AN APPROACH TO FORMAL REASONING ABOUT PROGRAMS

Peter Hitchcock

Department of Computer Science,
University of Warwick,
Coventry, England

A dissertation submitted for the degree of Doctor of Philosophy.
June 1974.

PAGE NUMBERS CLOSE TO THE EDGE OF THE PAGE. SOME ARE CUT OFF

PREFACE

I would like to acknowledge the help and encouragement of my supervisor, David Park. Chapters 2, 3, 4 and 5 are an extended form of work done jointly with him, which was first published in Hitchcock and Park [1972].

I would also like to thank the Science Research Council for an industrial studentship and my employers, IBM UK Laboratories, for support under their Advanced Education Programme.

Finally, special thanks are due to my wife and son, whose patience and co-operation have been of the greatest help.

ABSTRACT

This thesis presents a formal apparatus which is adequate both to express the termination and correctness properties of programs and also the necessary induction rules and axioms of their domains. We explore the applications of this formalism with particular emphasis on providing a basis for formalising the stepwise development of programs.

The formalism provides, in some sense, the minimal extension into a second order theory that is required. It deals with binary relations between tuples and the minimal fixpoints of monotone and continuous functionals on them. The correspondence between common constructs in programming languages and this formalism is shown in an informal manner.

To show correctness of a program it is necessary to find an expression for its termination properties which will depend on the induction rules for the data structures of the program. We show how these rules may be formally expressed and manipulated to derive other induction rules, and give a technique for mechanically deriving from a schema an expression for its domain which may be expressed in terms of given induction rules by the manipulations referred to above.

We give axiomatic definitions, including an induction rule, for some domains which commonly occur in programs, these being finite sets, trees, structures, arrays with fixed bounds, LISP S-expressions, linear lists, and the integers.

In developing a program one may start by defining the basic operations and domains in an axiomatic manner. Development proceeds by finding satisfactory representations for this domain in terms of more specific domains and their operations, until finally one has domains which are representable in a target language. We discuss what is meant by a representation in an attempt to formalise this technique of data refinement, and also mention the less general notion of simulation which requires that a representation is adequate for a particular program to work.

A program may have been developed in a recursive manner and if the

target language does not contain recursion as a basic primitive it will be necessary to simulate it using stacks. We give axioms for such stacks, and give a mechanical procedure for obtaining from any recursive program, a flowchart program augmented by stacks, which simulates it.

CONTENTS

1 INTRODUCTION

1.1 Objectives

The aim of this thesis is to introduce a formalism which is capable of describing the correctness, termination properties, equivalence etc of programs and also is capable of specifying the necessary formal assertions about their data domains, principally the induction axiom. We then use this to derive useful theorems about programs. Part of our motivation has been to provide a formal basis for the techniques of structured programming, Dijkstra [1969], Jones [1972], Hoare [1971a], Wirth [1971a], and the theorems we have derived have been slanted towards this application.

We differ from existing formal approaches in that the formalism provides, in some sense, the minimal extension into a second order theory that is required for our purposes. The relationship to other formal approaches and the limitations of the formalism will be discussed briefly later.

1.2 Structure of the Thesis

The formalism is introduced in Chapter 2 where its position with respect to other formal systems is also discussed.

Chapter 3 shows how this formalism may be used to describe flowchart and equation schema in terms of their constituent blocks, and gives the relational form of commonly occurring constructions in programming languages. We also show the form of common assertions about programs.

To show the correctness of a program it is necessary to find an expression for its termination properties which will depend on the induction rules of the data structures of the program. Chapter 4 shows how induction rules may be expressed and manipulated, and Chapter 5 gives a technique for mechanically deriving from a schema an expression for its domain. This can be related to the induction rule of

the domain of interpretation by the manipulations of Chapter 4.

Chapter 6 considers interpretations in more detail, both the non-constructive interpretation of schema blocks by means of the first order predicate calculus, and the explicit specification of basic operations and data structures by means of axioms. The chapter owes much to the work of Hoare [1972a] in the axiomatic definition of Pascal.

Chapter 7 formalises the process of the refinement of data and introduces a simulation theorem. The idea of simulation is carried further by Chapter 8 which presents procedures which mechanically derive from recursive programs, flowchart programs augmented by stacks which simulate the original program.

1.3 Notation

The following notation and the associated familiar theories will be assumed.

1.3.1 First Order Predicate Calculus

True, False	truth values
$\neg$	negation
&	conjunction
$\vee$	disjunction
$\equiv$	equivalence
$\rightarrow$	implication
$\exists$	existential quantifier
$\forall$	universal quantifier

1.3.2 Set Theory

ϕ	the empty set
$\in$	membership
$\subset$	proper containment
$\subseteq$	containment

$\cup$	union
$\cap$	intersection
$\times$	direct product
$\{x \mid p(x)\}$	the set of all x such that p (x), implicit set definition

1.3.3 Tuples

$\langle\rangle$	the zero tuple denoted by Λ
D^n	the set of tuples from D of length n
D°	the set whose only member is the zero tuple
$\langle d_1, d_2 \ldots d_m \rangle$	an element from D^m
$\cap$	concatenation between tuples $\langle d_1, \ldots d_m \rangle^\cap \langle e_1, \ldots e_m \rangle = \langle d_1, \ldots d_m, e_1, \ldots e_n \rangle$.

1.3.4 Relations between tuples

We include here a summary of the notation introduced in Chapter 2.

$m \overset{U}{\rightarrow} n = \{\langle a, b\rangle \mid a \in D^m \ \& \ b \in D^n\}$	universal relation
$m \overset{\Omega}{\rightarrow} n = \phi$	empty relation
$m \overset{E}{\rightarrow} m = \{\langle a, a\rangle \mid a \in D^m\}$	identity relation
$m \overset{E_i}{\rightarrow} 1 = \{\langle a, a_i\rangle \mid a = \langle a_1, \ldots a_m\rangle \in D^m\}$	selector relation
$m \overset{N}{\rightarrow} o = \{\langle a, \Lambda\rangle \mid a \in D^m\}$	nullifier relation
$R^{-1} = \{\langle b, a\rangle \mid \langle a, b\rangle \in R\}$	inverse
$R; S = \{\langle a, c\rangle \mid \exists b \ \langle a, b\rangle \in R \ \& \ \langle b, c\rangle \in S\}$	composition
$[R, S] = \{\langle a, b^\cap c\rangle \mid \langle a, b\rangle \in R \ \& \ \langle a, c\rangle \in S\}$	concatenation
$[R \mid S] = \{\langle a^\cap c, b^\cap d\rangle \mid \langle a, b\rangle \in R \ \& \ \langle c, d\rangle \in S\}$	direct product

The domain of $R = \{\langle a, \Lambda\rangle \mid \langle a, b\rangle \in R\} = R; N$

The range of $R = \{\langle b, \Lambda\rangle \mid \langle a, b\rangle \in R\} = R^{-1}; N$

1.3.5 Substitutions

$\Phi(\sigma/X)$ is the result of substituting σ for all free occurrences of X in Φ.

$\Phi(\sigma_1/X_1, \sigma_2/X_2 \ldots \sigma_n/X_n)$ is the result of simultaneously substituting σ_1 for X_1, ... σ_n for X_n in Φ.

2 DESCRIPTION OF THE FORMALISM

The formalism is a relational calculus based on binary relations between tuples which when we talk about schema may be identified with the relations which hold between state vectors across program blocks. The syntax of the system is given using an informal BNF grammar, and the context sensitive parts of the syntax follow. The semantics are explained using a set theoretic model of the system, rather than by axioms, assuming an arbitrary, non-empty, interpretation. Some of the operations on, and between relations, have direct analogues in programming languages. These will be pointed out in an informal manner in Chapter 3.

2.1 The Relational Calculus

2.1.1 Interpretations

The interpretation of a term τ is determined by a structure $\mathfrak{D} = \langle D, f \rangle$ where D is known as the domain of interpretation and f is a function from the set of typed relation variables to the set of binary relations between tuples from D, such that $f({}_{m}\underset{\rightarrow}{A}{}_{n}) \subseteq D^m \times D^n$. The interpretation of τ by a structure $\mathfrak{D}$ is denoted by $I(\tau, \mathfrak{D})$. We will also talk of the structure $\mathfrak{D}$ as being an interpretation of τ. We define $\mathfrak{D}[R/X]$ to be the structure $\langle D, \hat{f} \rangle$ where $\hat{f}(Y)$ = if Y = X then R, else f (Y) with the obvious extension for multiple replacements. The type of R must be the same as that of X.

2.1.2 Typed Relation Variables

<typed relation variable>::= ${}_{m}\underset{\rightarrow}{A}{}_{n} \mid {}_{p}\underset{\rightarrow}{B}{}_{q} \ldots$ m,n,p.q $\geq o$

We assume that an infinite set of distinct identifiers exist. $I({}_{m}\underset{\rightarrow}{A}{}_{n}, \mathfrak{D}) = f({}_{m}\underset{\rightarrow}{A}{}_{n})$.

It is some relation between tuples from D^m and D^n, whose elements are denoted by:

$$\langle\langle d_1 \ldots d_m\rangle, \langle c_1 \ldots e_n\rangle\rangle.$$

We identify the special case of relations of type $m \to o$, $m > o$ with predicates or sets. We use ${}_{m}\overset{A}{\to}{}_{o} = \{\langle a, \wedge\rangle | X(a)\}$ in place of the predicate $X(x_1, \ldots x_m)$. If the domain D is not empty there are just two $o \to o$ relations which may be considered as truth values, true is identified with $\langle\wedge, \wedge\rangle$ and false with the empty set of type $o \to o$.

2.1.3 Typed Relation Constants

<typed relation constant>::=
<universal relation>|<empty relation>|
<identity relation>|<selector relation>|
<nullifier relation>

2.1.3.1 Universal Relation

<universal relation>::= ${}_{m}\overset{U}{\to}{}_{n}$

$I({}_{m}\overset{U}{\to}{}_{n}, \mathfrak{D}) = \{\langle a, b\rangle | a \in D^m \;\&\; b \in D^n\}$

2.1.3.2 Empty Relation

<empty relation>::= ${}_{m}\overset{\Omega}{\to}{}_{n}$

$I({}_{m}\overset{\Omega}{\to}{}_{n}, \mathfrak{D}) = \phi$

2.1.3.3 Identity Relation

<identity relation>::= ${}_{m}\overset{E}{\to}{}_{m}$

$I({}_{m}\overset{E}{\to}{}_{m}, \mathfrak{D}) = \{\langle a, a\rangle | \langle a \in D^m\}$

2.1.3.4 Selector Relation

<selector relation>::= ${}_{m}\overset{E_i}{\to}{}_{1}$ $\quad 1 \le i \le m$

$I({}_{m}\overset{E_i}{\to}{}_{1}, \mathfrak{D}) = \{\langle a, a_i\rangle | a = \langle a_1 \ldots a_m\rangle \in D^m\}$

This operation corresponds to the selection of variables from a state vector by identifiers.

2.1.3.5 Nullifiers

<nullifier relation>::= $\underset{m}{\overset{N}{\rightarrow}}_{o}$

$I\left(\underset{m}{\overset{N}{\rightarrow}}_{o}, \mathfrak{D}\right) = \{\langle a, \Lambda\rangle \mid a \in D^{m}\}$

We will use the nullifier relation to stand for the complete domain of interpretation. We will often use the prefix is- as a mnemonic device for the indication of such relations; eg is-integer, is-stack, is-binary-tree.

2.1.4 Terms

<terms>::=

<typed relation variables>|<typed relation constants>|
<negated terms>|<inverse terms>|
<composition terms>|<concatenation terms>|
<product terms>|<union terms>|
<intersection terms>|<μ-terms>

To specify context sensitive restraints we assume, for this section, that A is a term of type $m \rightarrow n$ and B is a term of type $p \rightarrow q$.

2.1.4.1 Negated Terms

<negated term>::= <term>'

$\underset{m}{\overset{A'}{\rightarrow}}_{n}$ is a term of type $m \rightarrow n$.

$I\left(\underset{m}{\overset{A'}{\rightarrow}}_{n}, \mathfrak{D}\right) = \{\langle a, b\rangle \mid \langle a, b\rangle \notin I\left(\underset{m}{\overset{A}{\rightarrow}}_{n}, \mathfrak{D}\right)\}$

2.1.4.2 Inverse Terms

<inverse term>::= <term>$^{-1}$

$_{m} \overset{A^{-1}}{\rightarrow} _{n}$ is a term of type $n \rightarrow m$

$I\ (_{m} \overset{A^{-1}}{\rightarrow} _{n}, \mathfrak{D}) = \{\langle b, a\rangle | \langle a, b\rangle \in I\ (_{m} \overset{A}{\rightarrow} _{n}, \mathfrak{D})\}$

The special case of $_{m} \overset{A^{-1}}{\rightarrow} _{o}$ corresponds to the introduction into a program of a set of constants, and $_{m} \overset{N^{-1}}{\rightarrow} _{o}$ to the introduction of new variables into the state vector, possibly by declarations in inner blocks.

2.1.4.3 Composition Terms

<composition term>::= <term>; <term>

$_{m} \overset{A}{\rightarrow} _{n}; _{p} \overset{B}{\rightarrow} _{q}$ is a term of type $m \rightarrow q$ iff $n = p$.

