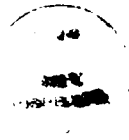
Radical Computing II

Saul Amarel
Curtis G. Callan, Jr.
Alvin M. Despain
Oscar S. Rothaus



June 1984

JSR-83-701

JASON
The MITRE Corporation
1820 Dolley Madison Boulevard
McLean, Virginia 22102

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER JSR-83-701	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Radical Computing II		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) S. Amarel, C. Callan, A. Despain, O. Rothaus		8. CONTRACT OR GRANT NUMBER(s) F19628-84-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation 1820 Dolley Madison Blvd. McLean, VA 22102		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE July 1984
		13. NUMBER OF PAGES 75
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report continues JASONs investigation of techniques that might be used to achieve a radical improvement in digital computer performance. The 1982 report investigated "residue arithemetic" and "symbolic computing;" this report extends the discussion of "symbolic computing" into "source program transformations" and a new topic "reversible computing."

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

"Acknowledgement"

We are grateful for discussions with Nils Nilsson of SRI and Edward Fredkin of M.I.T. Dr. Nilsson consulted with us on the artificial intelligence approaches to program transformations. Dr. Fredkin, who pioneered the reversible logic developments, provided extensive consultation in that area. We are also grateful to all our colleagues in JASON for inspiring discussions and helpful criticism. In particular, Peter Banks, Ken Case, Freeman Dyson, Doug Eardley, Paul Horowitz, Allen Peterson, William Press and John Vesecky all made contributions to this report.

TABLE OF CONTENTS

1. INTRODUCTION 1

1.1. History 1

1.2. Scope 1

1.3. Nature of Difficult Computing Problems 1

1.4. Limitations to High Performance 2

1.5. Possibilities for Radical Improvement 2

1.5.1. Smaller sized components 2

1.5.2. Transformation of programs 2

2. PROGRAM TRANSFORMATIONS 3

2.1. Motivations 3

2.1.1. Program development techniques 3

2.1.2. Performance 3

2.1.2.1. Algorithm improvement 4

2.1.2.2. Algorithm concurrency 4

2.2. Requirements for Transformation 4

2.2.1. Language 5

2.3. PROLOG 5

2.3.1. History 5

2.3.2. Analysis 6

2.3.3. PROLOG Structure 6

2.4. PROLOG Transformation Rules 8

2.5. Recurrences 12

8. APPENDIXES 59

1. INTRODUCTION

1.1. History

During the 1983 JASON summer study we continued our investigation of techniques that might be used to achieve a radical improvement in digital computer performance. In our previous study we investigated "residue arithmetic" and "symbolic computing"[1]. In this report we will discuss an extension of our "symbolic computing" into "source program transformations" and a new topic, "reversible computing".

1.2. Scope

The general goal of our earlier study was to identify new approaches for computer system development. The goal of this report is the same. We will try to identify critical mathematical and computer concepts that could lead to a radical increase in future computer performance, in calculating important, and currently difficult, problems.

1.3. Nature of Difficult Computing Problems

There is a set of difficult computing problems that have great economic importance. This domain is characterized by massive numerical calculations, symbolic calculations and search. For example, the design of a modern VLSI circuit involves symbolic calculations (calculus, etc.), numeric simulation of analog circuit properties, and search over a design space to find a near-optimal (or even sometimes just a feasible) solution. Today such problems are solved by a combination of human labor and machine calculations. Symbolic calculations are either done by hand or by use of a symbolic system such as MACSYMA[2]. Then these results are hand converted into a FORTRAN program, or parameters for a SPICE program[3] run. The results from the above analysis would then be examined by hand, and new computer runs would be made. Slowly, a design for a VLSI chip would evolve. A single design for a VLSI can cost tens of million of dollars for the human labor alone. The greatest difficulty in improving this process is the difficulty of implementing an automatic search that is efficient, i.e., not exhaustive. As we shall see, symbolic manipulation will aid the solution of this problem.

1.4. Limitations to High Performance

The primary factors that limit the performance of today's computers are:

(1) The speed of light limits how fast signals can be propagated throughout a computer.

(2) The serial nature of computer calculations limits parallel execution.

In the past, computer performance has been improved by improving the speed of the logic gates; this approach is becoming more difficult every year as integrated circuit techniques are maturing. Currently, the heat dissipation of logic circuits prevents them from being packed closer together, and the resulting separation causes, due to the finite speed of light, an inherent propagation delay that limits the performance of serial machines. To break this limitation, either a totally new technology of radically smaller dimensions and efficiency in power dissipation is required; and/or new organizational principles for parallel execution of computer algorithms are necessary.

1.5. Possibilities for Radical Improvement

1.5.1. Smaller sized components

To achieve a radical improvement then, we can seek radically smaller logical components with radically improved efficiencies. A speculative approach to this problem will be considered later in this report. If these components are sufficiently fast, then our current, serial designs for computers will suffice. If not, then concurrent execution techniques will be needed to achieve a radical improvement.

1.5.2. Transformation of programs

Since humans are not always good at expressing tasks in either efficient or concurrent form, we will need to develop techniques to transform source programs. This will require symbolic manipulation of source program fragments.

2. PROGRAM TRANSFORMATIONS

The main task of a compiler is to transform a source language into an object language. Improving the efficiency of the resulting program is the purpose of compiler optimizations. The ideas for this stem from compiler theory, structured programming[4], and artificial intelligence[5, 6]. Cocke and Allen[7] discuss about twenty transformations for compilers. The transformation of source programs into new source programs is not a new idea. It has been widely advocated as a method of developing programs, of improving the efficiency of programs and of discovering concurrency[8].

In the past, such efforts have only been partly successful. First, classic methods of optimization in compilers have been very successful and have relieved some of the pressure for source-to-source transformations. Second, classic source languages (such as FORTRAN) were ad-hoc designs and the corresponding algebra of the programs was extraordinarily difficult. Backus, the father of FORTRAN, has examined this problem and suggested some new directions[9]. The computer language 'fp' was a result of this effort. Finally, it has been only recently that an increasingly powerful drive to use parallel computers has existed. As a result, transformations to convert serial constructs into concurrent ones are becoming increasingly important.

2.1. Motivations

2.1.1. Program development techniques

The "Operational Program Development" technique[10] is a good example of a new method for developing software. The basic idea is to begin with a specification of a program that can be executed, at least at a high level. Then this specification is transformed into a complete, detailed, and efficient source-program. The proponents of this method claim it is a fast, inexpensive, and relatively error-free method of software engineering.

2.1.2. Performance

In this report, our primary focus is on performance. Techniques that could lead to a radical improvement are of especial interest. There are two approaches to this that we discuss next.

2.1.2.1. Algorithm Improvement

It is difficult (perhaps impossible) to prove that source-to-source transformations can always provide a radical improvement, even to programs that are very large and highly structured. However, some examples can provide evidence that such transformations have *potential* for a radical improvement on very large and difficult computing problems. For example, consider the discrete Fourier transform (DFT) of size N. It has a computational complexity of $0(N^2)$. This transform can be transformed (by humans) into the fast Fourier transform (FFT) of complexity $0(N\log n)$[11]. A typical value of N might be N = 1000, so that a hundred-fold improvement results. Such a transformation is currently beyond the capabilities of any automatic system, but as we shall see, such systems might be developed in the future.

2.1.2.2. Algorithm concurrency

Sometimes it is easy for a programmer to envision the potential concurrency in a program he is writing. The concurrency in a large matrix multiply is easy to see, for example. At other times it is very difficult to envision and express concurrency even when the programmer knows it must exist. For example, consider the addition of two very big numbers. We all have learned a *serial* algorithm to perform such additions. It is universally known among programmers that modern digital computers have *parallel* hardware to perform addition, yet almost no programmer could describe in detail the highly-concurrent algorithm that is embedded in the machine hardware. The serial carry operation for a 32-bit adder requires approximately 64 time steps, while the parallel (carry-lookahead) algorithm requires only about 10 time steps. Humans sometimes can articulate only the serial form of a calculation to be performed. Thus the need for automatic transformation of programs from serial to concurrent form.

2.2. Requirements for Transformation

There are a number of requirements of a language so that it will facilitate source-to-source transformations. The language should be at a high enough level that important structures of the original problem have not been lost. There needs to be a clean algebra of the language that

describes permissible transformations[9].

2.2.1. Language

In the past, the programming languages employed in the programs that were transformed generally included both popular languages such as FORTRAN where the need was great, and languages with special features that made them attractive for transformation, for example, LISP[12], APL[13], and SETL[14]. More recently, it has been suggested that program languages should be designed with transformation in mind. John Backus has been the main proponent of this and has proposed the language 'fp'[9]. The recently developed languages LUCID[15], KRC[16], and PROLOG[17] all show a regard for the "algebra" of the language. In addition, there have been attempts to improve the algebraic properties of popular languages; for example, Loveman's work with a FORTRAN-like language[18]. We will choose PROLOG for our work in this report.

2.3. PROLOG

2.3.1. History

Interest in the PROLOG language is growing very rapidly, roughly doubling every year. Recently, in Japan, it was adopted as the primary language for the "Fifth-Generation Computer" project. It is not yet very popular in the United States, probably due to the very mature programming environment surrounding the LISP language and also to "NIH" [1] factors. In the discussion that follows, it is assumed that the reader has some familiarity with the PROLOG language. A good tutorial on PROLOG by Clocksin and Mellish[19] is recommended for those readers not familiar with PROLOG. The basic idea for employing Predicate Calculus as a basis for a programming language can be credited to Kowalski[20] (England). Since the algebra of Predicate Calculus is especially well defined, the algebra of programming languages based on it are likely to also have an especially well defined algebra. Colmerauer, in France, adapted the theoretical ideas of Kowalski to a practical programming language, PROLOG[17].

[1] NIH signifies "Not Invented Here".

2.3.2. Analysis

An analysis of the PROLOG language will reveal several important features. First, the language is based on Predicate Calculus and inherits much of its formal structure. Second, any useful computer language must deal with 'side-effects', particularly data input and output. These 'side-effects' disturb the otherwise clean algebra of the language. Third, PROLOG generally subsumes the LISP language without any particular difficulty and without much violence to desirable algebraic properties. Fourth, the 'cut' operator of PROLOG, a difficult construct to handle in algebraic transformations, is used in several very distinct ways. The 'cut' does not modify the semantic of the constructs it appears in, provided they contain no 'side-effect' operators. Its most pervasive use is in situations where only one of several possible choices are to be made. This is the 'CASE' statement of more familiar languages. Unfortunately, 'cut' is also used for much less transparent constructs. The main use of the 'cut' is to improve the efficiency of the program by preventing useless calculations. The 'cut' operator, in general, causes grave problems during transformation of program fragments that contain it.

We have two reasons to employ transformations: First, we want to improve program efficiency, and second we want to increase program concurrency. Since "side-effects" generally inhibit concurrency, it is natural for us to partition a PROLOG program into parts that are separated by "side-effect" operations. Within the "side-effect"-less parts, we can freely apply a large repertoire of methods to achieve efficiency and concurrency. Unfortunately, we cannot use many of our transformation methods on those parts that contain i/o and other "side-effect" operations.

2.3.3. PROLOG Structure

The structure of a PROLOG program is illustrated in Figure 1. At the top level is the data base and the query. The data base is a collection of procedures. Procedures are collections of clauses, all of which have the same name. Clauses are Horn clauses from the predicate calculus, sometimes augmented with 'cut' operators. There are two kinds of clauses, 'facts' and 'rules'. A fact has only a head consisting of a predicate name and any arguments surrounded by

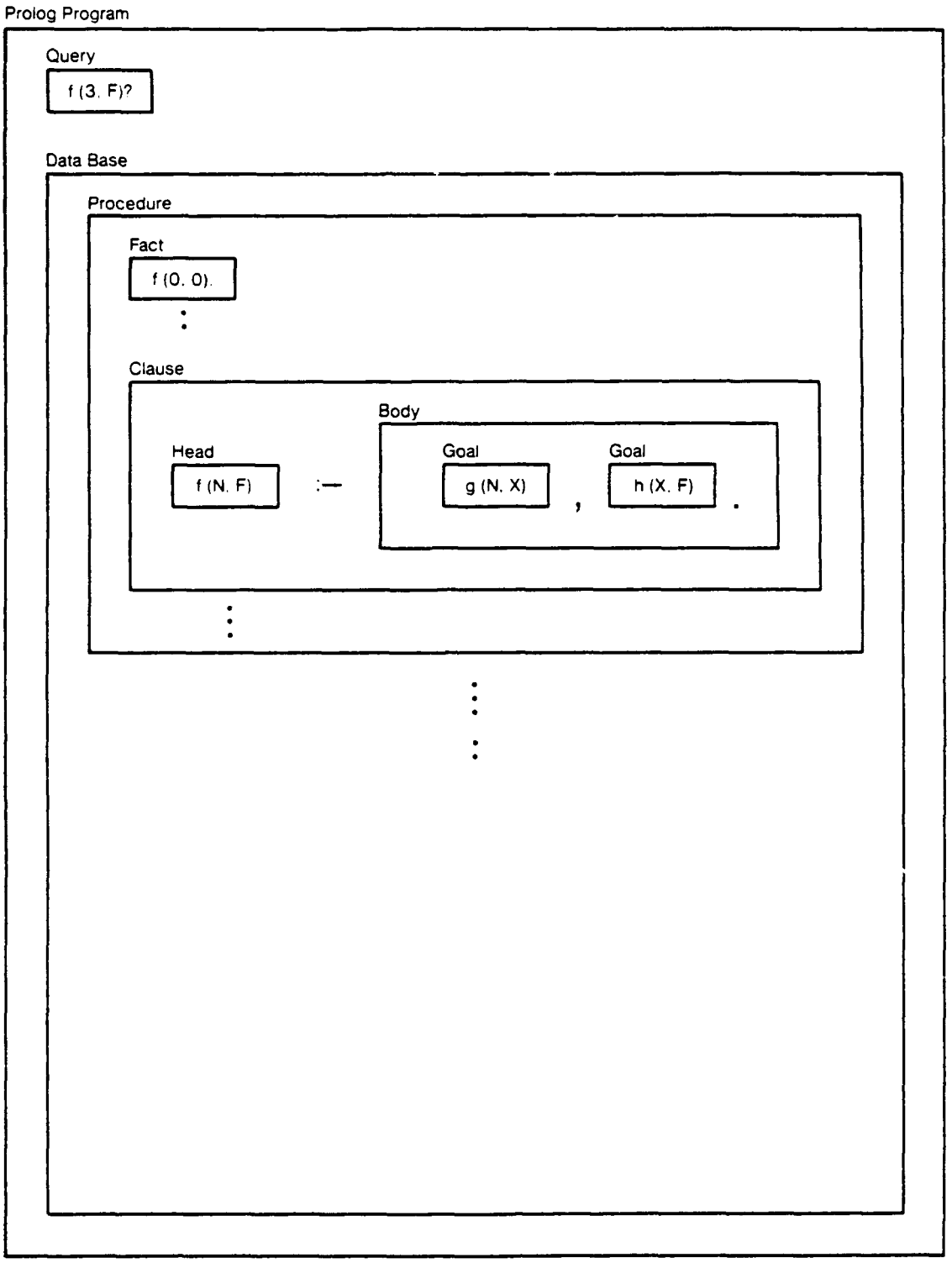


Figure 1. Prolog program structure.

parentheses. For example, 'f(0,0).', is a fact. A rule is a 'head', as described above, connected to a 'body' by the IF operator ':-'. The body is a collection of 'goals', each of which has the syntax of a 'head'. Thus, for example,

f(A,B) :- g(A,X), h(X,B).

is a rule that is read as "Function *f* with argument *A* and *B* will be *true IF* function *g* of arguments *A* and *X* is *true AND* if function h with arguments *X* and *B* is *true* ." The variable arguments, indicated by an initial letter that is upper case, are all assumed to be universally quantified.

Rules can be recursive as in the following function to sum the positive integers up to N.

f(0,0).

s(N,S) :- M is N-1, s(M,G), S is N+ G.

PROLOG constructs can be related to concepts of conventional programming languages as follows in Table 1.

Table 1. Comparison of Programming Languages.

PROLOG	CONVENTIONAL LANGUAGES
cut	if-then-else; case
goal	procedure call
clause	entry point of a procedure
unification	assignment, data selector and constructor
recursion	iteration and recursion

The special form of PROLOG constructs are especially helpful in the program transformations to be discussed below.

2.4. PROLOG Transformation Rules

PROLOG supports several forms of mathematics. We have previously mentioned that predicate calculus is the basic form of the language. Table 2 illustrates the basic definition of the predicate functions and the corresponding PROLOG functions.

Table 2. Definition of Predicate Functions.						
Predicate Calculus	X	Y	X ^ Y	X v Y	X ⊃ Y	~X
PROLOG	x	y	x,y	x;y	y:-x	not(x)
Values	false* false true true	false true false true	false false false true	false true true true	true true false true	true true false false

*In PROLOG, 'false' is replaced by 'fail'.

In predicate calculus there are a large set of transformation rules that can be directly adapted to transform PROLOG fragments. Table 3 summarizes all these transformations,

Table 3. Summary of PROLOG Transformations.		
Type	Classification	Components
Basic	Definitions	not, and, or, implies
	Propositional Calculus	association, commutation, distribution, contraposition, DeMorganization, negation
	Resolvents	modus ponens, merge, tautology, chaining, equivalence
Derived	Recurrences	head, tail, mixed, multiple
	Derivations	chaining
	In-line	expansions, contractions
	Caching	
	Partial Compilation	

and Table 4

Table 4. Basic Transformations.

