarrow E$ but before x or any of the variables of E are reassigned.

1.2.4.a example

```
x←2*y-a;                          empty;
u←t*x;                            u←t*(2*y-a) ;
v←q/(w-3)            =>           v←q/(w-3)
w←3*x+c;                          w←3*(2*y-a)+c ;
x←u+f(2);                         x←u+f(2) ;
```

1.2.4.b remark: By means of "symbolic execution", we can collapse any straight line sequence of assignment statements into n assignment statements, one for each of the n "output" variables whose values need to be used subsequently. Also, using this transformation, any variable which is assigned a single constant value may be systematically eliminated throughout its scope.

1.2.5 Reordering Assignments to Eliminate Temporary Storage

1.2.5.a Renaming:

Suppose x and y are distinct variables having disjoint scopes. Here, in a straight line sequence of assignments, the scope of a variable lasts from the place it is assigned, up to and including the last place it is referenced. Then y can be replaced with x everywhere in y's scope.

1.2.5.b example

```
x←2*a+b;                          x←2*a+b;

t← x/4-c;                         t←x/4-c;

u←x+m*i;                          u←x+m*i;
   ⋮ no more references              ⋮
   . to x beyond                     .
   . this point                      .

y←a-2*b;                  ⟹      x←a-2*b;

t←c-y/4;                          t←c-x/4;

u←m*i+y;                          u←m*i+x;
   ⋮ no more references              ⋮
   . to y beyond
     this point
```

try first §1.2.5.c.

1.2.5.c Changing Overlapping Scopes to Disjoint Scopes

Pairwise Permutation:

In order for the transformation of §1.2.5.a to apply, two distinct variables must have disjoint scopes. In some cases, a straight line sequence of assignments can be reordered so that overlapping scopes become disjoint. This may be accomplished by repeated application of pairwise permutation of sequential statements.

provided (x does not occur in E_2) and (y does not occur in E_1) and (x≠y):

```
x←E1;          =>      y←E2;
y←E2;                  x←E1;
```

1.2.5.d example

```
x←2*a-c;                x←2*a-c;
y←2/b+d;        =>      u←f(x);
u←f(x);                 y←2/b+d;
v←g(y);                 v←g(y);
```

try next §1.2.5.a.

1.2.6 Common Subexpression Elimination

Suppose the subexpression E is used two or more times on the right hand sides of assignments in a straight line sequence of assignments. Let t be a distinct new variable not appearing elsewhere in the program.

1.2.6.a <u>provided</u> no variable in E is reassigned in the statements between the assignments to x and y respectively, and t is a distinct new variable:

$$
\begin{array}{lcl}
x \leftarrow \ldots E \ldots ; & & t \leftarrow E; \\
\vdots & \Rightarrow & x \leftarrow \ldots t \ldots ; \\
y \leftarrow \ldots E \ldots ; & & \vdots \\
 & & y \leftarrow \ldots t \ldots ;
\end{array}
$$

1.2.6.b <u>example</u>

$$
\begin{array}{lcl}
x \leftarrow a - \sqrt{a^2+b^2} ; & & t \leftarrow \sqrt{a^2+b^2} ; \\
m \leftarrow x/(-b); & \Rightarrow & x \leftarrow a - t ; \\
y \leftarrow a + \sqrt{a^2+b^2} ; & & m \leftarrow x/(-b); \\
 & & y \leftarrow a + t;
\end{array}
$$

2. Go To Forms & Labels

2.1 Go To Chain Elimination

2.1.a example

```
    go to L1;                    go to L3;
       :                            :
L1: go to L2;     =>    L1: go to L3;
       :                            :
L2: go to L3;                L2: go to L3;
```

2.1.b example

```
go to L; L:S   =>   L:S
```

2.2 Elimination of Inaccessible Go To's & Labels

2.2.a Label Elimination - Let L be a label which does not appear in any go to statement in the scope* of L. Then L can be eliminated. try next 2.2.b

2.2.b Inaccessible Go To Elimination provided execution of S results in executing a Return or a go to L', and S is not a declaration:

```
... S; go to L ...   => ... S;...
```

---

* The scope of a label L is defined to be the lexicographically least enclosing block containing the label excluding any nested blocks using the same label. This convention follows ALGOL 60 usage.

2.3 Eliminating the Empty Loop

2.3.a provided B is false:

```
L: if B then
   begin          =>      L: empty
     S⃗;
     go to L
   end
```

2.3.b *while false do S* => *empty*

2.3.c try next §8.1

2.4 Other Transformations That Eliminate Go To's

2.4.a remark See §4.1.2, §4.3.1, §4.4.2, and §5.2 for transformations that eliminate go to's (namely, inverses of transforms that eliminate for, while, and repeat forms by reduction to explicit loops).

3. Conditional Forms

3.1 Trivial Conditional Simplifications

3.1.a *if B then true else false* => *B*

3.1.b *if B then false else true* => ∿*B*

3.1.c *if true then P else Q* => *P*

3.1.d *if false then P else Q* => *Q*

3.1.e *if B then P else P => P*

3.1.f *if B then P else empty => if B then P*

3.1.g *if B then empty else Q => if ∿B then Q*

3.1.h *if ∿B then P else Q => if B then Q else P*

3.2 Distribution and Factoring on Conditionals

3.2.a Let α be a unary operator in {+,-,∿}.
Let β be a binary operator in

{+,-,*,/,↑,=,≠,>,≥,<,≤,∨,∧,←}

Then,

provided a conditional is an operand of an operator:

α(if b then c else d) => (if b then αc else αd)

xβ(if b then c else d) => (if b then xβc else xβd)

(if b then c else d)βx => (if b then cβx else dβx)

3.2.b examples

∿(if a>5 then P(a) else Q(a)) =>

(if a>5 then ∿P(a) else ∿Q(a))

x+(if a>5 then y else z) => (if a>5 then x+y else x+z)

(if a>5 then x else y)←5 => (if a>5 then x←5 else y←5)

3.2.c Let the conditional *(if b then c else d)* be an argument to a function call $F(a_1, a_2, \ldots, a_n)$.
Then,

$F(a_1, \ldots, (\text{if } b \text{ then } c \text{ else } d), \ldots, a_n) \Rightarrow$

$(\text{if } b \text{ then } F(a_1, \ldots, c, \ldots, a_n) \text{ else } F(a_1, \ldots, d, \ldots, a_n))$

3.2.d Let the conditional *(if b then c else d)* be a subscript in a subscript expression. Then,

$A[i_1,\ldots,$ *(if b then c else d)*$,\ldots,i_k]$ =>

(if b then $A[i_1,\ldots,c,\ldots,i_k]$ *else* $A[i_1,\ldots,d,\ldots,i_k]$*)*

3.2.e *if(if b then c else d) then w* =>

if b then (if c then w) else (if d then w)

3.2.f In general, let $F(\ldots x\ldots)$ be a program form. Then,

$F(\ldots$*(if b then c else d)*$\ldots)$ =>

(if b then $F(\ldots c\ldots)$ *else* $F(\ldots d\ldots)$*)*

provided b commutes with any subexpression of F whose order of execution with respect to b is changed by application of this general transformation.

3.2.g example

provided v∉Names(b):

```
for v←1 step 1 until N do
   A[v,(if b then c else d)]←B[v]
```

⇓

```
for v ← 1 step 1 until N do
   if b then A[v,c]←B[v]else A[v,d]←B[v]
```

⇓

```
if b then
   for v←1 step 1 until N do A[v,c]←B[v]
else
   for v←1 step 1 until N do A[v,d]←B[v]
```

3.2.h remark: Factoring and distribution are inverse transformations. The rightward transformations above (=>) illustrate distribution. Their inverses (<=) are factoring.

3.3 Conditionalization of Boolean Expressions

3.3.a McCarthy Conditional Transformations:

provided a and b are side-effect-free:

$a \wedge b$ => *(if a then b else false)*

$a \vee b$ => *(if a then true else b)*

$\sim a$ => *(if a then false else true)*

3.3.b Variable-Directed Conditionalization

Let $F(a_1, a_2, \ldots, a_n)$ be a Boolean expression over Boolean primaries $a_1, a_2, \ldots, a_n$ *(for $n \geq 1$)* using the operators $\{\wedge, \vee, \sim\}$. Define Φ as follows:

provided $n \geq 2$:

$$\Phi F(a_1, a_2, \ldots, a_n) \Rightarrow \underline{if}\ a_1\ \underline{then}\ \Phi(\nabla(F(\underline{true}, a_2, \ldots, a_n)))\ \underline{else}\ \Phi(\nabla(F(\underline{false}, a_2, \ldots, a_n)))$$

provided $n = 1$:

$$\Phi(F(a_1)) = \nabla F(a_1)$$

Where ∇ is simplification with respect to elimination of Boolean constants as given in §11.3.