$I\ (_{m} \overset{A}{\rightarrow} _{n}; _{n} \overset{B}{\rightarrow} _{q}, \mathfrak{D}) = \{\langle a, c\rangle | (\exists b)\ (\langle a, b\rangle \in I\ (A, \mathfrak{D})\ \&\ \langle b, c\rangle \in I\ (B, \mathfrak{D}))\}$

This operation is basic to schemas and programming languages. It may appear as the sequencing of statements or as functional composition, eg f (g (x)) has the relational form G; F if $_{1} \overset{F}{\rightarrow} _{1,1}$ $\overset{G}{\rightarrow} 1$ are relations corresponding to the functions f and g.

2.1.4.4 Concatenation Terms

<concatenation term>::= [<term>, <term>]

$[_{m} \overset{A}{\rightarrow} _{n}, _{p} \overset{B}{\rightarrow} _{q}]$ is a term of type $m \rightarrow n + q$ iff $m = p$.

$I\ ([_{m} \overset{A}{\rightarrow} _{n}, _{m} \overset{B}{\rightarrow} _{q}], \mathfrak{D}) = \{\langle a, b^{\frown} c\rangle | \langle a, b\rangle \in I\ (A, \mathfrak{D})\ \&\ \langle a, c\rangle \in I\ (B, \mathfrak{D})\}$

This operation is complementary to selection and is used both to build up

state vectors and to express conditional statements or case statements.

2.1.4.5 Product Terms

<product term>::= [term | term]

$[_{m \to n} A \mid_{p \to q} B]$ is a term of type $m + p \to n + q$.

$I([_{m \to n} A \mid_{p \to q} B], \mathfrak{D}) = \{\langle a \frown c, b \frown d\rangle \mid \langle a, b\rangle \in I(A, \mathfrak{D}) \ \& \ \langle c, d\rangle \in I(B, \mathfrak{D})\}$

This operation can also be specified using selection and concatenation, eg $[_{1 \to 1} A \mid_{1 \to 1} B] = [E_1 A, E_2 B]$.

2.1.4.6 Union Terms

<union term>::= <term> ∪ <term>

$m \overset{A}{\to} n \cup p \overset{B}{\to} q$ is a term of type $m \to n$ iff $p = m$, $q = n$.

$I(m \overset{A}{\to} n \cup m \overset{B}{\to} n, \mathfrak{D}) = \{\langle a, b\rangle \mid \langle a, b\rangle \in I(A, \mathfrak{D}) \vee \langle a, b\rangle \in I(B, \mathfrak{D})\} = I(A, \mathfrak{D}) \cup I(B, \mathfrak{D})$

We use the union operation to separate alternative paths in a program. For conditional expressions, case statements, the domains of the subterms are disjoint, but we also allow non-deterministic programs where the subterms may overlap.

2.1.4.7 Intersection Terms

<intersection term>::= <term> ∩ <term>

$m \overset{A}{\to} n \cap p \overset{B}{\to} q$ is a term of type $m \to n$ iff $p = m$, $q = n$

$I(m \overset{A}{\to} n \cap m \overset{B}{\to} n, \mathfrak{D}) = \{\langle a, b\rangle \mid \langle a, b\rangle \in I(A, \mathfrak{D}) \ \& \ \langle a, b\rangle \in I(B, \mathfrak{D})\}$

2.1.4.8 μ-terms

<μ-term>::= μ_i <typed relation variable list> (<term list>)

<typed relation variable list>::=
<typed relation variable>|
<typed relation variable>
<typed relation variable list>

<term list>::= <term>|<term>, <term list>

If $A_1 \ldots A_n$ is a term list and $Y_1 \ldots Y_n$ is a typed relation variable list of the same length n, if the type of each Y_j, $1 \leq j \leq n$, is the type of A_j and if $1 \leq i \leq n$, then $\mu_i\ Y_1 \ldots Y_n\ (A_1, \ldots A_n)$ is a term of the type of Y_i.

The semantics of μ-terms are given in section 2.1.5.

2.1.5 Well Formed Terms

<well formed term>::=<term>.

A term A is well formed, if for all μ-terms of the form $\mu_i\ Y_1 \ldots Y_n\ (A_1, \ldots A_n)$ occurring as subterms of A, each A_k, $1 \leq k \leq n$, is syntactically monotone in each Y_j, $1 \leq j \leq n$.

An occurrence of a variable X in a term τ is free if it is not part of a subterm of the form μ_i ... X ... (...).

An occurrence of a variable X in a term τ is bound if it occurs in a subterm of the form μ_i ...X ... (...).

A term A is <u>syntactically monotone</u> in X if each free occurrence of X in A occurs within an even number of subterms of the form B'.

A term A is <u>syntactically continuous</u> in X if

i No free occurrence of X in A lies within a subterm of the form B'.

ii No free occurrence of X in A lies within a subterm of the form $\mu_i Y_1 \ldots Y_n (A_1, \ldots A_n)$ with some A_j not syntactically continuous in some Y_k.

Condition (ii) arises since there are terms, say $\tau(X,Y)$, where τ is monotone in Y and continuous in X such that the term $\mu Y (\tau (X, Y))$ is not continuous in X.

Consider $\tau (_1 \underset{\rightarrow}{X},_0 1 \underset{\rightarrow}{Y}_0) = (\cup; (Y \cup A)')' \cup X$

This is syntactically continuous in X and monotone in Y.

$\mu Y (\tau (X, Y)) = X$ if $X \cup A \neq N$

$= N$ if $X \cup A = N$

Let this be $F(X)$.
$F(X)$ is not continuous in X.

Consider sets x_i such that $x_i \not\subseteq A'$ and $\bigcup_i x_i = A'$. These may be found for any interpretation by a structure which has an infinite domain, then $\bigcup_i F(x_i) = A' \neq F(\bigcup_i x_i) = N$.

If the μ-term is well formed then $I(\mu_i Y_1 \ldots Y_n (A_1, \ldots A_n), \mathfrak{D})$ is the i'th component of the <u>minimal fixpoint</u> of the functional $I((A_1, \ldots A_n), \mathfrak{D})$. This functional $F_{A_1 \ldots A_n}$ is from an n-vector of relations to an n-vector of relations such that their j'th

components are of type Y_j, and is defined by:

$$F_{A_1 \ldots A_n}(\langle R_1, R_2 \ldots R_n \rangle) = \langle S_1, \ldots S_n \rangle$$

where $S_i \equiv I(A_i, [R_1/Y_1, R_2/Y_2, \ldots R_n/Y_n])$

Vectors of relations form a lattice with the operations $\subseteq$, $\cap$, $\cup$ defined componentwise, and since in a well formed term the functional $F_{A_1 \ldots A_n}$ is monotone the fixpoint of this functional always exists. Tarski[1955]

It is important not to confuse the algebra of vectors with the direct product operator defined earlier, ie $m \overset{R}{\to} n\ (D) \times p \overset{R}{\to} q\ (D)$ is not isomorphic with $m + p \overset{R}{\to} n + q\ (D)$ where $m \overset{R}{\to} n\ (D)$ is the set of $m \to n$ relations over D, eg with $m = n = p = q = 1$ and $D = \{ a \}$,

$1 \overset{R}{\to} 1\ (D) = \{\phi, \{\langle a, a \rangle\}\}$, $1 \overset{R}{\to} 1\ (D) \times 1 \overset{R}{\to} 1\ (D) =$
$\langle \phi, \phi \rangle, \langle \phi, \{\langle a, a \rangle\} \rangle, \langle \{\langle a, a \rangle\}, \phi \rangle \langle \{\langle a, a \rangle\}, \{\langle a, a \rangle\} \rangle$

whereas $2 \overset{R}{\to} 2\ (D) = \{\phi, \{\langle\langle a, a \rangle, \langle a, a \rangle\rangle\}\}$ which has fewer elements.

All the functionals corresponding to schemas are continuous and we will show in the next chapter how the fixpoint operator can be used to characterise the programming constructs of iteration and recursion.

2.1.6 Atomic Formula

<atomic formula>::= <well formed term> $\subseteq$ <well formed term>

An atomic formula is satisfied by a structure if the inclusion holds between the interpreted terms, ie $\mathfrak{D} \models \sigma \subseteq \tau \iff I(\sigma, \mathfrak{D}) \subseteq I(\tau, \mathfrak{D})$.

2.1.7 Assertions

<assertion>::= <atomic formula set> $\vdash$ <atomic formula set>

<atomic formula set>::= ϕ | <atomic formula>|<atomic

formula>, <atomic formula set>

An assertion $\phi \vdash \psi$ is valid iff every structure which satisfies all of ϕ also satisfies all of ψ.

2.2 Logical Properties

The logical properties of the relational calculus can be divided into first order and second order properties. Given an interpretation one can consider the corresponding interpretation for the pure predicate calculus in which (m → n)-ary typed relations are replaced by (m + n)- ary relations (relations in the normal set theoretic sense).

2.2.1 First Order Properties

Theorems are stated without proofs which are sketched in Hitchcock and Park (1972).

2.2.1.1 Theorem

There is an effective method which, given an atomic formula $\sigma \subseteq \tau$ of the relational calculus, not involving μ-terms, provides a sentence $F_{\sigma, \tau}$ in the corresponding pure first order predicate calculus with equality which is satisfied precisely by those interpretations which correspond to those satisfying $\sigma \subseteq \tau$.

2.2.1.2 Theorem

There is an effective method which, given a sentence F of the pure first order predicate calculus with identity with at most m variables, provides an atomic formula of the form $_{m \overset{\upsilon}{\rightarrow} o} \subseteq \tau_F$, containing only relation variables of the type n → o, which is satisfied by

precisely those interpretations which correspond to those satisfying F.

2.2.2 Second Order Properties

2.2.2.1 Theorem (Park)

There is an effective method for translating atomic formulae involving μ-terms into the second order predicate calculus which preserves satisfaction in the sense of the previous two theorems. The proof may be found in Park [1970].

2.2.2.2 Theorem (Park)

There exist sentences in the second order predicate calculus which cannot be translated into the relational calculus, in the sense of 2.2.2.1.

The proof is along the following lines.

The property that a domain is finite can be expressed as a sentence in the second order predicate calculus.

$$\neg\exists X(\forall x \exists y.X(x,y) \wedge (\forall x \forall y \forall z.((X(x,y) \wedge X(y,z) \rightarrow y=z) \wedge (X(x,z) \wedge X(y,z) \rightarrow x=y))) \wedge \forall x.\neg X(x,x))$$

There exists no set of assertions Φ, finite or infinite, such that an interpretation satisfies Φ iff its domain is finite.

This is known to be true for a set of first order assertions. If the assertions contain free relation variables then these can be set to Ω and eliminated,

since we must be able to assert the finiteness of any structure. It can be shown that for any μ-term, say $\mu X\, F(X)$ which contains no free relation variables that $(\exists n) \vdash \mu^X\ F(X) = F^n(\Omega)$. This means that any set of assertions which does not contain free relation variables can be replaced by a first order set of assertions.

2.2.2.3 Theorem (Park)

There exist assertions involving syntactically monotone μ-terms which cannot be expressed by assertions involving only syntactically continuous μ-terms, such that both assertions are satisfied by precisely the same set of structures.

The proof is sketched below.

Syntactically continuous μ-terms are representable in the language $L\omega_1\omega$ since $\mu X F(X) = \bigcup_{n=0}^{\infty} F^n(\Omega)$.

A result from logic, Keisler [1971], states that well foundedness is not

representable in $L\omega_1\omega$

We show in chapter 4 how it is possible to assert well foundedness using syntactically monotone μ-terms.

2.2.2.4 Theorem (Park)

There exist sentences in $L\omega_1\omega$ which cannot be translated into the relational calculus, in the sense of 2.2.2.1.

The property that a domain is finite can be expressed as a sentence in $L\omega_1\omega$, and the proof is then along the lines of 2.2.2.

2.3 Formal Reasoning

2.3.1 First Order Reasoning

i To show the validity of any assertion, not involving μ-terms, we show the validity of the corresponding predicate calculus formula. That is we assume $\phi_1 \vdash \phi_2$ whenever $[\bigwedge_{(\sigma\subseteq\tau)\,\in\phi_1} F_{\sigma,\tau} \rightarrow \bigwedge_{(\sigma\subseteq\tau)\,\in\phi_2} F_{\sigma,\tau}]$ is valid in the pure first order predicate calculus. $F_{\sigma,\tau}$ is given in 2.2.1.1.

ii From $\phi_1 \vdash \phi_2$ and $\psi, \phi_2 \vdash \phi_3$ we can deduce $\psi, \phi_1 \vdash \phi_3$.

iii For any relation variable $_m\overset{X}{\rightarrow}{}_n$, and any $(m \rightarrow n)$ary term τ, we can deduce from $\phi\vdash\psi$ that $\phi\ (\tau/X) \vdash \psi\ (\tau/X)$, where $\phi\ (\tau/X)$, $\psi\ (\tau/X)$ are the result of replacing all free occurrences of X by τ, after a suitable alphabetic change of bound relation variables in ϕ and ψ.