NAME	Predicate Calculus			PROLOG (no side-effects)		
Definition	X ⊃ Y	<=>	~X v Y	y:-x.	<=>	
	~(~Z)	<=>	Z	t:-not(z). w:-not(t).	<=>	w:-z.
Association	(X^Y)^Z	<=>	X^(Y^Z)	w:-x,y,z.	<=>	w:-y,z,x.
	(XvY)vZ	<=>	Xv(YvZ)	v:-x. v:-y. v:-z.	<=>	v:-y. v:-z. v:-x.
Distribution	X^(YvZ)	<=>	(X^Y)v(X^Z)	w:-y. w:-z. u:-x,w.	<=>	r:-x,y. s:-x,z. u:-r. u:-s.
	Xv (Y^Z)	<=>	(XvY)^(XvZ)	u:-x. u:-y,z.	<=>	r:-x. r:-y. s:-x. s:-z. u:-r,s.
Commutation	X^Y	<=>	Y^X	u:-x,y.	<=>	u:-y,x.
	XvY	<=>	YvX	u:-x. u:-y.	<=>	u:-y. u:-x.
DeMorganization	~(X^Y)	<=>	~X v ~Y	u:-x,y. v:-not(u).	<=>	u:-not(x). v:-not(y).
	~(XvY)	<=>	~X^~Y	u:-x. u:-y. v:-not(u).	<=>	v:-not(x),not(y).
Contrapositive	X ⊃ Y	<=>	~Y ⊃ ~X	y:-x. u:-not(x).	<=>	y:-x. z:-not(y). u:-z.
Modus Ponens	P^(~PvQ)	<=>	Q	u:-not(p). u:-q. v:-p,u.	<=>	v:-q.
Merge	(PvQ)^(~P vQ)	<=>	Q	u:-p. u:-q. v:-not(p). v:-q. r:-u,v.	<=>	r:-q.
Tautology	(PvQ)^(~PvQ)	<=>	(Q v ~Q)	u:-p,q. v:-not(p),not(q).	<=>	r.
	(PvQ)^(P v ~Q)	<=>	(P v ~P)	r:-u. r:-v.		
Nil	~P^P	<=>	nil	r:-p,not(p).	<=>	r:-fail.
Chaining	P⊃Q^Q⊃R	<=>	P⊃R	q:-p. r:-q.	<=>	r:-p.

and Table 5 provide more detail.

Table 5. PROLOG Derived Transformations (Examples).

NAME	Transformation		
Chaining	x:-ax. y:-ay. u:-x,y.	<=>	u:-ax,ay.
RECURRENCES head	r(0). r(A):-r(B),f(A,B),g(A,B).	<=>	r(0). r(A):-f(A,B), r(B), g(A,B).
mixed	r(0) r(A):-f(A,B), r(B), g(A,B).	<=>	r(0). r(A):-f(A,B), g(A,B), r(B).
tail	r(0). r(A):-f(A,B), g(A,B), r(B).	<=>	r(0). r(A):-r(B),f(A,B),g(A,B).
multiple <=> single	r(0). r(1). r(A):-f(A,B,), r(A), r(B).	<=>	s(0,1). s(A,B):-f(A,B), s(A,B). r(A):-s(A,B).
In-line	f. r:-a,f,z.	<=>	r:-a,z.
Expansion & Contraction	r:-a,s,t,z.	<=>	f:-s,t. r:-a,f,z.
Caching	a(0). a(1). b(x):-a(y), x is 2*y. b(0)? b(1)?	<=>	a(0). a(1). b(0). b(1):-fail. b(x):-a(y), x is 2*y.
Partial Compilation	a(0). a(1). b(x):-a(y), x is 2*y.	<=>	a(0). a(1). b(0). b(2).

For example, in Predicate calculus if a $\supset$ b (read this as a implies b) and b $\supset$ c , then it can be concluded that a $\supset$ c. Similarly, the PROLOG fragment

b :- a.

c :- b.

can be transformed (if there is no other use of b) into the simplified fragment:

c :- a.

PROLOG also supports the algebra (and arithmetic) of the reals (both integer and floating point) within a goal. Thus a goal can be an arithmetic construction such as "Sum is A + 3 * B". In

the usual algebraic way, fragments such as

... , B is A + 1, sum is A + 3 * B, ...

can be simplified (if there is no other use of B) into

... , sum is 4 * A + 3, ...

2.5. Recurrences

Recurrences can appear in three forms, as in the following function to sum the positive integers up to N. This is a mixed recurrence.

```
s(0,0).
s(N,S) :- M is N-1, s(M,G), S is N+ G.
```

Tail recursion is an especially desirable form because it is very efficient in terms of use for memory. It is equivalent to a 'DO' loop in FORTRAN. An example of tail recursion for the same sum-the-positive-integers task is;

```
s(N,S) :- s(N,O,S).
s(O,S,S).
s(N,A,S) :- M is N-1, B is A+ N, s(M,B,S).
```

Both of these recursions have roughly the same number of execution steps; however, the tail recursion has need for much less memory (space) and so is, all other factors being equal, more desirable than the first form. Note however the cost. The first form is a bit more compact and is easier to comprehend.

2.5.1.1. Head Recursion

Above we discussed general (or 'mixed') recursion and tail-recursion. A form we will name 'head-recursion' will also be important for our transformations. Consider our previous mixed-

recursion example of summing the possible integers up to N. By transforming the first goal, (an arithmetic statement) into N is M + 1, and re-arranging the goals according to the predicate calculus commutative rule we obtain our 'head-recursion' form.

```
s(0,0).
s(N,S) :- S(M,G), N is M + 1, S is N + G.
```

Because of the default computation rule employed by the PROLOG evaluator, this is not an efficient form but does, of course, have the same semantics as the original form. This 'head-recursion' form will be important in later transformation examples.

2.6. Comments

These three forms illustrate the goal of our transformation method. We seek transformation algorithms to convert at will between these forms.

3. EXAMPLES

In this chapter we will examine five different transformation problems. Each represents an important process for achieving efficiencies and/or concurrency.

3.1. Triple Append

This problem, in a LISP environment, has been examined by Wegbreit[21]. Because of the differences between LISP and PROLOG, it is instructive to see how the PROLOG version differs from the LISP one. The problem is to join together three lists, using the usual algorithm for joining two lists. Then the result is transformed into a more efficient form. The definition for appending one list to another is:

append ([],Z,Z). append $([X_H \mid X_T], Y, [X_H \mid Z_T])$:- append (X_T, Y, Z_T) . [2]

In order to more easily manipulate our programs, we will use an abbreviated form as follows.

$$a([],Z,Z).a([X_H \mid X_T],Y,[X_H \mid Z_T]):-a(X_{T,Y,Z_T}).$$

The result is that list Z is the list Y appended to list X. To join three lists A,B,C into the single list D, we employ the above PROLOG procedure and define our three list append as:

$$b(A,B,C,D):-a(A,B,E),a(E,C,D).$$

Now this is a perfectly acceptable program. It can be improved however. Notice that first list A must be traversed in order to append list B. Then in the next goal, list E, composed of lists A and B must be traversed to append list C. The cost is then $2l_A + l_B$ procedure calls to a, where l_x is the length of list X. This cost might be reduced to $l_A + l_B$ by program transformation. To do this consider two cases:

Case 1 A = [].

List A is null.

[2] In PROLOG, given a list, the notation [H|T] indicates that H is the first item of the list and T is the remainder.

b([],B,C,D) :- a([],B,E), a(E,C,D).

but a([],B,E) :- true if E + B. (by partial evaluation)

Thus

b([],B,C,D) :- a(B,C,D).

Case 2 $A \neq []$

$$b([A_H \mid A_T],B,C,[D_H \mid D_T]) :- a([A_H \mid A_T],B,[E_H \mid E_T]), a([E_H \mid E_T],C,[D_H \mid D_T])$$

but a

$$([A_H \mid A_T],B,[E_H \mid E_T]) :- a(A_T,B,E_T) IF E_H = A_H$$

(by partial evaluation). Thus

$$b([A_H \mid A_T],B,C,[A_H \mid A_H \mid D_T]) :- a(A_T,E,E_T), a(E_T,C,D_T).$$

Now by the derived-chaining rule, the r.h.s. is defined by the original definition of b;

$$b([A_H \mid a_T],B,C,[A_H \mid D_T]) :- b(A_T,B,C,D_T).$$

This new procedure is thus

$$b([],B,C,D) :- a(B,D,D).$$

$$b([A_H \mid A_T],B,C,[A_H \mid D_T]) :- b(A_T,B,C,D_T).$$

This is more efficient. It calls itself l_A times and calls a, l_B times, a saving of l_A calls.

3.2. Fibonacci Recurrence

In the next example, we define a procedure to calculate the N^{th} value of the Fibonacci sequence. The Fibonacci sequence is

$$1 , 1 , 2 , 3 , 5 , 8 , ...$$

when $f_i \leftarrow f_{i-1} + f_{i-2}$.

In PROLOG this is

fib(0,1).

fib(1,1).

fib(N,F) :- M is N-1, fib(M,G), L is N-2, fib(L,H), F is G+ H.

This is very inefficient. The separate calculations of fib on the r.h.s. often calculate the same values. It has a complexity of $0(2^N)$. We will transform it to a more efficient form. For notational purposes we will compress the above procedure as follows.

f(0,1).

f(1,1).

f(N,G+ H) :- f(N-1,G), f(N-2,H).

Now let us define a new clause to represent the r.h.s.

g(N-1,G,N-2,H) :- f(N-1,G,), f(N-2,H).

thus

g(N,F,N-1,G) :- f(N,F), f(N-1,G).

f(N,G+ H) :- g(N-1,G N-2,H).

and

g(1,1,1-1,1) :- f(1,1), f(0,1).

now since

f(N-1,G) :- f(N-1,G).

f(N-1,F) :- f(N-1,G),f(N-2,H) F is G+ H,

then by the derived-chaining rule,

f(N,F), f(N-1,G) :- f(N-1, G), f(N-2,H) F is G+ H

thus

g(N,F,N-1,G) :- g(N-1,G,N-2,H), F is G+ H.

In canonical form:

g(1,1,0,1).

g(N,F,M,G) :- M is N-1, L is M-1, g(M,G,L,H), F is G+ H.

Note that L now has no particular function.

Thus

g(1,1,1).

g(N,F,G) :- M is N-1, g(M,G,H), F is G+ H.

This is a much improved algorithm, with a complexity of 0(N). Thus it has only about N calls, far fewer than the number of calls of the original algorithm. It is possible to find an algorithm that is of only $0(log_2 N)$ complexity as opposed to the $0(N)$ complexity of our present algorithm. It turns out that for large N (approximately N = 50) this new algorithm is an improvement. The basic idea of the new algorithm is to turn the 'mixed-recurrence' into a 'head-recurrence' and solve it using matrix techniques. The improved recurrence derived above is:

g(M,F,G) :- L is M-1, g(L,G,H), F is G+ H.

This is equivalent to:

g(1,M,F,G) :- L is M-1, g(1,L,G,H), F is G+ H.

g(1,M,F,G) :- g(1,L,G,H), M is L+ 1, F is G+ H.

g(1,M,F,E) :- g(1,L,G,H), M is L+ 1, F is G+ H, E is G.

```
g(1,M,F,E) :- g(1,L,G,H),
              1 is 1,
              M is L+ 1,
              F is G+ H,
              E is G.
```

We now solve for the initial condition:

g(1,1,1,1).

Now, converting to matrix form:

$$g(1,1,1,1)^M = \left[\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}\right]^M \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The matrix can be expanded by calculating successive squarings with only (log M) complexity.

3.3. Arithmetic Series

This example illustrates the power that transform techniques can sometimes achieve. Consider a general arithmetic series involving polynomials. Such a form often appears in the 'do' loops of FORTRAN programs. An example as it would appear in a language similar to FORTRAN is:

```
f(0) = 0.
do ( n <- 1,N )
 {f(n)<-f(n-1)+ n**4-4*n**3+ 3*n**2-2*n+ 1}.
```

This can be expressed in PROLOG as:

```
f(0,0).
f(N,F) :- M is N - 1,
        f(M, F1),
        F is F1 +  N*N*N*N - 4*N*N*N +  3*N*N - 2*N +  1.
```

Note that the complexity of this program is 15N arithmetic operations.

It is possible to automatically transform any such arithmetic series by means of a PROLOG program written by Peter Van Roy (unpublished). This program is provided in Appendix A-1.

The improved form for f as automatically generated by Van Roy's program is:

```
f(_3,_6):-_6 is
((((-6*_3+ 15)*_3+ 20)*_3+ 15)*_3+ -14)*_3/-30.
```

In more readable form this is:

```
f(N,F):- F is
```

((((-6*N+ 15)*N+ 20)*N+ 15)*N-14)*N/-30.

Now note that this solution is *much* simplified. Its complexity is only 10 arithmetic operations, independent of the value of N.

3.4. Head Caching

In a simple PROLOG interpreter, once a goal fails, all the context of its sub-goals, both those that failed and those that succeeded, is destroyed by popping both the environment and the value stacks.

Later, if these same sub-goals are encountered, all the work of proving them must be repeated. This work could be avoided by storing the heads of the clauses and the result of the evaluation. In effect we wish to add a new clause to the data base.

It can be seen that this technique enhances performance at the cost of memory space (to store the cache entries). The modern trend seems to be ever decreasing costs for memory, so this may be an attractive method to improve performance for some applications that are more time-bound than space-bound.

This technique is not a general one that should be universally applied. For example if a procedure is being used as a generator of values, it is not appropriate to cache its intermediate results because they could be scrambled (in order of appearance) by caching. Also, if side-effect operations occur, such as assert or read, then caching can change the expected behavior of the executing program. It is also true that some cached values are much more valuable than others, so some selectivity in caching is desirable to optimally utilize memory space.

Our method of overcoming these drawbacks is to add a new mode declaration to the PROLOG interpreter language. Warren[22] used this method when he introduced his operator "mode". Clark and McCabe [23] elaborated on this and introduced several more mode operators. We propose a "cache" mode operator to declare the desirability of caching a named procedure. Thus if we wished to cache the results of the "ancestor" procedure, we would include "cache (ancestors)." in the program.

As a result, the programmer can select only those procedures for caching, that are likely to be greatly improved by caching. Procedures at too low a level to benefit from caching, generator procedures or procedures with side-effects can be avoided (in the most natural way) for caching purposes.

In order to illustrate the potential of this method, we will examine expand an example program, a procedure to determine if two people are related. It is:

```
related (X,Y):- ancestor (X,Y).
related (X,Y):- ancestor (Y,X).
related (X,Y):- ancestor (X,Z), ancestor (Y,Z).
```

Thus two people, X and Y are related if one is the ancestor of the other or if they have a common ancestor, Z. In order to compress the bulk of the programs to follow, we will reduce all the names in our example to simple letters, the first letter of the name. The compressed program follows where m represents male, c represents child, and f represents father. The letters t,j,g,b,v, all represent individual people.

```
m(t).
m(j).
m(g).
m(b).
c(v,g).
c(g,b).
c(j,g).
c(t,j).
f(X,D):-m(D),c(X,D).
a(A,X):-f(A,X).
```

```
a(A,X):-f(Y,X),a(A,Y).

r(X,Y):-a(X,Y).

r(X,Y):-a(Y,X).

r(X,Y):-a(X,Z),a(Y,Z).
```

As a measure of performance we will count calls to clauses. This is roughly proportioned to the number of logical inferences (LI)[3] since for our example there are about two logical inferences (LI) per clause.

For purposes of illustration, we will be interested in a compound query: 'Is t related to v?' followed by: 'Is v related to t?'. We wish to know the number of calls required to answer this query.

It is a simple matter to instrument our example program and count each call. We define a count procedure 'cnt (x)' and call it just as we start the execution of the body of the clause (See Appendix A-2). A much more elaborate version of this will be discussed later. When the query is then executed, the first half of the query 'r(t,v)?' requires 59 calls and the second, 'r(v,t)?', 35 calls for 94 total.

How much could caching reduce this figure? One way to find out would be to re-write a PROLOG interpreter to include the "cache" operator as discussed above. In this study however, we re-wrote the example program to call a simulated cache system. One might imagine that each of the statements was specified to be cached, and the 'logical' consequence is the re-written program.

This system was implemented to produce the measurements of the numbers of calls and a trace during execution of queries. The simulated cache was written in PROLOG and our measurements were made with a conventional PROLOG interpreter. The re-written example is:

---

[3] LI for Logical Inferences seems to have become the accepted measure of work in executing logic programs. The more common form is "LIPS" for "Logical Inferences per Second". We assume an LI is the unification of a simple variable.

```
m(t):- cnt(m).
m(j):- cnt(m).
m(g):- cnt(m).
m(b):- cnt(m).
c(v,g):- cnt(c).
c(g,b):- cnt(c).
c(j,g):- cnt(c).
c(t,j):- cnt(c).
f(X,D):- hf(f,X,XP,D,DP), m(D),c(X,D),   tf(f,X,XP,D,DP).
a(A,X):- hs(a,A,AP,X,XP), f(A,X),        ts(a,A,AP,X,XP).
a(A,X):- hf(a,A,AP,X,XP), f(Y,X),a(A,Y), tf(a,A,AP,X,XP).
r(X,Y):- hs(r,X,XP,Y,YP), a(X,Y),        ts(r,X,XP,Y,YP).
r(X,Y):- hs(r,X,XP,Y,YP), a(Y,X),        ts(r,X,XP,Y,YP).
r(X,Y):- hf(r,X,XP,Y,YP), a(X,Z),a(Y,Z), tf(r,X,XP,Y,YP).

cache(f).
cache(a).
cache(r).
```

The cache simulation program can be found in Appendix A-2.

