

RELATIONAL PROGRAMMING(U) NAVAL POSTGRADUATE SCHOOL
MONTEREY CA B J MACLENNAN SEP 83 NP552-83-012

5

F/G 12/1 NL

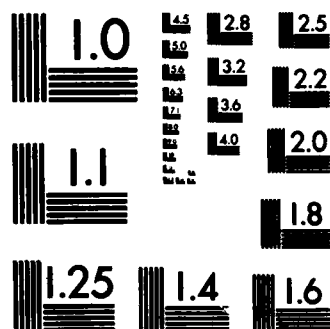
END
1980

END

2

品名

01 000 100



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Ad-A133643

2

NPS52-83-012

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
OCT 17 1983
S B

RELATIONAL PROGRAMMING

Bruce J. MacLennan

September 1983

DTIC FILE COPY

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, VA 22217

88 10 17 151

NAVAL POSTGRADUATE SCHOOL
Monterey, California

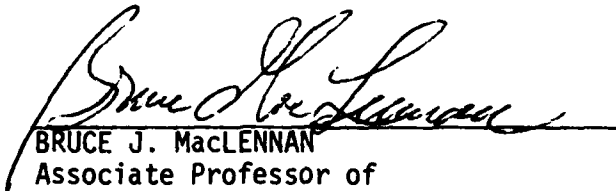
Rear Admiral J. J. Ekelund
Superintendent

D. A. Schrady
Provost

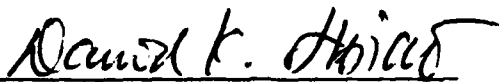
The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

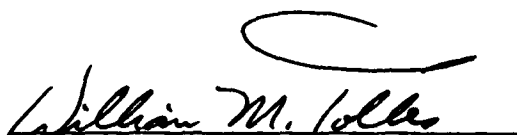
Reproduction of all or part of this report is authorized.

This report was prepared by:


BRUCE J. MacLENNAN
Associate Professor of
Computer Science

Reviewed by:


DAVID K. HSIAO, Chairman
Department of Computer Science


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 NPS52-83-012	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 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 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N: RR000-01--100 N0001483WR30104
11. CONTROLLING OFFICE NAME AND ADDRESS Chief of Naval Research Arlington, VA 22217		12. REPORT DATE September 1983
		13. NUMBER OF PAGES 77
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, relations, relational algebra, relational calculus, applicative language, logic programming, combinator, very-high-level language.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report describes <u>relational programming</u> , a style of programming in which entire relations are manipulated as data, and in which programs are represented as relations. The use of relational operators on both data and programs is illustrated, and implementation issues are discussed.		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

RELATIONAL PROGRAMMING

Bruce J. MacLennan

Computer Science Department

Naval Postgraduate School

Monterey, CA 93940

CONTENTS

1. Introduction	1
2. Classes and Relations	2
2.1 basic concepts	2
2.2 relational descriptions	2
2.3 converse	3
2.4 arrow diagrams	3
2.5 tables	4
3. Domains	4
4. Functions	5
4.1 basic concepts	5
4.2 higher level functions	7

5. Boolean Operations	9
5.1 logical connectives	9
5.2 empty class	10
5.3 Cartesian product	11
5.4 subset relation	11
6. Limiting and Restriction	11
7. Relative Product and Composition	13
8. Structures	15
8.1 initial and terminal members	15
8.2 higher level operations	17
9. Sequences	19
9.1 ordinal couples	19
9.2 catenation and consing	20
9.3 alternative definitions of sequences	22
10. Binary Operations	23
10.1 basic concepts	23
10.2 operations on binary operations	24
11. Combinators	26

11.1	paralleling of relations	26
11.2	conditional union	26
11.3	combinatory logic	27
11.4	Curried functions	29
12.	Records	30
12.1	basic operations	30
12.2	functional records	31
12.3	relational databases	33
13.	Ancestral Relations	35
13.1	definition	35
13.2	applications	37
13.3	iteration	37
14.	Arrays	38
14.1	definition and basic operations	38
14.2	relation to sequences	40
14.3	other array operations	42
15.	Isomorphic and Homomorphic Images	43
15.1	images	43

15.2 images of functional structures	44
15.3 isomorphism and the structure function	46
16. Data Structures	47
16.1 definition	47
16.2 operations on data structures	48
17. Reducing Structures	52
17.1 basic concepts	52
17.2 reduction of arrays	53
17.3 reduction of sequences	54
18. Examples	55
18.1 payroll	55
18.2 check issuing	57
18.3 pseudo-natural notation	57
19. Implementation	58
19.1 introduction	58
19.2 computability	59
19.3 extensional representation	59
19.4 intensional representations	60

19.5 eliminating polymorphism	60
19.6 extensional operators	62
19.7 intensional operators	64
20. Conclusions	64
21. References	67

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



RELATIONAL PROGRAMMING

1. Introduction

In this report¹ we discuss *relational programming*, i.e. a style of programming in which entire relations are manipulated rather than individual data. This is analogous to functional programming [Backus78], wherein entire functions are the values manipulated by the operators. We will see that relational programming subsumes functional programming because every function is also a relation. It is appropriate at this point to discuss why we have chosen to investigate relational programming².

As we have noted, relational programming subsumes functional programming; hence, anything that can be done with functional programming can be done with relational programming. Furthermore, relational programming has many of the advantages of functional programming: for instance, the ability to derive and manipulate programs by algebraic manipulation. A well developed algebra of relations dates back to Boole's original work and has been extensively studied since then. Although relations are more general than functions, their laws are often simpler. For instance, $(f.g)^{-1} = g^{-1}.f^{-1}$ is true for all relations, but true only for functions that are one-to-one. Also, relational programming more directly supports non-linear data structures, such as trees and graphs, than does functional programming. In relational programming the basic data values are themselves relations, whereas in functional programming there is a separate class of objects (lists) used for data structures. One final reason for investigating relational programming is that it provides a possible paradigm for utilizing associative and active memories. As a teaser for what is to come, we present the following example of a relational program. We will take a text T , represented as an array of words (i.e., $T(i)$ is the i -th word), and generate a frequency table F so that $F(w)$ is the number of occurrences of word w in T . Now we will see (§4) that all $T(w)$ is the set of all indices of the word w . If we let $\text{size}(C)$ be the cardinality of a set C , then

1. The work reported herein was supported by the Office of Naval Research under contract number N00014-82-WR-20162, and by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.
2. The reader can find a shorter introduction to relational programming in [MacLennan83]. That report is a revision and extension of [MacLennan81a] and [MacLennan81b].

RELATIONAL PROGRAMMING

the number of indices (occurrences) of w is just $\text{size}[\text{all } T(w)]$. Therefore we can write $F = \text{size.}(\text{all } T)(\S 7)$.

2. Classes and Relations

2.1 basic concepts

Our relational calculus will deal with three sorts of things: individuals, classes and relations. These can best be illustrated by example. If ' x ' is the name of an individual and ' C ' is the name of a class (set), then ' $x \in C$ ' means that the individual denoted by ' x ' is a member of the class denoted by ' C ' (i.e., that x has property C). Thus ' $Aristotle \in \text{man}$ ' would indicate that Aristotle is a man, and ' $2 \in \text{even}$ ' would mean that 2 is an even number. Some authors (e.g., Russell and Whitehead) use ' $x \varepsilon P$ ' for ' $x \in P$ '. The symbol ' ε ' and its alternate ' ϵ ' are abbreviations for ' $\epsilon\sigma\tau\iota$ ', which is the Greek word for 'is'.

If ' x ' and ' y ' are names of individuals and ' R ' is the name of a relation, then ' $x R y$ ' means that x bears the relation R to y . For example,

Aristotle student Plato

means that Aristotle is a student of Plato. Also, ' $2 < 3$ ' means that 2 bears the less-than relation to 3. A relation is just a set of pairs. Therefore, if we use $x:y$ to denote the basic pair-making operation, then xRy if and only if $x:y \in R$. The notation that we have introduced above will be extended to classes of classes, classes of relations, relations among classes, relations among relations, etc.

2.2 relational descriptions

There are several ways to describe classes and relations. One of the easiest is to list its elements, for example:

$$S = \{ 1, 3, 5, 7 \}$$

$$R = \{ 1:a, 2:b, 3:c, 4:d \}$$

RELATIONAL PROGRAMMING

This is called an *extensional* description of the class. Obviously, this is only possible if the class or relation is finite and only practical if it's small. Therefore we also have *intensional* descriptions of classes and relations.

If $S(x)$ is a sentence involving ' x ', then a *class description* is an expression of the form ' $\{x | S(x)\}$ '. This denotes the class of all individuals, a , for which $S(a)$ is true, i.e.,

$$a \in \{x | S(x)\} \Leftrightarrow S(a)$$

Similarly, if $S(x,y)$ is a sentence involving ' x ' and ' y ', then ' $\{x,y | S(x,y)\}$ ' is a *relation description* which describes the relation that holds between a and b whenever $S(a,b)$ is true, i.e.,

$$a\{x,y | S(x,y)\}b \Leftrightarrow S(a,b)$$

To illustrate this notation we will define the converse of a relation.

2.3 converse

The relation R^{-1} is called the *converse* of R , i.e. $xR^{-1}y \Leftrightarrow yRx$. Using our notation for descriptions we can define $R^{-1} = \{x,y | yRx\}$. As an example of a relation among relations, we define 'inv' as the relation that holds between converses: $s \text{ inv } r \Leftrightarrow r = s^{-1}$. Hence, $\text{inv} = \{s:r | r = s^{-1}\}$. Some examples of converses are $\text{parent}^{-1} = \text{child}$ and $\leq^{-1} = \geq$.

EXERCISES: Prove the following properties of the converse:

$$(r^{-1})^{-1} = r$$

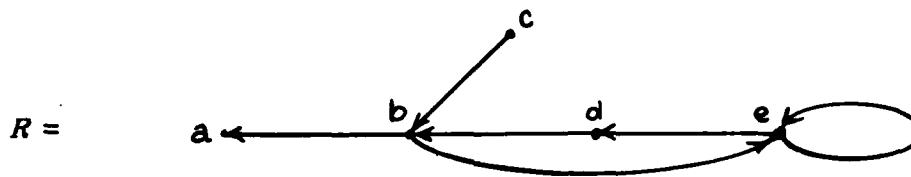
$$r \text{ inv } s \Leftrightarrow s \text{ inv } r$$

$$\text{inv}^{-1} = \text{inv}$$

2.4 arrow diagrams

Relations can be portrayed by *arrow diagrams* (*Hasse diagrams*). In such a diagram there is a node for each individual related by the relation and an arrow from x to y whenever xRy . For instance,

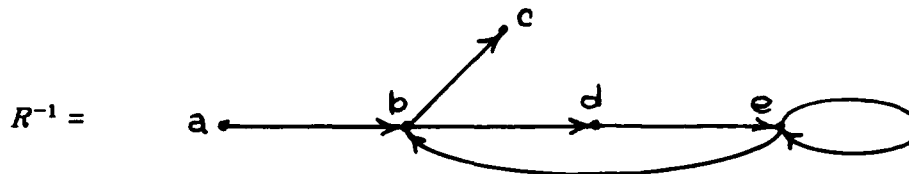
RELATIONAL PROGRAMMING



represents the relation R such that $bRa, cRb, dRb, eRd, eRe, bRe$ and $\neg xRy$ for all other cases:

$$R = \{ b:a, c:b, d:b, e:d, e:e, b:e \}$$

The effect of the converse operator is to reverse all of the arrows. Hence, R^{-1} is diagrammed:



2.5 tables

Relations can often be viewed as tables. For instance, the relation R of the previous section can be shown as the table in Figure 1.

R	
b	a
c	b
d	b
e	d
e	e
b	e

Figure 1. Relation Viewed as a Table

Of course, it makes no difference in what order we write the rows of the table.

The converse of a relation is obtained by simply exchanging the columns of the table (see Figure 2). Of course, classes are represented by one column tables. For instance the class C of primes less than ten is shown in Figure 3.

3. Domains

We often need to talk of the individuals that can occur on the right or left of a rela-

RELATIONAL PROGRAMMING

$$R^{-1}$$

a	b
b	c
b	d
d	e
e	e
e	b

Figure 2. Converse of a Relation

tion. We say that x is a *left-member* of R whenever there is a y such that xRy .

$$x \text{ Lm } R \Leftrightarrow \exists y(xRy)$$

For instance, if ' x parent y ' means that x is a parent of y , then 'Socrates Lm parent' means that Socrates is a parent. *Right-member* and *member* are defined analogously:

$$y \text{ Rm } R \Leftrightarrow \exists x(xRy)$$

$$z \text{ Mm } R \Leftrightarrow z \text{ Lm } R \vee z \text{ Rm } R$$

EXERCISES: Prove that these satisfy the identities:

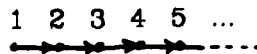
$$x \text{ Lm } R \Leftrightarrow x \text{ Rm } R^{-1}$$

$$y \text{ Rm } R \Leftrightarrow y \text{ Lm } R^{-1}$$

4. Functions

4.1 basic concepts

Functions and relations are closely related. Consider the successor relation, 'succ': $x \text{ succ } y \Leftrightarrow x+1 = y$. Thus, $x \text{ succ } y$ says that x 's successor is y . The corresponding arrow diagram is:



and the corresponding table is shown in Figure 4. since $1 \text{ succ } 2$, $2 \text{ succ } 3$, etc. Notice that, in this case, for each left member x there is a unique right member y such that $x \text{ succ } y$. This y can be written using Whitehead and Russell's [Whitehead70] *definite description*:

RELATIONAL PROGRAMMING

<i>C</i>
1
2
3
5
7

Figure 3. Set Viewed as a Table

$$\iota y (x \text{ succ } y)$$

This can be read: *the y such that x's successor is y*. A more convenient way to write this is $\text{succ}(x)$. In general, $R(x)$ means 'the unique y such that $x R y$ ', i.e. $R(x) = \iota y (x R y)$. When no confusion will result we write Rx instead of $R(x)$. This notation is left-associative, that is, $Fxy = (Fx)y$. When we need to make the application operation explicit we write $R @ x$ (R at x or R applied to x) for Rx .

The functional notation is meaningful only if there is a unique y such that $x R y$, i.e. $x R y \wedge x R z \supset y = z$. That is, there is only one arrow leading from x . When this condition is satisfied for all x we call R *right univalent*, symbolized by 'run':

$$R \in \text{run} \Leftrightarrow \forall xyz [x R y \wedge x R z \supset y = z]$$

The right univalent relations are more commonly called *functions*. In a *left* univalent relation there is exactly one arrow leading *to* each node. Consider the 'absolute reciprocal' relation: $x R y \Leftrightarrow y = |1/x|$. This is diagrammed in Figure 5. Since $R \in \text{run}$ it is meaningful to write $R(x)$, so we observe $R(-3) = 1/3$. We can find Rx by following the arrow pointing from x or by looking down the left column for x and taking the corresponding element from the right column.

The concepts of left univalence and bi-univalence are defined analogously:

$$R \in \text{lun} \Leftrightarrow \forall xyz [y R x \wedge z R x \supset y = z]$$

$$R \in \text{bun} \Leftrightarrow R \in \text{lun} \wedge R \in \text{run}$$

Bi-univalent relations are also called bijections and one-one mappings.

1	2
2	3
3	4
4	5
⋮	⋮

Figure 4. Function Viewed as a Table

4.2 higher level functions

Of course, the converse of a function is not necessarily a function. The 'sin' relation, defined so that $x \sin y$ means that y is the sine of x , is right univalent but not left univalent. Hence, we can write either $y = \sin x$ or $x \sin y$, but can express the arcsine only by $y \sin^{-1} x$. The notation $\sin^{-1} y$ is meaningless. Since $f(x)$ is meaningful only when $f \in \text{run}$ we will be careful to write $f(x)$ only when we have previously shown (or it is obvious) that $f \in \text{run}$ and $x \text{ Lm } f$.

The fact that $F(x)$ may be meaningless makes it convenient to use several other relations derived from F . One of these is the *image*. If F is any relation and C is a class then $\text{img } F C$ is the set of all y such that $x F y$ for some x in C , i.e.,

$$\text{img } F = \{C : z \mid z = \{y \mid \exists x (x F y \wedge x \in C)\}\}$$

The tabular interpretation of $\text{img } F C$ is shown in Figure 6. We see that, if F is any function, then $\text{img } F S$ is the image of the class S under that function. Notice that the operation $\text{img } F S$ is defined for all relations F and classes S , regardless of whether $F \in \text{run}$ or the members of S are left members of F . For these reasons, it is generally safer to write $\text{img } F C$ than $F x$.

