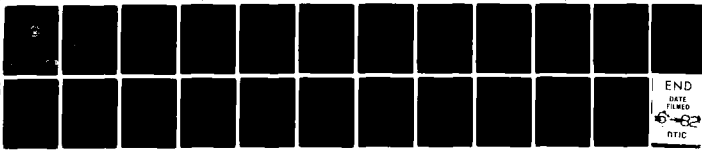


AD-A114 462 NAVAL POSTGRADUATE SCHOOL MONTEREY CA  
OVERVIEW OF RELATIONAL PROGRAMMING. (U)  
NOV 81 B J MACLENNAN  
UNCLASSIFIED NPSSB-81-817

F/6 12/1

NL

10/1  
12/1



END  
DATE  
FILMED  
6-82  
NTIC

①

NPS52-81-017

# NAVAL POSTGRADUATE SCHOOL

Monterey, California



ADA 114462

Overview of Relational Programming  
Bruce J. MacLennan  
November 1981

DTIC FILE COPY

Approved for public release; distribution unlimited

Prepared for:  
Chief of Naval Research  
Arlington, Virginia 22217

DTIC  
ELECTE  
MAY 17 1982  
S D E

82 05 17 173

NAVAL POSTGRADUATE SCHOOL  
Monterey, California


Rear Admiral J. J. Ekelund  
Superintendent

D. A. Schrady  
Acting Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Reproduction of all or part of this report is authorized.

This report was prepared by:

  
BRUCE J. MacLENNAN  
Assistant Professor  
of Computer Science

Reviewed by:

  
GORDON W. BRADLEY, Chairman  
Department of Computer Science

Released by:

  
WILLIAM M. TOLLES  
Dean of Research

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-81-017	2. GOVT ACCESSION NO. AD-A22 4462	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Overview of Relational Programming	5. TYPE OF REPORT & PERIOD COVERED Technical Report	
	6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) Bruce J. MacLennan	8. CONTRACT OR GRANT NUMBER(s)	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N; RR000-01-10 N0001482WR20043	
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940	12. REPORT DATE November 1981	
	13. NUMBER OF PAGES	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED	
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
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) relational programming, functional programming, relational algebra, relations, relational calculus, applicative languages, combinators, very-high-level languages, logic programming		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report provides a brief overview of the method of programming in a relational calculus. This is a style of programming in which entire relations are manipulated as data, and in which the program itself is represented as a relation. This report describes the use of the relational operators to manipulate both data and functions, and introduces an improved notation for relational programming.		

DD FORM 1473/

1 JAN 73

EDITION OF 1 NOV 66 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

# OVERVIEW OF RELATIONAL PROGRAMMING\*

B. J. MacLennan

Naval Postgraduate School

Monterey, CA 93940

November 10, 1981

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/ _____	
Availability Codes	
Dist	Avail and/or Special
A	



## 1. Introduction

Relational programming is a method of programming based on the use of a relational calculus. We begin by explaining why we have chosen to investigate relational programming.

We began investigating relations to try to find a high level way of manipulating complex data structures. Languages such as APL are very successful in the manipulation of vectors and matrices, and languages such as Snobol are useful in the manipulation of strings. Unfortunately, these are both examples of linear data structures, and many problems in computer science require non-linear data structures, such as trees and networks. Proposed extensions to APL and Snobol to handle non-linear data structures have not been very successful.

It is well known that almost any data structure can be described by a relation. In effect, then, any operation on relations can be thought of as an operation on data structures. Therefore, it seemed that the high level relational operators

\* The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

provided by a relational calculus might provide a source of high level operations for manipulating non-linear data structures. This has proved to be the case.

Backus [1] has described the advantages of programming with functionals, that is, with functions which operate on other functions. Functionals allow the high level combination of programs to yield new programs. Now notice, since every function is a relation, every relational operator is in effect a functional. Therefore, the same set of operators that are used for manipulating data can also be used for manipulating programs. The result is great economy of linguistic mechanism in combination with powerful means of manipulating both code and data.

A final goal in the development of relational programming has been the attempt to find a means of programming that permits practical proofs of real programs. The fact that relations are mathematically tractable, and that there is an well-developed theory of relations, has encouraged this study.

## 2. Background

Relational programming has been based on naive set theory. This is the set theory that most people are exposed to in every mathematics class from freshman calculus on. It is hoped that by basing this programming method on a simple and well-known mathematical basis, it will be more understandable to people without an extensive mathematical background.