There are four kinds of calls to the cache system. These are hs,hf,ts,tf. For a procedure with only a simple clause such as 'f', we employ 'hf' and 'tf'. The first, 'hf', creates a cache entry that represents the failure of this clause with the variable bound as the original clause was called. If the clause indeed fails, then nothing further happens and the cache entry remains. On the other hand, if the clause should succeed, then this clause entry must be replaced by a entry representing success, but with the new binding determined by the body of the clause. This is the function of

'tf' which appears at the end of each clause. If backtracking within the clause occurs, then each successful result must also be entered into the cache.

For multiple clause procedures, only the last clause can indicate a failure, so there is no caching at the head of any clause except the last clause of a procedure. The call 'hs' is just used for instrumentation.

The call 'ts' is similar to 'tf' but since no "fail" was entered into the cache for this clause none should be extracted.

The 'primed' variables (XP,ZP) that appear in the cache calls are needed because the variable bindings cached by the "fail" at the beginning of a clause are not the same as those cached by the "succeed" at the end. Thus to remove a previously cached "fail", those bindings must be propagated from 'hf' to 'tf'.

In inserting a new entry to the cache, duplicates (if any) should be removed. Also if a more general result is cached, and subservient ones should be deleted. For example if a(i,t):-fail. is initially in the cache when a(X,t):-fail is to be cached, a(i,t):-fail should be removed as it is dominated by a(X,t):-fail.

The cache program that accomplishes the above objections is shown in Appendix A-2.

The performance results of our example with the cache system, are shown in Table 6.

Table 6. Summary of Results.

INITIAL CALL	TOTAL CALLS		
	No Cache	CACHE	
		r(t, v), r(v, t)	r(v, t), r(t, v)
r(v, t)	35	5	32
r(t, v)	59	47	5
Both	94	52	37

Only 37 total calls are required in the cached system as compared to the 94 required in the uncached system. It is interesting to note that if r(v,t) is called before r(t,v), then 52 calls are needed. The cache scheme clearly saves calls in this simple example.

The state of the program after the query is shown below.

EXECUTED PROGRAM LISTING

```
m(t) :-
   cnt(m).
m(j) :-
   cnt(m).
m(g) :-
   cnt(m).
m(b) :-
   cnt(m).

c(v,g) :-
   cnt(c).
c(g,b) :-
   cnt(c).
c(j,g) :-
   cnt(c).
c(t,j) :-
   cnt(c).

f(j,g) :-
   cnt(f).
f(v,g) :-
   cnt(f).
f(t,j) :-
```

```
    cnt(f).
f(_1512,_1513) :-
    eq(_1512,t),
    eq(_1513,v),
    !,
    fail.
f(_1512,_1513) :-
    eq(_1512,_1522),
    eq(_1513,t),
    !,
    fail.
f(_1512,_1513) :-
    eq(_1512,v),
    eq(_1513,j),
    !,
    fail.
f(_1512,_1513) :-
    eq(_1512,_1522),
    eq(_1513,v),
    !,
    fail.
f(_1512,_1513) :-
    hf(f,_1512,_1522,_1513,_1523),
    m(_1513),
    c(_1512,_1513),
    tf(f,_1512,_1522,_1513,_1523).

a(v,g) :-
```

```
   cnt(a).
a(t,g) :-
   cnt(a).
a(t,j) :-
   cnt(a).
a(_1516,_1517) :-
   eq(_1516,t),
   eq(_1517,v),
   !,
   fail.
a(_1516,_1517) :-
   eq(_1516,v),
   eq(_1517,j),
   !,
   fail.
a(_1516,_1517) :-
   eq(_1516,v),
   eq(_1517,t),
   !,
   fail.
a(_1516,_1517) :-
   hs(a,_1516,_1526,_1517,_1527),
   f(_1516,_1517),
   ts(a,_1516,_1526,_1517,_1527).
a(_1516,_1517) :-
   hf(a,_1516,_1526,_1517,_1527),
   f(_1528,_1517),
   a(_1516,_1528),
```

```
   tf(a,_1516,_1526,_1517,_1527).

r(v,t) :-
   cnt(r).
r(t,v) :-
   cnt(r).
r(_1520,_1521) :-
   hs(r,_1520,_1530,_1521,_1531),
   a(_1520,_1521),
   ts(r,_1520,_1530,_1521,_1531).
r(_1520,_1521) :-
   hs(r,_1520,_1530,_1521,_1531),
   a(_1521,_1520),
   ts(r,_1520,_1530,_1521,_1531).
r(_1520,_1521) :-
   hf(r,_1520,_1530,_1521,_1531),
   a(_1520,_1532),
   a(_1521,_1532),
   tf(r,_1520,_1530,_1521,_1531).

count(level,2).
count(a,2).
count(r,3).
count(_1522,0).
```

CALL COUNTS

Total calls = 5

Notice how it has been transformed. Both 'success-goals' and 'failure-goals', are evident, as are the original clauses. This idea of transforming a PROLOG program leads us to another related technique.

3.5. Partial Compilation

If all possible results of executing a procedure, called with all of its argument unbound, are cached, then in some sense we have transformed a procedure into a 'partially-compiled' form that executes particular queries very quickly. It may, of course, use enormous memory space. Again, selective programmer control of such a facility could be effective. Thus we propose the mode "pcompile(Procedure-name)."

To illustrate the potential of this technique, we will include the statement "compile (a)" with the example program, and the following procedures with the cache program.

```
% Partial-compiler for PROLOG Example.

% head(X,Y,Z) :- X(Y,Z).
% compensation for the principle
% functor not being a variable

head(f,Y,Z) :- f(Y,Z).
head(a,Y,Z) :- a(Y,Z).
head(r,Y,Z) :- r(Y,Z).

head(fp,Y,Z) :- fp(Y,Z).
head(ap,Y,Z) :- ap(Y,Z).
head(rp,Y,Z) :- rp(Y,Z).

% compiler
p_compile(all) :- pcompile(X),
```

```
        p_compile(X),fail.

p_compile(X) :- head(X,Y,Z), pexec(X,Y,Z),fail.

p_compile(X) :- abolish(X,2),restore(X),!.

pexec(X,Y,Z) :- trans(X,XP),
        ycache(XP,Y,Z),!.

restore(X):-trans(X,XP),rts(XP,Y,Z),
        ass(X,Y,Z),restore(X).
restore(X).

trans(f,fp).
trans(a,ap).
trans(r,rp).

ass(fp,YQ,ZQ):- asserta((fp(YQ,ZQ))).
ass(ap,YQ,ZQ):- asserta((ap(YQ,ZQ))).
ass(rp,YQ,ZQ):- asserta((rp(YQ,ZQ))).

rts(fp,YQ,ZQ):- retract((fp(YQ,ZQ))).
rts(ap,YQ,ZQ):- retract((ap(YQ,ZQ))).
rts(rp,YQ,ZQ):- retract((rp(YQ,ZQ))).
```

Execution of the compiler results in the following transformed program. Not counting the compiler itself but only the original and program, 72 calls are required during the compilation.

EXECUTED PROGRAM LISTING

```
m(t) :-
   cnt(m).
m(j) :-
   cnt(m).
m(g) :-
   cnt(m).
m(b) :-
   cnt(m).

c(v,g) :-
   cnt(c).
c(g,b) :-
   cnt(c).
c(j,g) :-
   cnt(c).
c(t,j) :-
   cnt(c).

f(_41,_42) :-
   hf(f,_41,_51,_42,_52),
   m(_42),
   c(_41,_42),
   tf(f,_41,_51,_42,_52).

a(t,j) :-
   cnt(a).
a(v,g) :-
   cnt(a).
a(j,g) :-
```

```
   cnt(a).
a(g,b) :-
   cnt(a).
a(t,g) :-
   cnt(a).
a(v,b) :-
   cnt(a).
a(j,b) :-
   cnt(a).
a(t,b) :-
   cnt(a).

r(_49,_50) :-
   hs(r,_49,_59,_50,_60),
   a(_49,_50),
   ts(r,_49,_59,_50,_60).
r(_49,_50) :-
   hs(r,_49,_59,_50,_60),
   a(_50,_49),
   ts(r,_49,_59,_50,_60).
r(_49,_50) :-
   hf(r,_49,_59,_50,_60),
   a(_49,_61),
   a(_50,_61),
   tf(r,_49,_59,_50,_60).

count(m,20).
count(level,7).
count(f,18).
```

```
count(a,18).
count(c,16).
count(_51,0).
count(_51,0).
count(_51,0).
count(_51,0).
count(_51,0).
CALL COUNTS

Total calls = 72
```

Now the compound query requires (without further caching) 11 calls as compared to the 94 original calls. Note that even for this query, fewer total calls are needed (72 + 11 = 83).

It is a waste of effort to also compile 'f' once 'a' is compiled. Some speed up (from 11 to 2 calls) could result if 'r' were compiled, but this would cost considerable space for little gain. It can be seen that selective pseudo-compiling can sometimes be very helpful in improving performance.

For similar reasons, employing caching after compiling 'a' would not help performance.

3.6. Comments

The above examples illustrate the potential of the transformation techniques. However, all of the examples were quite simple and half of the examples were transformed by hand, not automatically. The problem of automatically controlling which transformations should be applied is a very difficult open problem.

A future goal is the automatic transformation of the DFT algorithm into the FFT algorithm. Another is the development of the Strassen algorithm[24]. Both of these problems have a common background; roughly speaking, they both appear to be connected to certain questions in

the theory of representations of finite groups, a fairly well developed body of mathematical knowledge. To solve these and related transformation problems, a program, similar to the Van Roy solver program discussed above, would need to be developed. Such a program would have to know not only all the facts about group representations but would also have to be able to sense that this particular area held some facts and techniques which might be relevant to the problem at hand. Such a program is likely to be very complex. It is not at all clear that it could be developed as an expert (mathematician) system, even if massive resources could be provided. It may even be the case that such a program could not be developed without some new breakthrough in the theory of artificial intelligence, or the development of some new kind of mathematics.

On the other hand, it is possible that a clever mathematician or computer scientist just might discover a new approach. Such a discovery could have tremendous consequences for high performance computing.

3.7. Conclusions for Transformations

The above examples achieved performance enhancements ranging from speed-ups of 1.5 to 3000, on very simple problems. However, it is true that these examples constitute plausibility arguments, not proofs, that transformation techniques may be important for achieving a radical improvement in performance. To have a big impact, such transformations would need to be automatically controlled during program execution, so that as more elements of the solution are developed, new transformations can be applied. There is currently very little theory to guide such dynamic applications of transformations. If future research is able to accomplish this, then a radical improvement in performance could indeed occur.

4. REVERSIBLE COMPUTING

4.1. Introduction

Reversible computing was investigated in an attempt to determine if radically smaller and more efficient logical circuits might be possible. As we shall see, density and heat dissipation improvements up to $\approx 10^6$ may someday be possible. We consulted one of the pioneers in this subject, Dr. Edward Fredkin of MIT, at some length and brought ourselves up to date on the (not very extensive) literature. Our basic conclusion is that the importance of reversible logic depends crucially on the physical architecture of the computer: It is irrelevant to the current scheme in which packets of charge are stored on, and moved between, structures of order one light wavelength in size, but might be relevant and even essential if the basic information-handling units were of molecular or atomic size (a distant but not necessarily unattainable goal). The question of physical realization of reversible logic elements has been almost completely neglected [4] in favor of the abstract questions of how, given the existence of reversible logic elements, one could wire them up to make a useful computer and how one would program it. We think that the problem of how to physically realize reversible computation at something like the atomic scale should be the next question to be attacked in this area. We also think that the very framework of reversible logic suggests some interesting new approaches to the problem of ultra-small-size computing elements which might be worth exploring for their own sake. Although practical payoff on any of these ideas is surely far off, the computer science and physics issues raised are fascinating and of fundamental importance.

4.2. Energy Dissipation in Computing

Contemporary computers dissipate at least 10^{-12} joules (about 10^8 kT if T equals room temperature) per logical operation. The reason is that bits are stored as charges on capacitors charged to about one volt (the typical operating voltage of solid state electronic devices). Since there is a lower limit to the size and capacitance of circuit elements that can be fabricated on a

[4] apart from some interesting "existence proof" work of Fredkin et. al.

chip using optical techniques, there is a lower limit to the energy associated with storing one bit. That limit turns out to be the above mentioned 10^{-12} joules, and the current style of computer logic causes that entire energy to be dissipated each time the state of a bit is changed [25]. The resulting heat load is a major barrier to high speed computation. A major question is the extent to which this dissipation is an inescapable concomitance of computation and to what extent it is due to "inefficient" physical or logical design of the computer[26]. Information theoretic/thermodynamic arguments have been used to suggest that there is a fundamental dissipation limit of kT per operation for computers designed on current principles.

In thermodynamics there is a well-known connection between dissipation and the reversible operation of heat engines. Standard computer logic elements, the NAND gate in particular, are not even reversible as abstract logical operations, let alone as physical devices. It has been suggested that if reversible logic functions are used, it is in principle possible to do computing with zero dissipation[27, 28]! In this scenario, the entire computing operation would have to be carried out reversibly in analogy with the dissipationless operation of a reversible heat engine. It is hard to evaluate the relative merits of two schemes which promise to reduce dissipation to $0*kT$ (the demand limit for reversible logic) and $1*kT$ (the demand limit for standard logic) per operation, respectively, when the best dissipation achieved to date is 10^8 kT! We think it is worthwhile to pursue the reversible logic scenario, not so much because it promises superior practical benefits, but rather because it raises unfamiliar questions about the nature of computing and suggests some interesting new approaches to the physical realization of computation.

There are two types of questions which arise when you pursue this line. First, there is the question of what are useful reversible-logic functions, how they might be tied together to make a useful computer and how such a computer might be programmed. These questions are all answerable in the abstract, without any reference to the physical realization of the system. This sort of question is the major subject of the work of Fredkin and other pioneers in reversible logic and the results are that manageable reversible-logic computers can be designed although they are in many interesting ways different from conventional computers. The second question has to do with physical realization of reversible computation: What sort of physical system can be used, what

calculation speeds can be achieved, etc? Here very little is known, although many interesting questions arise. We think this is the most important aspect of the reversible computation problem and have attempted to construct a framework for a serious exploration of these questions.

4.3. Physical Realizations of Computers

To establish a useful framework for our discussion it is helpful to remark that there are at least two broad classes of physical realizations of computing machines. The most important distinction is between open (dissipative) systems and closed (conservative) systems. The distinction is between systems in which the computational degrees of freedom are coupled to a "heat bath" with which energy can be exchanged and systems in which the computational degrees of freedom are effectively isolated from the rest of the world. The other essential distinction is between systems in which the computational degrees of freedom can be described classically versus those in which they must be described quantum mechanically.

A dissipative system will behave in many respects like a heat engine. In particular it should be possible to design it so that it is more and more reversible and less and less dissipative the slower it runs. This suggests an interesting tradeoff between dissipation and speed of operation about which we will be more quantitative in the next section. (The *logical* architecture of such a machine could be either reversible or not.)

A conservative system is necessarily reversible because any closed Hamiltonian system is reversible. In fact, it is physically reversible whatever its speed of operation and it would hardly make sense for the *logical* architecture of such a machine to be anything other than reversible!

Any device in which the computational degrees of freedom are realized on a scale much larger than atomic size will inevitably be dissipative: the total number of physical degrees of freedom vastly outnumber those directly involved in computation, and it is impossible to prevent leakage of energy between the computer and the "heat bath". This is the case with all present-day machines.

On the other hand, if the computational system were realized at the atomic scale, as some kind of cleverly constructed lattice, for instance, then the computational degrees of freedom would be a major fraction of the total number of degrees of freedom. In that case, the system might function as a good approximation to a closed reversible Hamiltonian system and the choice of reversible logic structure would be essential. Needless to say, no one has any practical ideas on how to realize such a computing system, though of course, the entire thrust of the development of faster computation is toward physically smaller computing elements. The point is that if atomic scale computing elements are ever achieved, reversible logic ideas may be most appropriate for doing computation. The other important dichotomy in thinking about physical realizations of computers is that between classical and quantum mechanical systems. This leads to a two-by-two classification scheme which is shown in Table 7.

Table 7. "Two-By-Two" Classification Scheme.

	OPEN	CLOSED
Classical	Conventional Machines	Fredkin's Billiard Ball Machine
Quantum Mechanical	Josephson Junction	Future Atomic Scale Machines ?
	MACROSCOPIC	MICROSCOPIC

Current computers are macroscopic and therefore classical and dissipative. Computers constructed at the atomic scale are surely quantum-mechanical and might well, for the reasons discussed earlier, be effectively closed, reversible systems.

Non-dissipative classical systems are consistent with Newtonian mechanics and represent internally consistent idealized systems which turn out to be a useful framework for demonstrating general features of reversible computation. We will be discussing Fredkin's billiard ball model in that light. Finally, there exist macroscopic (i.e., dissipative) but quantum-mechanical logic devices based on the Josephson junction which we will use to illustrate more precisely the theoretical limits on dissipative devices.