3.3.c example of Variable-Directed Conditionalization:

Let $F(a,b,c,d) = (a \wedge \sim b) \vee (d \wedge (a \vee c))$. Then,

$\Phi((a \wedge \sim b) \vee (d \wedge (a \vee c))) =$ *if* a

then $\Phi(\nabla((\textit{true} \wedge \sim b) \vee (d \wedge (\textit{true} \vee c))))$

else $\Phi(\nabla((\textit{false} \wedge \sim b) \vee (d \wedge (\textit{false} \vee c))))$

Applying removal of Boolean constants (∇w.r.t §11.3) yields:

then arm: $\nabla((\textit{true} \wedge \sim b) \vee (d \wedge (\textit{true} \vee c))) \Rightarrow (\sim b) \vee (d \wedge \textit{true})$

$\Rightarrow \sim b \vee d$

else arm: $\nabla((\textit{false} \wedge \sim b) \vee (d \wedge (\textit{false} \vee c))) \Rightarrow (\textit{false}) \vee (d \wedge c)$

$\Rightarrow d \wedge c$

whence

$\Phi((a \wedge \sim b) \vee (d \wedge (a \vee c))) \Rightarrow$

if a *then* $\Phi(\sim b \vee d)$ *else* $\Phi(d \wedge c)$

and $\Phi(\sim b \vee d) =$ *if* b *then* $\Phi(\nabla(\sim \textit{true} \vee d))$ *else* $\Phi(\nabla(\sim \textit{false} \vee d))$

$\Phi(d \wedge c) =$ *if* d *then* $\Phi(\nabla(\textit{true} \wedge c))$ *else* $\Phi(\nabla(\textit{false} \wedge c))$

So the entire net transformation is:

$(a \wedge \sim b) \vee (d \wedge (a \vee c)) \Rightarrow$ *if* a *then* (*if* b *then* d *else* *true*)

else (*if* d *then* c *else* *false*)

3.4 Reordering Nested Conditionals

3.4.a example

if a then (if b then c else d) else(if b then c else e)

=> if b then c else (if a then d else e)

3.4.b remark §3.4.a is profitable if b is almost always true and is cheap.

4. Looping Forms

4.1 Reduction of For-Forms

4.1.1 to While-Forms

4.1.1a *for v←a step b until c do S* =>

```
v←a;
while (v-c)*sign(b)≤0 do
    begin
    S;
    v←v+b
    end
```

try next: ∇w.r.t. §11.1 and §11.2

4.1.1.b example

for v← v+1 *while* v≤N *do* =>

```
[ v←v+1;

  while v≤N do

    [S; v←v+1]
```

4.1.2 To Explicit Loops

4.1.2.a Let L1 and L2 be distinct new labels not used elsewhere in the program:

```
for v←a step b until c do S  =>
  begin
      v←a;
  L1: if (v-c)*sign(b)>0 then go to L2;
      S;
      v←v+b;
      go to L1;
  L2: empty
  end
```

try next: ∇w.r.t. §11.1 and §11.2.

4.1.2.b Let L1 and L2 be distinct new labels not used elsewhere in the program.

```
for v←a while b do S  =>
begin
L1: v←a;
    if ∿b then go to L2;
```

```
    S;
    go to L1;
L2: empty
    end
```

4.1.3 with Plural For-List Elements

Let P be a distinct, new procedure name, not used elsewhere in the program.

4.1.3.a example Suppose $e_1, e_2, \ldots, e_n$ is a for-list where the e_i are each arithmetic expressions:

```
for v←e1,e2,...en do S  =>

    begin
    procedure P;S;
    v←e1;P;
    v←e2;P;
      ⋮
    v←en;P
    end
```

4.1.3.b example

```
for v←a step b until c, e while f do S =>

    begin
    procedure P;S;
    v←a;
    while (v-c)*sign(b)≤0 do[P; v←v+b ];
    v←e;
    while f do [P;v←e]
    end
```

try next ∇w.r.t. §11.

4.1.4 Simplifying Reduced For-Forms

4.1.4.a remark: When the for list elements are of a simple variety, simplification of a reduced for-form with respect to §11.1 and §11.2 is often possible. For example,

4.1.4.b example

for $i \leftarrow 1$ *step* 1 *until* N *do* $A[i] \leftarrow \emptyset$ =>

$$\nabla\left[\begin{array}{l} i \leftarrow 1; \\ \underline{while}\ (i-N)*sign(1) \leq \emptyset\ \underline{do}\ [A[i] \leftarrow \emptyset; i \leftarrow i+1] \end{array}\right]$$

Here, w.r.t. §11.1 and §11.2, ∇ yields the steps:

$(i-N)*Sign(1) \leq \emptyset$ => $(i-N)*1 \leq \emptyset$

=> $(i-N) \leq \emptyset$ => $(i \leq N)$.

So the above result simplifies to

$$\left[\begin{array}{l} i \leftarrow 1; \\ \underline{while}\ i \leq N\ \underline{do}\ [A[i] \leftarrow 0; i \leftarrow i+1] \end{array}\right]$$

4.2 Transforming Controlled Variables in For-Forms

4.2.1 Manipulating the Range

4.2.1.1 Shifting

provided v is local to the for-statement:

for $v \leftarrow a$ *step* b *until* c *do* $S(v)$ =>

for $v \leftarrow a \pm k$ *step* b *until* $c \pm k$ *do* $S(v \mp k)$

4.2.1.2 Dialating (=>) and Contracting (<=)

4.2.1.2.a provided *v* is local to the for-statement:

*for v←a step b until c do S(v*k)* <=>
*for v←a*k step b*k until c*k do S(v)*

4.2.1.2.b example (of contracting the range:)

for nickels ←5 step 5 until 100 do
P(nickels)
=> *for nickels ←1 step 1 until 20 do*
*P(nickels*5)*

4.2.1.3 Renaming

provided *i* and *j* are local, and

$j \notin Names(S(i))$

for i←a step b until c do S(i) =>
for j←a step b until c do S(j)

4.2.1.4 Simplifying Final Test to Check for Zero

4.2.1.4.a example Using §4.2.1.1 and §4.2.1.2 we can produce Final Tests on Zero:

for i←1 step 2 until 101 do P(i+6)
= *first shift range down by 101 (using 4.2.1.1)*

for i←1-101 *step* 2 *until* 101-101 *do*

P((i+101)+6)

⇓ then reduce to while loop and simplify:

i ← -100;

while (i-∅)*sign(2)≤0 *do* [P((i+101)+6); i←i+2]

⇓ ∇ gives final form:

i ←-100;

while i≤0 *do* [P(i+107);i←i+2]

4.3 While-Form Introduction & Elimination

4.3.1 Reduction to Explicit Loops

provided L is a distinct new label:

while b *do* S => L: *if* b *then*

begin

S;

go to L

end

See also §4.1.1, §4.4.1

4.4 Repeat-Form Introduction & Elimination

4.4.1 Reduction to While-Forms

repeat S until B => S; while ∿B do S

4.4.2 Reduction to Explicit Loops

provided L is a distinct new label:

repeat S until B => L:S;
if ∿ B then go to L

4.5 Loop-Form Transformations

4.5.1 Loop Fusion

4.5.1.a example

for i←1 step 1 until 20 do A[i]←B[i];

for j←5 step 5 until 100 do C[j/5]←∅;

transforming the second by renaming (j→i)
and contraction of the range (see §4.2.1.2.b)

for i←1 step 1 until 20 do A[i]←B[i];

for i←1 step 1 until 20 do C[i]←∅

now by loop fusion

for i←1 step 1 until 20 do
begin
A[i] ←B[i];
C[i] ←∅
end

4.5.2 Loop Doubling

4.5.2.a example

for $i \leftarrow 1$ *step* 1 *until* 100 *do* $A[i] \leftarrow B[i]$

=> *for* $i \leftarrow 1$ *step* 1 *until* 50 *do*

$$\begin{cases} A[i] \leftarrow B[i]; \\ A[i+50] \leftarrow B[i+50] \end{cases}$$

4.5.3 Loop Case Splitting

4.5.3.a example

for $i \leftarrow 1$ *step* 1 *until* 30 *do*

if $i \leq 15$ *then* $A[i] \leftarrow B[i]$ *else* $A[i] \leftarrow -B[i]$

$\Downarrow$

for $i \leftarrow 1$ *step* 1 *until* 15 *do* $A[i] \leftarrow B[i]$;

for $i \leftarrow 16$ *step* 1 *until* 30 *do* $A[i] \leftarrow -B[i]$

4.5.3.b remark contrast this with distribution of conditionals in §3.2.g.

4.5.4 Loop Unrolling

4.5.4.a example

for $i \leftarrow 0$ *step* 5 *until* 15 *do* $P(i+2)$ =>

```
P(2);
P(7);
P(12);
P(17)
```

4.5.5 Removal of Invariants

4.5.5.a example

```
for v←1 step 1 until N do
  begin
  x←y+2;
  A[v]←B[v-x]
  end
```

⇓

```
x←y+2;
for v←1 step 1 until N do A[v]←B[v-x]
```

4.5.6 Reduction in Operator Strength

4.5.6.a example Let k be a distinct, new variable:

```
for i←1 step 1 until 100 do
  A[i*25]←∅
```

‖ ∨

```
k←25;
for i←1 step 1 until 100 do
  [A[k]←∅;k←k+25]
```

4.5.6.b remark: compare with §4.1.2.1.a

applied to above input, which yields

$$\Downarrow$$

for $i \leftarrow 25$ *step* 25 *until* 2500 *do* $A[i] \leftarrow \emptyset$

4.5.7 Test Replacements & Loop Removal

4.5.7.a example

```
begin local i;                    begin
i←1;                              k←25;
k←25;                             while k≤2500 do
                          =>          [A[k]←∅;
while  i≤100 do                       k←k+25;
  [A[k]←∅;                        end
  k←k+25;
  i←i+1;
end
```

4.5.7.b remark the right side is the reduction of 4.5.6.a after i→j using While-Forms.