There are three sorts of objects with which relational programs deal:

- \* Individuals
- \* Sets
- \* Binary Relations

The individuals are the indivisible data values with which we compute. Typically they will include integers, real numbers, characters, and Boolean values. Both the sets and the relations may be either finite or infinite; the latter being represented on a finite computer using intensional methods (discussed later). Both the sets and relations are typeless, which means that there are no restrictions on what sets or relations can be members of other sets and relations. Axiomatizations of set theory often included intricate type systems (such as Russell's "Ramified Type Theory") to prevent contradictions. However, as is discussed in [3], there are other methods of preventing contradictions that do not depend on elaborate type systems. Some of the factors that have convinced us that a typeless system is more appropriate to programming are discussed in [3].

We use the notation  $x \in S$  to mean that  $x$  is a member of the set  $S$ , and  $xRy$  to mean that the pair  $\langle x,y \rangle$  is a member of the relation  $R$ . The functional notation  $Fx$  denotes the unique  $y$  (if it exists) such that  $xFy$ . In general spaces and the case of letters will be used to improve readability. Parentheses are used for grouping in the usual way.

### 3. Relations and Functions

#### 3.1 Functionals

Since every function is a relation, every operation on relations is also an operation on functions, i.e., a functional. In this section we will investigate several relational operators and show that they have useful functional interpretations.

The relative product operation on relations performs the composition of functions. That is,

$$f.g(x) = f(g(x))$$

We will sometimes also write this in its rightward form:

$$f;g(x) = g(f(x))$$

The union operation, when applied to functions, combines them. This is most useful when the functions have disjoint domains. For example,

$$f|g(x) = \begin{cases} f(x), & \text{if } x \in \text{dom } f \\ g(x), & \text{if } x \in \text{dom } g \end{cases}$$

(We write 'dom f' for the domain of f.)

If the functions do not have disjoint domains, the ordered union, or overlying operation,  $f/g$ , is often useful:

$$f/g(x) = \begin{cases} f(x), & \text{if } x \in \text{dom } f \\ g(x), & \text{otherwise} \end{cases}$$



That is, the pairs in  $f$  supercede the corresponding pairs in  $g$ .

The converse of a relation, when applied to a function, produces the inverse function. That is,

$$x = f^{-1}(y) \quad \text{iff} \quad y = f(x)$$

Notice that this operation is always defined since a relation always has a converse. Of course, the inverse of a function will be a function only if the original function was one-to-one. Nevertheless, because the converse is always defined it satisfies simpler properties.

The restrictions are useful operations on relations; they define subrelations of the given relation whose members satisfy a given property. When applied to functions, the restriction operations limit the domain, range, or both the domain and range of a function. They are defined:

$$y = s \rightarrow f(x) \quad \text{iff} \quad y = f(x) \quad \text{and} \quad x \in s$$

$$y = f \leftarrow s(x) \quad \text{iff} \quad y = f(x) \quad \text{and} \quad y \in s$$

$$f \langle \rangle s = s \rightarrow f \leftarrow s$$

where  $s$  is any set. As will be shown later, the restriction operations are often useful for constructing conditionals.

The image operation, when applied to a function, gives the image of a set under that function. This is defined:

$$\text{img } f(S) = \{ y \mid \exists x \in S: y = f(x) \}$$

The parallel application operation applies functions to corresponding elements of a sequence:

$$f||g(x,y) = (f x, g y)$$

The dual application or construction operation applies several functions to one argument, returning a sequence of the results:

$$f#g(x) = (f x, g x)$$

This is equivalent to Backus' construction operation,  $[f,g]$ .

The closure operators effectively iterate the application of a function. The transitive and non-transitive closures are defined:

$$\begin{aligned} f^* &= f^0 | f^1 | f^2 | \dots \\ f^+ &= f^1 | f^2 | f^3 | \dots \end{aligned}$$

where  $f^n$  means the composition of  $f$  with itself  $n$  times. Thus the result of  $f^+(x)$  is whichever of  $f^1(x)$ ,  $f^2(x)$ , ... are defined. (If more than one are defined we can use the restriction operations to pick the one we want.)