The image operation is also useful for working with relations. For example, $\text{img.in}(\lt) S$ is the set of all numbers that are *less* than some element of S . The reversal of the sense of the ordering occurs because $\text{img}(\gt) S$ is the set of all y such that for some $x \in S$, $x > y$. Thus $\text{img.in}(\lt) S$ is the set of all x such that for some $y \in S$, $x < y$.

Related ideas are the image and converse image of an individual. If R is a relation, then $c = \text{unimg } R x$ means that c is the class of individuals related to x . This class is

RELATIONAL PROGRAMMING

1	1
-1	1
2	1/2
-2	1/2
3	1/3
-3	1/3
⋮	⋮

Figure 5. Right-Univalent Relation Viewed as Table
called the *unit image* of x , and is defined $\text{unimg } R \ x = \{y \mid xRy\}$. Alternately we can define $\text{unimg } Rx = \text{img } R\{x\}$.

The converse idea is that of the *inverse unit image* of y :

$$\text{unimg.inv } R \ y = \{x \mid xRy\}$$

Like the image, $\text{unimg } R$ and $\text{unimg.inv } R$ are defined for all R and all arguments.

We can also apply the unit image operations to general relations. Therefore $\text{unimg.inv}(<)x$ is the set of all numbers less than x . This is sufficiently common that we define $\text{all} = \text{unimg.inv}$. Then $\text{all}<x$ is the set of all numbers less than x .

Next consider the function $\text{all}(=)$:

$$\text{all}(=)x = \{y \mid y=x\}$$

Hence, $\text{all}=x$ is just the *unit class* containing x , which we will abbreviate this $\text{un } x$. Conversely, if C is a single element class, then $\text{un}^{-1}C$ selects the unique member of that class: $\text{un}^{-1}C = \{x(x \in C)\}$. It is thus a uniqueness filter. We will write this as ϑC where $\vartheta = \text{un}^{-1}$. The expression ϑC can be read '*the C*.'

EXERCISES: Show the following:

$$\text{unimg } R^{-1} = \text{unimg.inv } R$$

$$\text{unimg.inv } R^{-1} = \text{unimg } R$$

$$\text{unimg } R \ y = \text{img } R(\text{un } y)$$

It is often convenient to have names for domain extracting functions, e.g., $\text{dom } R$ is the class of left members of R . These are simply defined using images:

RELATIONAL PROGRAMMING

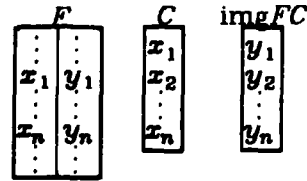


Figure 6. Image Operation Applied to Tables

$\text{dom} = \text{all Lm}$

$\text{dom.inv} = \text{all Rm}$

$\text{mem} = \text{all Mm}$

Of course the left members (domain) and right members (domain-inverse) of a relation can be obtained by taking its left and right columns, respectively, and deleting duplicates (Figure 7).

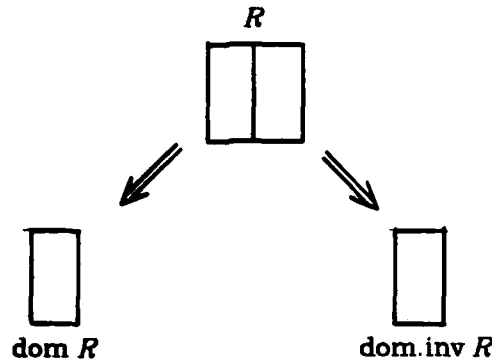


Figure 7. Domain Extracting Operators

5. Boolean Operations

5.1 logical connectives

We will next investigate ways of *combining* relations and classes. The simplest methods are just abstractions of the logical connectives used between propositions: Therefore, we define the intersection, union, negation and difference of classes and relations:

$$x \in (S \cap T) \Leftrightarrow x \in S \wedge x \in T$$

$$x \in (S \cup T) \Leftrightarrow x \in S \vee x \in T$$

$$x \in (\sim S) \Leftrightarrow \neg(x \in S)$$

RELATIONAL PROGRAMMING

$$x \in (S \setminus T) \Leftrightarrow x \in S \wedge \neg(x \in T)$$

$$x \in (S \supset T) \Leftrightarrow x \in S \supset x \in T$$

As an example of the use of these operations, consider our previous definition of Mm :

$$z \text{ Mm } R \Leftrightarrow z \text{ Lm } R \vee z \text{ Rm } R$$

Using the union operation this can be written $Mm = Lm \cup Rm$. Similarly, $bun = lun \cap run$. The logical connectives satisfy the usual properties of a Boolean algebra (e.g., DeMorgan's theorem).

As an example of the use of these operations, we will define the *closed interval* function, $[m..n]$, which is the set of integers $m, m+1, \dots, n$. It is just:

$$[m..n] = \text{all} \geq m \cap \text{all} \leq n$$

where \leq and \geq are the relations on integers. In general we will allow $[m..n]$ for any types on which a strict order is defined.

EXERCISES: Define the analogous notations $(m..n)$, $[m..n)$, and $(m..n]$.

5.2 empty class

It is useful to have a name for the empty class: $\phi = S \setminus S$, for any set S . Hence, $x \in \phi$ is always false. This is most often used for stating properties of relations and classes. For instance, $S \cap T = \phi$ means that classes S and T have no members in common.

The universal class is also useful: $\bigcirc = \sim \phi$. For instance, $S \cup T = \bigcirc$ means that every individual is either a member of S or of T . Notice that the class of the right members of a relation is just the image of the universe under that relation, i.e.,

$$\text{dom.inv } R = \text{img } R \bigcirc$$

$$\text{dom } R = \text{img } R^{-1} \bigcirc$$

$$\text{mem } R = \text{img } (R \cup R^{-1}) \bigcirc$$

EXERCISES: Prove these properties of the domain functions.

5.3 Cartesian product

It is often useful to have the maximum relation that can hold between two classes, i.e., the *Cartesian product* of those classes. This is defined:

$$S \times T = \{x:y \mid x \in S \wedge y \in T\}$$

EXERCISES: Show the Cartesian product satisfies the following properties:

$$(s \times t)^{-1} = t \times s$$

$$\text{dom}(s \times t) = s$$

$$\text{dom.inv}(s \times t) = t$$

$$\text{mem}(s \times t) = s \cup t$$

$$s \times (t \cap u) = (s \times t) \cap (s \times u)$$

$$s \times (t \cup u) = (s \times t) \cup (s \times u)$$

$$s \times (t \setminus u) = (s \times t) \setminus (s \times \sim u)$$

$$s \times (t \supset u) = (s \times \sim t) \cup (s \times u)$$

$$s \times \phi = \phi \times s = \phi \times \phi = \phi$$

$$s \times t = (s \times \emptyset) \cap (\emptyset \times t)$$

5.4 subset relation

Finally, we define the subclass operation:

$$S \subseteq T \Leftrightarrow \forall x (x \in S \supset x \in T)$$

EXERCISES: Show the following are true:

$$s \subseteq t \supset (s \times u) \subseteq (t \times u)$$

$$s \subseteq t \supset (r \times s) \subseteq (r \times t)$$

$$s \subseteq t \wedge u \subseteq v \supset (s \times u) \subseteq (t \times v)$$

6. Limiting and Restriction

It is often useful to limit the left or right domain of a relation. Consider the relation

RELATIONAL PROGRAMMING

$y \sin^{-1} x$, which means that x is an arcsine of y . We cannot write $x = \sin^{-1} y$ because \sin^{-1} is not right univalent (i.e. it is not a function). If we restrict y , the argument of \sin , to the range $-\pi/4$ to $\pi/4$, then there is a unique x such that $y \sin^{-1} x$. Let S be the class of reals in the range $-\pi/4$ to $\pi/4$:

$$S = (-\pi/4.. \pi/4] = \text{all} > (-\pi/4) \cap \text{all} \leq (\pi/4)$$

then we will write $S \rightarrow \sin$ for the sine function with its arguments restricted to S . This function is bi-univalent, so it is invertible. If we call the inverse of this restricted sine Arcsin:

$$\text{Arcsin} = (S \rightarrow \sin)^{-1}$$

then it is perfectly meaningful to write $\text{Arcsin } x$ (if $x \text{ Rm } \sin$). The left-restriction operation is defined:

$$x(S \rightarrow R)y \Leftrightarrow x \in S \wedge xRy$$

In other words,

$$y = (S \rightarrow R)x \Leftrightarrow y = Rx \wedge x \in S$$

The right-restriction is defined analogously:

$$x(R \leftarrow S)y \Leftrightarrow xRy \wedge y \in S$$

These notations can be combined to restrict both domains:

$$x(S \rightarrow R \leftarrow T)y \Leftrightarrow x \in S \wedge xRy \wedge y \in T$$

The combination $s \rightarrow R \leftarrow s$ is so common that a special notation is provided for it: $R \uparrow s = s \rightarrow R \leftarrow s$. For instance, $< \uparrow P$, where $x \in P \Leftrightarrow x > 0$, is the less-than relation restricted to positive numbers. Notice that $x \text{ succ } y$ if and only if y is the successor of x . Therefore we can define the sequence of integers $(m, m+1, \dots, n)$ by restricting the succ relation:

$$[m..n] = \text{all} \geq m \rightarrow \text{succ} \leftarrow \text{all} \leq n$$

See Figure 8 for a tabular representation of the domain restricting operations.

RELATIONAL PROGRAMMING

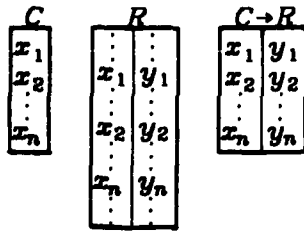


Figure 8. Domain Restricting Operations

EXERCISES: Show that the restriction operations can be defined in terms of intersection and Cartesian product:

$$s \rightarrow r \leftarrow t = r \cap (s \times t)$$

$$r \uparrow s = r \cap (s \times s)$$

$$s \rightarrow r = r \cap (s \times O)$$

$$r \leftarrow s = r \cap (O \times s)$$

EXERCISES: Show that other properties satisfied by these operations are:

$$s \times t = s \rightarrow O \times O \leftarrow t$$

$$\text{dom}(s \rightarrow r) = s \cap \text{dom}(r)$$

$$\text{dom.inv}(r \leftarrow s) = s \cap \text{dom.inv}(r)$$

$$\text{dom}(r \leftarrow s) = \text{img}(r^{-1})s$$

$$\text{dom.inv}(s \rightarrow r) = \text{img } r s$$

$$(s \rightarrow r)^{-1} = (r^{-1}) \leftarrow s$$

$$(s \rightarrow r \leftarrow t)^{-1} = t \rightarrow (r^{-1}) \leftarrow s$$

$$(r \uparrow s)^{-1} = (r^{-1}) \uparrow s$$

$$r \leftarrow s \cap r \leftarrow t = r \leftarrow (s \cap t)$$

$$r \leftarrow s \cup r \leftarrow t = r \leftarrow (s \cup t)$$

$$(r \times O) \leftarrow s = r \times s$$

7. Relative Product and Composition

If $x \text{son} y$ is the relation 'x is a son of y' and $x \text{brother} y$ is the relation 'x is a brother of y', then the *relative product*, 'son|brother', is the relation 'a son of a brother of'.

RELATIONAL PROGRAMMING

More formally,

$$R|S = \{x:z \mid \exists y(xRy \wedge ySz)\}$$

We will also write $S.R$ and $S \circ R$ for $R|S$. The reason for this is that if F and G are functions it is easy to see that $F.G$ is the composition of these functions:

$$\begin{aligned} z = F.G x &\Leftrightarrow x F.G z \\ &\Leftrightarrow xG|Fz \\ &\Leftrightarrow \exists y(xGy \wedge yFz) \\ &\Leftrightarrow \exists y[z = Fy \wedge y = Gx] \\ &\Leftrightarrow z = F(Gx) \end{aligned}$$

Hence, $F.G x = F(Gx)$.

It is convenient to have a notation for relative products of a relation with itself. For instance, the 'grandparent' relation can be written 'parent|parent', which we abbreviate parent^2 . In general,

$$\begin{aligned} R^0 &= (=)\uparrow(\text{mem } R) \\ R^1 &= R \\ R^{n+1} &= (R^n)|R = R|(R^n) \\ R^{-n} &= (R^n)^{-1} \end{aligned}$$

EXERCISES: Show these obvious properties of the relative product:

$$\begin{aligned} (r.s).t &= r.(s.t) \\ r.(s \cup t) &= r.s \cup r.t \\ (r \cup s).t &= r.t \cup s.t \\ r.(s \cap t) &\subseteq r.s \cap r.t \\ (r \cap s).t &\subseteq r.t \cap s.t \\ \exists(r.s) &\Leftrightarrow \exists(\text{dom.inv } r \cap \text{dom } s) \\ \text{where } \exists r &\text{ means } \exists x[x \in r] \end{aligned}$$

RELATIONAL PROGRAMMING

$$(r^{-1})^{-1} = r$$

$$(r.s)^{-1} = (s^{-1}).(r^{-1})$$

$$r^m.r^n = r^{m+n} \quad (m, n \geq 0)$$

$$(r^m)^n = r^{mn} \quad (m, n \geq 0, \text{ or } r \in \text{bun})$$

$$r^m.r^n \subseteq r^{m+n} \quad (r \in \text{bun})$$

$$r.r^{-1} = r^{-1}.r = r^0 \quad (r \in \text{bun})$$

$$\text{dom}(r.s) \subseteq \text{dom } r$$

$$\text{dom.inv}(r.s) \subseteq \text{dom.inv } s$$

$$\text{Lm} = \text{Rm} | \text{inv}$$

$$\text{Rm} = \text{Lm} | \text{inv}$$

$$r.(\phi \times \phi) = (\phi \times \phi).r = \phi \times \phi$$

$$r.\text{Id} = \text{Id}.r = r \quad \text{where Id} = (=) \text{ (the identity function)}$$

8. Structures

We have previously seen the use of arrow diagrams to represent a relation. For instance, the diagram in Figure 9 represents the relation R shown in Figure 10.

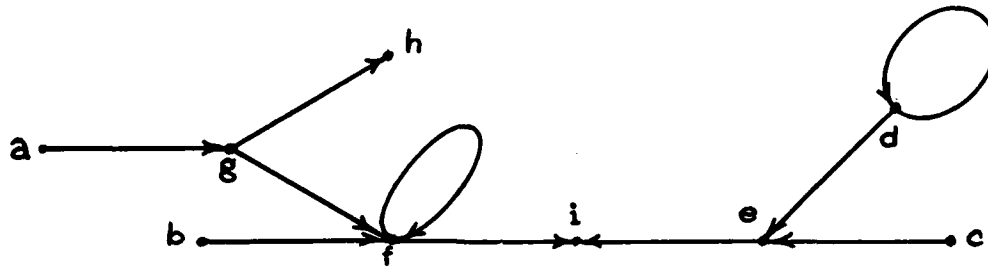


Figure 9. Arrow Diagram for a Relation

8.1 initial and terminal members

Now, notice that the domain (left) and codomain (right) members of R are:

$$\text{dom } R = \{ a, b, c, d, e, f, g \}$$

$$\text{dom.inv } R = \{ g, f, e, d, i, h \}$$

We define the *initial* members of R to be those members which are not pointed at by an arrow. Therefore, the initial members of R are the left members that are not right

RELATIONAL PROGRAMMING

$$R$$

a	g
b	f
c	e
d	d
d	e
e	i
f	f
f	i
g	f
g	h

Figure 10. Tabular Representation of a Structure members, that is, the domain members that are not codomain members.

$$\text{init } R = \text{dom}(R) \setminus \text{dom.inv}(R) = \{a, b, c\}$$

The *terminal* members of a relation are defined analogously:

$$\text{term } R = \text{dom.inv}(R) \setminus \text{dom}(R) = \{h, i\}$$

When a relation is used to represent a data structure, the above functions become important.

For instance, a sequence is represented by a relation with the structure:

$$S = \begin{array}{ccccccc} a_1 & a_2 & a_3 & \dots & a_{n-1} & a_n \\ \bullet & \bullet & \bullet & \dots & \bullet & \bullet \end{array}$$

In this case $\text{init } S$ is the unit class containing the head (first element) of the relation (i.e., a_1) and $\text{term } S$ is the unit class containing the last element of the sequence (i.e., a_n). Similarly, $(\sim.\text{init } S) \rightarrow S$ is the sequence with its first element deleted:

$$\begin{array}{ccccccc} a_2 & a_3 & \dots & a_{n-1} & a_n \\ \bullet & \bullet & \dots & \bullet & \bullet \end{array}$$

Hence, the following common sequence manipulation functions can be defined:

$$\begin{aligned} \alpha S &= \vartheta.\text{init } S && \text{first} \\ \omega S &= \vartheta.\text{term } S && \text{last} \\ \Omega S &= (\sim.\text{init } S) \rightarrow S && \text{final} \end{aligned}$$

RELATIONAL PROGRAMMING

$$AS = S \leftarrow (\sim.\text{term } S) \quad \text{initial}$$

EXERCISES: Prove the following properties of these relations:

$$\alpha = \omega.\text{inv}$$

$$\omega = \alpha.\text{inv}$$

$$A.\text{inv} = \text{inv}.\Omega$$

$$\Omega.\text{inv} = \text{inv}.A$$

More operations on sequences are discussed in the next section.