5. Theoretical Limits For Dissipative Machines

The single junction superconductor interferometer provides an example of a dissipative logic device whose properties can be quantitatively analyzed in some detail. In this section we summarize the results of Likharev[29] on the device schematized in Figure 2. It consists of a superconducting ring, broken by a Josephson junction (the cross in the figure), with provision for controlling the maximum current, I_M that can flow through the junction by varying an external current, I_C . The superconducting ring is subject to an external magnetic field with a flux, ϕ due to the combined effects of I and ϕ_e through it. The ring carries a current, I, and has a net flux, ϕ , due to the combined effects of I and ϕ_e through it. If the self-inductance of the ring is L, the net flux satisfies

$$\phi=\phi_e-LI.$$

The net flux, ϕ , is proportional to δ the difference across the junction of the superconducting order parameter phase and can be thought of as the variable describing the "state" of this system. To be precise, $\delta=2\pi\phi/(\phi_o)$, where

$$\phi_o=\frac{h}{2e}$$

is the magnetic flux quantum. The system is made to function as a logic device by manipulating

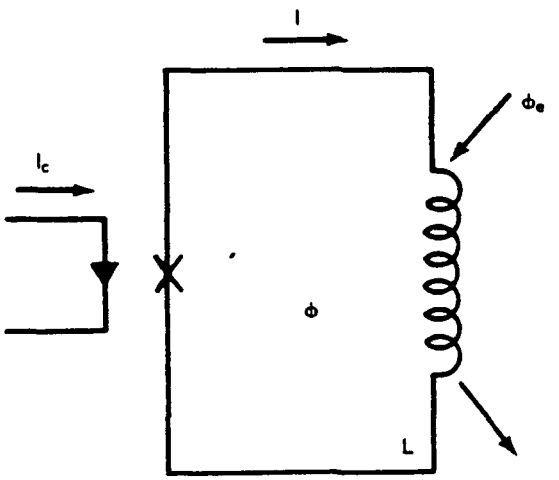


Figure 2. Josephson Junction Logic Device.

the state variable through changes in the external parameters I_c and ϕ_e .

The energy of this system is the sum of the magnetic field energy.

$$U_m = \frac{1}{2} LI^2 = \frac{1}{2L} (\phi - \phi_e)^2$$

and the energy of a junction with a phase difference δ across it

$$U_J = + I_M \frac{\phi_o}{2\pi} \cos\delta$$

$$I_M \phi_o / 2\pi \cos \left[2\pi \phi/\phi_0 \right]$$

(I_M is the maximum junction current, which, as we have said can be manipulated from the outside). The total energy functional,

$$U(\phi) = \frac{1}{2L}(\phi - \phi_e)^2 + \frac{I_M \phi_o}{2\pi} \cos\frac{2\pi\phi}{\phi_o}$$

generically has two minima. The situation when $\phi_e = 0$ and $I_M > 0$ is shown in Figure 3. This two-fold degeneracy of the lowest energy state can be used in principle to store one binary bit of information.

Better yet, we can, by changing the external parameters, I_M and ϕ_e , manipulate the shape of the potential in such a way as to smoothly switch the system point from one degenerate

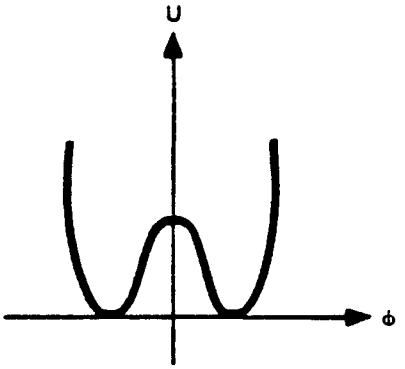


Figure 3. Two-fold Degeneracy.

ground-state to the other. This gives an explicit way of switching our bit-storage device, or carrying out as an elementary logical operation. A possible switching sequence is shown in Figure 4. where the system point (the heavy dot) starts in the right-hand well and finishes in the left-hand well. In this sequence, the system point always sits at a local potential minimum and the rate of change of the system coordinate is always completely controlled by the external parameters and can be made as small as we like at the price of dragging the switching event out over a longer and longer time. In Figure 5. we display a switching sequence where this is not true. In the third step of the sequence when the barrier finally disappears, the system point is at a large positive energy with respect to the left-hand minimum. It will roll down the hill and eventually settle down in the left-hand minimum only after dissipating its extra energy. The rate of this motion and the energy dissipated in it are not controllable from the outside, and to minimize dissipation in switching we must avoid this sort of sequence.

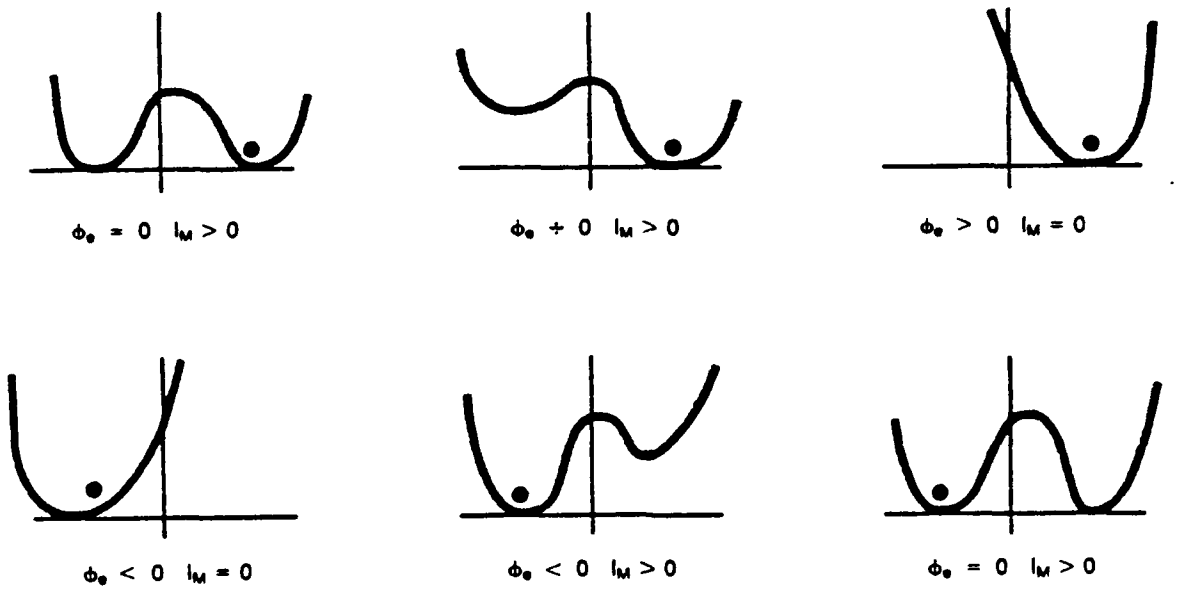


Figure 4. Switching Sequence.

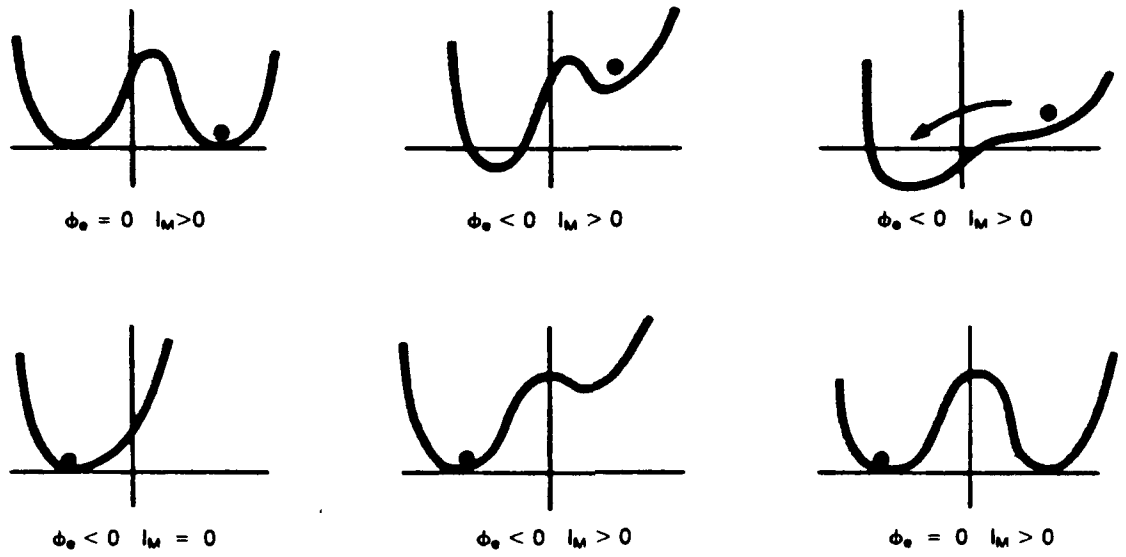


Figure 5. Dissipative Switching Sequence.

We finally come to the quantitative evaluation of dissipation in the switching event. This device has many more co-ordinates than the single flux co-ordinate ϕ , in which we are primarily interested. The effect of these degrees of freedom can be summarized by a viscous force

$$F_v = -K\dot{\phi}$$

which leads to damping of motions of the system co-ordinate (and dissipation of energy from the ϕ degree of freedom) at a rate determined by K. The total energy loss in some time evolution of ϕ is just

$$W = -\int dt\ F_v\ \dot{\phi} + K\int dt\ \dot{\phi}^2 > 0.$$

It is particularly convenient to characterize the damping by the time τ_c it takes small amplitude oscillations about a minimum to decay by e^{-1} instead of by K. In either of the switching scenarios described above, $\dot{\phi}$ necessarily is non-zero and there is necessarily some dissipation. The shape of the potential during the switching event is constrained by the requirement that spontaneous switching into the wrong well due to classical thermal fluctuations must be negligible (this means that the energy barrier between the two local minima must always be much greater than

kT).

Given this information, it is a straightforward matter to calculate the minimum energy dissipation (corresponding to the sequence of figure 5) in a switching event carried out in a time interval τ. The result is, roughly $W \approx kT\frac{\tau_c}{\tau}$ so long as $\tau \approx \tau_c$. In other words, the total energy dissipated in a switching cycle can be made as small as we like by making the switching time arbitrarily long compared to the basic dissipation time scale. This is analogous to the situation with heat engines: dissipation or entropy production can be made arbitrarily small by running the engine arbitrarily slowly. We can also determine the energy dissipated in a switching cycle like that of figure 6. In that case it turns out that $W \approx kT$ no matter how slowly we carry out the transition (at some stage the system executes free fall down a potential hill whose height is scaled by kT so that the system must dissipate energy of order kT to come into equilibrium). When this sort of device is used to make a computer, the question of overall logical organization inevitably arises. It turns out that if we use the conventional organization based on (logically irreversible) NAND gates (which can be simulated by appropriately connecting together several of the above-described switches), then switching cycles of the type of are inescapable and dissipation at the rate of roughly kT per operation is the theoretical limit. However, if a logically reversible organization is used, it turns out that only switching sequences of the type of need be encountered and the dissipation per operation can be reduced arbitrarily below kT, at the price of reducing the rate of computation. Since the motivation for reducing dissipation was to increase the rate of computation, this seems rather self-defeating. Later on we will discuss possibilities in which, at least in principle, dissipationless reversible computation can be carried out at arbitrary speed. In the next section we will finally make explicit what we mean by reversible logical architecture and devices.

5.1. Abstract Issues

It known that a computer can be built entirely out of a Boolean logic device called a NAND gate. The action of such a device is symbolized in Figure 6. The inputs a and b take on the values 0 or 1 as does the output. The output is computed by the function $\overline{(ab)}$ where the bar

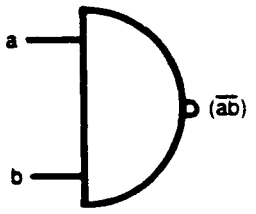


Figure 6. NAND Gate.

means logical "not" ($\overline{0}$=1, $\overline{1}$=0). This logical function is clearly not reversible or invertible since several input states produce the same output state. For this reason, a conventional computer cannot be run backwards. The previous section implies that the operation of a physical NAND gate entails a dissipation of at least kT per operation.

The discussion of the logical organization of strictly reversible computers was initiated by Bennett in 1973[27]. In pursuing this subject, Fredkin[28] developed a simple abstract reversible logical function which gave promise of being a universal building block for reversible computers. The structure and action of this function, called the Fredkin gate, is shown in Figure 7. As in the case of the NAND gates, the input and output lines take on the values 0 or 1. An examination of the truth table for this device shows that it is invertible: the correspondence between input and

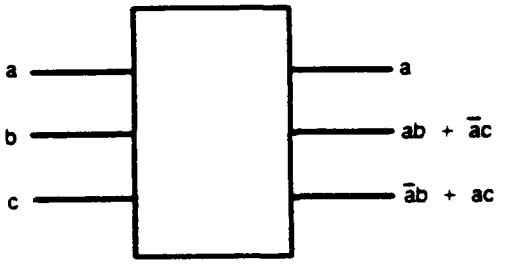


Figure 7. Fredkin Gate.

output states is one-to-one.

By ignoring some outputs and fixing some inputs the Fredkin gate can be made to perform any standard logical function. For instance, the AND of a,b, can be obtained by sitting c = 0 and keeping only the middle output line, as in Figure 8. This procedure requires a supply of input constants and a way of disposing of the unwanted outputs, known as "garbage". The brute force method of carrying out reversible computation is to record every one of the garbage constants which is produced during a computation. This is not a very satisfactory proceeding since the number of elementary logical operations required to carry out even a simple arithmetic operation, let alone a complicated program, is enormous and memory resources would be swamped.

Fredkin, Toffoli and students[30, 31] have shown how to get round this problem by really making use of the reversibility of the system. The point is that if one is doing some machine instruction such as computing the sum of two numbers which involves a large number of logical operations, one may: a) do the calculation, producing a large quantity of garbage, b) record the result, producing a very small amount of garbage c) run the computation backwards, eating up the garbage produced in a). *If the machine instruction itself is logically reversible, as in* $(A,B)\rightarrow\left(\frac{A+B}{2}\right)\left(\frac{A-B}{2}\right)$, one doesn't even have to accumulate garbage in step b). The only true

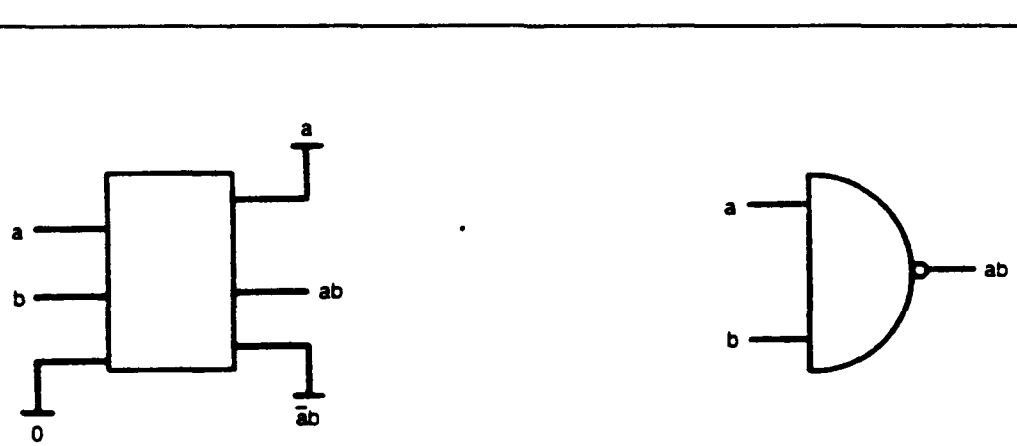


Figure 8. AND (a,b).

garbage which needs special memory allocation and has to be kept to the end of the program is that associated with truly non-invertible machine instructions. By careful design of the machine instruction set and programming practices, it appears possible to reduce the garbage accumulated in a typical program to a manageable size. We are not aware of a quantitative answer to the question, if a program requires a total of N steps to execute, what is the minimum number of garbage bits that must be accumulated? We suspect that the answer is logN, which would mean that only a trivial amount of memory has to be devoted to true garbage accumulation, but we don't have a proof.

Finally, as a result of this experience, Fredkin and students have been able to produce sketchy but credible designs for real computers. These designs are explicit two-dimensional wiring diagram layouts of Fredkin gates, and have been demonstrated in computer simulation exercises to work as expected.

To summarize, although computers based on reversible logic elements have some unfamiliar features, machines whose effective operation is nearly a carbon copy of conventional computers can be laid out as explicit two-dimensional hook-ups of the logically reversible Fredkin gate. In the next section we will take up the question whether the Fredkin gate is physically realizable.