2.3.1.1 Derived Rules

First order reasoning in the remainder of this thesis will be given informally, rather than by following the formal reasoning outlined above. We list some first order results which will be found useful.

i $\vdash A; E = E; A = A$

ii $\vdash (A; B); C = A; (B; C)$

iii $\vdash (A')' = A$

iv $\vdash (A^{-1})^{-1} = A$

v $\vdash (A')^{-1} = (A^{-1})'$

vi $\vdash \Omega = \mathcal{V}'$

vii $\vdash A \subseteq \mathcal{V}$

viii $\vdash A; \Omega = \Omega$

ix $\vdash [E_1, [E_2, [\ldots E_n]]] = E$

x $\vdash [A, [B, C]] = [[A, B], C]$

xi $\vdash [A; B|C; D] = [A|C]; [B|D]$

xii $\vdash (A; B)^{-1} = B^{-1}; A^{-1}$

xiii $\vdash (A \cup B)' = A' \cap B'$

xiv $\vdash A; (B \cup C) = A; B \cup A; C$

xv $\vdash A; (B \cap C) \subseteq A; B \cap A; C$
$A^{-1}; A \subseteq E \vdash A;(B \cap C) = A;B \cap A; C$

xvi $\vdash (A;\ {}_{m}\overset{X'}{\rightarrow}_{o})' \subseteq (A;\ N)' \cup A;X$

$A^{-1};A \subseteq E \vdash (A;X')' = (A;N)' \cup A;X$

The atomic formula $A^{-1};\ A \subseteq E$ asserts that the relation A is single-valued.

2.3.1.2 Conventions

i Elision of parentheses
$A;\ (B;\ C) \equiv A;\ B;\ C$
$[A,\ [B,\ C]] \equiv [A,\ B,\ C]$

ii Composition semicolon will be omitted and concatenation used.
$A;\ B \equiv AB$

iii Type indications will be dropped whenever possible. The rules governing well formed terms will usually enable them to be restored.

iv Strictly the relation constants E, Ω etc should be distinguished by types. This will not be done. Two occurrences of E in a term may be of different types.

2.3.2 Second Order Reasoning

The rules are presented for well formed μ-terms of order n.

i Fixpoint Property:
$\vdash \mu_i X_1 \cdots X_n\ (\tau_1, \ldots \tau_n) = \tau_i\ (\mu_i X_1 \cdots X_n\ (\tau_1, \ldots \tau_n), \ \ldots\ \mu_n X_1 \ldots X_n\ (\tau_1, \ldots \tau_n))$
$1 \leq i \leq n.$

ii Minimality Property.

a if $\phi \vdash \psi(\Omega_1, \ldots \Omega_n)$

b and $\phi, \psi(X_1, \ldots X_n) \vdash \psi(\tau_1(X_1 \ldots X_n), \ldots \tau_n(X_1 \ldots X_n))$

c then $\phi \vdash \psi(\mu_1 X_1 \ldots X_n (\tau_1, \ldots \tau_n), \ldots \mu_n X_1 \ldots X_n (\tau_1, \ldots \tau_n))$

provided that each atomic formula in ψ has the form $\sigma_1 \subseteq \sigma_2$, with σ_1 syntactically continuous in $X_1, \ldots X_n$, and σ_2 syntactically monotone in $X_1 \ldots X_n$, and $X_1 \ldots X_n$ are not free in ϕ.

The validity of this extended form of Scott Induction, Scott and de Bakker [1969], is shown in Hitchcock and Park [1972], together with a counter example when σ_1 is allowed to be syntactically monotone.

2.3.2.1 Derived Rules

i Substitutivity.

If τ_2 is the term obtained from τ_1 by substituting a relational variable Y for an occurrence of a variable X in a context where neither is bound then:

$X \subseteq Y \vdash \tau_1 \subseteq \tau_2$ or $\tau_2 \subseteq \tau_1$ depending on whether the occurrence of X is within an even or odd number of complemented subterms.

$X = Y \vdash \tau_2 = \tau_1$

ii Elimination of Multiple Fixpoints.

$\vdash \mu_i X_1 \ldots X_{i-1} \; XX_i \ldots X_n (\tau_1, \ldots \tau_{i-1}, \tau, \tau_i, \ldots \tau_n)$

$$= \mu X\ (\tau\ (\mu_1 X_1 \ldots X_n\ (\tau_1, \ldots \tau_n)/X_1, \ldots \mu_n X_1 \ldots X_n\ (\tau_1, \ldots \tau_n)/X_n))$$

iii Fixpoint Induction.

$$\{\tau_i\ (\sigma_1/X_1, \ldots \sigma_n/X_n) \leq \sigma_i \mid 1 \leq i \leq n\} \vdash \mu_i X_1 \ldots X_n\ (\tau_1, \ldots \tau_n) \leq \sigma_i$$

Derivations of the above rules may be found in Hitchcock and Park [1972].

2.3.2.2 Conventions

In situations where no confusion can arise we will often use $\mu_i X_1 \ldots X_n$ to abbreviate the term $\mu_i X_1 \ldots X_n\ (\tau_1, \ldots \tau_n)$.

2.4 Other formal techniques

Manna and Pnueli [1970] adopt an essentially first order approach. They obtain from a program two first order sentences which contain unspecified predicates. If Floyd assertions are "guessed at" and used to replace the unspecified predicates in the first formula, then a first order sentence is obtained whose satisfiability implies the partial correctness of the program. The termination properties of a program are given by the unsatisfiability of the second first order sentence. To show this unsatisfiability it is usually necessary to assume a second order induction axiom for the domain of interpretation. This approach is first order in the sense that once predicates have been "guessed at" first order formula are obtained. There is however an implicit quantification of the unspecified predicate symbols. The fact that the termination properties of a program are not partially decidable shows that the problem cannot be reduced to the proving of a first order theorem. The existence of a second order induction rule is required.

Cooper [1969] uses the second order predicate calculus and makes explicit the implied quantification of unspecified predicate symbols, but says nothing about the necessary induction rules for proofs of termination.

The approach we have taken follows closely that of Park [1970], but is expressed in a relational form suggested by Scott and de Bakker [1969]. Park [1970] shows that some of the predicates corresponding to Floyd assertions must have additional fixpoint properties and that fixpoints can be used to express induction rules.

We have not gone as far as the more sophisticated languages of Milner's LCF [1972] and Scott's Lambda [1972] which have higher types but which use only continuous μ-forms and so are not capable of expressing and manipulating induction rules and hence of talking about termination properties.

We must also mention the similar formalism of de Bakker [1971], de Bakker and de Roever [1972] which is confined so far to monadic relations, to a more restricted class of operations on them, and to continuous μ-forms.

De Roever [1973] describes a polyadic relational calculus which does not contain monotone μ-terms, but which is otherwise essentially similar to ours. Whereas we have derived our first order reasoning via translation to the predicate calculus, de Roever gives axioms for first order reasoning.

3 RELATIONS AND PROGRAMS

3.1 Relational Forms of Program Constructs

Our development process proceeds by postulating a program which is composed of the familiar constructions below. The program is not completely specified, blocks of code may be defined non-constructively by the relation that holds across them. Ultimately we arrive at a program in a target programming language. The justification of this final transition requires a semantic definition of the target language. We do not wish to consider this problem here, except to say that it will be easier to justify the transition if the semantics are given by axioms rather than by a mechanical interpreter, eg the Vienna Definition Language [Walk et al 1969]. For this reason the following treatment is rather informal.

3.1.1 Assignment

An assignment statement modifies the state vector and we consider it as defining a relation between the state vector before assignment, and the state vector after assignment.

As an example consider the statement a := f (a, b) in a program whose state vector consists of the variables a and b. Assume that $_2 \xrightarrow{F} _1$ is the relation corresponding to the function f. The relation between the input state vector and the first component of the output state vector is clearly F, and between the input and the second component of the state vector, the selector relation E_2, as the variable b is unchanged. The concatenation operator is then used to build up the output state vector, resulting in the term $[F, E_2]$.

We may prefer to be less explicit about an assignment statement, or group of statements and define them by

the relation which holds across them, realising this relation more explicitly at a lower level in the development.

McCarthy [1962] gave axioms for a contents function $c(u,\xi)$ which gives the contents of location u in the state vector ξ and an assignment function $a(u, \alpha, \xi)$ which modifies the value of location u in the state vector ξ, to α. The contents function is modelled by E_i and the assignment function by $[E_1 \ldots E_i; A \ldots E_m]$ assuming that the state vector has m components, that u is the name for the i'th component and that the constant relation $_1 \overset{A}{\Rightarrow} _1$ represents the constant α. The axioms are:

i $\quad c(u, a(v, \alpha, \xi)) = \text{if } u = v \text{ then } \alpha \text{ else } c(u,\xi)$

ii $\quad a(v, c(v, \xi), \xi) = \xi$

iii $\quad a(u, \alpha, a(v, \beta, \xi)) = \text{if } u = v \text{ then } a(u, \alpha, \xi)$
$\qquad\qquad \text{else } a(v, \beta, a(\mu, \alpha, \xi))$

From the definitions of E_i and the concatenation operator it is easily shown that the assertions corresponding to these axioms are valid, ie

i $\quad \vdash [E_1, \ldots E_i; A, \ldots E_m]; E_j = \text{if } i = j \text{ then } E_i; A \text{ else } E_j$

ii $\quad \vdash [E_1, \ldots E_i, \ldots E_m] = {}_m \overset{E}{\Rightarrow} {}_m$

iii $\quad \vdash [E_1, \ldots E_i; B, \ldots E_m]; [E_1, \ldots E_j; A, \ldots E_m]$
$\qquad = \text{if } i = j \text{ then } [E_1, \ldots E_j; A, \ldots E_m] \text{ else}$
$\qquad [E_1, \ldots E_j; A, \ldots E_m]\ [E_1, \ldots E_i; B, \ldots E_m].$

Note however that we can deal only with state vectors of a known length whereas McCarthy's axioms refer to those of arbitrary length.

3.1.2 Branching

A conditional statement, if p then Q else R is represented by the term $[_{m} \overset{P}{\rightarrow}_{o\ m} \overset{Q}{\rightarrow} {}_{m}] \cup [_{m} \overset{P'}{\rightarrow}_{o}\ ,\ {}_{m} \overset{R}{\rightarrow} {}_{m}]$. The $m \rightarrow o$ relation P corresponds to the predicate p and acts as a 'filter' allowing only arguments which satisfy p to be applied to Q. The formalism also allows non-determinate branches, ie the domains of the sub-terms involved may overlap. Case statements are an obvious extension. We later use the equivalent formulation of $[P, E]\ Q \cup [P', E]\ R$.

3.1.3 Composition

The sequential execution of statements is straightforward. If R and S are the relations holding across two statements r and s, then R; S is the result of executing first r and then s.

3.1.4 Procedures

We will deal here only with procedures which are non-recursive. They may not access non-local variables other than those in the parameter list. Recursive and mutually recursive procedures are dealt with later.

The declaration of a procedure invoked as a function reference defines a relation between the formal parameters of the procedure and the result vector, provided that we allow only access to formal parameters and local variables in the body of the procedure. Invocation of the function is the selection of the appropriate actual parameters from the state vector of the calling program, composition with the relation representing the body of the procedure, and then assignment of the result state vector.

A procedure call differs only in that the assignment of results is made in the body of the procedure to formal parameters. The procedure declaration defines a relation between the input parameter list and the output parameters, those which are modified in the body of the procedure. Following Hoare [1971b] these two types of parameters should be distinguished. A procedure declaration could have the form p (x) : (v) proc Q where x is the list of formal parameters which are assigned to, and v is the list of formal parameters which supply values. The form of a procedure call is call p (a) : (e) where e is a list of expressions and a is a list of variable names. The relation which holds between the state vector of the calling program before and after such a statement is obtained as follows. The input expression list is formed and composed with the body of the procedure, and the list of variables, a, enables the correct final state vector to be built up using the concatenation operator. This is essentially Hoare's value and result model. We cannot handle calls by name.

As an example consider a program with variables a, b, c and a procedure declaration p (x, y) : (y, z) Q. The relation which holds across the statement call p (a, b) : (b, c) is given by $[[E_2, E_3]\ Q, E_3]$.

Hoare's restriction that the actual parameter list a contains a disjoint list of variables is essential. The simultaneous assignment of two results to the same location is not defined. However we do not have the restriction that none of the variables in a occur in e. This arises in Hoare's work from trying to identify mathematical variables, which have the same value whenever they occur in a formula, with program variables whose values change. This is only possible if the variables are not assigned to.

Since we regard the procedure body as a relation between the input and output parameter lists, and have a call by value mechanism, we circumvent the restriction that the actual input parameter list may not contain the same variable more than once. Consider the example:

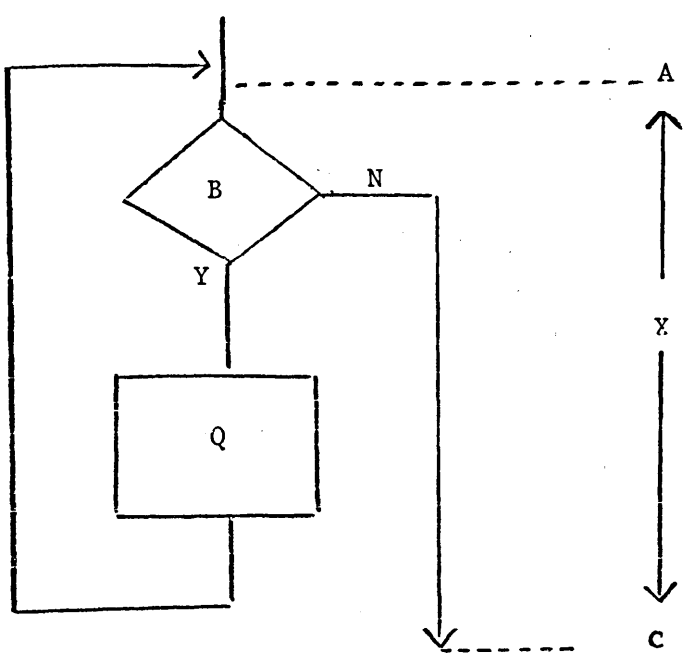
```
p (x) : (v, x) begin x := x + v
                     x := x - v
               end;
```

Clearly the body of the procedure is the $2 \rightarrow 1$ relation E_2. Hence call p (a) : (a, a) does nothing. Note that call p (a) : (a, a) with a body replacement mechanism as for example ALGOL 60, is rather different.