4.5.8 Nested Loop Simplification

4.5.8.a provided P is invariant over R* and P and Q are side-effect-free:

```
while P do                        while P do
  if Q then R else S      =>      [[while Q do R];S]
```

*P is invariant over R provided $W(R) \cap R(P) \doteq \emptyset$ (cf. page 9).

4.5.8.b provided P and R commute:

```
repeat                                  repeat
  if Q then R else S      =>              [while Q do R; S]
until P                                 until P
```

5. Compound Statements and Blocks

5.1 Eliminating Redundant Begin-End Pairs

5.1.a provided S is a single statement

```
begin
  S          =>      S
end
```

5.1.b

```
begin                                   begin
  begin                                   S
    S          =>
  end                                   end
end
```

try next §5.1.a

5.2 Statement Fusions

5.2.1 Forward Fusion of Conditionals

Provided S_1 and b commute:

$$S_1;\ \underline{if}\ b\ \underline{then}\ c\ \underline{else}\ d \ \Rightarrow\ \underline{if}\ b\ \underline{then}[S_1;c]\underline{else}[S_1;d]$$

5.2.2 Backward Fusion of Conditionals

$$(\underline{if}\ b\ \underline{then}\ c\ \underline{else}\ d);S_1 \Rightarrow\ \underline{if}\ b\ \underline{then}[c;S_1]\underline{else}[d;S_1]$$

5.2.3 Fusion of conditionals without Else Clauses

Provided b is invariant over c:

if b then c;
if ¬b then d $\Rightarrow$ *if b then c else d*

5.2.4 Nested Conditional Introduction

5.2.4.a example

$$\left[\begin{array}{l} \underline{if}\ a\ \underline{then}\ \{b;\underline{go}\ \underline{to}\ L\};\\ \underline{if}\ c\ \underline{then}\ \{d;\underline{go}\ \underline{to}\ L\};\\ e;\\ L: S_1 \end{array}\right] \Rightarrow \left[\begin{array}{l} \underline{if}\ a\ \underline{then}\ b\ \underline{else}\\ (\underline{if}\ c\ \underline{then}\ d\ \underline{else}\ e)\\ L: S_1 \end{array}\right]$$

try next §2.2.

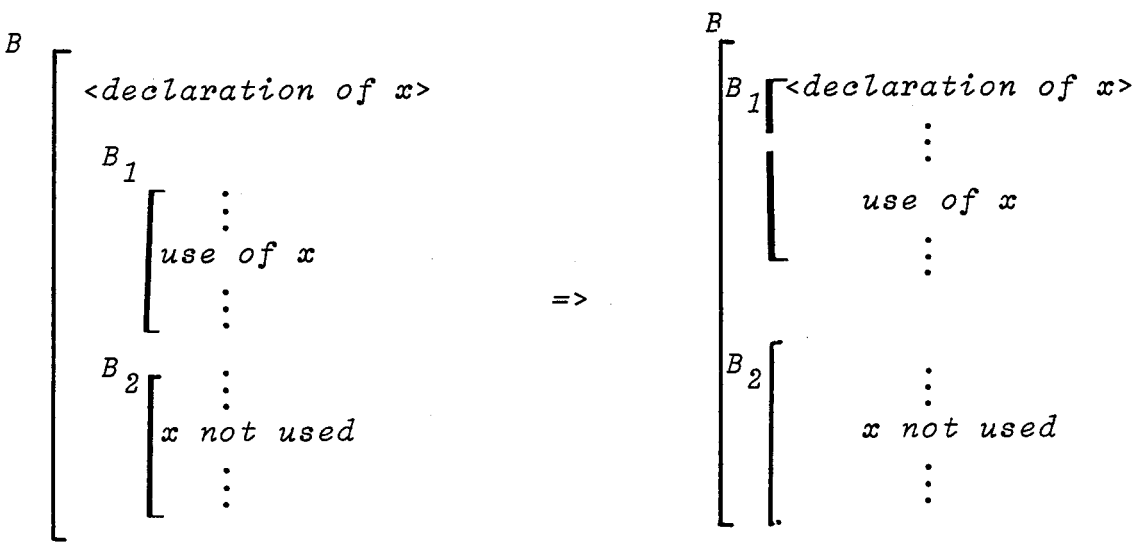
5.2.4.b example

provided (S_1 is a Return or a go to) and (S_2 is not labeled):

if a then $S_1;S_2$ => *if a then* S_1 *else* S_2

5.3 Using Block Structure to Save Space

Variables declared in disjoint blocks can use the same space since their activations occur at disjoint times. Hence, if a variable x is used in block B_1 but not in block B_2 and is declared in any block global both to B_1 and B_2, moving the declaration to the head of B_1 saves space:



6. Declarations

6.1 Eliminating Useless Declarations

If a variable is declared but never used within the scope of its declaration, the declaration can be eliminated.

```
begin <declaration of x> ;          begin empty;
  :                                   :
  x not used                          x not used
  :                          =>       :
end                                   end
```

6.2 Shifting Array Bounds

6.2.a If a k^{th} subscript i_k of an array $A[\ldots,i_k,\ldots]$ is declared to vary between m_k and n_k, then the declaration and use of A can be changed within its scope by applying the following transformations uniformly:

$$\underline{array}\ A[\ldots,m_k{:}n_k,\ldots] \Rightarrow \underline{array}\ A[\ldots,m_k \pm a{:}n_k \pm a,\ldots]$$

$$A[\ldots,i_k,\ldots] \Rightarrow A[\ldots,i_k \pm a,\ldots]$$

6.2.b example

```
begin integer array A[1:100];
  for i←1 step 1 until 100 do
    A[i]← ∅
end
```

⇓

```
begin integer array A[∅:99];
  for i←1 step 1 until 100 do
    A[i-1]← ∅
end
```

(Notice the latter can be improved by applying a range shift transformation given in §4.2.1.1)

⇓

```
begin integer array A[∅:99];
  for i←0 step 1 until 99 do
    A[i]←∅
end
```

6.3 Reduction in Array Dimension by Change of Declaration and Accessing

6.3.a Let $A[i_1,i_2,\ldots,i_r]$ be a subscripted expression for an array A of dimension r, where A is declared by array $A[m_1:n_1,m_2:n_2,\ldots,m_r:n_r]$. Let $f(i_1,i_2,\ldots,i_r)$ be a 1-1, onto mapping of subscripts $(i_1,i_2,\ldots,i_r)$ onto the range m:n. Then, the following transformations reduce A to linear dimension when applied uniformly in the scope of A.

array $A[m_1:n_1,m_2:n_2,\ldots,m_r:n_r]$ => *array* $A[m:n]$

$A[i_1,i_2,\ldots,i_r]$ => $A[f(i_1,i_2,\ldots,i_r)]$

6.3.b example

array $A[1:N,1:N]$; *array* $A[1:N^2]$;

$\ldots A[i,j]\ldots \Longrightarrow \ldots A[(i-1)*N+j]$

where $f(i,j) = (i-1)*N+j$

6.3.c more generally Let $g(m_1:n_1,\ldots,m_r:n_r) = m_1':n_1',\ldots,m_k':n_k'$ and $f(i_1,i_2,\ldots,i_r) = f'(i_1',i_2',\ldots,i_k')$ be a 1-1, onto mapping of r-tuples onto k-tuples in the respective domain and range Then,

array $A[m_1:n_1,\ldots,m_r:n_r]$ => *array* $A[m_1':n_1',\ldots,m_k':n_k']$

$\ldots A[i_1,\ldots,i_r]\ldots$ =>$\ldots$ $A[f(i_1,\ldots,i_r)]\ldots$

transforms A by changing its dimension, shape, or subscript ordering.