5.2. Physical Realization of the Fredkin Gate

In order to give an existence proof for reversible computation, Fredkin has introduced a stylized model based on perfectly elastic collisions of billiard balls moving on a frictionless plane[28]. Consider a two dimensional square grid as laid out in Figure 9 with unit spacing between the grid points and identical hard spheres of radius $\frac{1}{\sqrt{2}}$ moving at one lattice spacing per time step along the principal directions of this lattice as shown in At time $t = 0$, the center of every ball lies on a grid point and that will again be true at every integer-valued time. Balls will occasionally undergo right-angle elastic collisions at integer-valued times (see b). The balls emerging from the collision will again move along the principal lattice directions and their centers will coincide with lattice points at integer-valued times. At some lattice points a billiard ball will

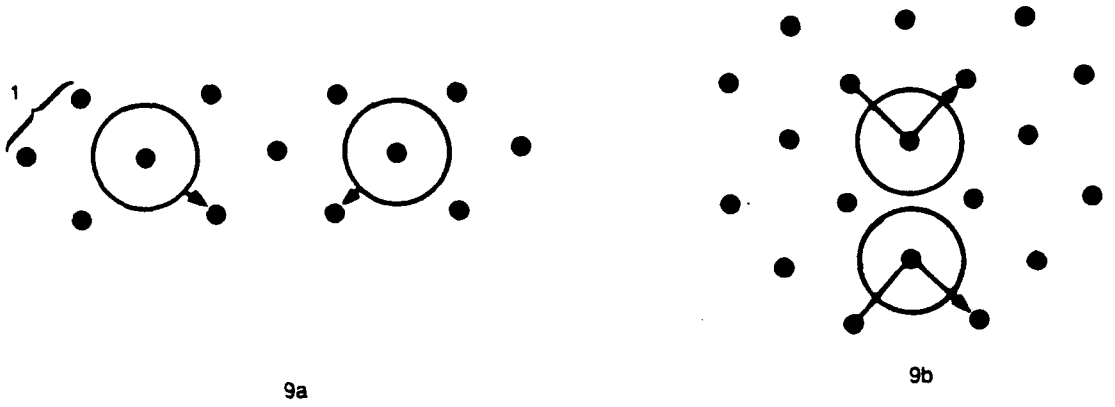


Figure 9. Square Grid.

be nailed down to function as a perfect reflector of anything that comes by. The presence or absence of a billiard ball at a lattice site at an integer time can be taken as a binary bit of information and the Newtonian evolution of such a system of billiard balls amounts to a "calculation" involving those bits.

The construction of the Fredkin gate goes in two steps. First construct the gate shown in Figure 10 where the bar represents a fixed reflector. This device lets a ball on the x path go through undeflected if no ball is on the c path, but switches it onto a different path if a ball is simultaneously present on the c path (and lets the c ball through undeflected). The information processing here amounts to switching bits between two output paths, depending on the context of a control path. If these interaction gates are strung together according to Figure 11 (where the connecting paths have appropriate delays in them to maintain proper synchronization), it is possible to verify that the overall system functions exactly like the Fredkin gate.

According to the previous section a useful reversible computer can be made by wiring together enough Fredkin gates. The same computer can therefore be realized as a two-dimensional arrangement of appropriately aimed and placed billiard balls and reflectors. The execution of a program on such a computer is just the carrying out of the Newtonian time evolution

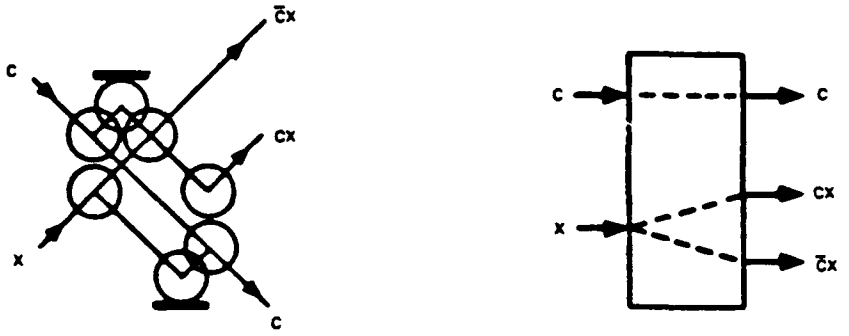


Figure 10. Interaction Gate.

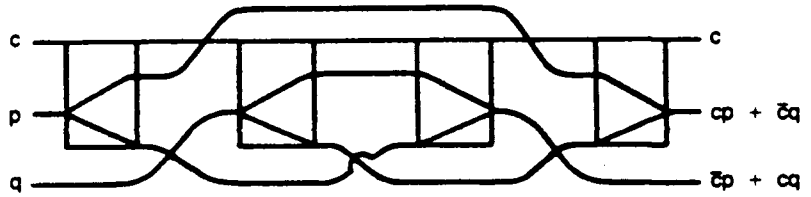


Figure 11. Gate Connections.

of the mechanical system.

By construction, this system is dissipation-free and since the billiard ball velocity is arbitrary, it can operate at any speed we like. This amounts to an existence proof for dissipation-free, fast computing via a classical conservative system.

5.3. Billiard Ball Machine as Cellular Automaton

The defects of the billiard ball model as a practical physical realization of reversible computing are fairly obvious. It does, however, have the virtue of suggesting a different abstract framework within which some interesting new possibilities for physical realization suggest themselves.

The essence of the billiard ball model is that at integer time steps billiard balls are located at lattice points only and the pattern of occupied lattice sites changes from one time step to the next according to some rule. The rule is not made explicit, but is the result of evolving the previous configuration according to Newtonian mechanics. The step by step evolution of the state of a lattice according to a local rule is the subject of cellular automaton theory, a particularly active branch of fundamental computer science. It is natural to ask whether the essence of the billiard ball model can be captured in some cellular automaton rule. For the moment, this is just an idle question, but in the next section we will see that the cellular automaton framework is one into which it might be possible to fit real atomic physics.

There is indeed a cellular automaton version of the billiard ball machine which we have reconstructed from remarks of Fredkin (the precise rule to be used is, we believe, due to Margolus). Consider a lattice divided up into individual cells by solid and dotted lines in the manner of Figure 12. Some of the cells are occupied and we want to devise a transitional rule to cause the pattern of occupation to change. If we look at the unit cells defined by the solid lines alone or the dotted lines alone, we see that they each contain four of the unit cells of the full lattice. The transition rule will be defined for such groups of four cells and applied alternately to the groups

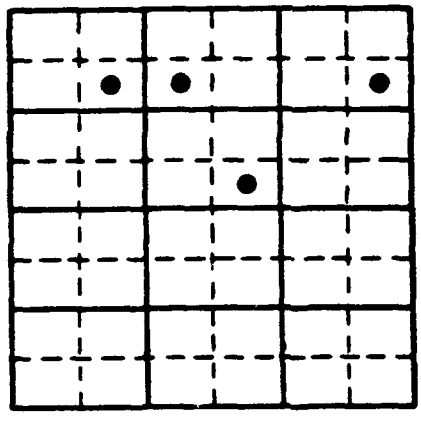


Figure 12. Dotted lines.

defined by the solid lines and dotted lines. The transition rules we will use are defined in Figure 13. Rotations of the rules presented are also valid. The transformation effected by these rules is obviously one-to-one within the group of four cells on which they act. By extension, the action of these rules on the lattice as a whole is one-to-one and reversible.

A bit of playing with the rules shows that single occupied cells propagate like billiard balls in the manner indicated in Figure 14. Single occupied cells however, do not collide with each other in the manner of billiard balls. In order for this to work out properly, it is necessary to consider a train of two similar occupied cells, as in Figure 15 and the three other versions, corresponding to the other possible directions of motion, propagate and collide exactly in the manner of billiard balls. One can also construct a configuration which does not propagate and reflects any billiard ball configuration incident on it. Figure 16

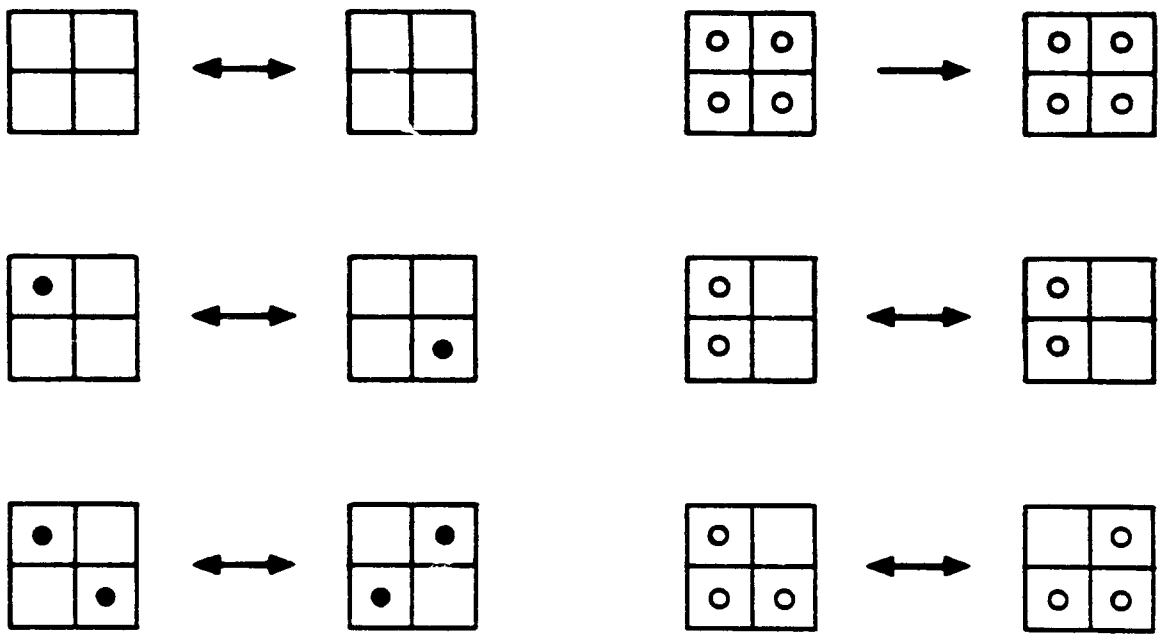


Figure 13. Transition Rules.

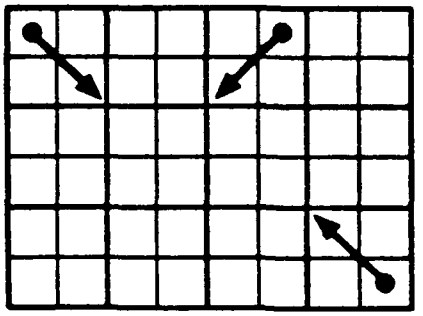


Figure 14. Propagation.

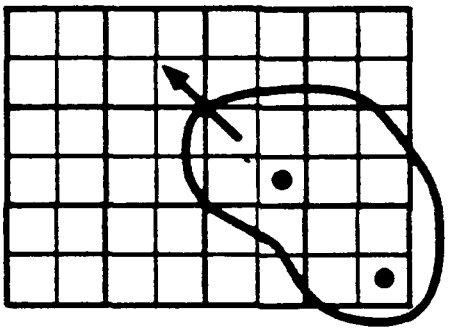


Figure 15. Cell Train.

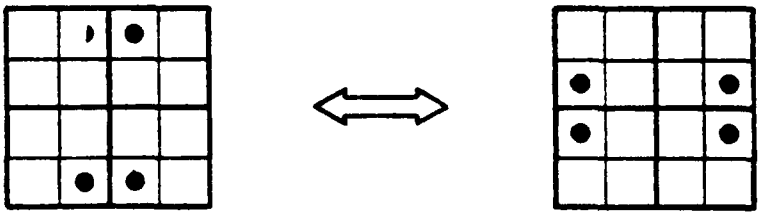


Figure 16. Non-Propagating Configuration.

As the previous sections have shown, an explicit reversible computer design is available once we have "billiard balls" and "mirrors". Now that we know that our cellular automaton rules produce these two types of object it is possible, in a perfectly explicit way, to construct a reversible cellular automaton computer. This is interesting because, as we shall argue in the next section, the cellular automaton framework seems particularly well-suited to realization at the atomic lattice scale.

5.4. Notional Atomic Scale Realizations

We have argued that reversible computing ideas are likely to be of most interest in the study of computers realized at the atomic scale, where the computational degrees of freedom are not vastly outnumbered by all the rest and a computer might function as a good approximation to a conservative Hamiltonian system. We would now like to explore a framework which suggests that cellular automaton rules of the type just discussed might actually be realizable at the atomic scale. We don't have a specific practical proposal, but rather some general notions about the sort of physical systems which it might be profitable to explore.

Under the right conditions, atoms or molecules will arrange themselves in a regular lattice. For a bulk material, this lattice will be three dimensional, while for material adsorbed on a convenient substrate the lattice will be two dimensional. Let us consider a two-layer (i.e. essentially two-dimensional) lattice of the type displayed in Figure 17.

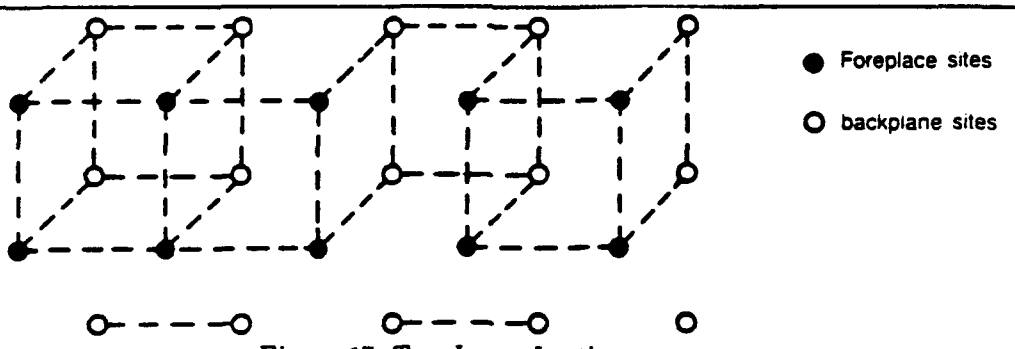


Figure 17. Two Layer Lattice.

The lattice sites in the the layers, (foreplane and backplane) are distinguished by open and filled circles. The basic idea is that the sites harbor some two-fold quantum mechanical degree of freedom (such as a spin, the presence of an atomic excitation, etc.) which can be manipulated and used as a token for computing. For convenience, we will refer to this degree of freedom as a spin, although it need not actually be one.

There are interactions between "spins" at neighboring sites, and we have indicated the desired pattern of interactions by dashed and wavy lines. They will cause the "spins" on the sites to change with time and our goal is to cause this time evolution to occur in a way which carries out the cellular automaton rules discussed in the previous section. The simplest way to do this is to imagine that all the wavy line interactions can be turned on or off simultaneously from outside by some macroscopically controllable agency such as a laser pulse. Suppose that the wavy line interactions can be turned on and then off in just such a way as to exchange spins between the foreplane and backplane sites (each wavy line connects just one foreplane and one backplane site). Suppose further that the dashed line interactions, which connect up cells of four sites, either all in the foreplane or all in the backplane, can be turned on and then off in such a way as to effect the transformation on spins corresponding to the cellular automaton rules of the previous section. Then by alternately activating the dashed and wavy bonds one would effect the cellular automaton rules as transformations on the "spins". Then by the discussion of all the previous systems, this microscopic device could be made to function as a reversible computer.

If we think of the site variables as really being elementary spins, it is easy to see what is involved in obtaining exchange. The most general interaction between two spins is

$$H_I = \alpha(t)\vec{\sigma_1} \cdot \vec{\sigma_2}$$

The bond strength, α, depends on t, since we must imagine being able to manipulate from outside. If we turn this bond on and then off in such a way that

$$\int_0^\infty dt\, \alpha(t) = \pi$$

(a matter of properly tailoring the laser pulse, or whatever it actually is, that manipulates the bond) then it is easy to show that the net effect is simply to exchange the spins between the two

sites. Although we have not done it explicitly, we believe it should be possible to construct a set of bonds for four spins which can be manipulated in such a way as to carry out the desired cellular automaton transformation.

If a scheme of the above type can be found, it suggests that a reversible atomic scale (and, therefore, one might hope, very fast) computer could be built. The obvious challenge is to find semi-realistic choices for sites, bonds and the extended driver of the bonds. We don't have any concrete response to this challenge, but we think that materials questions of the kind raised here are a rather natural sort of outcome of thinking about where reversible logic fits in the overall scheme of computing concerns. We have been struck by the extent to which previous work on reversible computing has focused on abstract questions and would strongly recommend that future work begin to focus on physics questions. The framework we have presented is not necessarily the best one, but does give a way of focusing on an interesting set of materials and physics questions, and might have the virtue of stimulating thought.

5.5. Quantum Mechanics Issues

The previous discussions have not made much of the fact that physics at the atomic scale is necessarily quantum mechanical. Indeed, the whole question of the role of quantum mechanical effects in small-scale computing devices has been only very sketchily explored in the literature. The scheme we have been discussing has one illuminating and bizarre quantum mechanical feature which we will explain, just to give an idea of the sort of issues involved.

The bonds of our lattice cellular automaton are alternately switched on and off by some external system which acts as a clock and driver for the whole system. This driver is itself some mechanical system executing periodic motion; let us for definiteness take it it be a rotator of some kind, rotating in some angular coordinate, Θ, such that every time Θ passes through some marker angle, Θ_0, the bonds responsible for switching spins on the lattice are activated.

We can write down a fairly explicit Lagrangian for this system:

$$L = \frac{1}{2}I\dot{\Theta}^2 + \left[\sum_{i=1}^{N}\vec{\sigma}_1^{(i)} \cdot \vec{\sigma}_2^{(i)}\right]\pi\dot{\Theta}\,\delta(\Theta - \Theta_0) + \cdots$$

The first term is just the rotator kinetic energy and says that, in the absence of other terms, the system just executes uniform rotational motion. The next term describes the interaction with the "wavy" bonds of the previous section: the spins are divided up into N pairs and the interaction of each pair with Θ is such as to effect the exchange transition every time Θ passes through Θ_0. The $\dot{\Theta}$ factor ensures the same action on the spins no matter how fast Θ is moving. The dots indicate the terms, not yet specified but similar in nature, responsible for the spin transformations on four spins at a time (needed to complete the cellular automaton rules).