Local variables are introduced into procedure bodies by the use of $0 \rightarrow 1$ relations to extend the state vector in the body of the procedure, or by use of the concatenation operator to extend the state vector, depending on whether the local variable is initialised or not.

3.1.5 Iteration

A simple iterative form is the program construct while B do Q, this may be represented as a flowchart.



The relational expression for this loop is obtained in the following manner. Let X be the relation between the points marked A and C in the flowchart.

We can then trace our way round the loop and obtain the equation:

$$X = [B', E] \cup [B, Q]; X$$

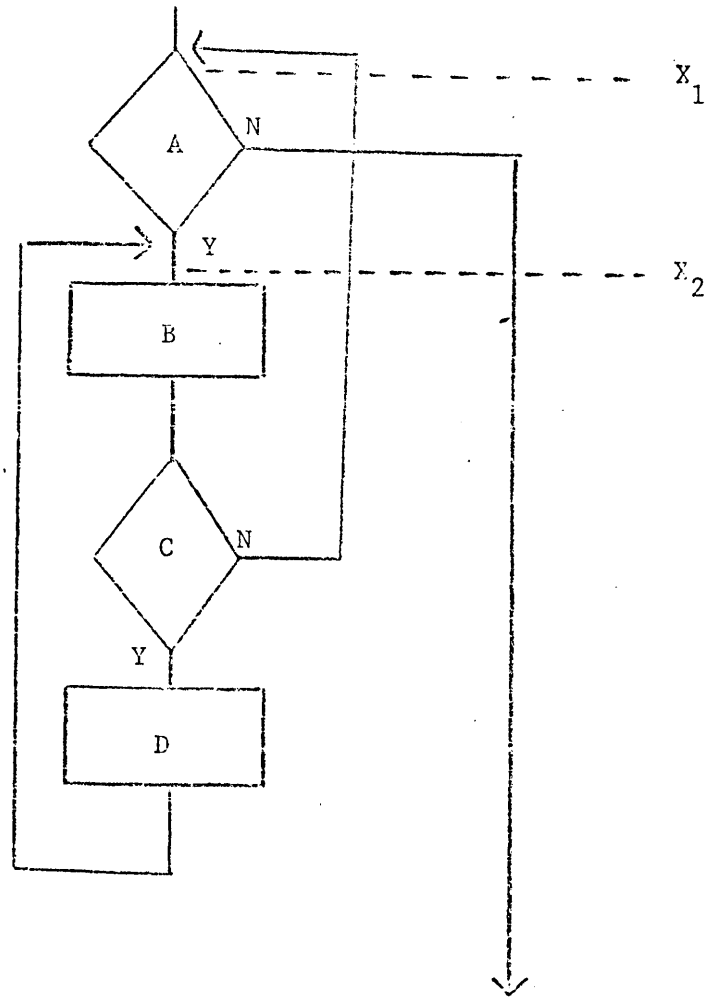
The solution which characterises this loop is given by the minimal fixpoint, ie

$$\mu X([B', E] \cup [B, Q]; X),$$

3.1.6 Flowcharts

The process shown for obtaining the relational form of a loop extends to any flowchart, and hence to languages which include goto statements and labels, but not label variables. Sufficient variables $X_1 \ldots X_n$ are chosen such that there is at least one occurring in each cycle, and a set of n mutually dependent equations is produced.

eg



$X_1 = [A', E] \cup [A, E] X_2$

$X_2 = B; [C', E] X_1 \cup B; [C, D] X_2$

and the relation across this program fragment is given by

$\mu_1 X_1 X_2 ([A', E] \cup [A, E] X_2 , B ([C', E] X_1 \cup [C, D] X_2)$

3.1.7 Recursion, Equation Schema

We treat recursion in a similar manner to iteration. A variable X_i is associated with each recursive procedure or function and equations similar to those above can be obtained.

eg, $f(x_1, x_2)$ = if $p(x_1)$ then $a(x_1, x_2)$
else $g(x_1, x_2, x_2)$

$g(x_1, x_2, x_3)$ = if $p(x_3)$ then $f(h(x_1), x_2)$

else j (g (x_1, x_2, h (x_3))

If X_1 and X_2 are the relations associated with $2 \to -1$ and $3 \to 2$ 1 then we can write the following equations.

$$X_1 = [E_1\ P,\ A] \cup [E_1\ P',\ E_1,\ E_2,\ E_2]\ X_2 = \mathcal{F}(X_1, X_2)$$
$$X_2 = [E_3\ P,\ E_1\ H,\ E_2]\ X_1 \cup [E_1,\ E_2,\ E_3\ H]\ X_2\ J = \mathcal{G}(X_1, X_2)$$

and the relation which characterises f is given by:

$$F = \mu_1 X_1 X_2\ (\mathcal{F}(X_1, X_2),\ \mathcal{G}(X_1,\ X_2))$$

3.1.8 Limitations

We must not pretend that we can describe all the familiar constructs of programming languages in this formalism. We have already shown that we are only able to describe a particular procedure calling mechanism and so cannot describe the body replacement rule of ALGOL 60. The formalism is such that the number of components of the state vector and control structure of a program must be capable of being determined statically. This means that we cannot handle such dynamic changes to the state vector as the creation of variables in SNOBOL IV nor the dynamic changes to control structure caused by label variables or the possibility of passing procedures as parameters. The lambda calculus based languages, and procedure variables need relations of higher types than we allow in our formalism.

It must be remembered however that the formalism was developed to reason about programs and program schemas rather than for the definition of the formal semantics of languages. There is still an element of informality in the transition between relational expressions and their realisation by an actual programming language, which would bear further investigation.

3.2 Properties of Programs

We need to express properties of programs in our formalism.

3.2.1 Correctness

The specification of a program is a relation between input and output variables.

If S is the specification of a program and R is the relation which characterises the program then the program is partially correct with respect to S, if $R \subseteq S$ and is correct with respect to S if $R = S$.

3.2.2 Termination

The domain of a program is the set of values for which it terminates.

If R is the relation which characterises the program then the domain is given by R; N.

The program is total if $RN = N$.

Notice that an argument is included in the domain if at least one computation with that argument terminates, not if all computations terminate.

3.3 Examples

3.3.1 Factorial

This form of a program to compute factorial is taken from Hoare [1971b].

The program is:

```
fact (r) : (a) begin
if a = o then r: = 1
else begin new  w;
```

```
call fact(w) : (a - 1);
r: = a * w   end
end
call fact (r) : (a)
```

The relational form of the body of the declaration is given by:

$$F = {}_{1}\mu X_{1 \to 1} (A \cup [E, BX] C)$$

with the interpretation:

$$A = \{<0, 1>\}$$

$$B = \{<a, a - 1> \mid a > 0\}$$

$$C = \{<<a_1, a_2>, a_1 * a_2> \mid a_1, a_2 \geq o\}$$

The term [E, BX] is of type $1 \to 2$, ie the state vector has been extended corresponding to the declaration new w, the term BX is a recursive call of the procedure with argument a - 1 whose result is placed in the location corresponding to w.

Let S be the relation

$$S = \{<a, a!> \mid a \geq o\}$$

We show by fixpoint induction that the procedure is contained in S.

i $\quad A \subseteq S$ since $0! = 1$

ii $\quad [E, BS] = \{<a, <c_1, c_2>> \mid c_1 = a \;\&\; (\exists b)\; b = a-1$
$\quad\quad \&\; c_2 = b!\}$
$\quad = \{<a, <a, (a - 1)!>> \mid a > 0\}$

iii $\quad [E, BS]\, C = \{<a, a * (a - 1)!> \mid a > 0\}$
$\quad = \{<a, a!> \mid a > 0\} \subseteq S$

iv $\quad A \cup [E, BS]\, C \subseteq S \vdash \mu X (A \cup [E, BX]\, C) \subseteq S.$

The specification of the program is:

$$T = \{<<a_1, a_2>, <b_1, b_2>> \mid a_1! = b_2 \& b_1 = a_1\}$$

The relation corresponding to call fact (r) : (a) is given by:

$$P = [E_1, E_1\ F]$$

Now $P = [E_1, E_1\ F] \subseteq [E_1, E_1\ S] \subseteq T$

Hence the factorial program is partially correct.

3.3.2 Park [1970]

Consider the pair of schemas

$G = \mu_1 X_1 X_2\ (A \cup B X_2,\ C X_1 \cup D X_2 F)$

$H = \mu_2 X_1 X_2\ (A \cup B X_2,\ C X_1 \cup D X_2 F)$

with the interpretation

$A = \{\langle\langle a_1, a_2\rangle, b\rangle \mid a_1 = 0 \ \&\ b = 0\}$

$B = \{\langle\langle a_1, a_2\rangle, \langle b_1, b_2, b_3\rangle\rangle \mid a_1 > 0 \ \&\ b_1 = a_1 \ \&\ b_2 = a_2 \ \&\ b_3 = a_2\}$

$C = \{\langle\langle a_1, a_2, a_3\rangle, \langle b_1, b_2\rangle\rangle \mid a_1 > 0 \ \&\ a_3 = 0 \ \&\ b_1 = a_1 - 1 \ \&\ b_2 = a_2\}$

$D = \{\langle\langle a_1, a_2, a_3\rangle, \langle b_1, b_2, b_3\rangle\rangle \mid a_1 > 0 \ \&\ a_3 > 0 \ \&\ b_1 = a_1 \ \&\ b_2 = a_2 \ \&\ b_3 = a_3 - 1\}$

$F = \{\langle a, b\rangle \mid b = a + 1\}$.

The specification of the program is the relation

$S = \{\langle\langle a_1, a_2\rangle, b_1\rangle \mid a_1 \geq 0 \ \&\ a_2 \geq 0 \ \&\ b_1 = a_1 * a_2\}$

We also need

$$T = \{\langle\langle a_1, a_2, a_3\rangle, b_1\rangle \mid a_1 > 0 \ \&\ b_1 = (a_1 - 1) * a_2 + a_3 \ \&\ a_2 \geq 0 \ \&\ a_3 \geq 0\ \}$$

To show partial correctness we will prove by fixpoint induction that $G \subseteq S$ and $H \subseteq T$.

i $\quad A \subseteq S$

ii $\quad BT = \{\langle\langle a_1, a_2\rangle, b_1\rangle \mid a_1 > 0 \ \&\ b_1 = (a_1 - 1) * a_2 + a_2 \ \&\ a_2 \geq 0\ \}$

$= \{\langle\langle a_1, a_2\rangle, b_1\rangle \mid a_1 > 0 \ \&\ b_1 = a_1 * a_2 \ \&\ a_2 \geq 0\} \subseteq S$

iii $A \cup BT \subseteq S$ from i and ii.

iv $CS = \{\langle\langle a_1, a_2, a_3\rangle, b_1\rangle \mid a_1 > 0 \;\&\; a_2 \geq 0 \;\&\; b_1 = (a_1 - 1) * a_2\} \subseteq T$

v $DT = \{\langle\langle a_1, a_2, a_3\rangle, b_1\rangle \mid a_1 \geq 0 \;\&\; a_2 > 0 \;\&\; a_3 > 0 \;\&\; b_1 = (a_1 - 1) * a_2 + a_3 - 1\}$

vi $DTF = \{\langle\langle a_1, a_2, a_3\rangle, b_1\rangle \mid a_1 > 0 \;\&\; a_3 > 0 \;\&\; b_1 = (a_1 - 1) * a_2 + a_3\} \subseteq T$

vii $CS \cup DTF \subseteq T$

viii $A \cup BT \subseteq S,\ CS \cup DTF \subseteq T \vdash \mu_1 X_1 X_2 \subseteq S,\ \mu_2 X_1 X_2 \subseteq T$

Hence from 'iii', 'vii', 'viii' we obtain the partial correctness of the program.

We could realise this either as a functional program.

s (x_1, x_2) = if $x_1 = 0$ then 0
else t (x_1, x_1, x_2)
t (x_1, x_2, x_3) = if $x_3 = 0$ then s $(x_1 - 1, x_2)$
else t $(x_1, x_2, x_3 - 1) + 1$

or as mutually recursive procedures.

```
s (x) : (a, b) begin
if a = 0 then x: = 0
else call t (x) : (a, a, b)  end
t (x) : (a, b, c) begin
if c = 0 then call s (x) : (a - 1, b)
else begin
call t (x) : (a, b, c - 1);
x: = x + 1
end  end
call s (a) : (a,b)
```

4 INDUCTION RULES AND WELL FOUNDED RELATIONS

In order to establish the correctness of a program it is necessary to obtain an expression for the domain of the program. This chapter shows how induction rules, necessary for termination proofs, can be expressed and manipulated in a schematic form, and the following chapter uses these results to obtain expressions for the domains of programs.