6.4 Fusion of Independent Modules into a Common Program with Suitable Renaming.

Let M_1 and M_2 be blocks using local names $L_1=(\ell_{11},\ell_{12},\ldots,\ell_{1m})$ and $L_2=(\ell_{21},\ell_{22},\ldots,\ell_{2n})$ respectively. Let $E=\{(e_i,e_i') \mid e_i\in L_1, e_i'\in L_2, (1\leq i\leq k)\}$ be an equivalence declaration which declares $e_i=e_i'$ for $(1\leq i\leq k)$, and let Left (E) $= \{e_i \mid \exists (e_i,e_i')\in E\}$. Compute a substitution $S=\{(e_i,n_i) \mid e_i\in$ Left(E) $\wedge n_i\notin L_1\cup L_2 \wedge n_i\neq n_j$ for $1\leq i,j\leq k\}$. S is a renaming of equivalence variables with distinct new names not conflicting with names in both L_1 and L_2. Now rename the modules and fuse them, as follows:

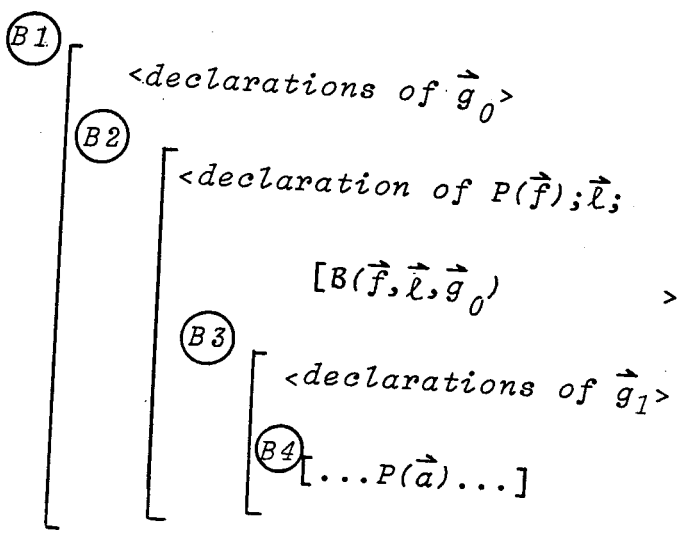
$$\left[\begin{array}{l} \langle \textit{declarations for variables in Left(E)}\rangle \\ \int_E \int_S [M_1 \\ \qquad [M_2 \end{array}\right.$$

7. Procedures

7.1 Eliminating Calls

7.1.1 Suitable Systematic Renaming

Suppose we have a procedure P which is declared and then called within the scope of its definition, as follows:



Here, P is declared with formal parameters $\vec{f}$ and local variables $\vec{\ell}$ at the head of a block (B2). Block (B1) is global to the definition and use of P, and it contains declarations of global variables $\vec{g}_0$. The body of P is a piece of text $B(\vec{f},\vec{\ell},\vec{g}_0)$

which is a function of $\vec{f}, \vec{\ell}$ and $\vec{g}_0$. The procedure P is called with a list $\vec{a}$ of actual parameter expressions at a point (B4) in the program. This point of call is embedded within a block (B3) containing declarations of variables $\vec{g}_1$ defined for the point of call, but not at the point of declaration of P. Thus, we can write $\vec{a}=a_1,a_2,\ldots,a_n$ where $a_i=A_i(\vec{g}_0,\vec{g}_1)$ for $(1\leq i\leq n)$ to indicate that each of the actual parameters in the call $P(\vec{a})$ can be a function of variables $\vec{g}_0$ and $\vec{g}_1$.

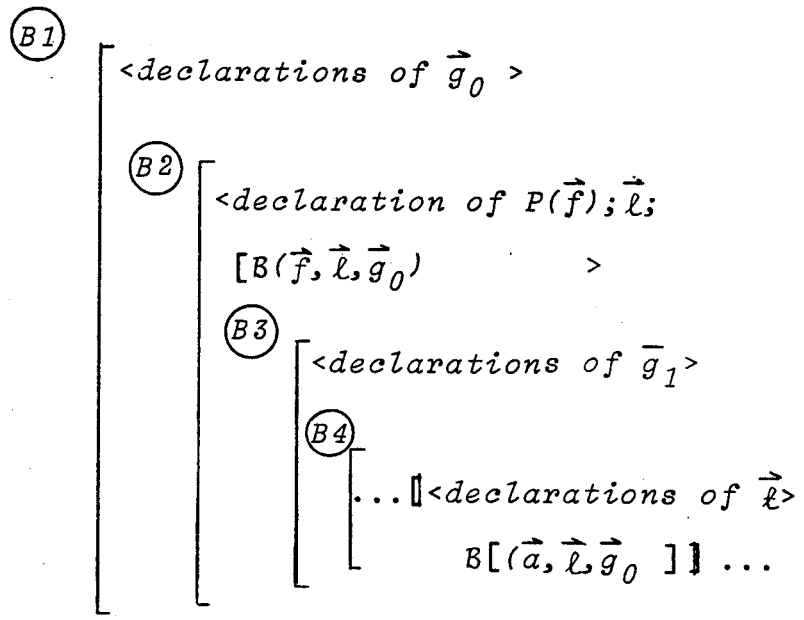
If the formal parameters $\vec{f}$ of P are called by name, we substitute the actual parameter expressions a used in the call $P(\vec{a})$ for respective occurrences of $\vec{f}$ in the body $B(\vec{a},\vec{\ell},\vec{g}_0)$, and we replace the call $P(\vec{a})$ at level (B4) with the block:

(B4) [...‖ <*declarations of* $\vec{\ell}$> $B(\vec{a},\vec{\ell},\vec{g}_0)$ ‖]*...

---

* The brackets ...[...]... are called "return intercept brackets". If an expression of the form Return(X) is executed within a pair of return intercept brackets, control returns to the text immediately surrounding the bracketed text, as if the bracketed text had been a procedure call (see §12.2.4).

Since this substitution may introduce name conflicts we must first employ "suitable systematic renaming" to eliminate them. The following diagram indicates the environments after replacing the call P($\vec{a}$) with the text of P:



Two forms of possible name conflicts arise when we compare this diagram to the original form of the program above. First, the proper bindings of the actual parameters $a_i = A_i(\vec{g}_0, \vec{g}_1)$ may be occluded by name conflicts with ℓ, so that names in $\ell \cap (\vec{g}_0 \cup \vec{g}_1)$ need to be changed. Second, the proper bindings of $\vec{g}_0$ may be occluded by name conflicts with $\vec{g}_1$, so that names in $\vec{g}_1 \cap \vec{g}_0$ need to be changed.

We adopt the policy of changing the most local declaration of a given conflicting name. We choose distinct new names, not used elsewhere in the program for this purpose. Thus, let $N = \bar{g}_0 \cup \bar{g}_1 \cup \bar{f} \cup \bar{\ell}$ be the set of names used in the program. Compute three substitutions S_1, S_2 and F as follows:

$$S_1 = \{(v_i, v_i') \mid (v_i \in \vec{g}_0 \cap \vec{g}_1),\ (v_i \text{ is used in } B(\vec{f}, \vec{\ell}, \vec{g}_0))$$

$$v_i' \notin N, \text{ and } v_i' \neq v_j' \text{ for } i \neq j\}$$

$$S_2 = \{(n_i, n_i') \mid n_i \in \vec{\ell} \cap (\vec{g}_0 \cup \vec{g}_1),\ (n_i \text{ is used in } \int_{S_1} \vec{a}),$$

$$n_i' \notin N, \text{ and } n_i' \neq n_j' \text{ for } i \neq j\}$$

$$F = \vec{f} \times \int_{S_1} \vec{a} = \{(f_i, \int_{S_1} a_i) \mid (1 \leq i \leq n)\}$$

Now create a substituted text as follows:

(B1) [<declarations of $\vec{g}_0$>

(B2) [<declaration of $P(\vec{f}); \vec{\ell}; B(\vec{f}, \vec{\ell}, \vec{g}_0)$ >

(B3) $\int_{S_1}$ [<declaration of $\vec{g}_1$ >

(B4) [$\iint_{F\ S_2}$ ‖ <declarations of $\vec{\ell}$>; $B(\vec{f}, \vec{\ell}, \vec{g}_0)$ ‖

If the parameters $\vec{f}$ of P($\vec{f}$) are called by value instead of by name, the block created at (B4) becomes:

$$\iint_{A\ S_2} \left[\!\!\left[\begin{array}{l} \text{<declarations of } \vec{f} \text{ and } \vec{\ell}\text{>;} \\ f_1 \leftarrow \alpha_1;\ f_2 \leftarrow \alpha_2; \ldots; f_n \leftarrow \alpha_n; \\ B(\vec{f}, \vec{\ell}, \vec{g}_0) \end{array} \right]\!\!\right]$$

where the substitution S_2 must be redefined to eliminate conflicts with $\vec{f}$ as well as $\vec{\ell}$, as follows:

$$S_2 = \{(n_i, n_i') \mid n_i \in (\vec{f} \cup \vec{\ell}) \cap (\vec{g}_0 \cup \vec{g}_1), (n_i \text{ is used in } \int_{S_1} \vec{a}),$$

$$(n_i' \notin N), \text{ and } n_i' \neq n_j' \text{ for } i \neq j\}$$

and where

$$A = \{(\alpha_i, \int_{S_1} a_i) \mid (\alpha_i \text{ are distinct new symbols } \alpha_i \notin N)$$

$$\text{and } a_i \in \vec{a}).$$

Before replacing the call P(a) with the derived text $\int_{S_2} [\!\![\text{<declarations of } \vec{\ell}\text{>}; B(\vec{a}, \vec{\ell}, \vec{g}_0)]\!\!]$ it is useful to attempt program simplication, since

formal parameters may be replaced by constants allowing partial evaluation of the surrounding procedure text as outlined in §7.1.2.