As another example of the use of 'init' and 'term', consider the relation, representing a tree, shown in Figure 11.

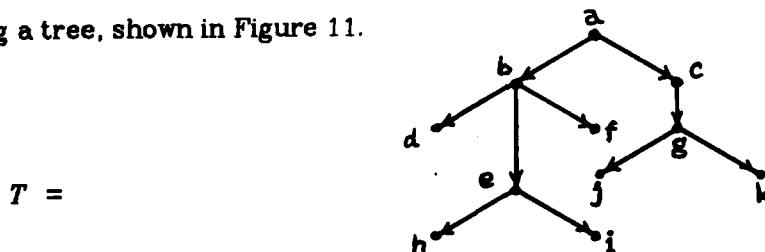


Figure 11. Relation Representing a Tree

Then, αT is 'a', the root of the tree, and term T is $\{d, h, i, f, j, k\}$, the leaves of the tree.

The result is analogous for forests.

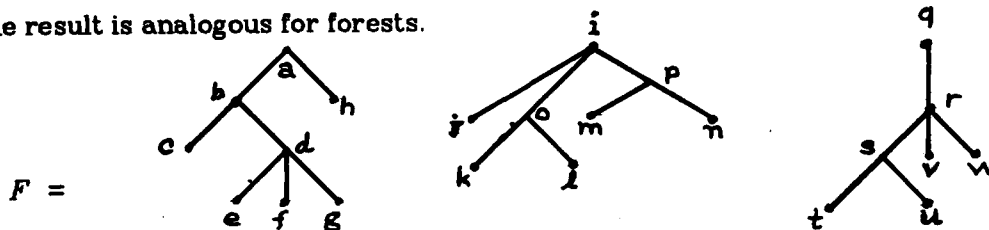


Figure 12. Relation Representing a Forest

Given the relation in Figure 12, the set of roots is $\text{init } F$ and the set of leaves is $\text{term } F$:

$$\text{init } F = \{a, i, q\}$$

$$\text{term } F = \{c, e, f, g, h, j, k, l, m, n, t, u, v, w\}$$

8.2 higher level operations

The set of nodes whose parent is n is just $\text{img } F n$. For instance, the set of nodes directly descended from a root is

RELATIONAL PROGRAMMING

$$(\text{img } F).\text{init } F = \{b, h, j, o, p, r\}$$

The set of nodes that point to leaves is

$$(\text{img } F^{-1}).\text{term } F = \{b, d, a, i, o, p, s, r\}$$

These operations can be used for obtaining the maximum and minimum of sets. Suppose ' $<$ ' is the less-than relation on integers and S is some set of integers, say $\{3, 5, 9\}$. Then

$$<\uparrow S = \begin{array}{c} \text{---} \curvearrowright \text{---} \\ 3 \quad 5 \quad 9 \end{array}$$

Now note that $\text{init}(<\uparrow S) = \{3\}$ and $\text{term}(<\uparrow S) = \{9\}$. Hence, if S is any set of numbers, then the minimum and maximum of this set are:

$$\min S = \alpha(<\uparrow S)$$

$$\max S = \omega(<\uparrow S)$$

Notice that we can select the maximum and minimum based on any relation that is a *series* (i.e., transitive, irreflexive and connected). If R is any series then $\alpha(R\uparrow S)$ is the minimum (relative to R) and $\omega(R\uparrow S)$ is the maximum.

EXERCISES: Show that the following are properties of these operations:

$$\text{init } r = \text{term}(r^{-1})$$

$$\text{term } r = \text{init}(r^{-1})$$

$$\text{init } \subseteq \text{dom}$$

$$\text{term } \subseteq \text{dom.inv}$$

$$\text{init}(r\uparrow s) = \text{term}(r^{-1}\uparrow s)$$

$$\text{init}(r \cup s) \subseteq \text{init } r \cup \text{init } s$$

$$\text{init } r \cap \text{init } s \subseteq \text{init}(r \cap s)$$

$$\text{term}(r \cup s) \subseteq \text{term } r \cup \text{term } s$$

$$\text{term } r \cap \text{term } s \subseteq \text{term}(r \cap s)$$

$$\text{init}(s \times t) = s \setminus t$$

$$\text{term}(s \times t) = \text{init}(s \times t)^{-1} = \text{init}(t \times s) = t \setminus s$$

9. Sequences

9.1 ordinal couples

In this section we will continue the discussion of sequences begun in the last section.

We saw that it was easy to define the following operations on sequences:

$$\alpha S = \varnothing.\text{init } S$$

$$\omega S = \varnothing.\text{term } S$$

$$A S = S \leftarrow (\sim.\text{term } S)$$

$$\Omega S = (\sim.\text{init } S) \rightarrow S$$

This provides us with functions for taking sequences apart. We will define the *ordinal couple* or *pair*, which puts them together. If x and y are two objects, then ' (x,y) ' is the relation that relates x and y but no other objects.

$$x,y = \begin{array}{c} \longrightarrow \\ x \quad y \end{array}$$

That is, $u(x,y)v$ if and only if $u=x$ and $v=y$. This is formally defined by:

$$x,y = \{u:v \mid u=x \wedge v=y\} = \text{un}(x) \times \text{un}(y) = \text{un}(x,y)$$

Notice that $x:y = \varnothing(x,y)$. Observe also that $\alpha(x,y) = x$ and $\omega(x,y) = y$. Finally, $xRy \Leftrightarrow (x,y) \subseteq R$.

Explicit relations can be described by a combination of the pair and union operations. For example, we have the identity:

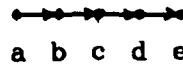
$$\{x_1:y_1, x_2:y_2, \dots, x_n:y_n\} = (x_1,y_1) \cup (x_2,y_2) \cup \dots \cup (x_n,y_n)$$

We will define a convenient notation for sequences of two or more elements:

$$(x_1, x_2, \dots, x_n) = \{x_1:x_2, x_2:x_3, \dots, x_{n-1}:x_n\}$$

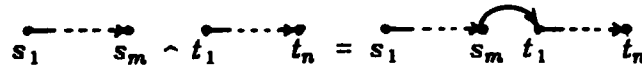
Therefore the sequence (a,b,c,d,e) is just

RELATIONAL PROGRAMMING



3.2 catenation and consing

If s and t are sequences then we can define an operation ' $s \sim t$ ', which is the catenation of s and t . To form this catenation we must hook the last element of s to the first element of t :

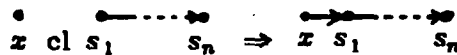


Therefore $x[s \sim t]y$ if and only if $x s y$, or $x t y$, or $x = \omega s$ and $y = \alpha t$. Hence,

$$s \sim t = s \cup (\omega s, \alpha t) \cup t$$

The catenation operation is only defined for sequences, which are required to have at least two elements (since an irreflexive relation with less than two elements is the empty relation). This issue is discussed in the following section.

How then do we add a single element to the left or right of a sequence? The 'cons left' and 'cons right' operations are easy to define:



$$x \text{ cl } s = (x, \alpha s) \cup s$$

$$s \text{ cr } y = s \cup (\omega s, y)$$

If S is a sequence and $x \text{ Mm } S$, then Sx is the successor of x in S and $S^{-1}x$ is the predecessor of x in S (if these exist).

$$Sx = \text{successor of } x \text{ in } S$$

$$S^{-1}x = \text{predecessor of } x \text{ in } S$$

These are convenient ways of moving around within a sequence. Also, note that if s is a subsequence of t then $s \subseteq t$. Some additional identities are:

RELATIONAL PROGRAMMING

$$(x,y)|(y,z) = (x,z)$$

$$\text{img } \alpha (S \times T) = S$$

$$\text{img } \omega (S \times T) = T$$

EXERCISES: Show that if s is a sequence, then:

$$\alpha(x \text{ cl } s) = x$$

$$\Omega(x \text{ cl } s) = s$$

$$\omega(s \text{ cr } y) = y$$

$$A(s \text{ cr } y) = s$$

$$(\alpha s) \text{ cl } (\Omega s) = s, \text{ if size } s > 2$$

$$(A s) \text{ cr } (\omega s) = s, \text{ if size } s > 2$$

Also, if s is a sequence, show that $s \cup (\omega s, \alpha s)$ is a ring formed by joining the last element of s to the first element.

If s is a sequence, then s^{-1} is the reverse of s . Hence, $\text{rev } s = s^{-1}$. Show the following:

$$\alpha s = \omega s^{-1}$$

$$\omega s = \alpha s^{-1}$$

$$A s = (\Omega s^{-1})^{-1}$$

$$\Omega s = (A s^{-1})^{-1}$$

$$(s \sim t)^{-1} = t^{-1} \wedge s^{-1}$$

$$(x \text{ cl } s)^{-1} = s^{-1} \text{ cr } x$$

$$(s \text{ cr } x)^{-1} = x \text{ cl } s^{-1}$$

$$(x,y)^{-1} = (y,x)$$

$$(x_1, x_2, \dots, x_n)^{-1} = (x_n, \dots, x_2, x_1)$$

RELATIONAL PROGRAMMING

$$\{x_1:y_1, x_2:y_2, \dots, x_n:y_n\}^{-1} = \{y_1:x_1, y_2:x_2, \dots, y_n:x_n\}$$

2.3 alternative definitions of sequences

We will state the formal definition of a sequence: a relation is a sequence if it is a connected irreflexive bijection. That is,

$$\text{sequence} = \text{connex} \cap \text{irrefl} \cap \text{bun}$$

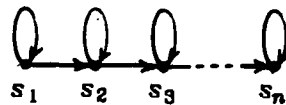
$$s \in \text{irrefl} \Leftrightarrow s^0 \subseteq s^{-1}$$

$$s \in \text{connex} \Leftrightarrow \text{dom } s = \text{init } s \cup \text{dom } s^{-1} \wedge \text{dom } s^{-1} = \text{term } s \cup \text{dom } s$$

Although the preceding definition of sequences is very convenient, it has a number of limitations. For example, the operations discussed above are only defined for sequences with two or more elements, since an irreflexive relation cannot relate less than two elements. In particular, $(2) = \emptyset$. One solution to this problem is to use a standard "end marker" for all sequences, say 'EOF'. For example, the sequence 1, 3, 5 would be represented by the relation $(1,3,5,\text{EOF})$. A one element sequence containing 3 would be represented by $(3,\text{EOF})$ and an empty sequence by $(\text{EOF}) = () = \emptyset$. This definition has some curious properties of its own. For example, the relation $(3,3,\text{EOF}) = \{3:3, 3:\text{EOF}\}$ has no initial members and in fact is not a sequence (since it's not irreflexive). Of course this objection also applies to our original definition of sequences.

A different solution is to extend the definition of sequences so as to allow length one sequences by making the relation reflexive.

$$s \cup (= \uparrow \text{ mem } s)$$



The one element sequence is then:

$$\begin{array}{c} \text{O} \\ \downarrow \\ s_0 \end{array} = (s_0, s_0)$$

This still does not solve the problem with repeating elements in sequences, however.

An alternative definition of sequences is based directly on the pair-making operation. Define $\langle \rangle$ to be some distinguished value 'nil'. Then define

$$\langle x_1, x_2, \dots, x_n \rangle = x_1 : \langle x_2, \dots, x_n \rangle$$

We can see that

$$\langle x_1, x_2, \dots, x_n \rangle = x_1 : x_2 : \dots : x_n : \text{nil}$$

(We have assumed ':' is right-associative in the above equation.) This is essentially the way lists are represented in LISP. A more comprehensive solution to these problems is discussed in Chapter 16, Data Structures.

10. Binary Operations

10.1 basic concepts

In this section we will discuss our approach to binary operations — that is, to functions with two arguments and one result. We have already seen how unary functions are connected to relations. For instance, we can write the fact that y is the sine of x by either $x \text{ siny}$ or $y = \sin x$. Since we only deal with binary relations, we will have to have a new convention for handling binary functions. This convention is: we will combine the two arguments of an operation into a pair. For instance, we can define a relation 'sum' such that $(x, y) \text{ sum } z$ if and only if z is the sum of x and y . More formally:

$$\text{sum} = \{a : z \mid \exists x, y [a = (x, y) \wedge z = x + y]\}$$

We can use our function application convention as usual, e.g.,

$$z = \text{sum}(x, y) \Leftrightarrow (x, y) \text{ sum } z$$

Now, it would be inconvenient to have to invent names, such as 'sum', for each operation, such as '+'. Hence, we will adopt a systematic convention for making such names: placing the conventional infix symbol for the operation in parentheses or other brackets. For instance,

RELATIONAL PROGRAMMING

$$(x,y)[+]z \Leftrightarrow z = [+](x,y) \Leftrightarrow z = x+y$$

In fact, if π is any infix operation symbol, we will explicitly define its meaning by

$$x\pi y = [\pi](x,y)$$

This notation will permit us to manipulate in a more regular fashion the usual arithmetic operations (+, -, \times , /) as well as the relational operations (e.g. \cap , \cup , \rightarrow , \leftarrow , \uparrow , \times). We omit the brackets when the meaning is clear without them. For instance, if S is a class of classes, then

$$\text{img} \cap (S \times S)$$

is the class of all pairwise intersections of members of S .

10.2 operations on binary operations

It is often convenient to be able to generate simple relations from a binary operation. Following Russell and Whitehead [Whitehead70], let π represent any binary operation. We define:

$$(\pi y) = \{x : x \pi y = z\}$$

$$(x \pi) = \{y : x \pi y = z\}$$

Hence, $x(-1)y \Leftrightarrow y = x-1$, therefore (-1) is the predecessor function. Similarly, $x(1+)y \Leftrightarrow y = 1+x$, therefore $(1+)$ and $(+1)$ are both the successor function. These can be used as functions: $(-1)x = x-1$ and $(+1)x = x+1$.

This convention makes it very easy to form more complex functions. For instance, if we want $f(x) = \sin(1/x)$ then we can define $f = \sin.(1/)$. To see that this works:

$$\begin{aligned} f(x) &= [\sin.(1/)]x \\ &= \sin[(1/)x] \\ &= \sin[1/x] \end{aligned}$$

Again, we omit the brackets when the meaning is clear from context or can be made clear by spacing. Furthermore, we adopt the convention that if two binary operators

RELATIONAL PROGRAMMING

occur together, then the first is taken in its unary sense and the second in its binary sense. For example, $s \times \cup t$ means $[s \times] \cup t$, not $s \times [\cup t]$. When a binary operator is used in its unary sense, it will be taken to be very binding; that is, $f.x \cup$ means $f.[x \cup]$, not $(f.x) \cup$.

Now observe the action of the $(x,)$ and $(,y)$ functions:

$$(x,.)y = (x,y)$$

$$(,y)x = (x,y)$$

Therefore, for any binary operation π (except ',') we can define

$$\pi y = [\pi].(,y)$$

$$x \pi = [\pi].(x,)$$

Let's see why this works:

$$\begin{aligned} (x \pi)y &= [(\pi).(x,)]y \\ &= (\pi)[(x,)y] \\ &= (\pi)[x,y] \\ &= x \pi y \end{aligned}$$

The form $(\pi y)x$ is analogous. In general, if f is a binary function, then $f.(x,)$ and $f.(,y)$ are the "partially instantiated" unary functions. This is the effect of Curry and Feys 'B' combinator [Curry58].

Since S^{-1} is the reverse of a sequence, $\pi.\text{inv}$ is the reverse form of an operation. For instance, $-. \text{inv}$ is the reverse subtract operation:

$$\begin{aligned} -. \text{inv}(x,y) &= -(\text{inv}(x,y)) \\ &= -(y,x) \\ &= y - x \end{aligned}$$

Thus $-. \text{inv}$ can be read 'subtract from' and $/ . \text{inv}$ can be read 'divide into'. This is Curry and Feys 'C' combinator (see the next section).

11. Combinators

11.1 paralleling of relations

In this section we will discuss several powerful operations for manipulating relations. These are called *combinators* because of their similarity to the combinators of Curry and Feys [Curry58].