4.1 Well Founded Relations

When describing data domains it is necessary to characterise them by first and second order axioms if we are to prove termination etc of programs operating on these domains. The induction axiom for the domain states that it is well founded with respect to some relation R, ie that there is no infinite sequence d_1, d_2 ... of elements from D such that d_1 R d_2 R .... For example the integers are well founded with respect to the predecessor relation, or LISP S-expressions are well founded with respect to the operations car and cdr.

The set of elements from D, all of which are well founded with respect to R, is called the initial part, or ι (R), of R. This can be characterised using the minimal fixpoint operator.

Defn: $\iota (R) \stackrel{D}{=} \mu X\ (RX')'$.

This definition can be justified by the following argument. Consider first the meaning of the relation $(RX')'$. Using the set model

$RX' = \{a | (\exists b) \langle a, b\rangle \in R \ \&\ b \notin X\}$

so $(RX')' = \{a | (\forall b) \langle a, b\rangle \in R \rightarrow b \in X\}$

ie $(RX')'$ is the set of elements all of whose R-predecessors (if any) are in X. $\iota (R)$ is closed under R, ie $d \in \iota (R)$ & $dRe \Rightarrow e \in \iota (R)$, and so all the R-predecessors of any element of $\iota (R)$ are themselves in $\iota (R)$, it is thus a fixed point of $(RX')'$. Conversely, for any x_0 not contained in $\mu X\ (RX')'$, there must be at least one R-predecessor, x_1,

not contained in $\mu X\,(RX')'$, likewise this too must have an R-predecessor not in $\mu X\,(RX')'$, and so we can produce an infinite sequence $x_0\ R\ x_1\ R\ x_2\ \ldots$. The original element x_0 cannot therefore be in $\iota\,(R)$. Hence $\iota\,(R) \subseteq \mu X\,(RX')'$. Since we have already shown that $\mu X\,(RX')' \subseteq \iota\,(R)$, $\iota\,(R) = \mu X\,(RX')'$.

4.2 Induction Rules

If we now state as an axiom that a domain is well founded with respect to a relation R, we can use an instance of Fixpoint Induction to derive the familiar induction rules.

Let S be some predicate, ie a $1 \to 0$ relation, and assume we are given. as an axiom, that $\iota\,(R) = N$. Then using fixpoint induction, ie that $(RS')' \subseteq S \Rightarrow \iota\,(R) \subseteq S$, we can derive that

$\{x \mid (\forall y)\ (\langle x, y\rangle \in R \to y \in S)\} \subseteq S \Rightarrow N \subseteq S$, or in predicate calculus terms:

$$(\forall x)\ \ ((\forall y)\ (\langle x, y\rangle \in R \to S\ (y)) \to S\ (x)) \to (\forall x)\ S(x)$$

eg, given that

$\iota\ (\underline{pred}) = N$ where $\underline{pred} = \{\langle x+1, x\rangle \mid x \geqslant 0\}$

$\iota\ (>) = N$ where $> = \{\langle x, y\rangle \mid x > y \geqslant 0\}$

we obtain

$S\ (0) \wedge (\forall x)\ (S\ (x) \to S\ (x+1)) \to (\forall x)\ S\ (x)$

$(\forall x)\ (((\forall y)\ (y < x \to S\ (y)) \to S\ (x)) \to (\forall x)\ S\ (x)$

which are the familiar forms of mathematical and course of values induction.

Burstall [1969] gives the structural induction rule. "If for some set of structures, a structure has a certain property whenever all of its proper constituents have that property, then all of the structures in the set have the property". This is saying that the domain of structures considered is well founded under the relation 'proper constituent'; the induction rule is an informal statement for an induction rule of the type derived above.

We have also formalised the familiar recursive definitions of data domains, eg LISP S-expressions are defined as:

> "An S-expression is either an atomic symbol or it is composed of these elements in the following order: a left parenthesis, an S-expression, a dot, an S-expression, and a right parenthesis." Given the operations car and cdr which select the constituents of an S-expression, the domain of S-expressions is given by the axiom ι (car $\cup$ cdr) = N.

The axioms for commonly occurring domains and their basic operations will be discussed in more detail in section 6.2.

4.3 Manipulations of well-founded relations

We list here, with proofs in a later section, some basic manipulations which establish or preserve well-foundedness.

4.3.1 Defn: $_{m} R^{*}_{m} \stackrel{D}{=} \mu X\,(E \cup RX)$

4.3.2 Defn: $_{m} R^{\tau}_{m} \stackrel{D}{=} RR^{*}$ transitive closure of R

4.3.3 Defn: $_{m} R^{o}_{m} \stackrel{D}{=} _{m} E_{m}$

$$R^{n+1}_{m \to m} \stackrel{D}{=} RR^{n} \qquad n \geq o$$

4.3.4 Defn: $_{\iota} (R)_{o} \stackrel{D}{=} \mu X(RX')'$ initial part of R

The standard rules for regular expressions hold for terms defined from variables $_{m} X_{m}$, $_{m} E_{m}$, $_{m} \Omega_{m}$ using ;, $\cup$, *, ie all those formulae deducible from the classical axioms listed in Conway (1971) p 25 by the usual rules for =, $\subseteq$, interpreting E as 1, Ω as 0 etc.

In addition we have:

4.3.5 $R^{*} = \mu X\,(E \cup XR)$

4.3.6 $(R^{*})^{-1} = (R^{-1})^{*} \stackrel{D}{=} R^{-*}$

4.3.7 $RR^{-1} \subseteq E \Rightarrow R^{*} \cup R^{-*} = R^{*}R^{-*}$

4.3.8 $\iota(R) \subseteq R^{*}(RN)'$, $\iota([_{m} A,_{o} R]) \subseteq R^{*}(A \cap RN)'$

4.3.9 $R^{-1}R \subseteq E \Rightarrow \iota(R) = R^{*}(RN)'$, $\iota([A, R]) = R^{*}(A \cap RN)'$

4.3.10 $R \subseteq S \Rightarrow \iota(S) \subseteq \iota(R)$

4.3.11 $\iota(R) = \Omega \Leftrightarrow (RN)' = \Omega$

4.3.12 $\iota(\Omega) = N$

4.3.13 $\iota(R^{n}) = \iota(R), n > o$

4.3.14 $\iota(R^{\tau}) = \iota(R)$

4.3.15 $S \subseteq R^{\tau} \Rightarrow \iota(R) \subseteq \iota(S)$

4.3.16 $R^{n} \subseteq S \subseteq R^{\tau} \Rightarrow \iota(S) = \iota(R), n > o$

4.3.17 $\iota(R) = N \Rightarrow R \cap E = R \cap R^{-1} = \Omega$

4.3.18 $\iota(R \cup S) \subseteq \iota(R) \cap \iota(S) \subseteq \iota(R) \cup \iota(S) \subseteq \iota(R \cap S)$

4.3.19 $\iota(R \cup S) \subseteq \iota(RS) \subseteq \iota(R \cap S)$

4.3.20 $\iota([R|S]) = [\iota(R)|N] \cup [N|\iota(S)]$

4.3.21 $\iota(S) = N \Rightarrow \iota([R|\mho] \cup [E|S]) = [\iota(R)|N]$

4.3.22 $\iota([R|E] \cup [E|S]) = [\iota(R)|\iota(S)]$

4.3.23 $\iota([\iota(R), R]) = N$ for any R

4.3.24 $fN = N, f^{-1}f \subseteq E, Sf \subseteq fR \Rightarrow f\iota(R) \subseteq \iota(S)$

4.3.25 $\iota([_{1} A,_{o} E]) = A'$

4.3.26 $\iota (R \cup [A, E]) = \mu X ((RX')' \cap A')$

4.3.27 $RA \subseteq A \Rightarrow \iota (R \cup [A, E]) = \iota (R) \cap A'$

Note:

4.3.8/4.3.9 The composition operator ';', here elided, was defined using an existential quantifier, this implies that if R is not single valued, then although there is at least one sequence of elements $d_1 \ldots d_n$ where $d_1 \in R^*(RN)'$ and $d_n \in (RN)'$ and $d_1 R d_2 \ldots d_n$, there may be other sequences which start at d_1 and do not terminate. This explains the inclusion of 4.3.8.

4.3.10 Note that the initial part operator is antimonotone.

4.3.14 This is a formalisation of the equivalence of mathematical and course of values induction, and of their analogues on other domains. If R is interpreted as the predecessor relation pred , i.e. $\{\langle x+1, x\rangle | x \geq 0\}$, then R is, by definition the relation > , and given that $\iota(R) = N$, we can derive the familiar induction rules shown in 4.2.

4.3.17 This states formally that if a total domain is well founded by R then there can be no element in the sequence $d_1 R d_2 \ldots d_n$ which is repeated, otherwise a loop would occur, and the total domain would not be well founded.

4.3.21 This is a formal statement of the induction rule corresponding to a lexicographical ordering which is used later to show termination of Ackermann's function If we interpret R and S as the relation >, then the pairs <a,b>and <c,d> are related by $(\lfloor R | \upsilon \rfloor \cup \lceil E | S \rceil)$ iff a>c or a=c and b>d.

4.3.23 This states that if the domain of any relation R is restricted to those elements which are well founded by R, then any element in the domain is well founded by this restriction of R. This is used later to show termination of programs which count up to a limit.

4.3.24 This is a formalism of part of the discussion concerning proofs of termination in Floyd 1967a , and is a special case of a more general simulation result, see 7.2. The normal use of this theorem is for the mapping function f to be total and single valued. It maps program states, related by S, into a domain which is known to be well-founded with respect to R, i.e. $\iota(R)=N$. Hence $fN=N=\iota(S)$ and the domain of the original program is well founded with respect to S.

4.4 Extension to Multiple Domains

We have discussed in 2.1.5 the concept of a multiple fixpoint of a functional acting on the direct product of relation algebras. We discuss here the special case where the functional can be represented by a matrix whose components are relations. The motivation for this special case will be found in section 5.6 where the termination properties of multiply recursive programs are expressed as the initial part of a square matrix of relations. The use of matrices is local to this section and is introduced as a convenient notation. Although this makes the algebra of matrices and vectors of relations look similar to that of relations it is important not to be misled into thinking that they are the same. The essential difference between the two algebras is in their treatment of the null element. In the case of relations a tuple containing a null element is itself considered to be null, whereas in the case of a vector a null element is a perfectly acceptable component. To extend the relational algebra so that there is a direct correspondance between tuples and vectors would mean introducing the concept of an object, whose value is undefined, to be an element of every domain of interpretation and a corresponding redefinition of the basic operations of the relational calculus. This exercise will not be attempted here. An example of the difference between the two algebras will be found in 2.1.5.

4.4.1 Basic constants and operations

An m-vector V is a column vector with m components which are given individually by V_i. It has type $m \times 1$.

We will only be interested here in vectors whose components are relations of type $n_i \to o$.

We consider here only those functionals on vectors which can be represented as $\mathfrak{F}(V) = A \cup BV$ where A and B are matrices. An $m \times n$ matrix A is applied to an n-vector

to produce an m-vector according to the rules of composition given below. $m \times n$ matrices can be built from the following constants, variables and operations.

$({}_{m \times n}\mathcal{V})_{ij} = \mathcal{V}_{ij}$

$({}_{m \times n}\Omega)_{ij} = \Omega_{ij}$

$({}_{m \times n}E)_{ij} = E_{ii}$ if $i = j$

$= \Omega$ if $i \neg = j$

$({}_{m \times o}N)_i = N_i$

$({}_{m \times n}R)_{ij} = R_{ij}$ where R_{ij} is a relation

$({}_{m \times n}A)'$ is a matrix of type $m \times n$ such that $(A')_{ij} = (A_{ij})'$

$({}_{m \times n}A)^{-1}$ is a matrix of type $n \times m$ such that $(A^{-1})_{ij} = (A_{ji})^{-1}$

Note that this is not the conventional matrix inversion. The following operations take place between two $m \times n$ matrices to produce an $m \times n$ matrix.

$({}_{m \times n}A \cup {}_{m \times n}B)_{ij} = A_{ij} \cup B_{ij}$

$({}_{m \times n}A \cap {}_{m \times n}B)_{ij} = A_{ij} \cap B_{ij}$

$([{}_{m \times n}A, {}_{m \times n}B])_{ij} = [A_{ij}, B_{ij}]$

$([{}_{m \times n}A | {}_{m \times n}B])_{ij} = [A_{ij} | B_{ij}]$

Composition takes place between $m \times n$ and $n \times p$ matrices to produce an $m \times p$ matrix.

$({}_{m \times n}A ; {}_{n \times p}B)_{ij} = \bigcup_{k=1}^{n} A_{ik} ; B_{kj}$.

μ-terms of the form $\mu X \mathfrak{F}(X)$ are formed from an m-vector of relation variables and a functional $\mathfrak{F}$ which acts on this m-vector using only the constants and operations given earlier. The result of this functional must be an m-vector. Note that not all functionals can be represented in this matrix form, in particular those corresponding to recursive schemas. This μ-term is an abbreviation for $\mu_i X_1 \ldots X_m(\mathfrak{F}(X)_1, \ldots \mathfrak{F}(X)_m)$ and is well formed if the individual components $\mathfrak{F}(X)_i$ are syntactically monotone in each X_j, $1 \leq i,j \leq m$. The functional $\mathfrak{F}(X)$ is syntactically monotone or continuous if the individual components $\mathfrak{F}_i$ are syntactically monotone or continuous in the components X_j of the vector.