7.1.1.a example

```
(B1) begin integer j,x,y; integer array A[1:100];
       ...x←y←∅;j←4;...
     (B2)
       procedure P(q,r);integer q,r;integer i,j,k
         begin i←x+z;j←y-1;k←∅
         ... A[q+i]←A[r-(j+k)]...
         end
       (B3)
         begin integer i,x,z;
         ...x←z←1;i←3;
                  (B4) ...P(x+i,z+j)...
         end
     end
```

To compute the substitutions, S_1,S_2 and F, first set:

$$\vec{g}_0 = j,x,y,A$$

$$\vec{f} = q,r$$

$$\vec{l} = i,j,k$$

$$\vec{g}_1 = i,x,z$$

Then, using v´ to denote a distinct new symbol corresponding to v, we get

$S_1 = \{(x, x')\}$

$S_2 = \{(i, i'), (j, j')\}$

$F = \{(q, x'+i), (r, z+j)\}$

Using these substitutions we get the final transformed text:

```
(B1)
begin integer j,x,y;integer array A[1:100];
   ...x←y←Ø;j←4;...
 (B2) procedure P(q,r);integer q,r;integer i,j,k;
      begin i←x+2;j←y-1;k←0;
        ...A[q+i]←A[r-(j+k)]...
      end
        (B3)
        begin integer i,x´,z;
          (B4)
          [ integer i´,j´,k;
          begin i´←x+2;j´←y-1;k←Ø ;
          ...A[(x´+i)+i´]←A[(z+j)-(j´+k)]... ]
          end
        end
end
```

7.1.2 Partial Evaluation

7.1.2.a example

When the actual parameters $\vec{a}$ of a procedure call $P(\vec{a})$ are substituted for the formal parameters $\vec{f}$ in the body $B(\vec{f},\vec{\ell},\vec{g}_0)$ of the procedure declaration $P(\vec{f})$, a piece of substituted text $B(\vec{a}, \vec{\ell},\vec{g}_0)$ results which can frequently be simplified. For example, let P be declared as follows:

```
integer procedure Modulo(x,y);integer x,y;
Return (if y=0 then x else x-y*Entier(x/y))
```

and suppose we expand the call

```
... Modulo(A[i]+1,2)...
```

The substituted body becomes

```
⟦ Return(if 2=0 then A[i]+1 else
      (A[i]+1) - Entier((A[i]+1)/2)) ⟧
```

But since 2=∅ evaluates to false, we can use if false then A else B => B to simplify this to:

```
⟦ Return(A[i]+1-2*Entier( (A[i]+1)/2) ⟧
```

Then, using ⟦Return(x)⟧ => x, we get the final form:

```
...(A[i]+1) - 2*Entier( (A[i]+1)/2)...
```

7.1.2.b The Inside-Out Method

Partial evaluation of a piece of program text may be accomplished using an "inside-out" method which simplifies program constituents in the order of deeper to shallower levels of nesting. This works as follows:

1. First, perform all possible arithmetic on constants. Eg. 2*3+5 => 11, x*sign (6) => x*1 (see §11.1).
2. Second, perform algebraic simplifications, particularly elimination of algebraic identities (as in x*1 => x, y+0 => y, 0*x => 0) and trivial cancellations (as in x/x => 1, 2y-3y+y => ∅), (see §11.1).
3. Third, simplify relational expressions (eg. 3>0 => true, x≠x => false, x-y<∅ => x<y, -a≤-b => a≥b), (see §11.2).
4. Fourth, simplify Boolean expressions (eg. (x>0)∧(true∨false) => (x>0), ∿(true∨a) => ∿(true) => false. In particular, Boolean constants true and false that are operands of Boolean operators may be eliminated (see §11.3).

5. Finally, simplify control forms (eg. *if* *true* *then* x *else* y => x, *while* *false* *do* S => *empty*)

Whenever a function $f(\vec{a})$ is called with actual parameters $\vec{a}=a_1,a_2,\ldots,a_n$ that are all constants, the entire call can be simplified to a constant by normal procedure evaluation, provided f is side-effect-free.

8. The Mechanics of the Empty and Undefined Program Forms

8.1 *Empty* Introduction and Elimination

The *empty* program is an identity under program transformation, much the same way that 0 and 1 are identities under addition and multiplication (x+0 = x, x*1 = x). The *empty* program is created by transformations that map program fragments that do nothing into the explicit constant *empty*. For instance,

while *false* *do* S => *empty*

and

if *false* *then* A => *empty*

The *empty* program combines with program syntax in which it is embedded according to certain simplification laws.

8.1.a Empty Block Elimination

begin
empty => *empty*
end

8.1.b Empty For Statement Elimination
provided i is local to the for-statement:

for $i \leftarrow a$ *step* b *until* c *do* *empty*; => *empty*

8.1.c Empty Statement and Empty Declaration Elimination

begin $S_1;S_2;\ldots S_{i-1};$*empty*$;S_{i+1};\ldots;S_n$ *end* =>

begin $S_1;S_2;\ldots;S_{i-1};S_{i+1};\ldots;S_n$ *end*

As an example of the latter transformation, note that a labelled empty statement can be eliminated by transferring its label to the following statement:

```
...                    ...
L:empty ;      =>      L:S2 ;
S2;                    ...
...
```

8.1.d Empty Procedure Elimination .

procedure $P(\vec{x})$;<*declarations*>;*empty*=>*empty*

provided all calls $P(\vec{a})$ in P are also replaced by empty everywhere in the scope of the declaration of P.

8.1.e Empty While-Do and Repeat-Until Elimination

provided b is side-effect-free:

while b do empty => *empty*

provided b is side-effect-free:

repeat empty until b => *empty*

8.2 Undefined Introduction and Elimination

The undefined program (or, special case, the undefined value) can result from executing or transforming certain program forms. For instance, if a=0 then

a^0 => *undefined* ,

x/a => *undefined* , *and*

Log(a) => *undefined*

The undefined value can propagate. Whenever an operand of an operator, or an argument of a function is undefined, then the result is undefined.

8.2.a *a* <*operator*> *undefined* => *undefined*

undefined <*operator*> *a* => *undefined*

<*operator*> *undefined* => *undefined*

$f(a_1, a_2, \ldots, a_n)$ => *undefined if* a_i = *undefined*

for some i such that $(1 \leq i \leq n)$.

8.2.b When undefined is a body of an iterative control form, or a procedure, the result is undefined.

while b do undefined => *undefined*

repeat undefined until b => *undefined*

for v←a step b until c do undefined => *undefined*

8.2.c When undefined is an arm of a conditional then the result is undefined unless the arm is not executable. Thus, for example,

if x>0 then undefined else P(a) => *undefined*

if false then undefined else P(a) => *P(a)*

if true then x else undefined => *x*

8.2.d A call in a procedure which has not been declared is undefined.

provided P has not been declared :

$P(\vec{a})$ => *undefined*

8.2.e example

In general, the undefined program form may appear as a result of an undefined operation, but it may, in turn, be eliminated through transformations of the text in which it is embedded. For instance, expanding the call Modulo(5,0) in the function

```
integer procedure Modulo(x,y) ;integer x,y;
  Return (if y=0 then x else x-y*Entier (x/y))
```

gives:

```
[ Return (if 0=0 then 5 else 5-0*Entier (5/0))]
     ⇓    8.2
[ Return (if true then 5 else 5-0*Entier (undefined))]
     ⇓    8.2.a
[ Return (if true then 5 else 5-0*undefined) ]
     ⇓    8.2.a  twice
[ Return (if true then 5 else undefined) ]
     ⇓    8.2.c
[ Return (5) ]
     ⇓
     5
```

In strict inside-out evaluation by mechanical means, it is convenient to be able to deal with the undefined program.

9. High-Level Forms

9.1 Reduction versus Recognition

Reduction is a transformation that maps programs in a language L into programs in a proper subset L´ of L. For instance, the reductive transformation:

```
while b do S   =>   L: if ¬b then
                       begin
                       S;
                       go to L
                       end
```

maps programs written in a superset of Algol 60 into pure Algol 60.

Recognition is the inverse of reduction. It replaces program fragments written in a language L´ with equivalent constructions in an extension L of L´ (where L´⊂L).

An example of recognition mapping Algol 60 into an extended Algol 60 is:

```
                                        i←1;
for i←1 step 1 until N do    =>     repeat A[i]←0; i←i+1
    A[i]←0                          until i>N
```

Reduction can be used to map programs written in an extension E of a base language B into programs written only in B. If accompanied by simplification, reduction can produce increased efficiency, but this may occur at the expense of reduced legibility. Conversely, recognition may improve legibility at the expense of efficiency.

While the transformations in §9.2 are given as reductions, they may be applied in the reverse direction to produce recognitions.