In the classical approximation to the motion of Θ, the rotator proceeds at constant velocity and one cellular automaton transformation is executed per cycle. The quantum-mechanical version of the motion of Θ is somewhat different. The rotator interacts with the computer coordinates through the sum

$$\sum\vec{\sigma}_1^{(i)} \cdot \vec{\sigma}_2^{(i)} + \cdots$$

and, as the calculation proceeds, this sum takes on an essentially random sequence of values. This is roughly equivalent to saying that Θ is moving in a one dimensional random potential.

In a random potential, there are no propagating states and all wave functions decay exponentially with distance. If a computation takes N steps, we prepare the system in a state localized around $\Theta = 0$ and the computation is completed when Θ is finally observed at $2\pi N$. The exponential decay of wave functions probably means that the time to complete long calculations increases exponentially with N ! To know under what circumstances this would be a practical problem, we would have to have a much more concrete model to work with. This observation could be elaborated further, but is meant to give an example of the peculiar phenomena that must be understood when we try to think about computing at the quantum mechanical level.

6. CONCLUSIONS

6.1. Transformations

The idea of employing source program transformations is not new, but has received renewed interest with the development of functional programming (the 'fp' language) and logic programming (the 'PROLOG' language). As discussed above, this technique has an interesting, if as yet unproved, potential.

Transformation techniques may be the path of choice for Soviet scientists. The Soviets have well-known problems in computer hardware, but have immense talent in mathematics. It just may be that the break-through needed in transformation techniques will be mathematical in nature. In addition, the Soviets have concentrated their software efforts in this area. There are really only two major language/compiler systems that have been developed by the Soviets. The rest are derivative of western software systems. The first unique Soviet software system is a language for program development and is not of particular interest here. The second is called 'ANALYTIK'[32] and has gone through at least three major revisions since 1970. Some of these can be traced in the bibliography of Appendix A-2. An example of output from ANALYTIK can be found in Appendix A-3. A brief reading of a *very* restricted sample of the open Soviet literature in this area did not reveal anything of especial interest, however.

In general, the development of transformation techniques should be closely followed. Rapid progress could occur once the right good idea is discovered. There is, of course, no guarantee that this will occur any time soon.

6.2. Reversible Computing

The ideas in reversible computing are very immature at present. The potential side-benefits from developments in this area could be very important however, even if the main ideas are not found to be feasible. The important areas to watch are technological. The key is some new molecular-scale technology[33, 34]. While there are developments in this area, they seem to be a very long way from any practical system.

6.3. Final Remarks

We have discussed two ideas about how a radical improvement in computer performance might come about. There are of course many other possibilities as well. The most important would be methods of organizing parallel calculations. This is an old, but very critical problem. The development of computer system ideas is proceeding at a rapid pace. It will take considerable effort to try to predict the likely direction of new developments.

7. REFERENCES

1. A. M. Despain, G. J. MacDonald, A. M. Peterson, O. S. Rothaus, and J. F. Vesecky, *Radical Computing,* Jason, McLean, Va (April, 1983). Tech. Rep. JSR-82-701
2. Richard J. Fateman and W.A. Martin, "The MACSYMA System," *SYMSAM-II,* pp.59-75..
3. L. Nagel, *SPICE: A Computer Program to Simulate Semiconductor Curcuits,* May 1975.
4. E. W. Dijkstra, "Notes on Structured Programming," TH. Rep., Technische Hogeschool, Eindhoven, The Netherlands (1970). 2nd ed.
5. Marvin Minsky, "Form and Content in Computer Science," *ACM (1970 Turing Lecture)* Vol. **17**(2), pp.197-215 (1970).
6. H. A. Simon, "The Heuristic Compiler," in *Representation and Meaning,* ed. L. Siklossy, Prentice Hall, New Jersey (1972).
7. Frances E. Allen and John Cocke, "A Catalogue of Optimizing Transformations," in *Design and Optimization of Compilers,* ed. R. Rustin, Prentice-Hall (1972).
8. H. Partsch and R. Steinbruggen, "Program Transformation Systems," *ACM Computing Surveys* Vol. **15N 3**, pp.199-237 (sept. 1983).
9. John Backus, "Can Programming be Liberated From the von Neumann Style?," *CACM* Vol. **21**(8), pp.614-641, Tenth Turing Lecture (Aug. 1978).
10. Pamela Zave, "The Operational Versus the Conventional Approach to Software Development," *CACM* Vol. **27**, pp.104-118 (Feb. 1984).
11. J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.* Vol. **19**, pp.297-301 (Apr. 1965).
12. J. McCarthy and et. al., *LISP 1.5 Programmer's Manual,* MIT Press, Cambridge, MA (1962).
13. K. Iverson, *A Programming Language,* John Wiley and Sons, New York, N.Y. (1962).
14. J. Schwarz, "Using Annotations to Make Recursion Equations Behave," Res. Memo, Dept. Artif. Intell. U. Edinburgh (1977).
15. E. A. Ashcroft and W. W. Wadge, "Lucid, A Nonprocedural Language with Iteration," *Communications of the ACM* Vol. **20**(7), pp.519-526 (July 1977).
16. David A. Turner, "Recursion Equations as a Programming Language," in *Functional Programming and its Applications,* ed. David A. Turner, Cambridge University Press, Cambridge (1982).
17. A. Colmerauer, H. Kanoui, and M. van Caneghem, *Etude et Realization d'un System Prolog,* Groupe de Researche en Intelligence Artificielle, Univ. d'Aix-Marseille, Luminy (1979).
18. David B. Loveman, "Program Improvement by Source to Source Transformation," *JACM* Vol. **24**(1), pp.121-145 (Nov. 1975).
19. W. F. Clocksin and C. S. Mellish, in *Programming in Prolog,* Springer-Verlag, New York (1981).
20. Robert A. Kowalski, "Predicate Logic as a Programming Language," *Proc. IFIPS 74,* IFIPS (1974).
21. Ben Wegbreit, "Goal Directed Program Transformation," *IEEE Trans. Soft. Eng.* Vol. **SE-2**(2), pp.69-80 (1976).
22. D. H. D. Warren, "Applied Logic - Its Use and Implementation as Programming Tool," Ph.D. Thesis, Univ. Edinburgh, Scotland (1977). Available as Tech. Note 290, AI Center, SRI International

23. Keith L. Clark and Frank G. McCabe, "The Control Facilities of IC-Prolog," in *Expert Systems in the Micro Electronic Age*, ed. D. Michie, Edinburgh Univ. Press (1979).

24. Volker Strassen, "Gaussian Elimination is not Optimal," *Numerische Mathematk* Vol. **13**, p.354 (1969).

25. Robert W. Keyes, "Fundamental Limits in Digital Information Processing," *Proc. IEEE* Vol. **69**(2), pp.267-278 (Feb. 1981).

26. R. Landauer, *IBM J. Res. Dev.* Vol. **3**, p.183 (1961).

27. C. H. Bennett, *IBM. J. Res. Dev.*, p.525 (1973).

28. Edward Fredkin and Tommaso Toffoli, "Conservative Logic," *International J. Theoretical Physics* Vol. **21**(3 & 4), pp.219-253 (1982).

29. K. K. Likharev, "Classical and Quantum Limitations on Energy Consumption in Computation," *Int'l J. Theor. Physics* Vol. **21**, p.311 (1982).

30. E. Barton, "A Reversible Computer Using Conservative Logic," 6.895 term paper, MIT (1978).

31. Andrew Lewis Ressler, *The Design of a Conservative Logic Computer and a Graphical Editor simulator*, MIT (Jan. 1981). MS thesis

32. V. M. Glushkov, V. G. Bodarchuk, T. A. Grinchenko, A. A. Dorodnitsyna, V. P. Klimenko, A. A. Letichevskii, S. B. Pogrebinskii, A. A. Stognii, and Yu. S. Fishman, "ANALYTIK (Algorithmic Language for the Descroption of Computing Processes Using Analytical Transformations," *Kibernetika*, pp.102-134 (May-June 1971).

33. Forrest L. Carter, "Prospects for Computation at the Molecular Size Level," *COMCON 84 Digest of Papers*, pp.110-119, Computer Society Press (Feb. 27-Mar.1, 1984).

34. M. Keith DeArmond and Kenneth W. Hanck, "Switching and Charge Storage in Metal Complexes- Smart Molecules?," *COMCON 84 Digest of Papers* Vol. **28**, p.Computer Society Press, IEEE (Feb. 27-Mar.1, 1984).

8. APPENDIXES

APPENDIX A-1

RECURRENCE SOLVER

```
% Solving Recurrences:
% Assignment 3, CS257
% Peter Van Roy
% Converts all functions that can be expressed as a polynomial
% to an efficient Horner form.
% The new form replaces the old in the PROLOG data base.
rgo :-
        solve(Func),      % Get function to be solved.
        retract(solve(Func)),      % Remove it from data base.
        funclist(Func, 20, FuncList),      % Get first 20 function values.
        get_succ_diff(FuncList, DiffList),  % Calculate successive differences.
        conv_to_poly(DiffList, Poly/D),  % Convert to polynomial representation.
        horner(Poly, N, Horner),  % Convert polynomial to efficient Horner form.
        abolish(Func,2),  % Remove old definition from data base.
        NewFunc=..[Func,N,F],          % Arrange the result to its final form.
        (D=1 -> Expr=Horner; Expr=Horner/D),
        NewClause=(NewFunc :- F is Expr),
        nl, write('The improved form for '), write(Func),
        write(' is: '), nl, write(NewClause), nl,
        assert(NewClause),  % Insert the new form in the PROLOG data base.
        fail.      % Continue with other functions.
rgo.

% Generate a list of values Func(i) for i=0, 1, ..., N-1.
funclist(Func, N, FuncList) :-
        funclist(Func, 0, N, FuncList).

funclist(Func, N, N, []) :- !.
funclist(Func, I, N, [F|FuncList]) :-
        Term=..[Func,I,F],
        call(Term), !,
        I1 is I+ 1,
        funclist(Func, I1, N, FuncList).


% Get the first elements of all rows of successive differences
% down to the row of zeroes.  This is enough to characterize
% the function completely.
get_succ_diff(Row, []) :- zero(Row), !.
get_succ_diff([A|Row], [A|DiffList]) :-
        next_row([A|Row], NextRow),
        get_succ_diff(NextRow, DiffList).

next_row([A,B|Row1], [D|Row2]) :- !,
        D is B-A,
        next_row([B|Row1], Row2).
next_row([_], []).

zero([0|List]) :- zero(List).
zero([]).
```

```
% Convert a representation of a function as a list of successive
% differences to a polynomial:
% Uses the recurrence:  Poly = Ai +  (N-i)/(i+ 1)*NextPoly.
conv_to_poly(DiffList, Poly/D) :-
        conv_to_poly(DiffList, 0, Poly/D).

conv_to_poly([Am], _, [Am]/1) :- !.
conv_to_poly([Ai|DiffList], I, Poly/D) :-
        I1 is I+ 1,
        conv_to_poly(DiffList, I1, NextPoly/DN),
        minus(NextPoly/DN, NegPoly/DN),
        mult(NegPoly/DN, I, Temp1/T1),
        add([0|NextPoly]/DN, Temp1/T1, Temp2/T2),
        div(Temp2/T2, I1, Temp3/T3),
        add([Ai]/1, Temp3/T3, Poly/D).


% Convert a polynomial in list form to a Horner's formula
% structure, using N as the variable:
% (The second, third, and fourth clauses are optimizations).
horner([A0], N, A0) :- !.
horner([An.1], N, N+ An) :- !.
horner([0|Poly], N, Horner*N) :- !,
        horner(Poly, N, Horner).
horner([Ai|Poly], N, Ai) :-
        zero(Poly), !.
horner([Ai|Poly], N, Horner*N+ Ai) :-
        horner(Poly, N, Horner).


% Polynomial arithmetic:
% Polynomials are represented as lists of integers divided by an
% integer.  This avoids (1) round-off error in C-PROLOG, and
% (2) truncation on UNSW PROLOG.

% Addition of two polynomials:
add(Poly1/D1, Poly2/D2, Poly/D) :-
        gcd(D1, D2, G),
        D is (D1/G)*D2, % D is lcm(D1,D2)
        F1 is D2/G, % multiplying factor for Poly1's terms.
        F2 is D1/G, % multiplying factor for Poly2's terms.
        addx(Poly1, Poly2, Poly, F1, F2).

addx([A1|Poly1], [A2|Poly2], [S|Poly], F1, F2) :- !,
        S is A1*F1+ A2*F2,
        addx(Poly1, Poly2, Poly, F1, F2).
addx([], Poly2, Poly2, _, _) :- !.
addx(Poly1, [], Poly1, _, _).

% Change sign:
minus(Poly/D, Res/D) :- minusx(P  .y, Res).

minusx([A|Poly], [R|Res]) :-
        R is -A,
        minusx(Poly, Res).
```

```
minusx([], []).
```

```
% Multiplication by a scalar:
mult(Poly/D, Scalar, Res/R) :-
        gcd(D, Scalar, G),
        R is D/G,
        S is Scalar/G,
        multx(Poly, S, Res).

multx([A|Poly], S, [P|Res]) :-
        P is S*A,
        multx(Poly, S, Res).
multx([], S, []).

% Division by a scalar:
div(Poly/D, Scalar, Res/R) :-
        DI is Scalar*D,
        gcd([DI|Poly], G),
        (G==1 -> divlist(Poly, G, Res), R is DI/G;
                Res=Poly, R=DI).

divlist([A|Poly], S, [R|Res]) :-
        R is A/S,
        divlist(Poly, S, Res).
divlist([], _, []).

% gcd calculation:
gcd(X, 0, Y) :- !, X=Y.
gcd(U, V, X) :-
        W is U mod V,
        gcd(V, W, X).

% gcd of a list:
gcd([A], A) :- !.
gcd([A,B|List], 1) :-
        gcd(A, B, 1), !.
gcd([A,B|List], Ans) :-
        gcd(A, B, G),
        gcd([G|List], Ans).
```