A containment $A \subseteq B$ between matrices is a representation of the set of atomic formulae which are the containments between its components.i.e. $\underset{m\times n}{A} \subseteq \underset{m\times n}{B} \vdash \{a_{ij} \subseteq b_{ij} \mid 1\leq i\leq m,\ 1\leq j\leq n\}$.

4.4.2 Formal reasoning about matrices of relations

First order reasoning: Since the algebras of relations and of matrices and vectors are different first order reasoning about matrices has been justified as required by translating assertions about matrices into a set of assertions about their relational components.

Second order reasoning: This is as before and again matrices are used to represent sets of assertions about relations.

4.4.3 Initial Part

The initial part of an m × m matrix is a vector whose components are $n_i \to o$ relations, such that for any element d_i from the i'th component there is no infinite sequence of elements $d_{i1}, d_{j2}, d_{k3} \ldots$ of elements from the i'th, j'th and k'th ... domains such that $d_{i1}\ R_{ij}\ d_{j2}\ R_{jk}\ d_{k3} \ldots$ We can characterise the initial part in a similar way to that of section 4.1.

If $\underset{m\times m}{R}$ is a matrix and $\underset{m\times 1}{X}$ a vector of $n_i \to o$ relations then the functional (R X')'expands

componentwise so that $(RX')'_i = (\bigcup_{j=1}^{n} R_{ij} X_j)'$
it is the set all of whose predecessors under the $1 \times m$ relation $(R_{i1}\ R_{i2}\ R_{im})$ are in the m vector X. By a similar argument to that in section 4.1 we can justify the following.

Let $\iota_i(R)$ represent the i'th component of the initial part of R, then:

$$\iota_i\,(_{m \times n} R_n) = \mu_i\, X_1 \ldots X_m\, ((RX')'_1, (RX')'_2, \ldots (RX')'_m)$$

This can be represented in the matrix formalism as $\iota(_{m \times m} R) = \mu X(RX')'$
where X is an m-vector.

4.5 Manipulations of well founded matrices

4.5.1 $(_{m \times m} R)^* \stackrel{D}{=} \mu_{m \times m} X\ (_{m \times m} E \cup RX)$

4.5.2 $(_{m \times m} R)^\tau \stackrel{D}{=} RR^*$

4.5.3 $_{m \times m} R^0 \stackrel{D}{=} {}_{m \times m} E$
$R^{n+1}_{m \times m} \stackrel{D}{=} RR^n \quad n \geq 0$

4.5.4 $\iota\,(_{m \times m} R) \stackrel{D}{=} \mu_{m \times o} X\ ((RX')'$

The development so far closely parallels that of section 4.3 and indeed all of the manipulations given are applicable to matrices.

4.5.5 $_{m \times m} R^* = \mu_{m \times m} X\ (E \cup XR)$

4.5.6 $(_{m \times m} R^*)^{-1} = (R^{-1})^* \stackrel{D}{=} R^{-*}$

4.5.7 $RR^{-1} \subseteq E \Rightarrow R^* \cup R^{-*} = R^* R^{-*}$

$RR^{-1} \subseteq E \Leftrightarrow R_{ik} R_{ik}^{-1} \subseteq E_{ii}$,

$(R_{ik})N \cap (R_{jk})N = \Omega$ for $i \neq j$

4.5.8 $\iota({}_{m\times m}R) \subseteq R^*(RN)'$

$\iota([A,R]) \subseteq R^*([A,R]N)'$

where A is an m×m matrix whose elements are $m_i \to 0$ relations, and so $([A,R]N)'_i = (\bigcup_{j=1}^{m} A_{ij} \cup R_{ij}N_j)'$

4.5.9 $R^{-1}R \subseteq E \Rightarrow \iota(R) = R^*(RN)'$

$R^{-1}R \subseteq E \Rightarrow \iota([A,R]) = R^*([A,R]N)'$

$R^{-1}R \subseteq E \Leftrightarrow R_{ki}^{-1}R_{ki} \subseteq E_{ii}$ and $(R_{ki})N \cap (R_{kj}N) = \Omega, i \neq j$

ie R is single valued iff its elements are single valued, and elements in any one row are disjoint.

4.5.10 ${}_{m\times m}R \subseteq {}_{m\times m}S \Rightarrow \iota(S) \subseteq \iota(R)$

$R \subseteq S \Leftrightarrow R_{ij} \subseteq S_{ij}$

4.5.11 $\iota({}_{m\times m}R) = {}_{m\times o}\Omega \Leftrightarrow (RN)' = {}_{m\times o}\Omega$

4.5.12 $\iota({}_{m\times m}\Omega) = {}_{m\times o}N$

4.5.13 $\iota({}_{m\times m}R^n) = \iota(R)\ n > o$

4.5.14 $\iota({}_{m\times m}R^T) = \iota(R)$

4.5.15 ${}_{m\times m}S \subseteq {}_{m\times m}R^T \Rightarrow \iota(R) \subseteq \iota(S)$

4.5.16 ${}_{m\times m}R^n \subseteq {}_{m\times m}S \subseteq {}_{m\times m}R^T \Rightarrow \iota(S) = \iota(R), n > o$

4.5.17 $\iota({}_{m\times m}R) = N \Rightarrow {}_{m\times m}R \cap {}_{m\times m}E = {}_{m\times m}R \cap {}_{m\times m}R^{-1} = \Omega$

4.5.18 $\iota({}_{m\times m}R \cup {}_{m\times m}S) \subseteq \iota(R) \cap \iota(S) \subseteq \iota(R) \cup \iota(S) \subseteq \iota(R \cap S)$

4.5.19 $\iota\,(_{m}R_{m} \cup {}_{m}S_{m}) \subseteq \iota\,(_{m}R_{m} \cap {}_{m}S_{m}) \subseteq \iota\,(R \cap S)$

4.5.20 $\iota\,([_{m}R_{m} | {}_{m}S_{m}]) = [\iota_{m}(R) | {}_{m}N_{o}] \cup [{}_{m}N_{o} | \iota_{m}(S)]$

4.5.21 $\iota\,(_{m}S_{m}) = N \Rightarrow \iota\,([_{m}R_{m} | {}_{m}U_{m}] \cup [{}_{m}E_{m} | {}_{m}S_{m}]) = [\iota_{m}(R) | {}_{m}N_{o}]$

4.5.22 $\iota\,([_{m}R_{m}|E] \cup [E | {}_{m\times m}S]) = [\iota_{m\times o}(R) | \iota_{m\times o}(S)]$

4.5.23 If R is a restriction of R such that $\hat{R}_{ij} = [\iota_i(R), R_{ij}]$ then $\iota(\hat{R}) = N$

4.5.24 If ${}_{m\times m}F$ is a matrix such that $F^{-1}F \subseteq E$, $FN=N$, $SF \subset FR$

ie $\bigcup_{j=1}^{m} S_{ij} F_{jk} \subseteq \bigcup_{j=1}^{m} F_{ij} R_{jk}$

then $F\,\iota(R) \subseteq \iota(S)$

We now leave the straight forward analogues of 4.2 and turn to the problem of obtaining expressions for the initial parts of matrices in terms of their components.

4.5.25 If we let $\iota_{1:p}(R)$ represent the vector of length p such that $\iota_{1:p}(R) = (\iota_1(R) \quad \iota_2(R) \ldots \iota_p(R))$, then if the matrix ${}_{m\times m}R$ is partitioned to be of the form

$$\begin{Bmatrix} {}_{p\times p}A & {}_{p\times m-p}B \\ {}_{m-p\times p}C & {}_{m-p\times m-p}D \end{Bmatrix}$$

then $\iota_{1:p}(R) = \mu_{1}\,{}_{p\times o}X \quad {}_{(m-p)\times o}Y \quad ((AX')' \cap (BY')', (CX')' \cap (DY')')$

and $\iota_{p+1:m}(R) = \mu_{2}\,XY((AX')'\cap(BY')', (CX')'\cap(DY')')$

It is only necessary to study matrices of the above form since any more complex matrix can eventually be expressed in its constituent parts by a succession of partitionings.

Let $({}_{m} R_{m})_{i*}$ represent the $1 \times m$ matrix $(R_{i1}\ R_{i2} \ldots R_{im})$, and let R be partitioned to

$$\begin{Bmatrix} A & B \\ C & D \end{Bmatrix}$$

as above.

We can then obtain the following theorems for particular forms of R.

4.5.26 If R is <u>triangular</u> ie. a special case of the form above where $C = \Omega$.

$$\iota_{1:p}\ (R) = \iota\,(A \cup S)$$

$$\iota_{p+1:m}\ (R) = \iota\ (D)$$

where $({}_{p} S_{p})_{ij} = \Omega$ if $i \neq j$

$= [B_{i*}\ (\iota_{p+1:m}\ (R))'\ ,\ E_{ii}]$ if $i = j$

4.5.27 If R is such that $A = \Omega$ then

$$\iota_{1:p}\ (R) = (B_{\iota p+1:m}\ (R)')'$$

$$\iota_{p+1:m}\ (R) = \iota\ (D \cup C\ B)$$

4.5.28 If R is such that the types of A B C and D are the same then:

$$\iota_{1:m/2}\ (R) \supseteq \iota(A \cup B \cup C \cup D)$$

$$\iota_{m/2+1:m}\ (R) \supseteq \iota\ (A \cup B \cup C \cup D)$$

4.5.29 If in addition $C = {}_{m/2 \times m/2} E$ then:

$$\iota_{1:m/2}\ (R) \supseteq \iota(A \cup B \cup D)$$

$$\iota_{m/2+1:m}\ (R) \supseteq \iota\ (A \cup B \cup D)$$

4.5.30 Finally if R is such that A=B=C=D then:

$$\iota_{1:m/2}\ (R) = \iota(A)$$

$$\iota_{m/2+1\,:\,m}(R) = \iota(A)$$

4.6 Example

Since the simplicity of the above theorems may have been obscured by the notation, we give the following example of a triangular compound relation

Let R be

$$\left\{\begin{array}{c|c} A\ B & C \\ \hline \Omega\ D & F \\ \hline \Omega\ \Omega & G \end{array}\right\}$$

partitioned as shown.

then $\iota_3(R) = \iota(G)$

$$\iota_{1\,:\,2}(R) = \iota\left\{\left\{\begin{array}{cc} A & B \\ \Omega & D \end{array}\right\} \cup \left\{\begin{array}{cc} [C\iota(G)', E] & \Omega \\ \Omega & [F\iota(G)', E] \end{array}\right\}\right\}$$

$$\iota_{1\,:\,2}(R) = \iota\left\{\begin{array}{cc} A \cup [C\iota(G)', E] & B \\ \Omega & D \cup [F\iota(G)', E] \end{array}\right\}$$

Again using 4.5.26

$\iota_2(R) = \iota(D \cup [F\iota(G)', E])$

$\iota_1(R) = \iota((A \cup [C\iota(G)', E]) \cup [B\iota(D \cup [F\iota(G)', E])', E])$

4.7 Proofs

Proofs Section 4.3

4.3.5 To Prove: $R^* = \mu X\,(E \cup XR)$

Proof: by induction on $P(X, Y) \equiv X = Y, RY = XR$ with $\mathfrak{H}(X) = E \cup RX$, $\mathfrak{H}(Y) = E \cup YR$.

4.3.6 To Prove: $(R^*)^{-1} = (R^{-1})^*$

Proof: by induction on $P(X, Y) \equiv X^{-1} = Y$ with $\mathfrak{H}(X) = E \cup RX$, $\mathfrak{H}(Y) = E \cup YR^{-1}$ and then using 4.3.5.

4.3.7 To Prove: $RR^{-1} \subseteq E \Rightarrow R^* \cup R^{-*} = R^*\,R^{-*}$

Proof: $\subseteq$) $R^* \subseteq R^* (E \cup R^{-1} R^{-*}) \subseteq R^* R^{-*}$
$R^{-*} \subseteq (E \cup R R^*) R^{-*} \subseteq R^* R^{-*}$
$\therefore R^* \cup R^{-*} \subseteq R^* R^{-*}$

$\supseteq$) by induction on $XR^{-*} \subseteq R^* \cup R^{-*}$ with $\mathfrak{F}(X) = (E \cup RX)$

4.3.8 To Prove: 1. $\iota(R) \subseteq R^* (RN)'$,
2. $\iota([A, R]) \subseteq R^* (A \cap RN)'$

Proof 1 Induct on $P(X) \equiv X \subseteq R^* (RN)'$ with $\mathfrak{F}(X) = (RX')'$ using $(RX')' \subseteq (RN)' \cup RX$.

Proof 2 Using 1
$\iota([A, R]) \subseteq [A, R]^* (A \cap RN)' \subseteq R^* (A \cap RN)'$ since $([A, R] N)' = (A \cap RN)'$, and $[A, R] \subseteq R$.

4.3.9 To Prove: $R^{-1} R \subseteq E \Rightarrow$ 1. $\iota(R) = R^* (RN)'$,
2. $\iota([A, R]) = R^* (A \cap RN)'$

Proof 1 Induct on $P(X) \equiv X = Y (RN)'$ with $\mathfrak{F}(X) = (RX')'$, $\mathfrak{G}(Y) = E \cup RY$, using $R^{-1} R \subseteq E \Rightarrow (RX')' = (RN)' \cup RX$.