9.2 Reductions

9.2.1 Parallel Assignments

A parallel assignment takes the form

$(x_1, x_2, \ldots, x_n) \leftarrow (e_1, e_2, \ldots, e_n)$

and assigns the values of e_i to x_i all at once (for $1 \leq i \leq n$). For instance, $(x,y) \leftarrow (y,x)$ exchanges the values of x and y.

provided $x_1, x_2, \ldots, x_n$ are variables, $x_1', x_2', \ldots, x_n'$ are distinct new variables not used elsewhere in the program, and e_i are side-effect-free ($1 \leq i \leq n$):

$(x_1, x_2, \ldots, x_n) \leftarrow (e_1, e_2, \ldots, e_n)$ =>

$x_1' \leftarrow e_1; x_2' \leftarrow e_2; \ldots; x_n' \leftarrow e_n;$

$x_1 \leftarrow x_1'; x_2 \leftarrow x_2'; \ldots, x_n \leftarrow x_n'$

If some of the x_i are subscripted expressions, then this reduction is valid provided no x_j is used in a subscript in x_i for $1 \leq j \leq i$. For instance, (A[i],i)← (i,A[i]) => A'[i]←i;i'←A[i];A[i]←A'[i];i←i' is valid, but (i,A[i])←(A[i],i) => i'←A[i];A'[i]←i;i←i';A[i]←A'[i] is invalid

(Minimizing the number of temporary variables used to serialize a parallel assignment has been shown to be NP-complete by R. Sethi[1]).

9.2.2 Iterators

9.2.2a example

provided A is an array [1:N,1:M];

for x∈ *(subscripts of A)* *do* A[x]←0 =>

for i← 1 *step* 1 *until* N *do*
for j← 1 *step* 1 *until* M *do* A[i,j]←0;

9.2.2.b example (printing prime pairs between N and M)

```
[ for x∈(the set of odd integers between N and M) do
    if ((x) is prime) and ((x+2) is prime) then
    Print(x, newline,x+2, newline) ]

=>  [ x←2*Entier(N/2)+1;
      while x≤M do
        begin
        if ((x) is prime) and ((x+2) is prime)
          then Print(x, newline, x+2, newline);
        x←x+2
        end ]
```

9.2.2.c remark The use of high level iteration clauses such as x∈(Subscripts of A) or x∈(The set of odd integers) can sometimes make a program easier for people to read since the "key idea" is described directly rather than being buried in the mechanics of sequencing. Using high-level iterators can occasionally make program text more concise, since a large amount of equivalent low-level program text is sometimes required to express the same idea.

If an iteration clause can be given a perfectly precise underlying meaning by means of mechanical transformation, there is no reason not to use

it in rendering high-level program text more legible and concise for people.

9.2.3 Zahn and Dahl Loop Reduction

We use the form of Zahn and Dahl loops given by Knuth [2].

9.2.3.a Zahn Loop Reduction

```
loop until <event>_1 or . . . or <event>_n:
        <statement list>_0 ;
repeat;
then <event>_1 → <statement list>_1;
   ⋮
     <event>_n → <statement list>_n;
end
```

where $\langle statement\ list \rangle_0$ contains occurrences of $\langle event \rangle_i$ for $(1 \leq i \leq n)$, and $\langle event \rangle_i$ are distinct Boolean variables.

⇓

```
<event>_1←<event>_2←...←<event>_n←false;

loop: ∫_S1 <statement list>_0 ;
      go to loop;

next: if <event>_1 then[<statement list>_1] else
        ⋮
      if <event>_n then[<statement list>_n] ;
```

where the substitution S_1 is given by

$$S_1=\{(\langle event\rangle_i, [\langle event\rangle_i \leftarrow true; go\ to\ next])\mid (1\leq i\leq n)\}$$

and where loop and next are distinct new labels not appearing elsewhere in the program.

9.2.3.b Dahl Loop Reduction

$$loop;\vec{S};while\ B:\vec{T};repeat;$$

$$\Downarrow$$

$$L1:[\vec{S}];$$
$$if\ \sim B\ then\ go\ to\ L2;$$
$$[\vec{T}];\ go\ to\ L1;$$
$$L2:\ empty$$

where $\vec{S}$ and $\vec{T}$ may be sequences of one or more statements and where L1 and L2 are distinct new labels not used elsewhere in the program.

10. High Level Transformations

10.1 Eliminating Search Exhaustion Tests by Data Structure Extension

10.1.1 example (See Introduction (pages 4-7) for explanation):

```
procedure Search(T,X); array T[1:N]; integer i;
 begin i←N;
 while i>0 do if T[i]=X then Return(true) else i←i-1;
 Return(false)
 end
```



```
procedure Search(T,X);array T[0:N]; integer i;
 begin T[0]←X; i←N;
 while T[i]≠X do i←i-1;
 Return(i≠0)
 end
```

10.2 Recursion Removal

10.2.1 Replacing the Final Call With a Go To

10.2.1.a example (of printing a binary tree T composed of nodes which are either empty, or contain an Info field which gives information to be printed, and Left and Right Subtree fields which give pointers to left and right subtrees, (cf. Knuth [2], p.281)).

```
procedure TreePrint(T); binary tree T; value T;
 begin
 if Empty(T) then Return;
 TreePrint(LeftSubTreeOf(T));
 Print(Info(T));
 TreePrint(RightSubTreeOf(T))
 end
```

```
procedure TreePrint(T); binary tree T; value T;
 begin
 if Empty(T) then Return;
L: TreePrint(LeftSubTreeOf(T));
 Print(Info(T));
 T←RightSubTreeOf(T); go to L
 end
```

10.2.2 Replacing a Recursive Call Using an Explicit Stack

10.2.2.a example (see Knuth [2], pp. 281-282):

The latter program in §10.2.1.a transforms to:

```
procedure TreePrint(T); binary tree T; value T;
 begin stack S; S←emptystack;
 L1: while~Empty(T) do
       [Push(T)onto(S); T←LeftSubTreeOf(T);
       go to L1;
    L2:T←TopOf(S); Pop(S);
       Print(Info(T));
       T←RightSubTreeOf(T)
 if ~Empty(S) then go to L2;
 end
```

remark This program contains an instance of a jump into the middle of a loop. For an interesting defense of this "mortal sin", see Knuth [2],p.282.

10.2.3 Replacing a Descending Recursion with a Descending Iteration

10.2.3.a example

```
procedure Factorial(N); integer N; value N;
  Return( if N=0 then 1 else N*Factorial(N-1))
```

```
procedure Factorial(N); integer N; value N;
  begin integer i;
  Factorial←1;
  while N>0 do
     [Factorial←N*Factorial;
     N←N-1
  end
```

remark Recursion removal can be done mechanically by introducing explicit stacks. Transformation to equivalent,stackless, iterative forms has received frequent attention in the literature. See for example, McCarthy [3], Manna and Waldinger[5], and Darlington and Burstall[4]. We do not attempt here a representative summary of its many facets.

10.3 Refinement of Abstract Data Structures

10.3.1 example (the set membership predicate $X \in Y$):

Given an algorithm P written using set notation (e.g. using expressions such as $A \cap (B \cup C)$, or if $X \in Y$ then $C \leftarrow C \cup \{X\}$), we may wish to choose one of many possible underlying representations R for sets, and to map P into a concrete program using R. Here, we exemplify what might occur if X were a character, Y were a set of characters, and we chose to represent Y using a list.

Thus, we assert:

(1) Let (X) be a (Character)
(2) Let (Y) be a (Set of (Character)s)
(3) Represent (Y) by a (List)

Now we apply the following deductive assertion to statement (3):

if (the Representation of (Y) is a (List)) then
Assert(the Representation of (Y) is
(Finitely Enumerable))

As a consequence, a new statement can be made:

(4) The Representation of (Y) is (Finitely Enumerable)

We now examine the following transformation:

provided (There exists a (Z) such that
((X) is a (Z)) and ((Y) is a set of (Z)s) and
(The Representation of (Y) is (Finitely Enumerable)):

(X∈Y) => [for each a such that a∈Y do
if X=a then Return(true);
Return(false)]

The enabling condition of this transformation is satisfied by assertions (1),(2) and (4) above. Hence, (X∈Y) can be rewritten as:

(5) [for each a such that a∈Y do
if X=a then Return(true);
Return(false)]

Now the following transformation applies:

provided The Representation of (Y) is a (List):

```
for each a such that aϵY do S(a)    =>
   [list t;  t←Y;
    while t≠NIL do
      begin
      S(head(t));
      t←tail(t)
      end ]
```

It maps the last given form of the program in (5) into:

```
‖ [list t; t←Y;
   while t≠NIL do
       begin
       if X=head(t) then Return(true);
       t←tail(t)
       end ];
   Return(false) ‖
```

The latter piece of text is the concrete implementation of the predicate XϵY, when Y is a list.