The first combinator we will discuss is the *paralleling* of relations, $R||S$, which is defined:

$$(u,v)R||S(x,y) \Leftrightarrow uRx \wedge vSy$$

So, if f and g are functions, $[f||g](x,y) = [f(x), g(y)]$. Hence, $f||g$ is the element-wise combination of f and g . For example, if we want $f(x,y) = \sin x + \cos y$, we can write $f = +.(sin||cos)$ since

$$\begin{aligned} f(x,y) &= [+.(sin||cos)](x,y) \\ &= +[(sin||cos)(x,y)] \\ &= +[\sin x, \cos y] \\ &= \sin x + \cos y \end{aligned}$$

11.2 conditional union

The restriction operations allow us to define the very useful *conditional union* or *overlay* operation [MacLennan75], $R;S = R \cup \sim.\text{dom}R \rightarrow S$. In other words, the value of $(R;S)x$ is Rx if $x \in \text{dom}R$, and Sx otherwise. This has many uses. For example, if f is a partial function, then $f;\text{Id}$ is the extension of f to the identity function. That is, $(f;\text{Id})x$ is fx if that is defined and x otherwise.

The conditional union is useful for defining conditional-like structures. For example $p \rightarrow f;g$ is a function that applies f if p of its argument is true, and applies g otherwise:

$$[p \rightarrow f;g]x = \begin{cases} fx & \text{if } px \\ gx & \text{if } \neg px \end{cases}$$

(This assumes $p \subseteq \text{dom}f$. Why?) Therefore we have the equivalent of Backus' conditional

combining form $p \rightarrow f;g$.

The overlay operation is also useful for updating functions representing tables. For example $(i,x);T$ is a table just like T except that Ti is now x , regardless of whether Ti was defined or not. Similarly, $S;T$ is a table in which all the entries of S have been added to T , possibly replacing corresponding elements already there.

11.3 combinatory logic

One of the simplest combinators described by Curry and Feys is the *elementary cancellator*, K , defined so that Kx is a function such that $(Kx)y = x$ for all y . That is, K generates constant functions. Since Kx is a relation that relates x to everything, we can define it: $K = (Ox).un$, where $un = \mathcal{V}^{-1}$ is the unit class generator. To see that this works, note that

$$Kx = Ox . un \ x = Ox(un \ x)$$

and therefore that

$$\begin{aligned} u(Kx)v &\Leftrightarrow u[Ox(un \ x)]v \\ &\Leftrightarrow u \in O \wedge v \in un(x) \Leftrightarrow v = x \end{aligned}$$

Therefore, for arbitrary u , $(Kx)u = x$.

Another combinator is the *elementary duplicator*, W , defined so that $(Wf)x = f(x,x)$. If we define $\Delta x = (x,x)$ then it is easy to see that Wf is just $f.\Delta$. For instance, $x.\Delta$ is the squaring function:

$$x.\Delta \ n = x(\Delta n) = x(n,n) = n \times n = n^2$$

It should be clear that Backus' $[f,g]$ combining form is just our $(f||g).\Delta$, since

$$(f||g).\Delta \ x = f||g \ (x,x) = (fx, gx)$$

Since this combination is so common we will adopt a special notation for it:

$$f \bar{;} g = (f||g).\Delta. \text{ Hence, } (f \bar{;} g) \ x = (fx, gx)$$

RELATIONAL PROGRAMMING

EXERCISES: Show that some of the properties satisfied by these combinators are:

$$(R||S).(T||U) = (R.T)||S.U$$

$$(R||S)^n = (R^n)||S^n$$

$$(R;S).T = (R.T);S.T$$

$$(R||S).(T;U) = (R.T);S.U$$

$$\Delta.R = R;R$$

$$(R||S).inv = (S||R) = inv.(R||S)$$

$$inv.(R;S) = S;R$$

$$\alpha.(R;S) = (dom S) \rightarrow R$$

$$\omega.(R;S) = (dom R) \rightarrow S$$

$$R||S = (R.\alpha);(S.\omega)$$

$$cl = (\alpha;\Omega)^{-1}$$

$$cr = (\Lambda;\omega)^{-1}$$

EXERCISES: Show that $f = +.(x.\Delta;2x)$ is the function $f(t) = t^2 + 2t$.

The *formalizing combinator*, Φ , is defined so that $[\Phi f(a,b)]x = f[a(x),b(x)]$. It is easy to see that $\Phi f(a,b) = f.(a;b)$. For instance,

$$f = \Phi + [x.\Delta;2x]$$

is just the function $f(t) = t^2 + 2t$. This can be written directly using the notation of our relational calculus:

$$f = +.(x.\Delta;2x)$$

The combination $\Phi\pi[f,g]$ occurs very frequently. Therefore, we define $\bar{\pi} = \Phi\pi$ to be the formalization of the operator π . Notice that $[f\bar{\pi}g]x = (fx)\pi(gx)$. In particular, the function $ft = t^2 + 2t$ can be written

$$f = x.\Delta \mp 2x$$

In general, $\pi.(f;g) = f\bar{\pi}g$.

RELATIONAL PROGRAMMING

Another combinator is the meta-application operator, $\bar{\otimes}$, which corresponds to Curry and Feys' S combinator: $(f \bar{\otimes} g)x = (fx) \otimes (gx)$. For instance, $\text{img} \bar{\otimes} \text{init}$ is the operation that gives the set of descendents of roots of a forest F , since

$$(\text{img} \bar{\otimes} \text{init})F = (\text{img } F) \otimes (\text{init } F) = (\text{img } F) \cdot \text{init } F.$$

Another combinator defined by Curry and Feys is the Ψ combinator:

$$[\Psi(f, g)](x, y) = f[g(x), g(y)]$$

This is simply defined by $\Psi(f, g) = f \cdot (g \parallel g)$. Therefore, if $f = \Psi[+, \times, \Delta]$ then $f(x, y) = x^2 + y^2$. This can also be written $f = \times \cdot \Delta \cdot \alpha \mp \times \cdot \Delta \cdot \omega$.

11.4 Curried functions

A function f is called a *Curried* derivative of g if $fx y = g(x, y)$. We define operators 'curry' and 'uncurry' such that $f = \text{curry } g$ and $g = \text{uncurry } f$. First consider uncurry:

$$\begin{aligned} (\text{uncurry } f)(x, y) &= g(x, y) \\ &= fxy \\ &= (fx) \otimes y \\ &= \otimes[(fx), y] \\ &= \otimes.[f \parallel \text{Id}](x, y) \end{aligned}$$

By canceling (x, y) from both sides we see that $\text{uncurry } f = \otimes.[f \parallel \text{Id}]$. If we wish, we can factor f out of this expression in this manner:

$$\begin{aligned} \text{uncurry } f &= \otimes.[f \parallel \text{Id}] \\ &= [\otimes.][f \parallel \text{Id}] \\ &= [\otimes.].[\parallel \text{Id}]f \end{aligned}$$

Hence, $\text{uncurry} = [\otimes.].[\parallel \text{Id}]$.

Next consider Currying. One solution is to simply define $\text{curry} = \text{uncurry}^{-1}$; we can learn more however by defining curry directly. Suppose we are given a Curried pair-

RELATIONAL PROGRAMMING

making function ' π ': $\pi xy = (x, y)$. Then,

$$fxy = (\text{curry } g)xy = g(x, y) = g(\pi xy) = g([\pi x]y) = g. [\pi x]y$$

Therefore, canceling y from each side we get:

$$\text{curry } g \ x = g. (\pi x) = [g.](\pi x) = [g.]. \pi x$$

Hence, $\text{curry } g = [g.]. \pi$.

As a final example we derive Curry and Feys' C combinator: $Cfxy = fyx$. Observe:

$$Cfxy = fyx = (fy)@x = [@x](fy) = [@x].fy$$

Therefore, by canceling y we get:

$$Cfx = [@x].f = [.f][@x] = [.f].[@]x$$

Hence, $Cf = [.f].@$.

EXERCISES: Show the following properties satisfied by these combinators:

$$(Kx).f = \text{dom } f \rightarrow Kx$$

$$f.(Kx) = K(fx)$$

$$C = \text{curry}. [\text{inv}]. \text{uncurry}$$

12. Records

12.1 basic operations

By a *record* we mean a finite function whose domain is other than a contiguous subset of the integers. For example, the following relation might represent a personnel record:

$R = \{ \text{name} : \text{"Don Smith"}, \text{age} : 40, \\ \text{hire-date} : \{ \text{mo} : \text{"Aug"}, \text{dy} : 31, \text{yr} : 1980 \}, \text{salary} : 40000 \}$

The *selectors*

name, age, hire-date, mo, dy, yr, salary

might be the strings "name", "age", etc. or the integers 1, 2, etc.. We are not concerned with their exact nature so long as they are distinct. A field is selected by applying the record to the field's selector: $R(\text{age}) = R@ \text{age} = 40$. Thus ' $R@ \text{age}$ ' is analogous to Pascal or Ada's ' $R.\text{age}$ '. Next we will consider how records can be manipulated using the relational operators.

Notice that if D is a record of default values (say, for a personnel record) and R is a record providing values for only some of the fields of a personnel record, then $R;D$ is a complete personnel record with defaults from D provided for the unspecified fields of R . If R and S are records with disjoint selectors (or with overlapping selectors whose values agree) then $R \cup S$ is a join or combination of these two records. Finally, if S is a set of selectors, then $S \rightarrow R$ is a subrecord of R containing only the fields whose selectors are in S . For example,

$$\{\text{age, salary}\} \rightarrow R = \{ \text{age: 40, salary: 40000} \}$$

12.2 functional records

A common situation is to apply the same function f to every field of a record R . For example, we might want to negate the coordinates of a two-dimensional point $P = \{X:10, Y:30\}$. This is easily accomplished by $[0-].P$. Therefore P 's Y coordinate is:

$$[0-].P @ Y = [0-](PY) = [0-](30) = 0-30 = -30$$

In general, we can see that the φ field of $f.R$ is $f(R\varphi)$: $f.R@ \varphi = f(R\varphi)$.

Now suppose that we have a record F whose fields are functions f_1, f_2, \dots, f_n :

$$F = \{\varphi_1:f_1, \varphi_2:f_2, \dots, \varphi_n:f_n\}$$

We want to compute a record R that has the same shape as F , but with fields whose values are $f_i x$, for a given x :

$$R = \{\varphi_1:f_1 x, \varphi_2:f_2 x, \dots, \varphi_n:f_n x\}$$

Therefore, for any selector φ ,

RELATIONAL PROGRAMMING

$$R\varphi = F\varphi x = (F\varphi)@x = [@x](F\varphi) = [@x].F\varphi$$

Hence, $R = [@x].F$. We define $\hat{\Theta}$ to be the application of a functional record to an argument, $F\hat{\Theta}x = @x.F$. Notice that $F\hat{\Theta}$ is the function derived from the functional record F . Further applications are discussed in §14, on arrays.

We have seen how $f.R$ applies a function to a record argument to yield a record result and $F\hat{\Theta}x$ applies a functional record to an argument to yield a record result. Next we will investigate the application of a functional record to a record argument to yield a record result. In the simplest case F and R are the same shape and we want to apply corresponding elements of F to corresponding elements of R to yield corresponding elements of the result. Thus, if we let S be the result record, then for any field φ :

$$S\varphi = (F\varphi)@(R\varphi) = (F\bar{\Theta}R)\varphi$$

using the meta-application operator $\bar{\Theta}$. Hence, $S = F\bar{\Theta}R$. Therefore, we can apply corresponding elements of F to corresponding elements of R by $F\bar{\Theta}R$.

For an example of this operation, suppose that we have the personnel record R defined in §12.1 and that we want to compute a new record S in which the age field of R has been incremented and in which the salary field has been increased by 10%. We can accomplish this by $S = F\bar{\Theta}R$, where

$$F = \{\text{name: Id, age: 1+, salary: 1.1x, hire-date: Id}\}$$

A common situation is to update one field and leave the rest unchanged. The R record with its age field incremented is just $[\text{age}, 1+; \mathbf{K} \text{Id}]\bar{\Theta}R$.

We next consider a generalization of meta-application: the *outer product* of a functional record and its argument record. Suppose we have a record F of functions and a record R of arguments; the records F and R are not assumed to have the same shape (i.e., the same domain members). We define $S = \text{outer}FR$, the outer product of F and R to be a record with the same shape as F , each of whose fields has the same shape as

R . That is, if φ is a field selector of F and ψ is a field selector of R , then $S\varphi$ is a record in which $S\varphi\psi$ is the result of applying $F\varphi$ to $R\psi$. That is, $S\varphi = (F\varphi).R$, since $(F\varphi).R$ applies $F\varphi$ to each field of R and forms a record of the results. Therefore, $S\varphi = (.R)(F\varphi) = .R.F\varphi$. This yields the definition of the outer product: $.R.F = \text{outer}FR$. Further applications of this operator will be discussed in §14, on arrays.

EXERCISES: Define an outer product that yields the transpose of this result. That is, $S\varphi\psi = (F\psi)(R\varphi)$.

12.3 relational databases

Next we consider *databases* composed of record-sets and define functions that are analogous to the relational operations of Codd [Codd70]. Let D be a record set whose elements have the selectors {name, age, hire-date, salary}; D might represent part of an employee database. Observe that if f is any operation applicable to a record then $\text{img}f$ is a corresponding function applicable to the entire record set.

For example, to form a *projection* composed of just the 'age' and 'salary' fields of D we write $\text{img}[\{\text{age}, \text{salary}\} \rightarrow]D$. To compute D' in which every employee in D has been given a 10% raise, we can write

$$D' = \text{img} [(\{\text{salary}: 1.1\times\}; \text{KId})\bar{\otimes}]D$$

In other words, we are applying a function to each record in D ; this function multiplies the 'salary' field by 1.1 and leaves the other fields intact.

Often we want to choose some selector φ of the records in D to be a *key* and generate a function F from D such that Fk is the record in D whose φ field is k . We write this $F = \text{index}_{\varphi}D$. Observe:

$$\begin{aligned} \tau = Fk &\Leftrightarrow \tau\varphi = k \wedge \tau \in D \\ &\Leftrightarrow \tau @ \varphi = k \wedge \tau \in D \\ &\Leftrightarrow [\text{@}\varphi]\tau = k \wedge \tau \in D \end{aligned}$$

RELATIONAL PROGRAMMING

$$\Leftrightarrow \tau = [\oplus \varphi]^{-1}k \wedge \tau \in D$$

$$\Leftrightarrow \tau = ([\oplus \varphi]^{-1} \leftarrow D)k$$

Hence, $F = [\oplus \varphi]^{-1} \leftarrow D$, so we define $\text{index}_{\varphi} D = [\oplus \varphi]^{-1} \leftarrow D$.

Another common operation is *selection*. For example, suppose we want P to be the set of all records in D whose 'age' field is greater than or equal to 65. The first step is to index the set on the age field: $A = \text{index age } D$. Notice that xAr if and only if record r from D has an age field equal to x . We can think of A as a multiple-valued function that takes ages into the records having those ages. Thus, if we apply $\text{img}A$ to a set of ages then we will get a set of all the records that have ages in the given set. Clearly, then

$$P = \text{img}A(\text{all} \geq 65) = \text{img}[\text{index age } D](\text{all} \geq 65)$$

This leads to a general definition of the selection function:

$$\text{select}_{\varphi} D = \text{img}[\text{index}_{\varphi} D]$$

Hence, $\text{select}_{\varphi} = \text{img}.(\text{index}_{\varphi})$, so $\text{select} = (\text{img}.)\text{index}$. With this definition of select we can write

$$\text{select age } D (\text{all} \geq 65)$$

to select all those records whose age is greater or equal to 65.

Finally, we consider the *join* of two record sets D and E , $\text{join}_{\varphi}(D, E)$. This is composed of records formed by combining all those records from D and E whose φ fields are equal. To accomplish this, first index D and E on their φ fields: $F = \text{index}_{\varphi} D$, $G = \text{index}_{\varphi} E$. Let k be any value of the field φ ; observe that $(F \parallel G)k$ is a pair (d, e) where $d \in D$, $e \in E$ and d and e both have their φ fields equal to k . Therefore, we want $d \cup e$ to be in the join. The set of all such pairs (d, e) is just the range (dom.inv) of the relation $F \parallel G$. Therefore, to get the join J we must apply the union operation to every record pair in the range of $F \parallel G$:

$$J = \text{img} \cup (\text{dom.inv}[F \parallel G])$$

RELATIONAL PROGRAMMING

This is the definition of the join operation. We can factor D and E out of the definition thus:

$$\begin{aligned}\text{join}\varphi(D,E) &= [\text{img}\cup].\text{dom}.\text{inv } F\|G \\ &= [\text{img}\cup].\text{dom}.\text{inv}[\text{index}\varphi D\|\text{index}\varphi E] \\ &= [\text{img}\cup].\text{dom}.\text{inv}.\|.(\text{index}\varphi D, \text{index}\varphi E) \\ &= [\text{img}\cup].\text{dom}.\text{inv}.\|.[\text{index}\varphi \parallel \text{index}\varphi](D,E)\end{aligned}$$

Therefore,

$$\text{join}\varphi = [\text{img}\cup].\text{dom}.\text{inv}.\|.[\text{index}\varphi \parallel \text{index}\varphi]$$

EXERCISES: Factor φ out of the definition of join.