### 3.2 Control Structures

So far in the development of relational programming there has been no need to introduce control structures in the conventional sense. This is because the relational operators are adequate to express most control flow situations. For example, suppose we wish to apply  $f(x)$  if  $x$  satisfies  $s$  and  $g(x)$  otherwise; this is

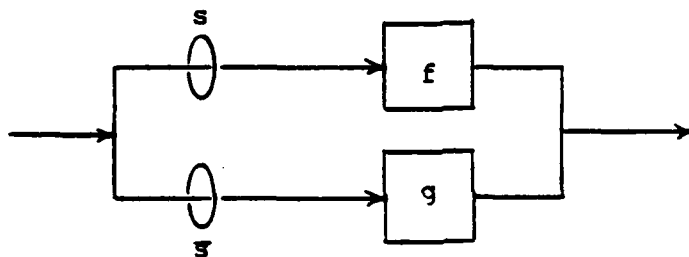
effectively a conditional construction. It can be written this way using the relational operators:

$$s \rightarrow f / g$$

This is equivalent to

$$(s \rightarrow f) ; (\text{non } s \rightarrow g)$$

('non s' returns the complement of the set s.) In other words, the domain of f is restricted to those things that do satisfy s and the domain of g is restricted to those things that don't satisfy s. This can be diagrammed like this:



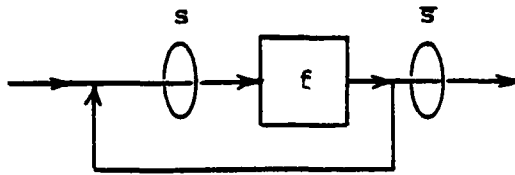
The  $s$  and  $\bar{s}$  can be thought of as filters on the inputs of  $f$  and  $g$ . Since they are mutually exclusive, it is guaranteed that at most one value will be produced for each value put in.

The relational equivalent of loops are constructed from the closure and restriction operators. Consider this function:

$$(s \rightarrow f)^+ \leftarrow \text{non } s$$

The application of  $s \rightarrow f$  will be iterated one or more times, which means that  $f$  will be applied one or more times, as long as

its input satisfies  $s$ . An output from this process is allowed only if it doesn't satisfy  $s$ . We can diagram this function:



This is the equivalent of a "repeat until" loop in Pascal. Similar expressions loop zero or more times, like a Pascal "while" loop.

### 3.3 Relations Obey Simple Laws

One of the reasons we have investigated relational programming is that it simplifies reasoning about programs. This is because relations obey many simple laws. For example,

$$(f.g)^{-1} = g^{-1}.f^{-1}$$

is true for all relations; it is only true for functions that are one-to-one.

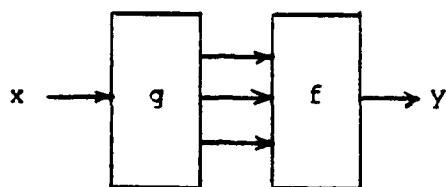
### 3.4 Multiple-Valued Functions

A relation can be thought of as a multiple-valued function. That is, there may be several  $y$  such that  $xFy$ . Functional approaches to programming often exclude multiple-valued functions and non-deterministic functions, even though these are often benign. Relational programming deals naturally with multiple valued functions. For example, suppose that  $g(x)$  is multiple-valued, e.g.,

there are three values, a, b, and c, such that  $x_g a$ ,  $x_g b$ , and  $x_g c$ . Further suppose that the function  $f$  has the same value,  $y$ , on each of a, b, c. That is,  $y = f(a)$ ,  $y = f(b)$ , and  $y = f(c)$ . Then it is perfectly meaningful to write

$$y = f.g(x)$$

even though  $g$  is not single-valued at  $x$ . This can be visualized:



#### 4. Relations and Data

##### 4.1 Finite Functions

We will now turn to the representation of data by relations and the high-level data manipulation functions provided by the relational operators. Although there are several ways that data can be represented by relations, one of the simplest is by finite functions, i.e., functions containing a finite number of pairs. This representation is particularly suitable for arrays and records. For example,

$$x = A(i)$$

is the application of an array  $A$  to its index  $i$ . Similarly,

$$x = z(re)$$

is analogous to a field selection operation  $z.re$ , but in rela-

tional terms it is also just the application of a function to an argument.

The value of viewing data in this way is that it makes data structures amenable to the relational operators. For example, the converse operator inverts a structure.

$$i = A^{-1}(x)$$

returns the index of the array element whose value is  $x$ . If  $x$  occurs several times in  $A$  then  $A^{-1}$  is multiple-valued. We can get a set of all the indices where  $x$  occurs by taking the image:  $\text{img } A^{-1}(x)$ .