APPENDIX A-2

CACHING SYSTEM

```
% CACHE:

% Generic program to cache heads of non-unit clauses,
%       FORM: X(Y,Z):- ...
% Must fix up correct number of variables "Y,Z,...".

% The following compensate for principle functor
% not being a variable.
asf(f,Y,Z):-asserta((f(A,B):-eq(A,Y),eq(B,Z),!,fail)).
asf(a,Y,Z):-asserta((a(A,B):-eq(A,Y),eq(B,Z),!,fail)).
asf(r,Y,Z):-asserta((r(A,B):-eq(A,Y),eq(B,Z),!,fail)).

rtf(f,Y,Z):-retract((f(A,B):-eq(A,Y),eq(B,Z),!,fail)).
rtf(a,Y,Z):-retract((a(A,B):-eq(A,Y),eq(B,Z),!,fail)).
rtf(r,Y,Z):-retract((r(A,B):-eq(A,Y),eq(B,Z),!,fail)).

ass(f,YQ,ZQ):- asserta((f(YQ,ZQ):-cnt(f))).
ass(a,YQ,ZQ):- asserta((a(YQ,ZQ):-cnt(a))).
ass(r,YQ,ZQ):- asserta((r(YQ,ZQ):-cnt(r))).

rts(f,YQ,ZQ):- retract((f(YQ,ZQ):-cnt(f))).
rts(a,YQ,ZQ):- retract((a(YQ,ZQ):-cnt(a))).
rts(r,YQ,ZQ):- retract((r(YQ,ZQ):-cnt(r))).

% FAIL CACHING.
%asf(X,Y,Z):-asserta((X(A,B):-eq(A,Y),eq(B,Z),!,fail)).
%rtf(X,Y,Z):-retract((X(A,B):-eq(A,Y),eq(B,Z),!,fail)).

% Cache a fail.
fcache(X,Y,YP,Z,ZP):- nvbind(Y,YP,Z,ZP),
                      cleanup(X),asf(X,Y,Z),!.

% Cleanup last cache insertion.
cleanup(X):- rtf(X,Y,Z), fremove(X,Y,YQ,Z,ZQ),asf(X,YQ,ZQ).
cleanup(X).

% Eliminate duplicates and submissive entries.
fremove(X,Y,YQ,Z,ZQ):-rtf(X,YP,ZP),
                      switch(Y,YP,YR,Z,ZP,ZR),
                      fremove(X,YR,YQ,ZR,ZQ),
                      fconassert(X,YQ,YP,ZQ,ZP).
fremove(X,Y,Y,Z,Z).

fconassert(X,Y,YP,Z,ZP):- dominate(Y,YP),dominate(Z,ZP),!.
fconassert(X,Y,YP,Z,ZP):- asf(X,YP,ZP).
```

```
% CACHE SUCCESS.
%ass(X,YQ,ZQ):- asserta((X(YQ,ZQ))).
%rts(X,YQ,ZQ):- retract((X(YQ,ZQ))).

% Remove failed head from the cache.
scache(X,Y,YP,Z,ZP):- fremv(X,YP,ZP), ycache(X,Y,Z),!.


% Remove exact fail head.
fremv(X,Y,Z):- rtf(X,YP,ZP),
          fremv(X,Y,Z), fcassert(X,Y,YP,Z,ZP).
fremv(X,Y,Z).

fcassert(X,Y,YP,Z,ZP):- eq(Y,YP),eq(Z,ZP),!.
fcassert(X,Y,YP,Z,ZP):- asf(X,YP,ZP).

% Success caching
ycache(X,Y,Z):- sremove(X,Y,YQ,Z,ZQ),ass(X,YQ,ZQ),!.

% Elim Dups.
sremove(X,Y,YQ,Z,ZQ):- rts(X,YP,ZP),
                  switch(Y,YP,YR,Z,ZP,ZR),
                  sremove(X,YR,YQ,ZR,ZQ),
                  sconassert(X,YQ,YP,ZQ,ZP).
sremove(X,Y,Y,Z,Z).

% Replace head in cache.
sconassert(X,Y,YP,Z,ZP):- dominate(Y,YP),
                    dominate(Z,ZP),!.
sconassert(X,Y,YP,Z,ZP):- ass(X,YP,ZP).
```

```
% CACHE UTILITIES.

% True if Y & YP are the same.
eq(Y,YP):-var(Y),var(YP),!.
eq(Y,YP):- Y==YP.

% True if Y is more general than YP.
dominate(Y,YP):-var(Y),!.
dominate(Y,YP):-Y==YP.

% Selects most general set of terms.
switch(Y,YP,YP,Z,ZP,ZP):- dominate(YP,Y),
                          dominate(ZP,Z),!.
switch(Y,YP,Y,Z,ZP,Z).

% Binds only non-variables.
nvbind(Y,Y,Z,Z):-nonvar(Y),nonvar(Z),!.
nvbind(Y,Y,Z,V):-nonvar(Y),!.
nvbind(Y,U,Z,Z):-nonvar(Z),!.
nvbind(Y,U,Z,V).


% Call functions for monitoring and caching.
% Head.
h(X,Y,Z):-cnt(X),inc(level),nl,
        count(level,N),tab(N),
        write(X),write('('),
        write(Y),write(','),
        write(Z),write(')'),!.

hs(X,Y,YP,Z,ZP):- h(X,Y,Z).
hf(X,Y,YP,Z,ZP):- h(X,Y,Z), confcache(X,Y,YP,Z,ZP),!.

% Tail.
t(X,Y,Z):-count(level,N),tab(N),dec(level),
        write(' , success: '),
        write(X),write('('),
        write(Y),write(','),
        write(Z),write(')'),nl,!.

ts(X,Y,YP,Z,ZP):- t(X,Y,Z), conycache(X,Y,Z).
tf(X,Y,YP,Z,ZP):- t(X,Y,Z), conscache(X,Y,YP,Z,ZP),!.


% Cache control.
conycache(X,Y,Z):-cache(on),cache(X), ycache(X,Y,Z),!.
conycache(X,Y,Z).

conscache(X,Y,YP,Z,ZP):-cache(on),cache(X), scache(X,Y,YP,Z,ZP),!.
conscache(X,Y,YP,Z,ZP).

confcache(X,Y,YP,Z,ZP):-cache(on),cache(X), fcache(X,Y,YP,Z,ZP),!.
confcache(X,Y,YP,Z,ZP).

cache_on :- assert((cache(on))).
```

```
cache_off:-retract((cache(on))),!.
cache_off.
```

```
%========================== END ==========================

%Program commands.
go:- cache_off, w('orig.p'),
     p_compile(all), w('compiled.p'),
     reset,
     rvt, w('vt.p'), reset,
     rtv, w('tv.p').

go1:-cache_on, w('orig.p'),
     rtv,w('tv.p'),reset,
     rvt,w('vt.p').

go2:-cache_on, w('orig.p'),
     rvt,w('vt.p'),reset,
     rtv,w('tv.p').

w(N):- tell(N),phead,lp,chead,
       tc,told,close(N).

rvt:- tell(tracevt),thead,r(v,t),
      nl,nl,nl,nl,chead,
      lc,tc,told,close(tracevt).

rtv:- tell(tracetv),thead,r(t,v),
      nl,nl,nl,nl,chead,
      lc,tc,told,close(tracetv).

tc:-count(m,M),count(c,C),count(f,F),
    count(a,A),count(r,R),
    T is M + C + F + A + R,
    write('Total calls = '),
    write(T),nl.

%Input/output.
phead:-write('  EXECUTED PROGRAM LISTING  '),nl,nl.
thead:-write('  TRACE OF PROGRAM EXECUTION    '),nl,nl.
chead:-write(' CALL COUNTS '),nl,nl.

%General purpose counters.
count(X,0).
cnt(X):-inc(X).
gencnt(0):-assertz((count(X,0))),!.
gencnt(_).
inc(X):-retract(count(X,N)),M is N + 1,
        asserta(count(X,M)),gencnt(N),!.
dec(X):-retract(count(X,N)),M is N - 1,
        asserta(count(X,M)),gencnt(N),!.
zero(X):-retract((count(X,M))).
reset:-zero(_),reset.
reset:-assert((count(X,0))).

%Program listings.
lc:-listing(count).
lf:-listing(f).
```

```
la:-listing(a).
lr:-listing(r).
lp:-listing([m,c,f,a,r,count]).
```

APPENDIX A-3

REPRINT OF "A BIBLIOGRAPHY OF SOVIET WORKS IN ALGEBRAIC MANIPULATIONS"

by

Alfonso M. Miola

[SIGSAM Bull., 15 (1), February 1981]

A BIBLIOGRAPHY OF SOVIET WORKS IN ALGEBRAIC MANIPULATIONS

Alfonso M. MIOLA
Istituto di Analisi dei Sistemi e Informatica
Via Buonarroti 12
00185 ROMA (ITALY)

In the June 1979 a Summer School on Programming has been organized by the Bulgarian Academy of Sciences in Primorsko (Bulgaria). Among other topics Symbolic and Algebraic Manipulations was covered with a few lectures by me and with some panel discussions. The lecturers were Professors Lavrov, Arato, Havel, Pottosin, Bauer, Pasula, Ershov, Andronico, Miola.
Recently Prof. Pottosin sent me a bibliography of the works done in Russia in Computer Algebra. I do think that this bibliography could be of interest of our community. Prof. Pottosin address is:
- 630090 Novosibirsk 90 - Computer Center - I.V. Pottosin.--USSR

1. S.A.Abramov. On Rational Functions Summing. J. Comput. Math. and Math. Phys., v. 11, No.4, 1971, pp. 1071-1075.

2. S.A.Abramov. On Some Algorithms for Algebraic Transformations of Functional Expressions. J. Comput. Math. and Comput. Mach., No.3, Kharkov, 1972, pp. 55-57.

3. I.R.Akselrod, L.F.Belous. Input Language for Automatic Programming System SIRIUS. In: "Automatization of Programming", No.3, Kiev, 1967.

4. I.R.Akselrod, L.F.Belous. Input Language for Automatic Programming System SIRIUS. Kharkov University, 1969.

5. I.R.Akselrod, L.F.Belous. On Reprocessing Literal-Analytical Information by Computer. J. Kibernetika, No.6, 1966.

6. I.R.Akselrod, L.F.Belous. On Symbolic Manipulation in Conversational Programming System SIRIUS. In: "Proc. 2-nd All Union Conference on Programming", Section H, Novosibirsk,1970.

7. I.R.Akselrod. Computation of Expressions in SIRIUS-System. In: "Automatization of Programming", No.2, Kiev, 1969.

8. I.R.Akselrod, L.F.Belous. Recursive Programs Organization in SIRIUS-System. In: "Automatization of Programming", No.3, Kiev, 1969.

9. I.R.Akselrod. On Syntax Analysis in SIRIUS-System. In: "Automatization of Programming", No.1, Kiev, 1969.

10. E.A.Arays, A.Shutenkov. Solution of Linear Algebra Problems in AVTO-ANALITIK-System of programming and Automatic Design. Tomsk, Tomsk University, 1971, pp. 191-196.

11. E.A.Arays, A.Shutenkov, G.V.Sibiriakov. Interpretation System for Solution of Large Problems. Issues of programming and Automatic Design, Tomsk, Tomsk University, 1971.

12. E.A.Arays, G.V.Sibiriakov. The AVTO-ANALITIK Programming System. J. Comput. Math. and Comput. Mach., No.3, Kharkov, 1972.

13. E.A.Arays, G.V.Sibiriakov. AVTO-ANALITIK. Novosibirsk University, 1973.

14. E.A.Arays, A.Shutenkov. The Realization of External Form with Cartan Method. Dokl. Akad. Nauk SSSR, 1974, 214, No.4, pp. 737-738.

15. I.O.Babaev. Some Extention of FORTRAN for Solving Celestial Mechanics Problems. Proc. 5-th Conf. on Math. and Mech., Tomsk, v.2, pp.145-146.

16. M.M.Bexhanova. On Some Aspects of Symbolic Manipulations. J. Comput. Math. and Comput. Mach., No.3, Kharkov, 1972, p. 60.

17. M.M.Bexhanova, I.V.Pottosin. Purpose of Difprocessor and its Input Language. Report of Programming Depart. of the Computing Center, Siberian Branch, USSR, Novosibirsk, 1966.

18. M.M.Bexhanova, K.I.Kostiukova, G.A.Plotnikova, I.V.Pottosin. Outline of Difprocessor Algorithms. Report of Programming Department of the Computing Center, Siberian Branch, USSR, Novosibirsk, 1966.

19. M.M.Bexhanova, V.L.Katkov, I.V.Pottosin. Researchs on Symbolic Manipulation in the Computing Center, Siberian Branch, Academy of Sciences, USSR. J. Comput. Math. and Comput. Mach., No.3, Kharkov, 1972, p. 21.

20. L.F.Belous, I.R.Akselrod. On Realization of SIRIUS Automatic Programming System. In: "Automatization of Programming", No.3, Kiev, 1967.

21. L.F.Belous. Analytical Differentiation in SIRIUS-System. In: "Automatization of Programming", No. 2, Kiev, 1969.

22. L.F.Belous. Dynamic Storage Allocation in SIRIUS-System. In: "Automatization of Programming", No. 2, Kiev, 1969.

23. Yu.V.Blagoveshchensky, V.G.Bondarchuk, Yu.S.Fishman. On Efficiency of Problem Solving Analytical Methods by Computer. In: "Issues of Accuracy and Efficiency of Comput. Algorithms", Proc. of Symposium, v. 5, Kiev, 1969.

24. Yu.V.Blagoveshchensky, Yu.S.Fishman, V.A. Shcherbakov. The Program for Analytical Solving of Nonlinear Ascillation Equations on MIR-2 Computer with ANALITIK Input Language. J. Kibernetika, No.6, Kiev, 1971.

25. V.G.Bondarchuk, S.V.Pogrebinsky. On Basic Principles of ANALITIK-Language Implementation. In: "Theory of Automata", No.2, Kiev, 1968.

26. V.G.Bondarchuk, Yu.S.Fishman. Integration Algorithms on ANALITIK-Language. J. Kibernetika, No. 4, 1968.

27. V.A.Brumberg. Celestial Mechanics Methods for Literal Manipulations. Tomsk University, 1974.

28. V.A.Brumberg, L.A.Isakovich. AMS-System for Analytical Manipulations of Poisson Series on Computer. Celestial Mechanics Algorithms, No.1, Theoretical Astronomy Institute, Leningrad, 1974.

29. A.V.Vasilieva. ALITA-System for Analytical Manipulations of Poisson Series on Computer. Celestial Mechanics Algorithms, No.7, Theoretical Astronomy Institute, Leningrad, 1975.

30. L.I.Goncharova. A Contribution Toward the Problem of General Organization of Symbolic Manipulation Systems. J. Comput. Math. and Comput. Mach., No.3, Kharkov, 1972, p. 62.

31. V.P.Gerdt. On Application of Symbol Manipulation Systems for Feinman Integrals Computation (Review). In Proc. "International Conference on Programming and Mathematical Methods for Solving Physical Problems", Dubna, 20-23 Sept. 1977, IINI, D10, 11-11264 Dubna, 1978.

32. V.P.Gerdt, O.V.Tarasov, D.V.Shirkov. Analytical Computations in Physics and Mathematics. Preprint IINI P2-11547, Dubna, 1978.

33. V.M.Glushkov, V.G.Bondarchuk, T.A.Grinchenko, A.A.Dorodnitsyna, V.P.Klimenko, A.A.Latichevsky, S.B.Pogrebinsky, A.A.Stogny, Yu.S.Fishman. ANALITIK (Algorithmic Language for Description of Computation Processes with algebraic manipulation). J. Kibernetika,No.3, 1971.

34. V.M.Glushkov, T.A.Grinchenko, A.A.Dorodnitsyna, A.M.Drakh, Yu.V.Kapitonova, V.P.Klimenko, L.M.Kress, A.A.Latichevsky, S.B.Progrebinsky, A.A.Stogny, Yu.S.Fishman, N.P.Tsariuk. ANALITIK-74 Algorithmic Language (Informational Part) Preprint 77/27, Institute of Cybernetics, Kiev, 1977.

35. V.M.Glushkov, T.A.Grinchenko, A.A.Dorodnitsyna, A.M.Drah, Yu.V.Kapitonova, V.P.Klimenko, L.M.Kress, A.A.Latichevsky, S.B.Pogrebinsky, A.A.Stogny, Yu.S.Fishman, N.P.Tsariuk. ANALITIK-74, J. Kibernetika, No. 5, 1978, pp. 114-147.

36. T.A.Grinchenko. Internal Representation and Analytical Expression Computations : "Theory of Automata", No. 2, Kiev, 1968.

37. T.A.Grinchenko, A.A.Dorodnitsyna, V.P.Klimenko, Yu.S.Fishman. Symbolic Manipulation Computer System for Engineering Calculations, MIR-2. J. Comput. Math. and Comput. Mach., No. 3, Kharkov, 1972, p. 26.

38. T.A.Grinchenko. Construction Principles and Computer Realization of APPLY-Operator. In: "Theory of Automata", No. 2, Kiev, 1968.

39. T.A.Grinchenko. Computer Realization Principles of Symbol Manipulation. J. Kibernetika, No. 1, 1968.

40. S.A.Ivanova. Language and Translator for Algebraic Manipulations with Polynomial from several Variables. Latvia annual, 1974, 17, pp. 230-237.

41. N.A.Kalinina. Some Algorithms and Methods in Symbol Manipulation Systems. Report of Programming Dept. of the Comput. Center, Siberian, USSR, Novosibirsk, 1972.

42. N.A.Kalinina. The ANALITIK System Program. J. Comput. Math. and Comput. Mach., No. 3, Kharkov, 1972, p. 33.

43. N.A.Kalinina. Symbol Manipulation Systems (Review). Report of Computing Center, Siberian Branch, USSR, Novosibirsk, 1972.

44. N.A.Kalinina. On Hierarchy in Symbol Manipulation Systems. J. Comput. Math. and Comput. Mach., No. 3, Kharkov, 1972, p. 70.

45. N.A.Kalinina. Some Aspects of Design of Symbol Manipulation Systems. In: "System and Theoretical Programming", Computing Center, Siberian Branch, URRS, Novosibirsk, 1973, pp. 103-123.

46. N.A.Kalinina. The Structure and Semantic Features of Symbol Manipulation Language. In: "Programming Problems", Computing Center, Siberian Branch, URRS, Novosibirsk, 1976, pp.8-34.

47. N.A.Kalinina, I.V.Pottosin. Architecture of General Purpose Symbol Manipulation Systems: Adaptability to Solved Problems and Interface with Programming System. In: "Theory and Practice of System Programming", Computing Center, Siberian Branch, URRS, Novosibirsk, 1977, pp. 5-12.

48. N.A.Kalinina. KANVA-Complex Analytical Evaluator. Organization and Key Algorithms. In: "Theory and Practice of System Programming", Computing Center, Siberian Branch, USSR, Novosibirsk, 1977, pp. 13-21.

49. L.V.Kantorovich. On Numeral and Analytical Computations on Computer. News of Academy Science of Armenia SSR, Section Phys. Math., 1957, No. 2.

50. L.V.Kantorovich. On a Mathematical Symbolism Suitable for Carring out Calculations on Computers. Dokl. Akad. Nauk SSSR, 1957, n.113, No. 4.

51. L.V.Katkov, N.I.Kostiukova. The Processor KINO. In: "Dynamic of Continuous Medium", Novosibirsk, 1969, v. 1.

52. V.L.Katkov, N.I.Kostiukova. The KINO System for Construction of Analytical Solutions of differential Equations on Computer. Proc. 1st All Union Conference on Programming, Kiev,1968.

V.L.Kotkov, N.I.Kostiukova. Calculations of Group on Computer. "Some Problems of Computing and Applied Mathematics", Novosibirsk, 1975, pp. 257-267.

V.L.Kotkov, M.D.Popov. Using Computer BESM-6 for Calculations of Group Admitted by Differential Equations System. Intern. Symposium "Theoretical-Group Methods in Mechanics",1978, p. 17.

V.P.Klimenko, S.B.Pogrebinsky, Yu.S.Fishman. A Contribution Toward the Problem Recognition of Functional Properties of Analytical Expressions on MIR-2 Computer. J. Kibernetika, No. 2, 1973, pp. 43-53.

N.I.Kostiukova. The Processor PASSIV. J.Comput. Math. and Comput. Mach., No. 3, Kharkov, 1972, p. 38.

G.P.Kozhevnikova. On Efficient Realization of Algorithmic Languages for Analytical Transformations. Proceedings of Symposium "Language Theory and Methods of Constructing Programming System", Kiev-Alushta, 1972, pp. 338-345.

G.P.Kozhevnikova. Computational Complexity of the Procedure "Compare" and "Differentiate" with Respect to the Languages of Lukashevicz and Kantorovich. J. Comput. Math. and Comput. Mach., No. 3, Kharkov, 1972, pp. 64-65.