13. Ancestral Relations

13.1 definition

Carnap [Carnap58] defines the relation of a property p being hereditary with respect to a relation r :

$$\begin{aligned}p \text{ Her } r &\Leftrightarrow \forall xy \{x \in p \wedge x r y \supset y \in p\} \\ &\Leftrightarrow \text{img}[r^{-1}]p \subseteq p\end{aligned}$$

This leads to the definition of the *ancestral of R of the first kind* as that relation which preserves all the hereditary properties of R . This is also called the *reflexive transitive closure* of R :

$$xR^*y \Leftrightarrow x \text{ Mmr } r \wedge \forall p [p \text{ Her } R \wedge x \in p \supset y \in p]$$

For example, if xPy means that x is a parent of y , then xP^*y means that x is an ancestor (or the same as) y . The *ancestral of the second kind* or *transitive closure* is also useful:

$$R^+ = R^*|R = R.R^*$$

Thus, P^+ means 'ancestor' in the colloquial sense. The easiest way to visualize the meanings of the ancestrals is by their expansion as infinite unions:

RELATIONAL PROGRAMMING

$$R^* = R^0 \cup R^1 \cup R^2 \cup R^3 \cup \dots$$

$$R^+ = R^1 \cup R^2 \cup R^3 \cup R^4 \cup \dots$$

EXERCISES: Here are some useful properties of the ancestrals. Prove them.

$$R^+ = R^* \setminus (=) = R^* \setminus R^0$$

$$xR^*y \Leftrightarrow \exists n[n \geq 0 \wedge xR^ny]$$

$$R^0 \subseteq R^*$$

$$R^n \subseteq R^*, \text{ for } n \geq 0$$

$$R^n \subseteq R^+, \text{ for } n > 0$$

$$R \cup R^+ = R^+$$

$$R^+ \subseteq R^*$$

$$R^+ = R \mid R^*$$

$$R^* = R^0 \cup R^+$$

$$(R^*)^{-1} = (R^{-1})^*$$

$$(R^+)^{-1} = (R^{-1})^+$$

$$(r \uparrow s)^* \subseteq r^* \uparrow s$$

Ancestral relations are always transitive. Notice that \leq and $<$ for integers can be defined:

$$\leq = (1+)^*$$

$$< = (1+)^+$$

That is, $x \leq y$ means that y can be reached from x by zero or more applications of the successor function $(1+)$. The ancestral "fills out" all of the paths in a structure. For instance, if

$$R = \begin{array}{cccc} a_1 & a_2 & a_3 & a_4 \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{array}$$

then



13.2 applications

Suppose that S is a sequence and we wish to find the first member of S which satisfies some property P . First form the closure S^+ , so that for any two members of S^+ we can tell which is first. Next, eliminate from S^+ any members that do not satisfy P : $S^+ \upharpoonright P$. Then, $\alpha(S^+ \upharpoonright P)$ is the first member of S satisfying P .

Next we will consider a simple character manipulation example: stripping leading blanks from a string. Note that $x (y \text{ cl})^* z$ means that z is a result of consing 0 or more y 's on the front of x . Hence, $z [(y \text{ cl})^*]^{-1} x$ means that x is the result of stripping one or more y 's from the front of z . To get the desired result it is only necessary to restrict the left domain of this function to be sequences that don't begin with a y . Suppose Y is the property of beginning with a y :

$$x \in Y \Leftrightarrow y = \alpha x \Leftrightarrow x \alpha y \Leftrightarrow x \in \text{all } y$$

Therefore, the function to strip leading y 's from a sequence is:

$$[(y \text{ cl})^*]^{-1} \leftarrow \sim.(\text{all } y)$$

13.3 iteration

Before we leave the topic of ancestral relations, it will be useful to investigate their use as a means of iteration. Suppose that F is a function (i.e., right univalent). Then, since

$$F^+ = F^1 \cup F^2 \cup F^3 \cup \dots$$

we will have $x F^+ y$ if and only if for some $n > 0$, $y = F^n x$. In general there may be many such n , so F^+ may not be a function. If F^+ is to be a function, it is necessary to pick a *termination condition* (a class) that is only true for one of $F^1 x$, $F^2 x$, $F^3 x$, Therefore

consider the relation $F^+ \leftarrow \sim \cdot \text{dom } F$. Let $\bar{D} = \sim \cdot \text{dom } F$ to see the effect of this function:

$$F^+ \leftarrow \bar{D} = (F^1 \cup F^2 \cup F^3 \cup \dots) \leftarrow \bar{D} = (F \leftarrow \bar{D} \cup F^2 \leftarrow \bar{D} \cup F^3 \leftarrow \bar{D} \cup \dots)$$

Then $(F^+ \leftarrow \bar{D})x$ is $F^n x$ where n is the unique $n > 0$ such that $F^n x$ is defined but $F^{n+1}x$ is not. This n is unique because $F^{n+1}x$ undefined implies that for all $m > n$ $F^m x$ is undefined. This leads to the definition of iter F :

$$\text{iter } F = F^+ \leftarrow \sim \cdot \text{dom } F$$

Notice that $\text{iter}[P \rightarrow F]$ will iterate the application of F so long as its argument satisfies P (and is in the domain of F). Since it always applies F at least once it is not like a **while** loop; the equivalent of the **while** loop

while $\neg P$ **do** F

is $\text{iter}[P \rightarrow F]; \text{Id}$, since any input not in $P \cap \text{dom } F$ will be passed through. Hence we define $\text{while}[P, F] = \text{iter}[P \rightarrow F]; \text{Id}$. Analogously, $F | \text{while}[P, F]$ is equivalent to

repeat F **until** $\neg P$

14. Arrays

14.1 definition and basic operations

An array is just a function from a contiguous subset of the integers to some set of values. If A is an array and $i \in \text{dom } A$ then $A(i)$ is the i -th element of A . Similarly, if $I \subseteq \text{dom } A$ is a set of index values then $\text{img } A|I$ is the corresponding set of array values and $I \rightarrow A$ is the subarray of A selected by those indices.

It is easy to define multi-dimensional arrays: they are just arrays whose elements are selected by sequences of integers, e.g. $M(i, j)$. If M is a two-dimensional array, then $M.(i,)$ is the i -th row of M and $M.(, j)$ is the j -th column of M . Also, if I is a set of row indices and J is a set of column indices then $I \times J \rightarrow M$ is the submatrix of M selected by these sets. It is easy to see that $M.\text{inv}$ is the transpose of M , since

$$M.\text{inv}(i, j) = M[\text{inv}(i, j)] = M(j, i)$$

RELATIONAL PROGRAMMING

More generally, if P is a permutation function (i.e. a bijection from an index set into itself) then $A.P$ is the result of permuting A by P .

APL-like array and matrix operations are easy to express with the relational operators. For example, if A is an array, then $f.A$ is the array resulting from applying f to every element of A . This follows from the definition of composition, $(f.A)i = f(Ai)$. Hence, $\sin.A$ applies \sin to every element of A . Conversely, if F is an array of functions, then $F \odot x$ is an array of results obtained by applying each element of F to x . That is, $(F \odot x)i = (Fi)x$. Also, if F is an array of functions and A is an array of arguments, then $F \bar{\odot} A$ is an array of results obtained by corresponding elements of F to corresponding elements of A . This follows from $(F \bar{\odot} A)i = Fi \odot Ai$.

Note that if A and B are two arrays with the same domain, then $A \bar{+} B$ is the element-wise sum of these two arrays. To see this, suppose that $C = A \bar{+} B$ and consider an arbitrary element of C :

$$Ci = (A \bar{+} B)i = Ai + Bi$$

In general, if π is an infix binary operation, then $\bar{\pi}$ is the element-wise extension of that operation to arrays. If f is any binary function, then $f.(A \bar{+} B)$ is the element-wise application of it to arrays A and B .

The same approach works for matrices and arrays of higher dimensionality. Suppose that M and N are two-dimensional matrices with the same domains. Then, $f.M(i,j) = f[M(i,j)]$ and

$$(M \bar{+} N)(i,j) = M(i,j) + N(i,j)$$

As for one-dimensional arrays, a matrix of functions can be applied to a single argument by $M \odot x$, and a matrix of functions can be applied to a matrix of arguments by $M \bar{\odot} N$.

If A and B are arrays, then $C = f.(A \parallel B)$ is an *outer product* by f of A and B , since $C(i,j) = f(Ai, Bj)$. For example,

$$\times.([1..12] \parallel [1..12])$$

is a 12 by 12 multiplication table. We can also form an outer product between an array of functions and an array of arguments. If F is an array of functions and A is an array of arguments, then $P = @.(F||A)$ is a matrix in which $P(i,j) = (Fi)(Aj)$.

EXERCISES: Prove that $@.(F||A) = \text{uncurry.outer } F \ A$.

Suppose x is an element of the array A (i.e., for some i , $x=Ai$). Then $\text{allA } x$ is the set of all indices for which $x=Ai$. Therefore we can find the index of the first occurrence of x in A (i.e. APL's iota operator) by $\text{min}(\text{allA } x)$. In general, if P is some property (i.e. class), then $\text{imgA}^{-1}P$ is the set of indices of all elements of A that satisfy P . A sorted reflexive sequence of these indices is just $\leq \uparrow \text{imgA}^{-1}P$

14.2 relation to sequences

It is easy to convert arrays to sequences and *vice versa*. Suppose all the elements of A are distinct, then A^{-1} is a function that returns the index of an element of A . We want to define a sequence S such that xSy if and only if x preceeds y in A , i.e. the index of x is one less than the index of y . To put this functionally, we want to define S so that $y=Sx$ means that y is the successor of x in A , i.e., that the index of y is one greater than the index of x .

$$y = Sx \Leftrightarrow A^{-1}y = A^{-1}x + 1$$

$$\Leftrightarrow A^{-1}y = (1+).A^{-1}x$$

$$\Leftrightarrow y = A.(1+).A^{-1}x$$

Hence, $S = A.(1+).A^{-1}$. Notice that this is just the image of the $(1+)$ structure under the function A^{-1} : $S = A^{-1}\$ (1+)$ (the $\$$ operation is discussed in the next chapter).

Next, we will consider the opposite process: converting a sequence to an array. Suppose we have a sequence:

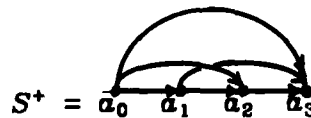
$$S = \underset{\cdot}{a_0} \underset{\cdot}{a_1} \underset{\cdot}{a_2} \underset{\cdot}{a_3}$$

RELATIONAL PROGRAMMING

We wish to convert this to an array:

$$A = \begin{array}{|c|c|} \hline 0 & a_0 \\ \hline 1 & a_1 \\ \hline 2 & a_2 \\ \hline 3 & a_3 \\ \hline \end{array}$$

Thus, for each element a_i in the sequence, we must find its index i in the resulting array. If we can define a relation R such that $R(a_i) = i$ then R^{-1} will be the array we seek. Now $R(a_i)$ is just the number of predecessors of a_i in S . That is, a_0 has no predecessors, so $R(a_0) = 0$; a_2 has two predecessors, so $R(a_2) = 2$, and so on. Since S defined an immediate predecessor relation, S^+ defines an ancestral predecessor relation:



Since xSy means x is a predecessor of y , $y = Sx$ means y is a successor of x . Thus the set of successors of any element a is then $\text{unimg } S^+a$, and the set of predecessors of a is $\text{unimg } \text{inv} S^+a$, e.g.

$$\text{unimg } \text{inv} S^+a_2 = \{a_0, a_1\}$$

Alternately, $\text{unimg } \text{inv} S^+a = \text{all } S^+a$ is the set of all elements that bear the S^+ relation to a . The size of this class is then the desired index:

$$\text{size}(\text{all } S^+a_2) = 2$$

Hence, $R(a) = \text{size}(\text{all } S^+a)$, so $R = \text{size} \circ (\text{all } S^+)$. Now, we know that A is R^{-1} , so we can define the function `sa0` which converts a sequence into a 0-origin array:

$$\text{sa } S = [\text{size}(\text{all } S^*)]^{-1}$$

To produce a 1-origin array, the only alteration is:

$$\text{sa } S = [\text{size}(\text{all } S^*)]^{-1}$$

14.3 other array operations

Next we will consider the concatenation of arrays. If A is an array such that $Ai = a_i$, then we can write A :

$$A = \{1:a_1, 2:a_2, \dots, m:a_m\}$$

where m is the length of the array. Similarly, suppose that B is an n element array, then the concatenation of these arrays is

$$A \text{ cat } B = \{1:a_1, \dots, m:a_m\} \cup \{m+1:b_1, \dots, m+n:b_n\}$$

We can see that $A \text{ cat } B = A \cup B'$ where B' results from B by shift its indices by m :

$$B' = \{m+1:b_1, \dots, m+n:b_n\}$$

How do we compute B' ? Observe:

$$B'i = B(i-m) = B[(-m)i] = B.(-m)i$$

Hence, $B' = B.(-m)$ and $A \text{ cat } B = A \cup B.(-m)$, where m is the length of A . The length of A is just $\text{size}.\text{dom } A$, so

$$A \text{ cat } B = A \cup B.(-\text{size}.\text{dom } A)$$

We will finish our discussion of arrays by investigating the generation of sorted arrays. Let S be a set of integers to be sorted, then $[\leq^* S]$ is a structure which relates lesser elements to greater elements. Now if x is any element of the set, $\text{all}[\leq^* S]x$ is the set of all elements less or equal to than x . Thus $\text{size}(\text{all}[\leq^* S]x) = \text{size}(\text{all } \leq^* S)x$ is the *number* of elements of S less than or equal to x . This is just the index of x in the sorted array we seek. Hence if A is the sorted array, iAx if and only if $x[\text{size}(\text{all } \leq^* S)]i$, so $A = [\text{size}(\text{all } \leq^* S)]^{-1}$. Of course this can be generalized to any ordering relation.

15. Isomorphic and Homomorphic Images

15.1 images

Consider any relation R and any biunivalent function f . If we take each node n of R and replace it by fn we get a relation closely related to R called the *image* of R under f , symbolized $f \mathcal{S} R$.

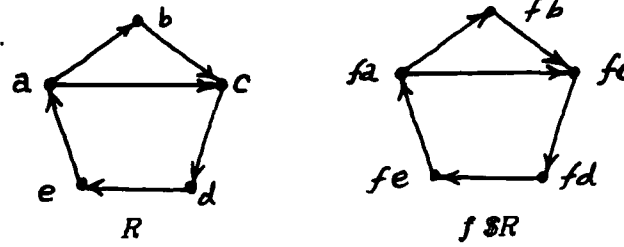


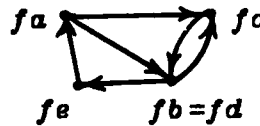
Figure 13. Image of a Relation

It is easy to define $f \mathcal{S} R$. Observe that if $S = f \mathcal{S} R$ then $(fx)S(fy)$ just when xRy . Conversely uSv whenever there are x and y such that xRy , $u=fx$ and $v=fy$. Hence,

$$\begin{aligned} uSv &\Leftrightarrow \exists xy [xRy \wedge u=fx \wedge v=fy] \\ &\Leftrightarrow \exists xy [xfu \wedge xRy \wedge yfv] \\ &\Leftrightarrow \exists xy [uf^{-1}x \wedge xRy \wedge yfv] \\ &\Leftrightarrow u[f^{-1}|R|f]v \end{aligned}$$

Hence, $f \mathcal{S} R = f^{-1}|R|f = f.R.f^{-1}$.

The image operation is also useful when f is not biunivalent. For example, if $fb=fd$ then $f \mathcal{S} R$ (with the R in Figure 13) is:



I.e., we merge the nodes corresponding to b and d .

The \mathcal{S} operation is clearly related to the img operation — they both compute the image of a structure. Since $\vartheta.[f||f].\text{un } x:y = (fx):(fy)$, we have this relationship between the images of relations and sets:

$$f \mathcal{S} R = \text{img}(\vartheta.[f||f].\text{un})R$$

RELATIONAL PROGRAMMING

That is, $f \mathcal{S} = \text{img}(\vartheta.[f || f].\text{un})$.