Proof 2 using 1, $\iota([A, R]) = [A, R]^* (A \cap RN)'$ then show that $[A, R]^* (A \cap RN)' = R^* (A \cap RN)'$

$\subseteq$) Clearly $[A, R]^* \subseteq R^*$

$\supseteq$) Induction on $P(X) \equiv X (A \cap RN)' \subseteq [A, R]^* (A \cap RN)'$ with $\mathfrak{F}(X) = E \cup RX$.
$P(\Omega)$ is true, assume $P(X)$,
$(E \cup RX) (A \cap RN)' \subseteq [A, R]^* (A \cap RN)' \cup [A, R] [A, R^*] (A \cap RN)'$

but $[A', R]\ [A, R]^*\ (A \cap RN)' \subseteq [A', R]\ N \subseteq A' \cap RN \subseteq (A \cap RN)'$

Hence P ($\Im(X)$) and P ($\mu X\ \Im(X)$).

4.3.10 To prove: $R \subseteq S \Rightarrow \iota(S) \subseteq \iota(R)$

Proof: Fixpoint Induction using anti-monotonicity of S in $(S\ \iota\ (R)')'$.

4.3.11 To prove:

$$\iota(R) = \Omega \Leftrightarrow (RN)' = \Omega$$

Proof: $\Rightarrow$) $\iota(R) = (R\ \iota\ (R)')' = (RN)' = \Omega$

$\Leftarrow$) using 4.3.8

4.3.12 To prove: $\iota(\Omega) = N$

Proof: 4.3.8

4.3.13 To prove: $\iota(R^n) = \iota(R)$, $n > o$

Proof: $\subseteq$) by Fixpoint induction with $\Im(X) = (R^n X')'$, $(R^n\ \iota\ (R)')' \subseteq (R^{n-1}\ \iota\ (R)')' \ldots \subseteq \iota(R)$.

$\supseteq$) using 4.3.14, 4.3.10 and $R^n \subseteq R^T$.

4.3.14 To Prove $\iota(R^T) = \iota(R)$

Proof: $\subseteq$) using 4.3.10 and $R \subseteq R^T$.

$\supseteq$) by induction on $P(X) \equiv X \subseteq \iota(R^T)$ with

$\Im(X) = (RX')'$

$P(\Omega)$, $P(X) \Rightarrow (RX')' \subseteq (R\ \iota\ (R^T)')'$,

$(RX')' \subseteq (R\ R^T\ \iota\ (R^T)')'$

$\therefore (RX')' \subseteq ((R \cup RR^T)\ \iota\ (R^T)')' \subseteq \iota(R^T)$

Hence $\iota(R) \subseteq \iota(R^T)$.

4.3.15 To prove: $S \subseteq R^{\tau} \Rightarrow \iota(R) \subseteq \iota(S)$

Proof: from 4.3.10, 4.3.14

4.3.16 To prove: $R^{n} \subseteq S \subseteq R^{\tau} \Rightarrow \iota(S) = \iota(R), n > 0$

Proof: from 4.3.10, 4.3.14, 4.3.13.

4.3.17 To prove: $\iota(R) = N \Rightarrow R \cap E = R \cap R^{-1} = \Omega$

Proof:

$N \subseteq \iota(R) \subseteq \iota(R \cap E) \subseteq (R \cap E)^{*}\ ((R \cap E)\ N)'$
using 4.3.8, 4.3.10.
but $(R \cap E)\ ((R \cap E)\ N)' = \Omega$
$\therefore\ ((R \cap E)\ N)' = N \therefore (R \cap E) = \Omega$
similarly $N \subseteq \iota(R) \subseteq \iota(R \cap R^{-1}) \subseteq (R \cap R^{-1})^{*}\ ((R \cap R^{-1})\ N)'$
but $(R \cap R^{-1})\ ((R \cap R^{-1})\ N)' = \Omega$
$\therefore\ (R \cap R^{-1}) = \Omega$

4.3.18 To prove: $\iota(R \cup S) \subseteq \iota(R) \cap \iota(S) \subseteq \iota(R) \cup \iota(S) \subseteq \iota(R \cap S)$

Proof: from 4.3.10 $\iota(R \cup S) \subseteq \iota(R)$, $\iota(R) \subseteq \iota(R \cap S)$, $\iota(R \cup S) \subseteq \iota(S)$, $\iota(S) \subseteq \iota(R \cap S)$,
Hence $\iota(R \cup S) \subseteq \iota(R) \cap \iota(S) \subseteq \iota(R) \cup \iota(S) \subseteq \iota(R \cap S)$.

4.3.19 To prove: $\iota(R \cup S) \subseteq \iota(RS) \subseteq \iota(R \cap S)$

Proof $\iota(R \cup S) = \iota((R \cup S)^{2}) \subseteq \iota(RS)$ 4.3.13, 4.3.10
$\iota(R \cap S) = \iota((R \cap S)^{2}) \supseteq \iota(RS)$ 4.3.13, 4.3.10.

4.3.20 To prove: $\iota([R|S]) = [\iota(R)|N] \cup [N|\iota(S)]$

Proof: Induction on $P(X, Y, Z) \equiv X = [Y|N] \cup [N|Z]$
with $\mathfrak{F}(X) = ([R|S]\ X')'$, $\mathfrak{G}(Y) = (RY')'$,

$\mathcal{H}(Z) = (SZ')'$
using $[A|B]' = [A'|N] \cup [N|B']$.

4.3.21 To prove: $\iota(S) = N \Rightarrow \iota([R|\mathcal{U}] \cup [E|S]) = [\iota(R)|N]$

Proof: $\subseteq$) $\iota([R|\mathcal{U}] \cup [E|S]) \subseteq \iota([R|\mathcal{U}]) = [\iota(R)|N]$

$\supseteq$) Induction with $P(X) \equiv [X|N] \subseteq \iota([R|\mathcal{U}] \cup [E|S])$, $\mathcal{H}(X) = (RX')'$.

$P(\Omega)$ is true, assume $P(X)$, then must show that $[(RX')'|N] \subseteq \iota([R|\mathcal{U}] \cup [E|S])$ this is done by an inner induction on $Q(Y) \equiv [(RX')'|Y] \subseteq \iota([R|\mathcal{U}] \cup [E|S])$, $\mathcal{H}(Y) = (SY')'$.

$Q(\Omega)$ is true, assume $Q(Y)$, then
$P(X) \Rightarrow ([R|\mathcal{U}]\,[X|N]')' \subseteq ([R|\mathcal{U}]\,\iota([R|\mathcal{U}] \cup [E|S])')'$.
$Q(Y) \Rightarrow ([E|S]\,[(RX')'|Y]')' \subseteq$
$([E|S]\,\iota([R|\mathcal{U}] \cup [E|S])')'$
taking the intersection
$[(RX')'|N] \cap ([(RX')'|(SY')'] \cup [N|(SY')']) \subseteq \iota([R|\mathcal{U}] \cup [E|S])$
$[(RX')'|(SY')'] \subseteq \iota([R|\mathcal{U}] \cup [E|S])$

Hence as a conclusion of the inner induction using that $\iota(S) = N$
$[(RX')'|N] \subseteq \iota([R|\mathcal{U}] \cup [E|S])$

Hence as a conclusion of the induction
$[\iota(R)|N] \subseteq \iota([R|\mathcal{U}] \cup [E|S])$.

4.3.22 To prove: $\iota([R|E] \cup [E|S]) = [\iota(R)|\iota(S)]$

Proof: $\subseteq$) $\iota([R|E] \cup [E|S]) \subseteq \iota([R|E]) \subseteq [\iota(R)|N]$
$\iota([R|E] \cup [E|S]) \subseteq \iota([E|S]) \subseteq [N|\iota(S)]$

Hence $\iota([R|E] \cup [E|S]) \subseteq [\iota(R)|\iota(S)]$.

$\supseteq$) by a similar nested induction to the above.

$P(X) = [X|\iota(S)] \subseteq \iota([R|E] \cup [E|S])$

$\varphi(X) = (RX')'$

$Q(Y) = [(RX')'|Y] \subseteq \iota([R|E] \cup [E|S])$

$\varphi(Y) = (SY')'$.

4.3.23 To prove: $\iota([\iota(R), R]) = N$

Proof: a) $\iota(R) \subseteq \iota([\iota(R), R])$ from 4.3.10

b) $\iota(R)' \cup (RN)' \subseteq \iota[\iota(R), R]$

Hence $N \subseteq \iota([\iota(R), R])$

4.3.24 To prove: $fN = N$, $f^{-1} f \subseteq E$, $Sf \subseteq fR \Rightarrow$
$f\iota(R) \subseteq \iota(S)$

Proof: Induction on $P(X) \equiv f X \subseteq \iota(S)$

$\varphi(X) = (RX')'$

$P(\Omega)$, is true, assume $P(X)$ then

$f(RX')' = fN \cap (fRX')' = (fRX')'$ using $f^{-1}f \subseteq E$, $fN=N$

$\subseteq (SfX')' = (S(fN \cap (fX)'))' = (S(fX)')'$

$\subseteq (S\iota(S)')' \subseteq \iota(S)$.

Hence $f\iota(R) \subseteq \iota(S)$.

4.3.25 To prove $\iota([{}_1 A_0, E]) = A'$

$[A, E]$ is single valued

$\therefore \iota([A, E]) = [A, E]^* A' = A'$

4.3.26 $\iota(R \cup [A, E]) = \mu X((RX')' \cap A')$

$\subseteq$) fixpoint induction

$((R \cup [A, E]) \mu X')' = (R \mu X' \cup [A,E] \mu X')'$

$= (R \mu X' \cup [A, E](R \mu X' \cup A))'$

$= (R \mu X' \cup A)' = \mu X$ since $\mu X' = R \mu X' \cup A$,

$[A,E]R \subseteq R$

$\supseteq$) fixpoint induction

$(R \iota (R \cup [A, E])')' \cap A'$

$= (R \iota (R \cup [A, E])' \cup A)'$ but $[A, E] \iota (R \cup [A, E])' \subseteq A$

$\subseteq (R \iota (R \cup [A, E])' \cup [A, E] \iota (R \cup [A, E])')'$

$= \iota (R \cup [A, E])$.

4.3.27 $RA \subseteq A \Rightarrow \iota (R \cup [A, E]) = \iota (R) \cap A'$

From 4.3.26 $\iota (R \cup [A, E]) = \mu X ((RX')' \cap A')$

Induct on $P (X, Y) \equiv X = Y \cap A'$

with $\mathcal{F}(X) = (RX')' \cap A'$, $\mathcal{G} (Y) = (RY')'$

$(RX')' \cap A' = (RY' \cup RA)' \cap A'$ induction hypothesis

$= (RY')' \cap (RA)' \cap A'$ but $(RA)' \supseteq A'$, given

$= (RY')' \cap A'$

Hence

$RA \subseteq A \Rightarrow \mu X ((RX')' \cap A') = \iota (R) \cap A'$.

Proofs Section 4.5

These mirror exactly those of section 4.3, except that we are now performing the induction on the lattices of vectors and matrices, rather than the lattice of simple relations. Since the induction predicates are the same we do not propose to give the proofs in detail, but merely establish some of the manipulations of formulae involving matrices and vectors.

4.5.5 Needs ${}_{m}R_{m} ({}_{m}A_{m} \cup {}_{m}B_{m}) = R A \cup R B$

Proof: $(R (A \cup B))_{ij} = \bigcup_{k=1}^{m} R_{ik} (A_{kj} \cup B_{kj})$

$= \bigcup_{k=1}^{m} R_{ik} A_{kj} \cup \bigcup_{k=1}^{m} R_{ik} B_{kj}$

$= (RA)_{ij} \cup (RB)_{ij}$

4.5.6 Needs $(RX)^{-1} = X^{-1} R^{-1}$

Proof $(RX)^{-1}_{ij} = ((RX)_{ji})^{-1} = \overset{m}{\underset{k=1}{U}}(R_{jk}X_{ki})^{-1}$

$= \overset{m}{\underset{k=1}{U}}(X_{ki})^{-1}(R_{jk})^{-1} = (X^{-1}R^{-1})_{ij}$

4.5.8 Needs $\underset{m\times m}{(RX')'} \subseteq \underset{m\times o}{(RN)'} \cup RX$

Proof $(RX')'_i = \overset{m}{\underset{j=1}{\cap}}(R_{ij}X_j')'$

$\subseteq \overset{m}{\underset{j=1}{\cap}}((R_{ij}N_j)' \cup R_{ij}X_j)$

$\subseteq \overset{m}{\underset{j=1}{\cap}}(R_{ij}N_j)' \cup \overset{m}{\underset{j=1}{U}}(R_{ij}X_j)$

$\subseteq (RN)'_i \cup (RX)_i$

4.5.9 Needs $R^{-1}R\subseteq E \Rightarrow (RX')' = (RN)' \cup RX$

Proof $R^{-1}R\subseteq E \Rightarrow (R^{-1}R)_{ij} \subseteq E_{ii}$ if $i=j$

$\subseteq \Omega$ if $i\neq j$

i.e. $\overset{m}{\underset{k=1}{U}} R^{-1}_{ki} R_{kj} \subseteq E_{ii}$ if $i=j$

$\subseteq \Omega$ if $i\neq j$

so $R^{-1}_{ki} R_{ki} \subseteq E_{ii}$ (1)

and $(R_{ki})N \cap (R_{kj})N = \Omega$ if $i\neq j$ (2)

Now $(RX')'_i = \overset{m}{\underset{j=1}{\cap}}(R_{ij}X_j')'$

$= \overset{m}{\underset{j=1}{\cap}}((R_{ij}N_j)' \cup R_{ij}X_j)$ using (1)

But $R_{ij}N \cap R_{ik}N = \Omega$ using (2)

and so $(R_{ij}N_j)' \cap R_{ik}X_k = R_{ik}X_k$

and $\bigcap_{k=1}^{m} (R_{ij}X_k) = \Omega$

and so $(RX')'_i = \bigcap_{j=1}^{m} ((R_{ij}N_j)') \cup \bigcup_{j=1}^{m} R_{ij}X_j$

$= (RN)'_i \cup (RX)_i$

4.5.17 Needs $\underset{m\times m}{R}$ $(R^{-1}N)' = \Omega$

Proof $(R^{-1}N)'_i = \bigcap_{j=1}^{m} (R^{-1}_{ji} N_j)'$

so $(R(R^{-1}N)')_i = \bigcup_{k=1}^{m} R_{ik} \bigcap_{j=1}^{m} (R^{-1}_{jk} N_j)'$

$\subseteq \bigcup_{k=1}^{U} R_{ik} (R^{-1}_{ik} N_j)'$

$\subseteq \Omega$

We now leave the proofs that were analogous to those of section 4.3, and give in detail proofs of the remaining theorems of section 4.5.