If the representation for Y had been a bit-vector, or an array instead of a list, the underlying concrete code generated would have been different. For example,

provided The Representation of (Y) is (Array[M:N]):

```
for each a such that aϵY do S(a)   =>
 [integer i;
  for i←M step 1 until N do S(Y[i])]
```

is a transformation which applies if Y is represented by a linear array. This transformation maps (5) above into the following concrete underlying program text:

```
[ [integer i;
   for i←M step 1 until N do
       if X=Y[i]then Return(true) ];
   Return(false) ]
```

10.3.2 remark In a more extensive fashion, Schwartz[6] has shown how abstract operations on sets can be implemented concretely, and the book by Dijkstra, Dahl, and Hoare[7] gives numerous examples of refinement of abstract data into concrete representations.

10.4 Replacing Loops that Sum Polynomials with Polynomials of Higher Degree

10.4.1 The general idea of the transformation is;

provided f is a polynomial of degree n:

```
S←0;
for i←a step b until c do    =>    S←g(c)
   S←S+f(i)                        where g is a polynomial
                                     of degree n+1.
```

10.4.2 example

```
S←0;
for i←1 step 1 until n do    =>    S←(n+1)*n^2
  S←S + 3*i^2-i/2
```

In many cases, the polynomial g can be determined using the calculus of finite differences (see Goldberg [8] for details).

10.5 Procedural Abstraction

Naive programmers often repeat the same, or nearly the same, sequence of instructions. For example, in the computation of the Cosine of the angle between two 3-vectors v and w one might find the code:

```
S←0;
for i←1 step 1 until 3 do
    S←S+v[i]*w[i];
R←0;
for i←1 step 1 until 3 do
    R←R+v[i]* v[i];
T←0;
for i←1 step 1 until 3 do
    T←T+w[i]*w[i];
Result←S/(R*T)
```

By recognizing the common loop one can replace this code by the following:

```
real procedure Dot(x,y);real array x,y[1:3];
  begin integer i;
  Dot←0;
  for i←1 step 1 until 3 do Dot←Dot+x[i]*y[i];
  end;
Result←Dot(v,w)/(Dot(v,v)*Dot(w,w))
```

Procedural abstraction is the inverse of expansion of a procedure call into a substituted, partially evaluated, procedure text. A program to identify unifiable pieces of code must be able to take a set of similar program fragments (say similar with respect to structure), $P_1, P_2, \ldots, P_n$, and to construct a least common generalization.

11. Manipulation of Expressions

11.1 Arithmetic Expressions

11.1.1 Performing Arithmetic on Constants

Let *a* and *b* be arithmetic constants, and let *execute(F)* be the result of evaluating program fragment *F* to produce a value. Let α be an operator such that $\alpha \in \{+, -, \times, \div, \uparrow, <, \leq, =, \geq, >, \neq, sin, cos, abs, entier, log, exp, sign\}$. Then,

$$(\alpha a) \Rightarrow execute(\alpha a)$$
$$(a \alpha b) \Rightarrow execute(a \alpha b)$$

11.1.2 Eliminating Arithmetic Identities

$a+0 \Rightarrow a$ $\qquad$ $a-0 \Rightarrow a$

$0+a \Rightarrow a$ $\qquad$ $0-a \Rightarrow -a$

$a*1 \Rightarrow a$ $\qquad$ $a/1 \Rightarrow a$

$1*a \Rightarrow a$ $\qquad$ $0/a \Rightarrow 0$

$0*a \Rightarrow 0$

$a*0 \Rightarrow 0$

remark: See §8.2 for cases that yield an undefined result.

$a\uparrow 0 \Rightarrow 1$

$1\uparrow a \Rightarrow 1$

$a\uparrow 1 \Rightarrow a$

11.1.3 Trivial Cancellations

```
x-x => 0          x+(-x) => 0          x/x => 1
```

11.1.4 Eliminating Unary Minus

```
-(-a) => a                    (-a)-b => -(a+b)
(-a)+(-b) => -(a+b)           (-a)*b => -(a*b)
(-a)-(-b) => b-a              a*(-b) => -(a*b)
(-a)*(-b) => a*b              (-a)/b => -(a/b)
(-a)/(-b) => a/b              a/(-b) => -(a/b)
(-a)+b  => b-a                a↑(-b) =>1/(a↑b)
a+(-b)  => a-b                (-a)↑N =>((-1)↑N)*(a↑N)
a-(-b)  => a+b                -(a-b) => b-a
```

11.1.5 Clearing Rational Fractions

```
(a/b)*(c/d) => (a*c)/(b*d)
(a/b)/(c/d) => (a*d)/(b*c)
(a/b)*c     => (a*c)/b
c*(a/b)     => (a*c)/b
(a/b)/c     => a/(b*c)
c/(a/b)     => (b*c)/a
(a/b)↑N     => (a↑N)/(b↑N)
(a/b)+(c/d) => (a*d+b*c)/(b*d)
(a/b)-(c/d) => (a*d-b*c)/(b*d)
(a/b)+c     => (a+c*b)/b
(a/b)-c     => (a-c*b)/b
a-(b/c)     => (a*c-b)/c
a+(b/c)     => (a*c+b)/c
```

11.1.6 Suggestions for Simplifying Rational Forms

(a) Perform arithmetic on constants (§11.1.1).

(b) Eliminate identities (0 and 1) and perform trivial cancellations (§11.1.2 and §11.1.3)

(c) Eliminate unary minus (§11.1.4)

(d) Clear fractions (§11.1.5). Now the expression is either in the form P or the form P/Q where P and Q are free of division operators.

(e) Multiply out P and Q, collect like terms, and cancel terms where possible.

(f) Try to factor P and Q and remove the polynomial gcd.

(g) Optionally express polynomials in the numerator (and denominator, if applicable) using Horner's Rule for quick evaluation. E.g.

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 \Rightarrow$$

$$a_0 + x*(a_1 + x(a_2 + \ldots x(a_n)\ldots))$$

11.1.7 remark The above algebraic transformations produce only mild simplifications of algebraic expressions. These are only the most rudimentary and skeletal indications of the sort of manipulations required. Not only is algebraic simplification unsolvable, in general, it requires a sizable amount of sophisticated code, in particular cases. The approach of using only pattern-directed transformations such as those above has been tried by Fenichel and found to be deficient in several respects. The code for the MACSYMA System, a sophisticated algebraic manipulation system, amounts to over 300,000 words on the DEC PDP-10.

This Catalogue does not aspire to swallow the whole field of algebraic manipulation as a special case of program manipulation, but instead emphasizes program transformations outside the realm of conventional algebraic manipulation.

11.2 Relational Expressions

11.2.1 Relational Simplifications

We define the following set: $\langle rel \rangle = \{<, \leq, =, \neq, \geq, >\}$.

11.2.1.a Cancellations:

$a \pm c \ \langle rel \rangle \ b \pm c \ \Rightarrow \ a \ \langle rel \rangle \ b$

provided $c>0$:

$a*c \ \langle rel \rangle \ b*c \ \Rightarrow \ a \ \langle rel \rangle \ b$

11.2.1.b Simplifications with 0 and 1

$a-b \ \langle rel \rangle \ 0 \ \Rightarrow \ a \ \langle rel \rangle \ b$

$0 \ \langle rel \rangle \ a-b \ \Rightarrow \ b \ \langle rel \rangle \ a$

provided $b>0$:

$a/b \ \langle rel \rangle \ 1 \ \Rightarrow \ a \ \langle rel \rangle \ b$

$1 \ \langle rel \rangle \ a/b \ \Rightarrow \ b \ \langle rel \rangle \ a$

provided $a>0$ and $b>0$:

$(1/a) \ \langle rel \rangle \ (1/b) \ \Rightarrow \ b \ \langle rel \rangle \ a$

11.2.1.c Simplifications to true and false

$a=a \Rightarrow true \quad a \leq a \Rightarrow true \quad a \geq a \Rightarrow true$

$a \neq a \Rightarrow false \quad a>a \Rightarrow false \quad a<a \Rightarrow false$

11.2.2 Arithmetic Negations of Relations

$$-a < -b \Rightarrow a > b$$
$$-a \leq -b \Rightarrow a \geq b$$
$$-a > -b \Rightarrow a < b$$
$$-a \geq -b \Rightarrow a \leq b$$
$$-a = -b \Rightarrow a = b$$
$$-a \neq -b \Rightarrow a \neq b$$

11.2.3 Logical Relational Negations

$$\sim(a = b) \Rightarrow a \neq b$$
$$\sim(a \neq b) \Rightarrow a = b$$
$$\sim(a < b) \Rightarrow a \geq b$$
$$\sim(a \leq b) \Rightarrow a > b$$
$$\sim(a > b) \Rightarrow a \leq b$$
$$\sim(a \geq b) \Rightarrow a < b$$

11.2.4 Relational Synonyms

Let $\langle ge \rangle = \{>, \geq\}$ and let $\langle le \rangle = \{<, \leq\}$:

$$(a \leq b) \Rightarrow (a < b) \vee (a = b)$$
$$(a \geq b) \Rightarrow (a > b) \vee (a = b)$$

$$a = b = c \Rightarrow (a = b) \wedge (a = c)$$
$$a \langle ge \rangle_1 b \ \langle ge \rangle_2 c \Rightarrow (a \langle ge \rangle_1 b) \wedge (b \langle ge \rangle_2 c)$$
$$a \langle le \rangle_1 b \ \langle le \rangle_2 c \Rightarrow (a \langle le \rangle_1 b) \wedge (b \langle le \rangle_2 c)$$