The image operations have many uses. For example, since $[1..n] = (1, 2, \dots, n)$, we can see that

$$(m+) \mathcal{S}[1..n] = (m+1, m+2, \dots, m+n)$$

Hence the identity $[m+1..m+n] = (m+) \mathcal{S}[1..n]$. To compute a list of the powers of two from 2^0 to 2^{16} we write $(2\uparrow) \mathcal{S}[0..16]$, where $x \uparrow y = x^y$. Finally, to compute a list of the sines of the angles from $0^\circ = 0 \text{ rad.}$ to $90^\circ = \pi/2 \text{ rad.}$ we write

$$\text{sin.}(\times\pi/180) \mathcal{S}[0..90]$$

To see that this works:

$$\begin{aligned} & \text{sin.}(\times\pi/180) \mathcal{S}(0, 1, \dots, 90) \\ &= (\text{sin.}(\times\pi/180)0, \text{sin.}(\times\pi/180)1, \dots, \text{sin.}(\times\pi/180)90) \\ &= (\text{sin}(0\times\pi/180), \text{sin}(1\times\pi/180), \dots, \text{sin}(90\times\pi/180)) \\ &= (\sin 0^\circ, \sin 1^\circ, \dots, \sin 90^\circ) \end{aligned}$$

EXERCISES: Show that $\text{inv} = [\vartheta.\text{inv}.\text{un}] \mathcal{S}$.

15.2 images of functional structures

We have seen how, given a function f and a relation of values V we can form a relation $f \mathcal{S} V$ in which the shape of V is the same as the shape of $f \mathcal{S} V$ and each member $v \in \text{mem} V$ corresponds to $f v$ in $f \mathcal{S} V$. Now we will address the converse problem: given a relation of functions F and a value v , how can we construct a relation $F!v$ such that the shape of F is the same as the shape of $F!v$ and each member $f \in \text{mem} F$ corresponds to $f v$ in $F!v$. This is clearly the image of F under some unknown function φ : $F!v = \varphi \mathcal{S} F$. We will solve for φ . Observe $\varphi f = f v = f @ v = (@v) f$. Hence $\varphi = (@v)$ and $F!v = (@v) \mathcal{S} F$. That is, $F!v$ is the image of F under the operation 'apply to v '. We can eliminate v from this definition:

$$(F!)v = F!v = (@v) \mathcal{S} F = (\mathcal{S} F)(@v) = (\mathcal{S} F) @ v$$

RELATIONAL PROGRAMMING

Therefore $F! = (\$F) \cdot @$. Notice that the $F!$ is a function derived from a functional structure just as $F\hat{}$ is a function derived from a functional record.

We now consider some applications of this operation. To form a sequence by applying each of a sequence of operations to the same argument we write, for example:

$$(\sin, \cos, \tan)! \vartheta = (\sin \vartheta, \cos \vartheta, \tan \vartheta)$$

In particular, $(f, g)!$ is just $f \cdot g$ and Backus' constructor $[f, g, \dots, h]$ is just our $[f, g, \dots, h]!$.

Recall our previous example in which we computed the sines of the angles from 0° to 89° by

$$\sin. (\times \pi / 180) \$ [0..89]$$

We can extend this to compute a table of the sines, cosines and tangents of the angles from 0° to 89° by using both of the image operations:

$$(\sin, \cos, \tan)! . (\times \pi / 180) \$ [0..89]$$

This produces a sequence of sequences of the form

$$((\sin 0^\circ, \cos 0^\circ, \tan 0^\circ), (\sin 1^\circ, \cos 1^\circ, \tan 1^\circ), \dots, (\sin 89^\circ, \cos 89^\circ, \tan 89^\circ))$$

In general $F!SR$ has an outer structure the same as R 's, each of the elements of which has a structure the same as F 's. Thus it is sort of an "outer product" between F and R in which the members are $f\tau$ for $f \text{ Mm } F$ and $\tau \text{ Mm } R$.

To convert a sequence of sequences such as this into a matrix requires an application of the sa operator at each level of structure. Let S be the sequence of sequences. First convert each of its elements to an array by $sa\$S$. Next, convert the resulting sequence to an array by $sa[sa\$S]$. The result of the latter operation is an array of arrays that can be converted to a two dimensional matrix by uncurrying. Thus the sequence to matrix conversion is

$$ssm S = \text{uncurry. } sa[sa\$S]$$

Therefore,

$$\text{ssm} = \text{sa} \$ | \text{sa} | \text{uncurry}$$

This can be read: To convert a sequence of sequences to a matrix, convert each of its elements to an array, convert the result to an array, and uncurry that result.

EXERCISES: Define an outer product operation PFR which has the outer structure of F but the inner structure of R . Thus, the matrix corresponding to PFR is the transpose of the matrix corresponding to $F! \$R$:

$$(\text{ssm } PFR).inv = \text{ssm}(F! \$R)$$

15.3 isomorphism and the structure function

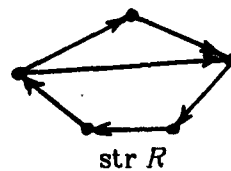
Carnap [Carnap58] defines two relations to be isomorphic when there is a biunivalent relation between their members that preserves their structure. That is, R is isomorphic to S :

$$R \simeq S \Leftrightarrow \exists f \in \text{bun}[R = f \$S]$$

Thus, two relations are isomorphic if one is a biunivalent image of the other. Equivalently, two relations are isomorphic if their arrow diagrams are equivalent when their node labels are removed. The isomorphism of sets is defined in the same way:

$$S \simeq T \Leftrightarrow \exists f \in \text{bun}[S = \text{img } f T]$$

The *structure* of a relation is arrow diagram for the relation with its node labels removed. For example, the structure of R in Figure 13 is:



Thus two relations are isomorphic if they have the same structure. Mathematically the structure of a relation is just the set of all relations isomorphic to the given relation: $\text{str } R = \{S | S \simeq R\} = \text{all } \simeq R$. Thus $\text{str } R$ is an equivalence class under \simeq . Alternately,

$$\begin{aligned}\text{str } R &= \{S \mid S \simeq R\} \\ &= \{S \mid \exists f \in \text{bun}[S = f \circ R]\} \\ &= \text{img}[R] \circ \text{bun}\end{aligned}$$

That is, $\text{str } R$ is the class of all biunivalent images of R . Note that $R \simeq S \Leftrightarrow \text{str } R = \text{str } S$.

The structure of a set is defined in exactly the same way. Since $\text{Cimg } SF = \text{img } FS$, we have

$$\text{str } S = \{T \mid T \simeq S\} = \text{all } S = \text{img}[\text{Cimg } S] \circ \text{bun}$$

Observe that if, following Russell and Whitehead [Whitehead70], we define a number as the class of all classes isomorphic to a given class, then the size of a class is just the set of all classes isomorphic to that class: $\text{size } S = \{T \mid T \simeq S\}$. But this is just the definition of the structure of a class. Hence for all sets S , $\text{size } S = \text{str } S$. In other words, the structure of a set is its cardinality. When the identity of its elements is ignored, the only structural characteristic still possessed by a set is its size:

$$\text{str}\{1,8,2\} = \text{str}\{\text{cat,dog,cow}\} = \{\cdot,\cdot,\cdot\} = 3$$

16. Data Structures

16.1 definition

Simple relations are not adequate for modeling all structures. For example, suppose we write this sequence: (1,2,3,2,4,5). This is defined to be the relation

$$\{ 1:2, 2:3, 3:2, 2:4, 4:5 \}$$

To make its structure more apparent, we will draw this as an arrow diagram:



This is certainly not what we expected, and it will not give the results we expect. For example, we cannot scan through this "sequence" because $R(2)$ is multiple valued.

RELATIONAL PROGRAMMING

To avoid this problem it is often better to use *data structures* (or interpreted structures). A data structure S is a pair (D, R) , where R is a relation (a simple structure) that defines the *form part* of the data structure, and D is a function that associates data values with the members of R ; it is called the *data part* of the data structure. Usually $\text{dom} D = \text{mem} R$, but this does not have to be the case; we will see examples later.

The structure that we intended by writing $(1, 2, 3, 2, 4, 5)$ can be correctly represented by a data structure (D, R) in which $R = (a, b, c, d, e, f)$ and

$$D = \{a:1, b:2, c:3, d:2, e:4, f:5\}.$$

It doesn't matter what a, b, c, d, e, f are, so long as they are distinct. We will write data structure sequences with angle brackets: $\langle 1, 2, 3, 2, 4, 5 \rangle$.

16.2 operations on data structures

We need functions for both interrogating and updating data structures. The data and form parts of data structures can be extracted by α and ω , respectively. In particular, if n is a node in S , $n \in \text{mem}(\omega S)$, then $\alpha S n$ is the value associated with that node. A common situation is to inquire the value of a node selected by applying a function f to the form of a data structure; we write this $\nu f S$. For example, $\nu \alpha S$ is the value of the first element of S and $\nu(\omega S) S$ is the value of the second element of S . In general,

$$\nu f (D, R) = D(f R) = D @ f R = @ (D, f R) = @. [Id || f] (D, R)$$

Therefore, $\nu f = @. [Id || f]$ and $\nu = (@.). (Id ||)$.

Next we define operators φ and δ that alter their argument function so that it operates on either the form or the data part of a data structure, but leaves the other part unchanged. That is, $\varphi f (D, R) = (D, f R)$ and $\delta f (D, R) = (f D, R)$. Therefore $\varphi f = Id || f$ and $\delta f = f || Id$, so $\varphi = Id ||$ and $\delta = || Id$.

We will define an operation Π such that $\Pi f S$ is the image of the structure S under the function f , that is, $\Pi f S$ is a structure with the same form as S but with values

RELATIONAL PROGRAMMING

derived by applying f to the data of S . Thus $\Pi f R$ is the analog for data structures of $f \mathcal{S} R$ for relations and $f.R$ for records and arrays. Suppose $S = (D, R)$ and $\Pi f R = (D', R')$. For any $n \in \text{mem} R$ we must have $D'n = f(Dn)$, so $D' = f.D = [f.]D$. Hence, we get $\Pi f S$ from S by applying $[f.]$ to the data part of S , so $\Pi f S = \delta[f.]S$, and $\Pi f = \delta[f.]$. For example, if S is any data structure whose values are numbers, then $\Pi[1+]S$ adds one to each element of the data structure.

The Π operator leaves the form of the data structure unchanged; next we consider operators that *reform* data structures. First we define operators that *filter* a data structure by removing some of its nodes. In the simplest case we just throw away the nodes we don't want, only retaining those that satisfy a given property P . Hence,

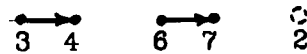
$$(D, R') = (D, R \uparrow P) = (D, [\uparrow P]R) = \varphi[\uparrow P](D, R)$$

Hence $\varphi[\uparrow P]$ filters a data structure by eliminating all those nodes that do not satisfy P . Suppose that we want to eliminate the negative nodes of a data structure. Thus we want $x \in P \Leftrightarrow \alpha S n \geq 0 \Leftrightarrow n(\alpha S) \geq 0$, so $P = \text{all}(\alpha S \geq 0)$.

This simple form of filtering will often lead to nodes becoming isolated. That is, if we filter the sequence

$$\langle 3, 4, -2, 6, 7, -1, 2, -4 \rangle$$

by the set $P = \text{all}(\alpha S \geq 0)$ then we will get



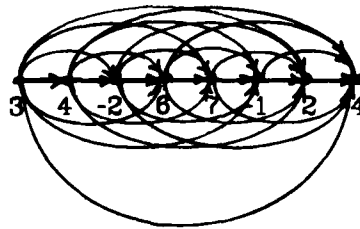
Note that the node whose value is 2 (a positive number!) is not even in the relation anymore since it has no neighbors (it is still in the data mapping, however). Usually we would prefer to connect up the remaining elements of the sequence, yielding $\langle 3, 4, 6, 7, 2 \rangle$. How can this be accomplished?

We will define an operator Φ such that $\Phi P S$ is the data structure resulting from filtering the data structure S by the predicate P . Suppose $S = (D, R)$ and $\Phi P S = (D, R')$. R' will be derived from R by adding some new pairs to $R \uparrow P$. In

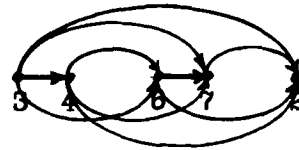
RELATIONAL PROGRAMMING

particular, we want to add just enough pairs to directly connect those nodes that were indirectly connected in R but are not indirectly connected in $R \uparrow P$. We will call this operation ξ , so $R' = \xi R$.

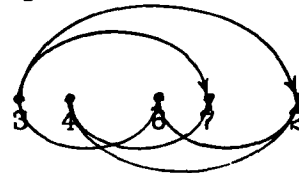
Observe that xR^+y if and only if y is reachable from x in one or more steps. Similarly $x(R|R^+)y$ if and only if y is reachable from x in *two* or more steps. Therefore, first take our original relation R and compute R^+ :



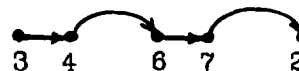
Then eliminate the undesirable members by restriction, $S = R \uparrow P$:



There are clearly many redundant edges here. We want to eliminate any edges that can be generated from the others; that is, we want a *minimal* set of edges. Since $S|S^+$ are all the edges of length two or greater, these are the redundant edges:



If we delete these edges from S we will have only the nonredundant edges left, so $S \setminus (S|S^+)$ is



We can now define ξ . First we define a useful operation μ that minimizes a relation by eliminating all of its redundant edges: $\mu R = R \setminus R|R^+$. To see that this works just

RELATIONAL PROGRAMMING

expand the transitive closure:

$$\begin{aligned}
 \mu R &= R \setminus R \mid R^+ \\
 &= R \setminus R \mid (R^1 \cup R^2 \cup R^3 \cup \dots) \\
 &= R \setminus (R^2 \cup R^3 \cup R^4 \dots) \\
 &= R \setminus R^2 \setminus R^3 \setminus R^4 \setminus \dots
 \end{aligned}$$

Hence, to filter a relation R by a predicate P we use $\mu[R^+ \uparrow P]$. Therefore,

$$\begin{aligned}
 \xi PR &= R' = \mu[R^+ \uparrow P] \\
 &= \mu.[\uparrow P][\text{trac} R] \\
 &= \mu.[\uparrow P].\text{trac } R
 \end{aligned}$$

where we have used $\text{trac } R = R^+$. Hence we have that

$$\xi P = \mu.[\uparrow P].\text{trac}$$

Notice that this definition is really quite readable. It says, "To filter a relation, compute the transitive closure (trac), eliminate undesirable nodes ($\uparrow P$), and eliminate redundant edges (μ)."

We now want to extend ξ into the operation Φ on data structures. Recall that ΦPS means that a node is to be included in the result only if its value satisfies P . Hence, if $S = (D, R)$ then we want to filter R by F where $n \in F$ if and only if $Dn \in P$. Now, the set of all nodes whose value is in P is just the inverse image of P under D , $F = \text{img.invd}P$. Therefore, we want to filter R by $\text{img.invd}P$, which we do by $\xi(\text{img.invd}P)R$. Hence, $\Phi P(D, R) = (D, \xi[\text{img.invd}P]R)$. We can factor (D, R) out of this equation:

$$\begin{aligned}
 \Phi PS &= \Phi P(D, R) \\
 &= (D, \xi[\text{img.invd}P]R) \\
 &= (D, \xi[(\Phi P).\text{img.invd}]R) \\
 &= (D, \xi.(\Phi P).\text{img.invd} R) \\
 &= (D, \text{uncurry}[\xi.(\Phi P).\text{img.invd}](D, R)) \\
 &= (\alpha S, \text{uncurry}[\xi.(\Phi P).\text{img.invd}]S)
 \end{aligned}$$

$$= (\alpha \bar{\cdot} \text{uncurry}[\xi.(\oplus P).\text{img}.\text{inv}])S$$

Therefore,

$$\Phi P = \alpha \bar{\cdot} \text{uncurry}[\xi.(\oplus P).\text{img}.\text{inv}]$$

17. Reducing Structures

17.1 basic concepts

In this section we will discuss several methods for *reducing structures*, that is, for applying a function to each element of a structure and accumulating the results. Since no one method has yet been selected, this section should be taken as a report of work in progress.

A general paradigm for processing a structure, such as a file, is the following:

1. Perform some initialization.
2. Read the next (or first) element of the file.
3. Take this value and the results of processing the previous values.
4. Process these to yield new cumulative values and continue from step (2).
5. When the end of the file is reached, return the accumulated result of processing all of its elements.