The relative product or composition operation can be used for many purposes, such as permuting arrays. If  $P$  is a permutation function (a bijection from the index set into itself), then  $A.P$  is the corresponding permutation of  $A$ . This operation can also be used for "cascading" data structures. For example, if 'address' is a table such that

$$a = \text{address}(n)$$

means that  $a$  is the address of the variable named  $n$ , and 'value' is a table such that

$$v = \text{value}(a)$$

means that  $v$  is the value contained by location  $a$ , then 'value.address' is a cascaded table such that

$$v = \text{value.address } (n)$$

means that  $v$  is the value of the variable named  $n$ .

The restriction operation can be used to define substructures. For example, suppose that  $M$  is a finite function representing a two-dimensional matrix:

$$x = M(i,j)$$

That is,  $M$  is a function that takes pairs of integers into the corresponding matrix elements. If  $I$  and  $J$  are index sets, the submatrix of  $M$  corresponding to these index values is just  $(IXJ) \rightarrow M$ , since this restricts the first and second indices of  $M$  to be in  $I$  and  $J$  respectively.

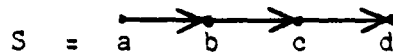
The union operation can be used to combine data structures. For example, if  $S$  and  $T$  are tables, then  $S|T$  is a table that contains the entries of both  $S$  and  $T$ . Also, if  $U$  and  $V$  are two arrays with consecutive index sets (which is not hard to arrange), then  $U|V$  is the catenation of  $U$  and  $V$ .

The overlaying operation  $U/V$  updates an array  $V$  according to the pairs in  $U$ . That is, if  $U/V(i) = U(i)$  if  $U(i)$  is defined, and  $U/V(i) = V(i)$  otherwise.

Finally, the image operation can be used for mass selections. For example, if  $A$  is an array and  $S$  is a set of indices, then  $\text{img } A(S)$  is the set of all elements of  $A$  selected by indices in  $S$ .

## 4.2 Sequences

Sequences and lists have a straight-forward representation as relations. If we draw the sequence of elements (a,b,c,d) like this:

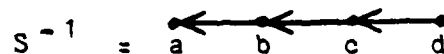


then you can see that this can be represented by the relation  $xSy$  that relates  $x$  to  $y$  just when there is an arrow from  $x$  to  $y$ . That is,

$$S = \{ \langle a,b \rangle, \langle b,c \rangle, \langle c,d \rangle \}$$

Next we consider the effect of the relational operators on such a sequence.

The converse of  $S$  is that relation  $S^{-1}$ , where  $yS^{-1}x$  if and only if  $xSy$ . The effect is to reverse the arrows:



so it can be seen that  $S^{-1}$  is just the reverse of  $S$ .

Like all relations, a sequence can be thought of as a function. The effect of functional application is to follow an arrow from one element of the sequence to another, e.g.,

$$c = S(b) \quad \text{and} \quad b = S^{-1}(c)$$

Of course,  $S^2$  goes two links:



$$d = S^2(b) \quad \text{and} \quad b = S^{-2}(d)$$

and so forth.

The restriction operation can be used to define subsequences of a given sequence. For example,  $S \langle \rangle P$  defines the subsequence of  $S$  all of whose elements satisfy the predicate  $P$ . That is, if  $P$  is the set of positive numbers, then this restriction has just the positive members of  $S$ .

The union operation can be used in various ways to combine sequences. For example, to concatenate the sequences  $S$  and  $T$  we can write

$$S \mid (\text{last } S, \text{ first } T) \mid T$$

This combines  $S$  and  $T$  with a third relation which is a sequence from the last element of  $S$  to the first element of  $T$ .

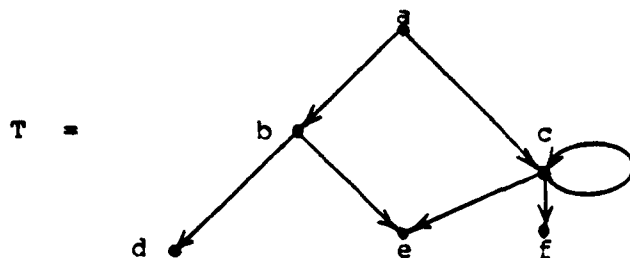
Finally, we can use the domain functions to find distinguished elements of a sequence. For example, the initial members of a sequence (of which there is exactly one) are those members that have an arrow leaving them, but not pointing at them. In other words the initial members are the elements of the domain that are not in the range:

$$\text{init}(S) = \text{dom}(S) - \text{dom}(S^{-1})$$

#### 4.3 General Data Structures

Since the manipulation of non-linear data structures was a major

reason for investigating relations, we would expect to find that the relational operators are useful. The same approach is used as for sequences. For example, the graph



is represented by the relation

$$T = \{ \langle a,b \rangle, \langle a,c \rangle, \langle b,d \rangle, \langle b,e \rangle, \langle c,e \rangle, \langle c,f \rangle, \langle c,c \rangle \}$$

Then, it is easy to see that the roots of this structure are just its initial members,  $\text{init}(T)$ , and the leaves are the initial members of the converse relation,  $\text{init}(T^{-1})$ . The latter are usually called terminal members.