11.2.5 Combining Ranges

11.2.5.a examples

$$(a \leq x \leq b) \wedge (c \leq x \leq d) \Rightarrow max(a,c) \leq x \leq min(b,d)$$
$$(a \leq x \leq b) \vee (c \leq x \leq d) \Rightarrow \underline{if}\ (b \geq c) \wedge (a \leq d)\ \underline{then}\ min(a,c) \leq x \leq max(b,d)$$
$$\underline{else}\ (a \leq x \leq b) \vee (c \leq x \leq d)$$

provided $a>b$: $(a \leq x \leq b) \Rightarrow \underline{false}$

11.3 Boolean Expressions

11.3.1 Eliminating Boolean Constants

$a \wedge \underline{true} \Rightarrow a$	$a \vee \underline{true} \Rightarrow \underline{true}$
$\underline{true} \wedge a \Rightarrow a$	$\underline{true} \vee a \Rightarrow \underline{true}$
$a \wedge \underline{false} \Rightarrow \underline{false}$	$a \vee \underline{false} \Rightarrow a$
$\underline{false} \wedge a \Rightarrow \underline{false}$	$\underline{false} \vee a \Rightarrow a$

$$\sim\underline{true} \Rightarrow \underline{false}$$
$$\sim\underline{false} \Rightarrow \underline{true}$$

11.3.2 Boolean Simplifications and Equivalences

$a \wedge b \Leftrightarrow b \wedge a$ $a \vee b \Leftrightarrow b \vee a$	*Commutative Laws*
$a \wedge (b \wedge c) \Leftrightarrow (a \wedge b) \wedge c$ $a \vee (b \vee c) \Leftrightarrow (a \vee b) \vee c$	*Associative Laws*
$a \wedge a \Leftrightarrow a$ $a \vee a \Leftrightarrow a$	*Idempotent Laws*
$a \wedge (b \vee c) \Leftrightarrow (a \wedge b) \vee (a \wedge c)$ $a \vee (b \wedge c) \Leftrightarrow (a \vee b) \wedge (a \vee c)$	*Distributive Laws*
$\sim(a \wedge b) \Leftrightarrow (\sim a) \vee (\sim b)$ $\sim(a \vee b) \Leftrightarrow (\sim a) \wedge (\sim b)$	*DeMorgan's Laws*
$\sim\sim a \Leftrightarrow a$	*Double Negation Law*
$a \wedge (a \vee b) \Leftrightarrow a$ $a \vee (a \wedge b) \Leftrightarrow a$	*Subsumption Laws*

$\sim a \vee (a \wedge b) \iff (\sim a) \vee b$	*Cancellation Laws*
$\sim a \wedge (a \vee b) \iff (\sim a) \wedge b$	
$(a \wedge \sim b) \vee (a \wedge b) \iff a$	*Ground Resolution Laws*
$(a \vee \sim b) \wedge (a \vee b) \iff a$	
$(a \vee \sim a) \iff \underline{true}$	*Excluded Middle Law*
$(a \wedge \sim a) \iff \underline{false}$	*Contradiction Law*

11.3.3 remark Boolean simplification is at least as hard as the NP-Complete problems and no method of simplification is known requiring less than exponential time.

12. Appendices

12.1 Notational Conventions

12.1.1 Sequences

Vector notation is used to denote sequences of statements or lists of identifiers or arguments:

statement sequences: $\vec{S} = S_1;S_2;\ldots;S_n$

identifier sequences: $\vec{v} = v_1,v_2,\ldots,v_n$

argument sequences: $\vec{a} = a_1,a_2,\ldots,a_n$

12.1.2 Substitution

A Substitution is a set of ordered pairs of the form (n,X), where n is a name, and X is an expression to be substituted for n.

$$S = \{(n_i,X_i) \mid n_i \text{ are distinct names } \}$$

Let F be a program fragment containing instances of the names n_i. Then the result of substituting X_i for every occurrence of n_i simultaneously and uniformly in F is denoted:

$$\int_S F$$

This is pronounced the "substitution of F with respect to S".

For example, Let $S = \{(x, a+b), (y, m*k), (z,F(y))\}$ and let $F =$ *begin* $q \leftarrow x*z$; *if* b *then* *Return*(y) *end* . Then,

$$\int_S F = \text{ begin } q \leftarrow (a+b)*F(y); \text{if } b \text{ then } Return(m*k) \text{ end}$$

12.1.3 Simplification

The simplification of a program fragment F with respect to a set of transformations x, is denoted

$$\nabla_x F$$

or, where the simplification transformations are apparent from context, just simply ∇ F.

12.2 Programming Language Forms

12.2.1 Use of Algol 60 Conventions

The syntax and terminology used in the Catalogue follow Algol 60 conventions, except for a few extensions, which have been added to provide coverage of structured programming syntax.

12.2.2 Structured Programming Extensions

12.2.2.a While-Do Forms:

While <*Boolean Expression*>*Do* <*Statement*>

12.2.2.b Repeat-Until Forms:

Repeat <*Statement*> *Until* <*Boolean Expression*>

12.2.2.c Return Expressions and Statements:

To exit from the execution of a procedure $P(\vec{a})$, the statement form *Return* can be used. To exit from a procedure $P(\vec{a})$ and, simultaneously, to return the value of the expression *E* the expression *Return(E)* can be executed. Executing *Return(E)* has the same effect as executing *begin P←E;Return end* where *P* is the procedure identifier.

12.2.3 Synonyms for Begin-End Brackets

Any form of parentheses or brackets can be used to stand for Begin-End pairs in Algol 60. The Catalogue uses such bracketing as

. . . [. . .] . . .

. . . (. . .) . . .

and

$$\left\lceil \begin{array}{l} S_1; \\ S_2; \\ \quad . \\ \quad . \\ \quad . \\ S_n \end{array} \right. .$$

In the latter case, indentation and the left square bracket in the margin, together indicate the scope of a Begin-End pair delimiting a compound statement or block.

12.2.4 Return-Intercept Brackets

The special brackets ...⟦ ... ⟧ ... are called "return-intercept brackets". Whenever a *Return* statement, or a return expression of the form *Return(E)*, is executed within return-intercept brackets, control passes to the text immediately surrounding the least enclosing pair of return-intercept brackets, as if the brackets delimited the text of a procedure call. Otherwise, the return-intercept brackets function exactly like Begin-End brackets. Return-intercept brackets can be used to set up "block expressions" as in CPL.

For example, the following text computes $X/\sum_{i=o}^{n} f(i)$:

```
X/[real S; integer i; S←0;
        for i←0 step 1 until n do S←S+f(i);
        Return(S)]
```

12.2.5 Extended Identifiers and Procedure Calls

Sequences of identifiers separated by spaces stand for single identifiers, and procedure calls and declarations may be given in the following extended forms;

12.2.5.a examples (of extended identifiers)

the Left Hand Side
Numerator of X
Output Buffer

12.2.5.b examples (of extended procedure calls)

Pop(T) From (S)

Write(M+5) On the (Mag Tape) Device

REFERENCES

[1] Sethi,R., A Note on Implementing Parallel Assignment Instructions, Information Processing Letters 2(1973),91-95.

[2] Knuth,D.E.,Structured Programming with Go To Statements, Computing Surveys 6,4(Dec. 1974),261-301.

[3] McCarthy,J., A Basis for a Mathematical Theory of Computation, in Computer Programming and Formal Systems, North Holland,(1973),33-70.

[4] Darlington,J. and Burstall,R.M., A System Which Automatically Improves Programs, Third International Joint Conference on Artificial Intelligence, Proceedings,(Aug. 1973),479-485.

[5] Manna,Z., and Waldinger,R., Towards Automatic Program Synthesis, CACM 14,3 (March 1971),151-165.

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

[7] Dahl,O-J.,Dijkstra,E., and Hoare,C.A.R., Structured Programming, Academic Press, London, (1972).

[8] Goldberg ,S. Introduction to Difference Equations,Wiley, New York, (1961).

[9] Aho,A.V. and Ullman,J.D., The Theory of Parsing, Translation, and Compiling,Vol.II, Prentice-Hall,Englewood Cliffs,N.J.,(1973).

Class No.	AUTHOR Standish, T.A.
List Price	TITLE THE IRVINE PROGRAM TRANSFORMATION CATALOGUE: A STOCK OF IDEAS FOR IMPROVING PROGRAMS USING SOURCE-TO-SOURCE TRANSFORMATIONS
Date Ordered	Edition or Series (Calif. Univ. Irvine. Dept. of Information & Computer Sci. Tech. report ; 161)
Date Rec'd. 9-30-85JM	Volumes
Dealer	Place Publisher Year Irvine : Dept. of Info. & Computer Sci., Univ. of Calif., 1976.
No. of Copies 2	Recommended by Fund Charged Cost Bk. Hr. from Serials
Order No.	HANDLING : 1 copy - stax 0-cui 1 copy - archives 0-pf
L.C. Card	