A simple form of this appears in APL's reduction operation:

$$+/V = V_1 + (\cdots (V_{n-1} + V_n) \cdots)$$

A more general form is Backus' insert:

$$/f : \langle x_1, \dots, x_n \rangle = f : \langle x_1, \dots, f : \langle x_{n-1}, x_n \rangle, \dots \rangle$$

Our first example of scanning structures will be to express this operation in the relational calculus.

RELATIONAL PROGRAMMING

17.2 reduction of arrays

We are given an n element array A and wish to compute:

$$t = A(n) + A(n-1) + \dots + A(2) + A(1)$$

where we have assumed that the domain of A is $[1..n]$. We saw in the section on ancestrals (§13) that $\text{iter}[\sim T \rightarrow F]$ will iterate the application of F with T used as the termination condition. Consider how the analogous loop would be written in Pascal:

```
S := 0; i := 0;
while i <= n do
begin S := S + A[i]; i := i + 1 end
```

On each iteration two functions are performed: S is increased by $A[i]$ and i is incremented by 1. Let's represent the state of the computation by a pair (s, i) , where s is the cumulative sum so far and i is the index of the next element to process. We will use F to represent one processing step, so that, if (s', i') is the new state, we can solve for F as follows:

$$\begin{aligned} F(s, i) &= (s', i') \\ &= (s + Ai, i + 1) \\ &= (+[s, Ai], [1+]i) \\ &= (+.[\text{Id}||A](s, i), [1+].\omega(s, i)) \\ &= (+.[\text{Id}||A] \cdot 1+. \omega)(s, i) \end{aligned}$$

Hence, $F = (+.[\text{Id}||A] \cdot 1+. \omega)$.

It remains to determine the termination condition, T . If x is a state, i.e., a pair (s, i) , then $x \in T$ when $i = n + 1$. Hence, $x \in T$ when $\omega x = n + 1$, so T is the set of all states mapped by ω into $n + 1$. Hence, $T = \text{all } \omega n + 1$. The final state, x_f , containing the sum is $\text{iter}[\sim T \rightarrow F]x_i$, where $x_i = (0, 1)$ is the initial state:

$$x_f = \text{iter}[\sim T \rightarrow F](0, 1)$$

Now, the total t is just ax_f , so

RELATIONAL PROGRAMMING

$$t = \alpha.\text{iter}[\sim T \rightarrow F](0,1)$$

We can generalize this to any function f with initial value i :

$$t = \alpha.\text{iter}[\sim T \rightarrow F](i,1)$$

$$\text{where } F = (f. [\text{Id} \parallel A]^{-1} 1 + \omega)$$

$$\text{and } T = \text{all } \omega (1 + \text{size } A)$$

This result can be improved by directly extracting the result from the final state. That is, we want to define a filter φ such that $t = \varphi.F^*(i,1)$. Hence we want $x_f \varphi t$, so $x_f \varphi t \Leftrightarrow (t, n+1) \varphi t$. Now, note that $[,n+1]t = (t, n+1)$, so $t [,n+1](t, n+1)$ by the definition of application. Therefore $\varphi = [,n+1]^{-1}$ and we have the simplified formula $t = [,n+1]^{-1}.F^*(i,1)$. This leads us to the following definition of the array reduction operation:

$$(f \text{ \texttt{ri}})A = [,n+1]^{-1}.F^*(i,m)$$

$$\text{where } F = (f. [\text{Id} \parallel A]^{-1} 1 + \omega)$$

$$\text{and } m = \min(\text{dom } A)$$

$$\text{and } n = \max(\text{dom } A)$$

Therefore, if A is an array indexed m to n , then $(+\text{t0})A$ is the summation of A ,

$$\sum_{i=m}^n A_i$$

Using this operation, the inner product of arrays A and B can be written simply as $+\text{t0}(A \times B)$.

EXERCISES: Show that $+\text{t0}(A \times B)$ is the inner product of A and B .

17.3 reduction of sequences

Next we will consider the scanning of sequences. Suppose S is a sequence:

$$S = (s_1, s_2, \dots, s_n, \text{EOF})$$

where EOF is an "end marker"; it can be any value. Now, we wish to find the result

RELATIONAL PROGRAMMING

$$i \ f \ s_1 \ f \ s_2 \ f \ \cdots \ f \ s_n$$

that is

$$f(f(\cdots f(i, s_1) \cdots), s_n)$$

for some function f and starting value i . The state can be represented by a pair (t, s) , where t is the result so far computed and s is the rest of the sequence to be processed. Hence, $(t', s') = F(t, s)$ where $t' = f(t, as)$ and $s' = \Omega s$. Therefore,

$$\begin{aligned} F(t, s) &= (t', s') = (f[t, as], \Omega s) \\ &= (f. [Id||\alpha](t, s), \Omega. \omega(t, s)) = (f. [Id||\alpha] \neg \Omega. \omega)(t, s). \end{aligned}$$

Hence, $F = f. [Id||\alpha] \neg \Omega. \omega$.

What is a terminal state? Notice that $\Omega(s_n, EOF) = \phi$, so a terminal state will have the form (r, ϕ) . Thus the set of terminal states is the set of all those states mapped into ϕ by ω : $\text{all}\omega\phi$. Hence,

$$r = \text{while}[\sim \text{all}\omega\phi, F](i, S)$$

To put this in a more useful form, we will define a function $f \S i$ such that $r = (f \S i)S$. This is simply

$$f \S i = \text{while}[\sim \text{all}\omega\phi, f. (Id||\alpha) \neg \Omega. \omega] . (i,)$$

Then, the sum of the elements of a sequence S is just $(+\S 0)S$.

18. Examples

In this section we will give several examples of relational programs.

18.1 payroll

Suppose we have a file Φ of employee records, where $r = \Phi n$ is the record for the employee with the employee number n . We will suppose that employee records are functions defined so that:

RELATIONAL PROGRAMMING

τN = employee name

τH = hours worked so far this week

τR = pay rate

We are given an update file U such that Un is the number of hours worked by employee n today. We wish to generate a new payroll file Φ' .

SOLUTION: Let $\tau = \Phi n$ and $\tau' = \Phi' n$ be the old and new employee records. It is clear that τ' is the same as τ except for its H field. In order to modify part of a relation, we will use the conditional union (or overlaying operation) defined by:

$$R;S = R \cup \sim.\text{dom}R \rightarrow S$$

Then, if h' represents the new value of the H field, the new employee record is $\tau' = (H, h'); \tau$, where h' is just the cumulative hours worked, $h' = \Phi n H + Un$. Therefore, by the definition of Φ' :

$$\Phi' n = \tau' = (H, h'); \Phi n$$

To find Φ' we must factor out the employee number n . To do this, note that $\Phi n H = [\odot H](\Phi n) = [\odot H].\Phi n$. That is, $[\odot H].\Phi = \Phi \odot H$ is a slice of the payroll file: the hours worked for each employee. Therefore,

$$\begin{aligned} h' &= \Phi n H + Un = [\odot H].\Phi n + Un \\ &= (\Phi \odot H \mp U)n \end{aligned}$$

Now, define the updating function u by

$$\begin{aligned} u(n) &= [H, (\Phi \odot H \mp U)n] \\ &= [H,].(\Phi \odot H \mp U)n \end{aligned}$$

Then, $\Phi' n = u(n); \Phi n = [\Phi; u]n$. Therefore, the solution to our problem, the new payroll file, is

$$\Phi' = u; \Phi, \text{ where } u = [H,].(\Phi \odot H \mp U)$$

RELATIONAL PROGRAMMING

18.2 check issuing

Suppose we wish to take the payroll file from the previous example and generate checks for the employees. We will assume that a function C is available such that $C(m, p)$ returns a check in the amount p made out to the name m .

SOLUTION: We will ignore overtime computations. Hence, if n is an employee number then $\Phi n N$ is his name and

$$p(n) = \Phi n H \times \Phi n R$$

is his pay. Therefore $p = \Phi \Phi H \times \Phi \Phi R$. Now observe that his check $c(n)$ is $c(n) = C(m, p n) = C(\Phi n N, p n) = C(\Phi \Phi N n, p n) = C.(\Phi \Phi N \neg, p) n$. Combining these we have the file F mapping employee numbers into checks:

$$F = C. [\Phi \Phi N \neg, (\Phi \Phi H \times \Phi \Phi R)]$$

from which we can factor out the old payroll file:

$$F = C. [\Phi N \neg, (\Phi H \times \Phi R)]. \Phi$$

If we just want a set of checks, this is $\text{dom. inv } F$.

18.3 pseudo-natural notation

Relational programs can be made less intimidating by using the pseudo-natural notation described in [MacLennan82]. This notation uses words in place of symbols and uses a comma convention to suppress many parentheses. The frequency table program from §1, $F = \text{size. (all } T)$, can be written:

'Freq-table' means all text then size.

Here, 'Freq-table' = F and 'text' = T .

The payroll example looks like this in the pseudo-natural notation:

'Updates' means:

Old-Master slice Hours, each add Hours-Worked,
then pair-with Hours.

RELATIONAL PROGRAMMING

'New-Master' means Updates each replace Old-Master.

Here, 'Updates' = \mathcal{U} , 'Old-Master' = Φ , 'slice' = Θ , 'Hours' = H , 'each add' = \mathcal{T} , 'Hours-Worked' = U , 'then' = $|$, 'pair-with' = π , 'New-Master' = Φ , and 'each replace' = \mathcal{T} .

The check issuing example is also easily put into this notation:

'Checks' means

Old-Master then:

something slice Name

also something slice Hours each times something slice Rate,

then Write-check.

Here, 'Checks' = F , 'something' represents an omitted argument, 'Name' = N , 'also' = \mathcal{A} , 'each times' = \mathcal{X} , 'Rate' = R , and 'Write-check' = C .

19. Implementation

19.1 introduction

The primary goal of our investigation has been to determine if relational programming is significantly better than conventional methods. It would be premature to devote much effort to implementation studies before it is even determined if relational programming is an effective programming methodology. However, a brief discussion of implementation possibilities is probably not out of line.

The most obvious representation of a relation is the *extensional* representation, in which all the elements of a relation or class are explicitly represented in memory. There are many kinds of extensional representations, such as hash tables, binary trees and simple sorted tables. Of course, performance can be improved through the use of associative memories and active memories (in which each memory cell has a limited processing capability).

Some relations and classes will be so large that it is uneconomical to represent them explicitly in memory. In these cases an *intensional* representation

RELATIONAL PROGRAMMING

[MacLennan73] should be used. Here a class or relation is represented by a formula or expression for computing that relation or class. Operations on the class or relation are implemented as *formal* operations on the expression. This is feasible because of the simple algebraic properties satisfied by relations. It can be seen that an intensional representation is really just a variant of a *lazy evaluation* mechanism [Henderson76, Henderson80]. Sometimes an intensional representation is necessary; for instance, relations of infinite cardinality, such as the numerical operators and relations, require an intensional representation.

19.2 computability

It can be shown on theoretical grounds that some of the operators we have described are not implementable in their full generality. For example, if *unimg* were applicable to all computable functions, it would be possible to solve the halting problem, since

$$\text{Halts}(f, x) \Leftrightarrow \text{unimg} f x \neq \phi$$

Since the halting problem is not solvable, we cannot implement *unimg* and the other operators used in the definition of *Halts* so that they work on all computable functions. Similar arguments set bounds on the implementability of many of the other operators.

These limitations do not prevent the use of the relational operators as a specification language. For this purpose it is only necessary that relational programs precisely specify the relationships between inputs and outputs, not that the programs be implementable. However, if we wish to use the relational operators for executable specifications or for a full-fledged programming language, then the issue of implementability becomes important.

19.3 extensional representation

It should be clear that all the operators are implementable on *extensionally*

represented sets and relations, that is, on sets and relations whose elements are explicitly listed in some form in the computer's memory. Obviously, only finite sets and relations can be represented extensionally. Suha Futaci [Futaci82] has analyzed the complexity of the algorithms associated with several different extensional representations.

19.4 intensional representations

Infinite sets and relations must be represented *intensionally*, that is, without explicitly listing their elements. There are several ways of accomplishing this. For example, infinite sets can be represented by their *characteristic functions*: total, computable, Boolean-valued functions that determine whether or not a given element is in the set. Since we require these functions to be computable they can be expressed in a finite algorithm and so are finitely representable in the computer's memory³.

Another intensional representation of infinite sets makes use of computable *enumeration functions*. If f is an enumeration function for a set then $f(1), f(2), \dots$ are distinct elements of the set. If n is greater than the cardinality of the set, then $f(n)$ might not halt.

One of the most common intensional representations of infinite relations makes use of the corresponding computable function. That is, the computable function f can be used to represent the relation R when $y = f(x) \Leftrightarrow (x, y) \in R$. Clearly, this representation can be used only when R is right univalent. Also, if $x \notin \text{dom} f$ then the computation of $f(x)$ might not halt.

19.5 eliminating polymorphism

When we investigate each of the various extensional and intensional representations of sets and relations, we find that different combinations of the operators are implementable on each representation. This could lead to a very confusing situation for the

3. Of course, *computable* characteristic functions only allow (by definition) the representation of *recursive* sets. There is little to be lost in restricting our attention to recursive sets, however.

RELATIONAL PROGRAMMING

TABLE 1. Sets Represented by their Characteristic Functions

Set Operation	Operation on Characteristic Function
$x \in S$	$S(x)$
$x \notin S$	$\neg S(x)$
$\sim S$	$\neg S$
$S \cap T$	$S \wedge T$
$S \cup T$	$S \vee T$
$S \setminus T$	$S \wedge \neg T$
$S \supset T$	$\neg S \vee T$
$S \times T$	$\wedge.(S T)$
xRy	$R(x,y)$
$\text{unimg} Rx$	$R.(x)$
$\text{all} Rx$	$R.(x)$
$\text{inv} R$	$R.\text{inv}$
$S \rightarrow R$	$S.\alpha \wedge R$
$R \leftarrow S$	$R \wedge S.\omega$
$R \uparrow S$	$R \wedge \wedge.(S S)$
$R S$	$R.\alpha \S \wedge S.\omega \S$
R, S	$R.(Id \alpha) \wedge S.(Id \omega)$

relational programmer. Without consulting a table of some kind the programmer would never be sure whether or not a particular combination was implementable. Therefore, relational programming will be simplified if we can divide the operators into disjoint classes in such a way that each operator is applicable and implementable on exactly one representation. Fortunately, when we investigate the use of the relational operators we find that certain operators are mostly used on finite sets and relations and others are mostly used on computable functions. Thus we have a basis for a division of the operators.

To accomplish this goal it is necessary to eliminate any *polymorphism*, that is, any operators that are both implementable and useful on more than one representation. For example, the set operations (\cap , \cup , \setminus , etc.) are useful and implementable on both finite sets and infinite sets represented by characteristic functions. However, the set operations on infinite sets are easily expressed as abstractions and compositions of the Boolean operations applied to the corresponding characteristic functions; see Table 1. The simplicity and directness of this representation of infinite sets and their operators permits us to eliminate them as basic objects in relational programming. Thus, the set operations (\cap , \cup , \setminus , etc.) will only be allowed on finite sets and relations.

RELATIONAL PROGRAMMING

Since we have eliminated characteristic functions as built-in representations of infinite sets and relations, we are left with only two others: enumeration functions and computable functions (for right univalent relations). We have chosen to eliminate enumeration functions because they have few uses and these can be easily expressed using the functional operations.

This leaves us with two classes of objects in relational programming:

- Finite sets (and hence relations)
- Computable functions

There are only a few operations that are both useful and implementable on both of these classes. For example, the application operation can be used both for applying a computable function to its argument and for looking up an item in a table (a finite relation). Therefore we define two versions of this operation: $f @ x$, which applies the computable function f to x , and $t \downarrow x$ (suggesting subscripting), which applies the finite relation (table) t to x . We allow $f @ x$ to be abbreviated fx and $t \downarrow x$ to be abbreviated t_x .

For some of the polymorphic operations either the intensional version or the extensional version can be easily expressed in terms of other operations. In these cases the easily expressible version can be dropped with little loss of convenience. An example of this is $\text{img.inv}fp$, where f is a total function and p is a characteristic function. This can be written $p.f$ since $p.f$ is the characteristic function of $\text{img.inv}fp$.