Notice that  $T(n)$  follows an arrow from node  $n$ , which may be multiple-valued. For example,  $T(b)$  could be  $d$  or  $e$ . Therefore, it is better to ask for all the descendents of a node  $n$ , which is just the image of  $T$  applied to  $n$ :

$$\text{descendents}(T) = \text{img } T(n)$$

## 5. Higher Levels of Abstraction

The relational programming style is open ended and easily admits even higher levels of abstraction. Observe that the relational

operators are themselves functions (in particular, functionals). Therefore, these functions can be manipulated and combined by the relational operators. Therefore, higher level operators can be built without the use of a "formal" (i.e., data based) representation, such as that used in LISP or Backus's FFP system [1]. This is a natural outgrowth of the fact that relational programming deals with a single kind of entities, relations, and uses them for all purposes. Second and higher level functionals have not been seriously investigated yet, although they seem to arise naturally from the attempt to eliminate variables.

## 6. Status

In this section we summarize the current status of our investigation into relational programming.

The operators are undergoing a continuing refinement. We began with the operators defined by Russell and Whitehead [7] and Carnap [2]. As the requirements of using a relational calculus for programming have emerged, we have modified the meaning of several of their operators, dropped some, and added others.

The notation is undergoing a continuing evolution, as is apparent in any comparison with our earlier reports [4, 5]. The notation used in this paper is more in conformity with mathematical custom and is easier to read and type. We anticipate that this evolution will continue; it would be premature to freeze it at this time.

In an attempt to better access the value of relational programming, we have begun the implementation of several trial applications. One of these is a table-driven syntax-directed-editor and generator of the type described in [6]. The resulting relational program is about a page long. It will be described in a future technical report.

We have consciously avoided allowing implementation considerations to influence the early development of relational programming. This is because we did not want to prejudice the study by particular assumptions about machine architecture. Rather, we have hoped that the investigation of relational programming will guide us to the machines we should be building. Recently, however, we have begun the investigation of some possible representations of relations along with an analysis of the complexity of the corresponding algorithms. This will be reported in a forthcoming thesis from the Naval Postgraduate School.

We have been attempting the practical proof of some relational programs. This simple properties which relations satisfy makes this a feasible undertaking.

Finally, we have begun the implementation of simple extensional and intensional representations and implementations of the relational operators. The goal here is to provide a system to allow "hands-on" experience with relational programming. This is a necessary part of the evaluation of any new programming style.

## 7. References

- [1] Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, CACM 21, 8 (August 1978), 613-641.
- [2] Carnap, R. Introduction to Symbolic Logic and its Applications, Dover, 1958.
- [3] MacLennan, B. J. Fen - an axiomatic basis for program semantics, CACM 16, 8 (August 1973), 468-474.
- [4] MacLennan, B. J. Programming with a Relational Calculus, Computer Science Department Technical Report NPS52-81-013, Naval Postgraduate School, September 1981.
- [5] MacLennan, B. J. Introduction to relational programming, Proceedings of ACM Conference on Functional Programming Languages and Computer Architecture, October 18-22, 1981.
- [6] MacLennan, B. J. The Automatic Generation of Syntax Directed Editors, Computer Science Department Technical Report NPS52-81-014, Naval Postgraduate School, October 1981.
- [7] Whitehead, A. N. and Russell, B. Principia Mathematica to \*56, Cambridge, 1970.