G.P.Kozhevnikova. On the Estimation of Efficiency of Symbol Manipulations. J. Cont. Systems and Machines, Kiev, No. 1, 1974.

G.P.Kozhevnikova, A.A.Stogny. Representation of Analytical Expressions under Algebraic Manipulations Performing on Computer. J. Kibernetika, No. 4, Kiev, 1975.

L.T.Petrova. On Execution of Algebraic Manipulations on Computer. High School Reports. Mathematics, No. 5, 1958, pp. 95-104.

L.T.Petrova. Some Applications of Scheme Symbolism. J. Comput. Math. and Comput. Phys., v. 1, No. 3, M. 1961, pp. 513-522.

L.T.Petrova, I.A.Platunova. Realization of Calculations in Source Lists Class on Computer. Proc. Math. Inst. Ac. Sci. USSR, v.66, 1962.

G.A.Plotnikova. Analytical Transformations in Difprocessor. Report of Programming Dept. of the Computing Center, Siberian Branch, USSR, Novosibirsk, 1966.

S.B.Pogrebinsky, Yu.S.Fishman. Dialog System for Analytical Solution of Some Problems of Algebra. Proc. of Symposium on "Language Theory and Methods of Constructing Programming System", Kiev-Alushta, 1972, pp.329-337.

66. E.N.Pashkin. Analytical Differentiation on Computer. In: "Computing Methods and Programming", v. 9, Moscow State Univ., 1967.

67. V.I.Skripnichenko. Operations with Literal Decompositions on Computer Results of Science and Tech. Astronomy, v. 11, p. 131, All Union Inst. Sci. and Tech. Information, Moscow,1975.

68. T.N.Smirnova. Polynomial PRORAB and Carrying out Analytical Transformations on Computer. Ph. D. Thesis, Leningrad, 1963.

69. T.N.Smirnova. Carrying out Analytical Transformations for on M-20 Computer with PRORAB-Program. Leningrad, Nauka, 1967.

70. V.V.Tumasonis. The System of Equivalent Transformations Expressions. J. Comput. Math. and Math. Phys., v. 11, No. 5, 1971, pp. 1272-1281.

71. V.V.Tumasonis. ALDA-Conversational System of Equivalent Transformations on Expressions. J. Comput. Math. and Comput. Mach., No. 3, Kharkov, 1972, pp. 52-54.

72. V.F.Turchin, V.V.Serdobolsky. REFAL Languages and its Application for Algebraic Expressions Transformations. J. Kibernetika, No.3, Kiev, 1969.

73. Yu.S.Fishman. Integration of Functions by Computer Performing Analytical Transformations. In: "Theory of Automaton", v. 2, Kiev, 1968.

74. Yu.S.Fishman, A.T.Kotziuba. The Realization of General-Purpose Symbol Integration Program on Computer. In: "Issue of Accuracy and Efficiency of Comput. Algorithms", v. 5, Kiev, 1969.

75. M.A.Chubarov. Polynomial Assembler. J. Comput. Math. and Comput. Mach., No. 3, Kharkov, 1972, pp. 42-44.

76. M.A.Chubarov. The ISP Interpretation System for Polynomial Manipulations. In: "Digital and Comput. Tech.", v. 5, Moscow, 1969.

77. V.A.Shurygin, N.N.Yanenko. On Realization of Algebraic Differential Algorithms by Computer. Problems of Cibernetics, v. 6, 1961.

78. D.Mordukhai-Boltovskoi, (trans. by Boris Korenblum and Myra Prelle), "A General Investigation of Integration in Finite Form of Differential Equations of the First Order"; Trans. in ACM SIGSAM Bulletin, v. 15, No. 2, pp. 20-32, May 1981.

APPENDIX A-4

AN EXAMPLE OF SOVIET WORK IN MACHINE SYMBOLIC MANIPULATION

KRUPNIKOV, ERNST DAVIDOVICH
POSTE RESTANTE
NOVOSIBIRSK 90
630090, SOVIET UNION

DOCTOR B.DAVID SAUNDERS
"SIGSAM BULLETIN" EDITOR
DEPARTMENT OF MATHEMATICAL SCIENCES
RENSSELAER POLYTECHNIC INSTITUTE
TROY, NEW YORK 12181, USA

March 5, 1982

DEAR DAVID,

THE INTEREST IN USING THE THEORY OF GENERALIZED HYPERGEOMETRIC FUNCTIONS IN COMPUTER ALGEBRA ALGORITHMS IS INCREASING. I OFFER A PROBLEM FOR CONSIDERATION BY READERS OF YOUR RESPECTABLE BULLETIN.

WHAT IS A MINIMAL SET OF IDENTITIES APPLICATION OF WHICH FACTORIZE EACH OF THE FOLLOWING TEN FUNCTIONS INTO THE PRODUCT OF TWO HYPER-GEOMETRIC FUNCTIONS WITH LEAST NUMBER OF PARAMETERS?

$${}_1F_1\left(\begin{matrix}1/2\\1/2-a,1/2+a\end{matrix};z\right), \qquad {}_1F_3\left(\begin{matrix}a-b,a-b\\a,1/2+a,2a\end{matrix};z\right),$$

$${}_1F_1\left(\begin{matrix}1/2\\1-a,1+a\end{matrix};z\right), \qquad {}_1F_3\left(\begin{matrix}a,1/2-a\\1/2+a-b,1/2+a-b,2a\end{matrix};z\right),$$

$${}_1F_2\left(\begin{matrix}a\\1/2-a,-1+2a\end{matrix};z\right), \qquad {}_3F_2\left(\begin{matrix}a,a-b,a-b\\1/2+a,2a\end{matrix};z\right),$$

$${}_1F_2\left(\begin{matrix}a\\1/2+a,2a\end{matrix};z\right), \qquad {}_3F_2\left(\begin{matrix}a,a-b,a-b\\1/2+a,-1-2a\end{matrix};z\right),$$

$${}_1F_2\left(\begin{matrix}a\\1/2+a,1-2a\end{matrix};z\right), \qquad {}_3F_2\left(\begin{matrix}a,1/2-a-b,1/2+a-b\\1/2+a,2a\end{matrix};z\right).$$

I BELIEVE IT WILL BE VERY INSTRUCTIVE TO BRING THE CORRESPONDING PROGRAMS IN REDUCE-2, MACSYMA, ANALITIK-71, AND OTHER HIGH-LEVEL LANGUAGES INTO COMPARISION. IF YOU WISH I SHALL IMMEDIATELY AIRMAIL MY PROGRAM IN ANALITIK-71 TO YOU.

HAVE YOU RECEIVED MY LETTER OF JANUARY 26?

WITH WARMEST REGARDS,

ERNST OF NOVOSIBIRSK
ALGEBRA PROGRAMMER

CARBON COPIES TO PROFESSORS ANTHONY CHARLES HEARN AND RICHARD J.FATEMAN
ENCLOSURE: TEST RUN OUTPUT WITH MY COMMENTS.

compiler · MMP-2
algorithmic language: ANALYTIK-71
turnaround time : 83 sec.
algebra programmer : Ernst Kniprikov
Novosibirsk, March 1982

F(U(1/2)=L(1/2-(A))=L(1/2+A),Z)=0

-1+2=F(L(1/2-(A)),1/4=Z)=F(L(1/2+A),1/4=Z)

$${}_2F_1\left(\begin{matrix}A,1/2\\1/2-A,1/2+A\end{matrix};Z\right)=-1+2\,{}_2F_1\left(\begin{matrix}-A,1/2\\1/2-A\end{matrix};Z/4\right){}_2F_1\left(\begin{matrix}A,1/2\\1/2+A\end{matrix};Z/4\right).$$

F(U(1/2)=L(1-(A))=L(1+A),Z)=0

F(L(1-(A)),1/4=Z)=F(L(1+A),1/4=Z)

$${}_2F_1\left(\begin{matrix}A,A\\1-A,1+A\end{matrix};Z\right)={}_2F_1\left(\begin{matrix}-A\\1-A\end{matrix};Z/4\right){}_2F_1\left(\begin{matrix}A\\1+A\end{matrix};Z/4\right);$$

F(U(A)=L(1/2+A)=L(-1+2=A),Z)=0

F(L(1/2+A),1/4=Z)=F(L(-1/2+A),1/4=Z)

$${}_2F_1\left(\begin{matrix}A\\1/2+A,-1+2A\end{matrix};Z\right)={}_2F_1\left(\begin{matrix}\\1/2+A\end{matrix};Z/4\right){}_2F_1\left(\begin{matrix}\\-1/2+A\end{matrix};Z/4\right);$$

F(U(A)=L(1/2+A)=L(2=A),Z)=0

F(L(1/2+A),1/4=Z)12

$${}_2F_1\left(\begin{matrix}A\\1/2+A,2A\end{matrix};Z\right)=\left[{}_2F_1\left(\begin{matrix}\\1/2+A\end{matrix};Z/4\right)\right]^2;$$

F(U(A)=L(1/2+A)=L(1+2=A),Z)=0

F(U(A)=L(1+2=A),2=ZI(1/2))=F(U(A)=L(1+2=A),-2=ZI(1/2))

$${}_2F_1\left(\begin{matrix}A\\1/2+A,1+2A\end{matrix};Z\right)={}_2F_1\left(\begin{matrix}A\\1+2A\end{matrix};2\sqrt{Z}\right){}_2F_1\left(\begin{matrix}A\\1+2A\end{matrix};-2\sqrt{Z}\right);$$

F(U(A-(B))=U(A+B)=L(A)=L(1/2+A)=L(2=A),Z)=0

F(U(A-(B))=L(2=A),2=ZI(1/2))=F(U(A-(B))=L(2=A),-2=ZI(1/2))

$${}_2F_1\left(\begin{matrix}A-B,A+B\\A,1/2+A,2A\end{matrix};Z\right)={}_2F_1\left(\begin{matrix}A-B\\2A\end{matrix};2\sqrt{Z}\right){}_2F_1\left(\begin{matrix}A-B\\2A\end{matrix};-2\sqrt{Z}\right);$$

F(U(A)=U(1/2+A)=L(1/2+A-(B))=L(1/2+A+B)=L(2=A),Z)=0

F(L(1/2+A-(B)),1/4=Z)=F(L(1/2+A+B),1/4=Z)

$${}_2F_1\left(\begin{matrix}A,1/2+A\\1/2+A-B,1/2+A+B,2A\end{matrix};Z\right)={}_2F_1\left(\begin{matrix}\\1/2+A-B\end{matrix};Z/4\right){}_2F_1\left(\begin{matrix}\\1/2+A+B\end{matrix};Z/4\right);$$

F(U(A)=U(A-(B))=U(A+B)=L(1/2+A)=L(2=A),Z)=0

F(U(1/2=A+(-1)/2=B)=U(1/2=A+1/2=B)=L(1/2+A),Z)12

$${}_2F_1\left(\begin{matrix}A,A-B,A+B\\1/2+A,2A\end{matrix};Z\right)=\left[{}_2F_1\left(\begin{matrix}(A-B)/2,(A+B)/2\\1/2+A\end{matrix};Z\right)\right]^2;$$

F(U(A)=U(A-(B))=U(A+B)=L(1/2+A)=L(-1+2=A),Z)=0

F(U(1/2=A+(-1)/2=B)=U(1/2=A+1/2=B)=L(1/2+A),Z)=F(U(1/2=A+(-1)/2=B)=U(1/2=A+1/2=B)=L(-1/2+A),Z)

$${}_2F_1\left(\begin{matrix}A,A-B,A+B\\1/2+A,-1+2A\end{matrix};Z\right)={}_2F_1\left(\begin{matrix}(A-B)/2,(A+B)/2\\1/2+A\end{matrix};Z\right){}_2F_1\left(\begin{matrix}(A-B)/2,(A+B)/2\\-1/2+A\end{matrix};Z\right);$$

F(U(A)=U(1/2+A-(B))=U(1/2+A+B)=L(1/2+A)=L(2=A),Z)=0

F(U(3/4+1/2=A+(-1)/2=B)=U(1/4+1/2=A+1/2=B)=L(1/2+A),Z)=F(U(-1/4+1/2=A+(-1)/2=B)=U(1/4+1/2=A+1/2=B)=L(1/2+A),Z)0

$${}_2F_1\left(\begin{matrix}A,1/2+A-B,1/2+A+B\\1/2+A,2A\end{matrix};Z\right)={}_2F_1\left(\begin{matrix}(1/2+A-B)/2,(1/2+A+B)/2\\1/2+A\end{matrix};Z\right){}_2F_1\left(\begin{matrix}(1-1/2+A-B)/2,(1/2+A+B)/2\\1/2+A\end{matrix};Z\right);$$

DISTRIBUTION LIST

Dr. Marv Atkins
Deputy Director, Science & Tech.
Defense Nuclear Agency
Washington, D.C. 20305

Dr. Robert Cooper [2]
Director, DARPA
1400 Wilson Boulevard
Arlington, VA 22209

Defense Technical Information [2]
Center
Cameron Station
Alexandria, VA 22314

The Honorable Richard DeLauer
Under Secretary of Defense (R&E)
Office of the Secretary of
Defense
The Pentagon, Room 3E1006
Washington, D.C. 20301

Director [2]
National Security Agency
Fort Meade, MD 20755
ATTN: Mr. Richard Foss, A05

CAPT Craig E. Dorman
Department of the Navy, OP-095T
The Pentagon, Room 5D576
Washington, D.C. 20350

CDR Timothy Dugan
NFOIO Detachment, Suitland
4301 Suitland Road
Washington, D.C. 20390

Dr. Larry Gershwin
NIO for Strategic Programs
P.O. Box 1925
Washington, D.C. 20505

Dr. S. William Gouse, W300
Vice President and General
Manager
The MITRE Corporation
1820 Dolley Madison Blvd.
McLean, VA 22102

Dr. Edward Harper
SSBN, Security Director
OP-021T
The Pentagon, Room 4D534
Washington, D.C. 20350

Mr. R. Evan Hineman
Deputy Director for Science
& Technology
P.O. Box 1925
Washington, D.C 20505

Mr. Ben Hunter [2]
CIA/DDS&T
P.O. Box 1925
Washington, D.C. 20505

The MITRE Corporation [25]
1820 Dolley Madison Blvd.
McLean, VA 22102
ATTN: JASON Library, W002

Mr. Jack Kalish
Deputy Program Manager
The Pentagon
Washington, D.C. 20301

Mr. John F. Kaufmann
Dep. Dir. for Program Analysis
Office of Energy Research, ER-31
Room F326
U.S. Department of Energy
Washington, D.C. 20545

Dr. George A. Keyworth
Director
Office of Science & Tech. Policy
Old Executive Office Building
17th & Pennsylvania, N.W.
Washington, D.C. 20500

MAJ GEN Donald L. Lamberson
Assistant Deputy Chief of Staff
(RD&A) HQ USAF/RD
Washington, D.C. 20330

Dr. Donald M. LeVine, W385 [3]
The MITRE Corporation
1820 Dolley Madison Blvd.
McLean, VA 22102

Mr. V. Larry Lynn
Deputy Director, DARPA
1400 Wilson Boulevard
Arlington, VA 22209

Dr. Joseph Mangano [2]
DARPA/DEO
9th floor, Directed Energy Office
1400 Wilson Boulevard
Arlington, VA 22209

Mr. John McMahon
Dep. Dir. Cen. Intelligence
P.O. Box 1925
Washington, D.C. 20505

Director
National Security Agency
Fort Meade, MD 20755
ATTN: William Mehuron, DDR

Dr. Marvin Moss
Technical Director
Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217

Dr. Julian Nall [2]
P.O. Box 1925
Washington, D.C. 20505

Director
National Security Agency
Fort Meade, MD 20755
ATTN: Mr. Edward P. Neuburg
DDR-FANX 3

Prof. William A. Nierenberg
Scripps Institution of
Oceanography
University of California, S.D.
La Jolla, CA 92093

Mr. Alan J. Roberts
Vice President & General Manager
Washington C^3 Operations
The MITRE Corporation
1820 Dolley Madison Boulevard
Box 208
McLean, VA 22102

Los Alamos Scientific Laboratory
ATTN: C. Paul Robinson
P.O. Box 1000
Los Alamos, NM 87545

Mr. Richard Ross [2]
P.O. Box 1925
Washington, D.C. 20505

Dr. Phil Selwyn
Technical Director
Office of Naval Technology
800 N. Quincy Street
Arlington, VA 22217

Dr. Eugene Sevin [2]
Defense Nuclear Agency
Washington, D.C. 20305

Dr. Joel A. Snow [2]
Senior Technical Advisor
Office of Energy Research
U.S. DOE, M.S. E084
Washington, D.C. 20585

Mr. Alexander J. Tachmindji
Senior Vice President & General
Manager
The MITRE Corporation
P.O. Box 208
Bedford, MA 01730

Dr. Vigdor Teplitz
ACDA
320 21st Street, N.W.
Room 4484
Washington, D.C. 20451

Dr. Al Trivelpiece
Director, Office of Energy
Research, U.S. DOE
M.S. 6E084
Washington, D.C. 20585

Mr. James P. Wade, Jr.
Prin. Dep. Under Secretary of
Defense for R&E
The Pentagon, Room 3E1014
Washington, D.C. 20301

Mr. Leo Young
OUSDRE (R&AT)
The Pentagon, Room 3D1067
Washington, D.C. 20301