19.6 extensional operators

The results of the separation process are displayed in Tables 2-5. Table 2 lists the primitive operations on finite, extensionally represented sets and relations. These operations are considered primitive because they are not simply defined in terms of other operations. Tables 3 and 4 show the non-primitive operations on extensionally represented sets and relations, that is, those that can be simply defined in terms of

RELATIONAL PROGRAMMING

TABLE 2. Primitive Extensional Operations

Operator	Meaning
$t \downarrow x$	application
$t \mid u$	relative product
$t \bar{u}$	construction
$x : y$	pair formation
$s \cup t$	union
$\text{un } x$	unit-set formation
$\text{cur } t$	Currying
$\text{unc } t$	un-Currying
$\forall x$	unique element selection
$\text{size } x$	cardinality
$\text{str } t$	structure of relation
t^+	transitive closure

TABLE 3. Non-primitive Extensional Operations (Part 1)

Operator	Definition
(x, y)	$\text{un}(x : y)$
$(x,)$	$\text{un.}(x :)$
$(, y)$	$\text{un.}(:y)$
Δx	(x, x)
$x \in t$	$\text{or. img}[x =]t$
$s \subseteq t$	$\text{and.}(\text{img}[\in t])s$
$s = t$	$s \subseteq t \wedge t \subseteq s$
$\text{inv } t$	$\text{img}[:.(\text{tl}, \text{hd})!]t$
$\text{dom } t$	$\text{img } \text{hd } t$
$\text{rng } t$	$\text{dom. inv } t$
$\text{mem } t$	$\text{dom } t \cup \text{rng } t$
$\text{lm}(x, t)$	$x \in \text{dom } t$
$\text{rm}(x, t)$	$x \in \text{rng } t$
$\text{mm}(x, t)$	$x \in \text{mem } t$
$\text{run } t$	$\text{and. img}(1 = .\text{size.}[\text{unimg } t]) . \text{dom } t$
$\text{lun } t$	$\text{run. inv } t$
$\text{bun } t$	$\text{run } t \wedge \text{lun } t$
$\text{init } t$	$\text{dom } t \setminus \text{rng } t$
$\text{term } t$	$\text{rng } t \setminus \text{dom } t$
t^*	$t^+ \cup (\text{img} :. \Delta) . \text{mem } t$
$p \rightarrow t$	$\text{filter}(p, \text{hd})t$
$t \leftarrow p$	$\text{filter}(p, \text{tl})t$
$t \mapsto p$	$p \rightarrow t \leftarrow p$

other operations. Although these operations are non-primitive, we would expect that they would be built-in in a relational programming system. These definitions make use of several new primitive operations, which are defined in Table 5. They also make use of the operations on elementary pairs: $\text{Hd} = \alpha.\text{un}$ and $\text{Tl} = \omega.\text{un}$.

RELATIONAL PROGRAMMING

TABLE 4. Non-primitive Extensional Operations (Part 2)

Operator	Definition
α	$\emptyset.\text{init}$
ω	$\emptyset.\text{term}$
$A t$	$t \rightarrow \notin \text{term} t$
Ωt	$\notin \text{init} t \rightarrow t$
$t;u$	$t \cup \notin \text{dom} t \rightarrow u$
$x \text{ cl } t$	$(x, \alpha t) \cup t$
$t \text{ cr } x$	$t \cup (\omega t, x)$
$\min s$	$\alpha.(\text{img} <) s \times s$
$\max s$	$\omega.(\text{img} <) s \times s$
$s \cap t$	$\text{dom}[s \rightarrow t \times \text{un} 0]$
$s \setminus t$	$\text{dom}[\notin t \rightarrow s \times \text{un} 0]$
$\{0..m\}$	$f^m(0, \text{un} 0)$ where $f(n, s) = (n+1, s \cup \text{un} n)$
$\{m..n\}$	$\text{img}[m+]\{0..n-m\}$
$[m..n]$	$\text{img}[:\Delta]\{m..n\}$
$t \oplus x$	$\text{img}(\text{Hd} \vdash \oplus x, \text{tl})t$
$t!x$	$\oplus x \mathcal{R}$
μt	$t \setminus t \mid t^+$
$\text{index} \varphi t$	$\text{img}[\downarrow \varphi \vdash \text{Id}]t$
$\text{select} \varphi$	$\text{img}(\text{index} \varphi)$
$\text{join} \varphi$	$\text{img} \cup \text{dom} \cdot \text{inv} \cdot \parallel \cdot [\text{index} \varphi \parallel \text{index} \varphi]$
$\text{as } t$	$\text{img}[A \downarrow \vdash A \downarrow + 1 +](\text{dom} t \setminus \text{un} \cdot \text{max} \cdot \text{dom} t)$
$\text{sa } t$	$f \S(1, \phi)$ where $f[x, (i, a)] = (i+1, a \cup [i, x])$
$\text{sa} 0 t$	$f \S(0, \phi)$ (f defined above)
$\text{rp} f t$	$\text{img}[\text{Hd} \vdash f, \text{tl}]t$
$\text{rpi} f t$	$\text{img}[f, \text{Hd} \vdash \text{tl}]t$
$t \text{ cat } u$	$t \cup \text{rpi}[+\text{size} \cdot \text{dom } t]u$
$\text{rsort } s$	$\text{img} \leq s \times s$
$\text{sort } s$	$\text{size} \cdot \text{inv} \cdot \text{all} \cdot \text{rsort } s$
$\text{unimg} t x$	$\text{rng}[\text{un} x \rightarrow t]$
$\text{unimg} t$	$\text{img} f(\text{dom} t)$ where $f x = x : (\text{unimg} t x)$
ssm	$\text{unc} \cdot \text{sa} \cdot \text{sa} \mathcal{S}$

19.7 intensional operators

All the intensional operators can be expressed using recursive definitions and lambda expressions. Nevertheless, it is useful to divide these operators into two classes, primitive and non-primitive, on the basis of whether they can be easily defined in terms of the other operators. The intensional operators are shown in Tables 6 and 7.

20. Conclusions

Of course, we are not the first to propose introducing aspects of a relational calculus into programming. Codd [Codd70] has used a relational calculus as the basis for data base systems. Although he defines several operations on relations (*viz.*, permutation,

RELATIONAL PROGRAMMING

TABLE 5. New Primitive Extensional Operations

$\text{and}\{\text{true}\} = \text{true}$
$\text{and}\{\text{false}\} = \text{false}$
$\text{and}\{\text{true}, \text{false}\} = \text{false}$
$\text{or}\{\text{true}\} = \text{true}$
$\text{or}\{\text{false}\} = \text{false}$
$\text{or}\{\text{true}, \text{false}\} = \text{true}$
$\text{union}\{S_1, S_2, \dots, S_n\} = S_1 \cup S_2 \cup \dots \cup S_n$
$\text{filter } pS = \{x \mid x \in S \wedge p(x)\} \text{ (a finite set)}$

TABLE 6. Primitive Intensional Operations

Operator	Definition
$f \odot x$	fx
$\text{img } fs$	$\{fx \mid x \in s\}$
$(f.g)x$	$f(gx)$
$x\pi$	$\pi.(x,)$
πx	$\pi.(,x)$
$(f \parallel g)x$	(fx, gx)
$f \S t$	$\text{img}[f \parallel f]t$
$(f \bar{\odot} g)x$	$(fx)(gx)$
$(p \rightarrow f; g)$	$\text{if } px \text{ then } fx \text{ else } gx$
$\text{curry } f$	$[f.].\pi$
$\text{uncurry } f$	$f. \alpha \bar{\odot} \omega$
$\Phi p(d, \tau)$	$(d, \mu[\tau \uparrow p.d+])$
$\text{iter}[p \rightarrow f]$	$(p \rightarrow \text{iter}[p \rightarrow f]; \text{Id}).f$

TABLE 7. Non-primitive Intensional Operations

Operator	Definition
$\text{while}[p, f]$	$p \rightarrow \text{iter}[p \rightarrow f]; \text{Id}$
$f \S i$	$\text{while}[\phi \neq \omega, (f. [\text{Id} \parallel \alpha] \parallel \Omega.\omega).\Delta].(i,)$
f^n	$\text{while}[n \neq \alpha, 1 + \ f\].(0,)$
νf	$\odot. [\text{Id} \parallel f]$
φ	$\text{Id} \parallel$
δ	$\parallel \text{Id}$
Πf	$\delta[f.]$
$\text{extend}(t, f)$	$\in \text{dom } t \rightarrow t \downarrow; f$
$\text{restrict}(s, f)$	$\text{img}[\phi. \text{Id} \parallel f. \Delta]s$
$\sim p$	$\neg p$

join, tie, composition, and restriction), this small set of operations is insufficient for general purpose programming. These remarks also apply to Childs' reconstituted definition of relations [Childs69], which is also oriented towards data bases. Feldman and Rovner [Feldman69] augmented Algol with several relational operators for associative access to a data base. Their operations, which are our plural description and image, are quite limited, being based on a traditional von Neumann language.

RELATIONAL PROGRAMMING

One general purpose language that does make extensive use of sets and relations is SETL [Kennedy75]. It provides most of the familiar operations on sets (e.g., union, intersection, difference, powerset, image). SETL differs from relational programming in three significant respects: (1) it can only handle *finite* sets, (2) many operations must still be performed in a word-at-a-time fashion using the *set former*, and (3) it resorts to conventional control structures.

Finally, we must mention logic programming systems, such as PROLOG [Kowalski79, vanEmden76], which use predicate logic to describe computational processes. These systems also differ from relational programming in two significant respects: (1) they have a word-at-a-time programming style due to the use of variables representing individuals in the clauses of the program, and (2) they are implemented using a resolution theorem prover, whereas a more conventional procedural implementation suffices for relational programming. Essentially the same remarks apply to Popplestone's relational programming [Popplestone79], which is like logic programming except that it uses "forward inference" rather than "backward inference."

In summary, no other programming style that we are aware of combines the universal use of relations with a rich set of operations on those relations that can be implemented in a deterministic, procedural way. It is hoped that the preceeding discussion has made plausible some of the advantages claimed for relational programming in the Introduction. Considerable work remains to be done in evaluating the effectiveness of a relational calculus as a programming tool. For instance, the optimum set of combinators and relational operators must be selected. Another non-trivial problem is the selection of a good notation for the relational calculus. More from convenience than conviction we have based our notation on [Whitehead70] and [Carnap58]. Making relational programming an effective tool will require designing a notation that combines readability with the manipulative advantages of a two-dimensional algebraic notation. This is all preliminary to any serious considerations of software or hardware implementation techniques.

RELATIONAL PROGRAMMING

21. References

- [Backus78] 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.
- [Carnap58] Carnap, R. *Introduction to Symbolic Logic and its Applications*, Dover, 1958.
- [Childs69] Childs, D.L. Feasibility of a set-theoretic data structure based on a reconstituted definition of relation. *IFIP 68 Proceedings*, 420-430, North-Holland, 1969.
- [Codd70] Codd, E.F. A relational model for large shared data banks, *CACM* 13, 6 (June 1970), 377-387.
- [Curry58] Curry, H.B., Feys, R. and Craig, W. *Combinatory Logic, I*, North-Holland, Amsterdam, 1958.
- [Feldman69] Feldman, J.A. and Rovner, P.D. An Algol-based associative language, *CACM* 12, 8 (August 1969), 439-449.
- [Futaci82] Futaci, Suha. *Representation Techniques for Relational Languages and the Worst Case Asymptotical Time Complexity Behaviour of the Related Algorithms*, MS Thesis, Computer Science Department, Naval Postgraduate School, June 1982.
- [Kennedy75] Kennedy, K. and Schwartz, J. An introduction to the set theoretical language SETL, *J. Computr. and Math. with Applications* 1 (1975), 97-119.
- [Kowalski79] Kowalski, R. Algorithm = logic + control, *CACM* 22, 7 (July 1979), 424-436.
- [Henderson76] Henderson, P. and Morris, J.H., Jr. A lazy evaluator, *Record 3rd ACM Symp. on Principles of Programming Languages*, 1976, 95-103.
- [Henderson80] Henderson, P. *Functional Programming Application and Implementation*, Prentice-Hall, 1980, 223-231.
- [MacLennan73] MacLennan, B.J. Fen - an axiomatic basis for program semantics, *CACM* 16, 8 (August 1973), 468-474.

RELATIONAL PROGRAMMING

- [MacLennan75] MacLennan, B.J. *Semantic and Syntactic Specification and Extension of Languages*, PhD Dissertation, Purdue University, December 1975.
- [MacLennan81a] MacLennan, B.J. Introduction to Relational Programming, *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, ACM, October 18-22, 1981, 213-220.
- [MacLennan81b] MacLennan, B.J. Programming with a Relational Calculus, Naval Postgraduate School Computer Science Department Technical Report NPS52-81-013, September 1981.
- [MacLennan82] MacLennan, B.J. A Simple, Natural Notation for Applicative Languages, *SIGPLAN Notices* 17, 10 (October 1982), 43-49.
- [MacLennan83] MacLennan, B.J. Overview of Relational Programming, *SIGPLAN Notices* 18, 3 (March 1983), 36-45.
- [Popplestone79] Popplestone, R.J. Relational programming, in Hayes, J.E. et al. (eds.), *Machine Intelligence 9*, Halsted Press, 1979, 3-26.
- [Schwartz75] Schwartz, J. Automatic data structure choice in a language of very high level, *CACM* 18, 12 (December 1975), 722-728.
- [vanEmden76] van Emde, M.H. and Kowalski, R.A. The semantics of predicate logic as a programming language, *JACM* 23, 4 (October 1976), 733-742.
- [Whitehead70] Whitehead, A.N. and Russell, B. *Principia Mathematica to *56*, Cambridge, 1970.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93943	40
Professor Bruce J. MacLennan, Code 52M1 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	12
Dr. Robert Grafton Code 433 Office of Naval Research 800 N. Quincy Arlington, VA 22217	1
Dr. David Mizell Office of Naval Research 1030 East Green Street Pasadena, CA 91106	1
Mr. John Backus IBM Research 5600 Cottle Road San Jose, CA 95193	1
Professor Peter Henderson Department of Computer Science SUNY at Stony Brook Long Island, NY 11794	1
Dr. Olle Olsson Department of Computer Science University of Uppsala Box 2059 S-750 02 Uppsala Sweden	1

Dr. Sueo Iimori Department of Mathematics Faculty of Education Saga University Saga City, 840 Japan	1
Professor Maurice Clint Department of Computer Science The Queen's University of Belfast Belfast BT7 1NN Northern Ireland	1
Professor Christopher Holt Department of Computer Science The Queen's University of Belfast Belfast BT7 1NN Northern Ireland	1
Professor D. A. Gustafson Department of Computer Science Fairchild Hall Kansas State University Manhattan, KS 66506	1
Professor Richard T. Snodgrass Department of Computer Science The University of North Carolina at Chapel Hill New West Hall 035A Chapel Hill, NC 27514	1
Mr. William Bex 726 Cowper St. Palo Alto, CA 94301	1
Professor Satish Thatte Department of Computer and Communication Sciences 221 Angell Hall 435 South State The University of Michigan Ann Arbor, MI 48109	1
Dr. Michael Wise Department of Computer Science University of New South Wales P.O. Box 1 Kensington NSW 2033 Australia	1

Professor Horst Kremers Department of Geodetic Science Stuttgart University Keplerstrasse 11 D-7000 Stuttgart 1 Federal Republic of Germany	1
Professor Harvey Abramson Department of Computer Science The University of British Columbia 2075 Wesbrook Mall Vancouver, B. C. Canada V6T 1W5	1
Dr. M. Sintzoff Philips Research Laboratory 2 av. Van Becelaere 1170 Brussels Belgium	1
Mr. A. Finelli Universite de Paris VI Bibliotheque Informatique Recherche 4 Place Jussieu, Tour 55 75230 Paris Cedex 05 France	1
Mr. W. H. Fisher Engineering Research Center Western Electric P.O. Box 900 Princeton, NJ 08540	1
Dr. Mehdi Jazayeri Synapse Computer Corporation 801 Buckeye Court Milpitas, CA 95035	1
Professor P. Raulefs Universität Kaiserslautern Fachbereich Informatik Postfach 3049 D 6750 Kaiserslautern West Germany	1
Professor John Conery Department of Computer and Information Science University of Oregon Eugene, OR 97403	1

Professor S. Ceri
Laboratorio di Calcolatori
Dipartimento di Elettronica
Politecnico di Milano
20133 - Milano
Italy

1

Mr. Mark Himmelstein
1323 Tulip Way
Livermore, CA 94550

1

Professor Werner Tractnig
ERL 457
Computer Systems Lab.
Stanford University
Stanford, CA 94305

1

Professor Rodney Farrow
Computer Science Department
Columbia University
New York, NY 10027

1

Professor Mark Linton
Computer Science Department
Stanford University
Stanford, CA 94305

1

CDR Mike Roth
Naval Data Automation Command
Code 40
Washington Navy Yard
Washington, D.C. 20374

1

Dr. Ted Glaser
849 Berkeley St.
Santa Monica, CA 90403

1

END

FILMED

11-83

DTIC