APPENDIX: RELATIONAL CALCULUS - REVISED NOTATION

Old Notation	Name	New Notation
$x \in C$	class membership	$x \in C$
$x C$	"	"
$x R y$	relation membership	$x R y$
$F : x$	function application	$F x$
$R^{-1}$	converse	$R^{-1}$
$\text{!} : R$	"	$\text{inv } R$
$\text{!em} : R$	domain	$\text{dom } R$
$\overrightarrow{\text{Lm}} : R$	"	"
$\text{rim} : R$	codomain	$\text{dom.inv } R$
$\overrightarrow{\text{Rm}} : R$	"	"
$\text{mem} : R$	members	$\text{mem } R$
$R!$	image	$\text{img } R$
$i : x$	unit class	$\text{un } x$
$\theta : C$	unit class selector	$\text{the } C$
$\overleftarrow{R}$	unit image	$\text{unimg } R$
$\overrightarrow{R}$	unit coimage	$\text{unimg.inv } R$
$R \rangle S$	right restriction	$R \leftarrow S$
$S \langle R$	left restriction	$S \rightarrow R$
$R \times S$	restriction	$R \langle \rangle S$
$R \wedge S$	intersection	$R \& S$
$R \vee S$	union	$R   S$
$R - S$	difference	$R \_ S$
$\_ R$	complement	$\text{non } R$
$x + y$	addition	$x + y$
$x - y$	subtraction	$x - y$

$x*y$	multiplication	$x*y$
$x/y$	division	$x\%y$
$\emptyset$	empty class	$\emptyset$
$\overline{\emptyset}$	full class	all, non $\emptyset$
$C*D$	Cartesian product	$CXD$
$R S$	relative product	$R;S$
$RS$	functional composition	$R.S, R'S$
$=, I$	identity	$=, Id$
$init:R$	initial members	$init R$
$term:R$	terminal members	$init.inv R$
$\infty:R$	first member	$first R$
$\omega:R$	last member	$first.inv R$
$Q:R$	final members	$final R$
$A:R$	initial members	$inv.final.inv R$
$min:C$	minimum	$min C$
$max:C$	maximum	$max C$
$x,y$	pair	$x:y$
$\begin{pmatrix} x \\ y \end{pmatrix}$	"	"
$\#:C$	size	$size C$
$Curry:C$	Curry	$graph^{-1} C$
$Curry^{-1}:R$	graph	$graph R$
$\begin{matrix} r \\ s \\ r \\ s \end{matrix}$	parallel application	$r  s$
	construction	$r\#s$
$(\pi)$	binary operator	$(\pi)$
$(x\pi)$	left binding	$(x\pi)$
$(\pi y)$	right binding	$(\pi y)$
$R^*$	reflexive trans. closure	$R^*, retrac R$



$R^+$	transitive closure	$R^+$ , trac R
$R^n$	relation power	$R^n$
$f::g$	meta-application	$f::g$
$fRf^{-1}$	isomorphic image	$f\$R$
$Md:(g,f)$	overlying	$f/g$
$R \subseteq S$	subset	$R \subseteq S$
$R \rightarrow S$	"	$R \subseteq S$
$\emptyset$	empty relation	$\emptyset X \emptyset$
$\bar{\emptyset}$	full relation	$\text{all} X \text{all}$
$m..n$	closed interval	$(\text{unimg } \underline{\geq}) m \ \& \ (\text{unimg } \underline{\leq}) n$
CXD	cross product	$\text{graph}(\text{CXD})$
$\begin{pmatrix} a \dots y \\ b \dots z \end{pmatrix}$	explicit relation	$(a:b \mid \dots \mid y:z)$
$s^t$	catenate	$s^t$
$\langle a,b,\dots,z \rangle$	sequence	$(a,b,\dots,z)$
$f@i$	reduction	$f@i$

Note! The major difference between the new and old notations is that  $y=Fx$  now means  $xFy$ , whereas previously it meant  $yFx$ . This means that separate operators are now needed for relative product and composition. These are related by  $R.S = S;R$ . This effects the interpretation of several other relations and classes. For example, functions are now the right-univalent (run) relations, whereas previously they were the left-univalent (lun) relations. Also, the domain of a function is its left members, rather than its right members.

## INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Bruce J. MacLennan, Code 52M1 Department of Computer Science Naval Postgraduate School Monterey, CA 93940	12
Mr. Jim Bowery Viewdata Corporation of America, Inc. 1444 Biscayne Boulevard, Suite 305 Miami, Florida 33132	1
Dr. Mehdi Jazayeri Synapse Computer Corporation 801 Buckeye Court Milpitas, CA 95035	1
Dr. M. Sintzoff Philips Research Laboratory 2 av. Van Becelaere 1170 Brussels Belgium	1
Professor Harvey Abramson Department of Computer Science The University of British Columbia 2075 Westbrook Mall Vancouver, B.C. Canada V6T 1W5	1
Dr. Charles D. Marshall Department K51 IBM Research 5600 Cottle Road San Jose, CA 95193	1