4.5.26 To prove: $\underset{m\times m}{R} = \begin{pmatrix} \underset{p\times p}{A} & \underset{p\times m-p}{B} \\ \Omega & \underset{m-p\times m-p}{D} \end{pmatrix} \Rightarrow {}_{1:p}(R) = {}_1(A \cup S)$ (1)

$${}_{p+1:m}(R) = {}_1(D) \quad (2)$$

where $(\underset{p\times p}{S})_{ij} = \Omega$ if $i \neq j$

$= [B_{i*}({}_{p+1:m}(R))', E_{ii}]$ if $i = j$

Proof:

Note first from 4.5.25

$${}_{1:p}(R) = \mu_1 X_1 X_2 ((AX_1')' \cap (BX_2')', (DX_2')')$$

$${}_{p+1:m}(R) = \mu_2 X_1 X_2 ((AX_1')' \cap (BX_2')', (DX_2')')$$

$\subseteq$) by fixpoint induction

(2) $(D_1(D')') \subseteq {}_1(D)$

(1) $(A({}_1(A \cup S)'))' \cap (B {}_1(D)')'$

$\subseteq (A({}_1(A \cup S)'))' \cap (S {}_1(A \cup S)')' \subseteq {}_1(A \cup S)$

since $(B\,\iota(D)')_i = [B_{i*}\,\iota(D)\,;E]\,N$

$\supseteq [B_{i*}\,\iota(D)\,;E]\,\iota(AuS)'$

$= S\,\iota(AuS)'$

$\supseteq$ Induction on: $P(X_1, X_2) \equiv X_1 \subseteq \iota_{1:p}(R)$

$X_2 \subseteq \iota_{p+1:m}(R)$

With $\Im_1(X_1, X_2) = (AX_1')' \cap ([B_{i*}\,X_2', E]\,X_1')'$

$\Im_2(X_1, X_2) = (DX_2')'$

$P(\Omega, \Omega)$ is true, assume $P(X_1, X_2)$ then

$(DX_2') \subseteq (D\,\iota_{p+1:m}(R)')' \subseteq \iota_{p+1:m}(R)$

and $((Au[B_{i*}(DX_2'), E])X_1')' \subseteq (A\,\iota_{1:p}(R)' \cup [B_{i*}\,\iota_{p+1:m}(R)\,;E]\,\iota_{1:p}(R)')'$

$\subseteq (A\,\iota_{1:p}(R)' \cup (B_{i*}\,\iota_{p+1:m}(R)' \cap \iota_{1:p}(R)'))'$

$\subseteq (A\,\iota_{1:p}(R)')' \cup B_{i*}\,\iota_{p+1:m}(R)')' \subseteq \iota_{1:p}(R)$

Hence the limit

$$(A_{1:p}(R)')' \cap ([B_{i*}\ \iota_{p+1:m}(R)', E])_{1:p}(R)')'$$

$$\subseteq (\iota_{1:p}(R)')'$$

and so by fixpoint induction

$$\iota(A \cup C) \subseteq \iota_{1:p}(R).$$

4.5.27 To prove: if $R_{m \times m} = \begin{Bmatrix} \underset{p\times p}{Q} & \underset{p\times m-p}{B} \\ \underset{m-p\times p}{C} & \underset{m-p\times m-p}{D} \end{Bmatrix}$ then

$$\iota_{1:p}(R) = (B(\iota_{p+1:m}(P))')'$$

$$\iota_{p+1:m}(R) = \iota(D \cup CB)$$

Proof:

$\subseteq$) $R \cup RR = \begin{Bmatrix} BC & BD \cup BD \\ CD \cup DC & DCB \cup DD \end{Bmatrix} \subseteq R^{\tau}$

$\therefore \begin{Bmatrix} Q & B \\ Q & D \cup CB \end{Bmatrix} \subseteq R^{\tau}$

$\therefore$ using 4.5.26 and 4.5.15

$$\iota_{p+1:m}(R) \subseteq \iota(D \cup CB)$$

$$\iota_{1:p}(R) \subseteq \iota([B\iota_{p+1:m}(R)', E]) \subseteq (B\iota_{p+1:m}(R)')'$$

$\supseteq$) By induction on $P(X_1, X_2) \equiv X_1 \subseteq \iota_{1:p}(R)$

$$X_2 \subseteq \iota_{p+1:m}(R)$$

with $\mathfrak{F}_1(X_1, X_2) = (BX_2')'$

$$\mathfrak{F}_2(X_1, X_2) = ((D \cup CB)X_2')'$$

4.5.28 To prove: if A, B, C, D have the same type then

$$\iota(A \cup B \cup C \cup D) \subseteq \iota_{1:m/2}(R) \text{ where } R = \begin{Bmatrix} A & B \\ C & D \end{Bmatrix}$$

$$\iota(A \cup B \cup C \cup D) \subseteq \iota_{m/2+1:m}(R)$$

Proof: $(R^{\tau})_{ij} \subseteq (A \cup B \cup C \cup D)^{\tau}$

result follows from 4.5.30

4.5.29 To prove: if A, B, C, D have the same type, and
D = E, then $\iota(A \cup B \cup C) \subseteq \iota_{1 : m/2}(R)$
$\iota(A \cup B \cup C) \subseteq \iota_{m/2 + 1 : m}(R)$

Proof: $(R^T)_{ij} \subseteq (A \cup B \cup C)^T$, result follows from 4.5.30

4.5.30 To prove if $R_{m \times m} = \begin{pmatrix} A & A \\ A & A \end{pmatrix}$ then
$\iota_{1 : m/2}(R) = \iota(A)$
$\iota_{m/2 + 1 : m}(R) = \iota(A)$

Proof: By induction on $P(X_1, X_2, Y) \equiv X_1 = Y, X_2 = Y$

with $\Im_1(X_1, X_2) = (AX_1')' \cap (AX_2')'$
$\Im_2(X_1, X_2) = (AX_1') \cap (AX_2')$
$\Im_3(Y) = (AY')'$

5 TERMINATION

5.1 Introduction

We will show how to derive from a schema, a relation whose initial part describes the domain ofthe schema. The manipulations of well founded sets derived in the previous section can then be used, together with the axioms for the domain of interpretation, to obtain expressions for the domain of the schema.

By way of informal motivation consider the following simple computation model and the deterministic schema described by the recursion equation $X_{m \to n} = \tau(X)$ with a solution $f = \mu X(\tau(X))$. The result of applying the schema to an argument x is given by $x_1 f$ and clearly this is the same as $x_1 \tau(f)$. Computation proceeds by presenting arguments modified by τ to nested occurrences of f. We will show how to obtain, by syntactic means, a derivative $\overset{\circ}{\tau}$ which describes this modification, ie the relation between state vectors before and after one 'cycle' of recursion. We will also derive a co-derivative $\underset{\circ}{\tau}$, of type $m \to o$, which gives those arguments for which application of $\tau(f)$ is undefined. A particular argument x_1 can then give rise to a non-terminating computation if either:

i the computation leads to an undefined result ie $x_1 \overset{\circ}{\tau} x_2 \ldots x_n$ where $x_n \in \underset{\circ}{\tau}$.

or:

ii there is an infinite sequence such that $x_1 \overset{\circ}{\tau} x_2 \overset{\circ}{\tau} \ldots$

If either of the above conditions is satisfied and if we let $R = [\overset{\circ}{\tau}, E] \cup \underset{\circ}{\tau}$, then there is an infinite sequence $x_1 R x_2 R \ldots$ i.e. the schema terminates on precisely the set which is well founded under R.

As an informal example consider the schema $X = A \triangleright BX$ where A is the relation <1, 0> and B is the relation $\{\langle a, b\rangle \mid a \neq 1 \wedge b = a - 1\}$ ie corresponding to the conditional expression $f = (x = 1) \to 0, f(x - 1)$. f is undefined for $C = (AN)' \cap (BN)'$ ie for x = 0, and the relation between successive calls to f is B. We would expect the domain of f to be given by $\iota(B \cup [C, E])$.

Using 4.3.27 and that $BC=\Omega$, $\iota(B \cup [C,E]) = \iota(B) \cap C'$, but $B \subseteq \underline{pred}$ and it is an axiom of the integers that ι (pred) = N. Hence, using 4.3.10, ι $(B \cup [C, E]) = C$ $= \{a | a > 0\}$.

5.2 Definitions

Simplicity:

A term τ is simple in a relation variable X if either:

i τ contains no free occurrences of X

ii $\tau = X$

iii $\tau = \rho\sigma$

iv $\tau = [\rho, \sigma]$

v $\tau = \rho \cup \sigma$.

where ρ and σ are terms simple in X.

Any term which is syntactically continuous in X and this includes any term corresponding to a schema, is reducible to a term simple in X. More details may be found in Hitchcock and Park [1972].

Derivatives

$\dot{\tau}$ is the derivative and $\underset{\bullet}{\tau}$ the co-derivative of a term τ simple in X if:

i τ contains no free occurrences of X, $\dot{\tau} = \Omega$, $\underset{\bullet}{\tau} = (\tau N)$

ii $\tau = X$, $\dot{\tau} = E$, $\underset{\bullet}{\tau} = \Omega$

iii $\tau = \rho\sigma$, $\dot{\tau} = \dot{\rho} \cup \rho\dot{\sigma}$, $\underset{\bullet}{\tau} = \rho \cup \rho_{\bullet}$

iv $\tau = [\rho, \sigma]$, $\dot{\tau} = \dot{\rho}\dot{\omega}$, $\underset{\bullet}{\tau} = \rho_{\bullet}\sigma_{\bullet}$

v $\tau = \rho \cup \sigma$, $\dot{\tau} = \dot{\rho} \cup \dot{\sigma}$, $\underset{\bullet}{\tau} = \rho_{\bullet} \cap \sigma_{\bullet}$

Goodness:

The definition of the domain of a non-deterministic schema is such that the schema is considered to terminate for a given argument if there is at least one terminating sequence from that argument, not that all possible computation sequences from that argument terminate. This is due to the use of an existential quantifier in the definition of the composition operator. The definition of well foundedness states that all sequences, starting with elements in the well founded set, terminate. It is natural therefore to expect the containment in theorem 5.4.1 which relates the initial part of a relation derived from a schema to the domain of that schema. Good terms are defined to be such that there is only one computation sequence from a given argument, and hence we would expect the equality of theorem 5.4.2. A deterministic computation sequence will arise if,firstly the individual relations in the sequence are single valued, with the exception of the terminating relation, and secondly if,when branching occurs in the schema, either the branch allows no parallel paths, or if it does then the computations along parallel paths follow the same sequence. These two conditions correspond to conditions (i) and (ii) in the definition of goodness below. We will show later, in section 5.5, that conditional expressions form an important subset of good terms.

We define $\overset{\circ}{\tau}$, $\underset{\circ}{\tau}$ as $\overset{\bullet}{\tau}(f/X)$, $\underset{\bullet}{\tau}(f/X)$

A term τ is good relative to X,f and a set of axioms Φ if τ is simple in X and:

i for any subterm of τ of the form $\rho\sigma$ in which X occurs free,
$\Phi \vdash (\rho^{-1}\rho)(f/X) \subseteq E$ ie $\rho\ (f/X)$ is single valued

ii For any subterm of τ of the form $\rho \cup \sigma$,
$\Phi \vdash \overset{\circ}{\sigma} \subseteq \overset{\circ}{\rho} \cup \underset{\circ}{\rho}\ \mho$
and $\Phi \vdash \overset{\circ}{\rho} \subseteq \overset{\circ}{\sigma} \cup \underset{\circ}{\sigma}\ \mho$

5.3 Lemmas

if τ is simple in X then:

5.3.1 $(\tau N)' \subseteq \dot{\tau}(XN)' \cup \iota$

The proof is by induction on the formation rules for τ.

i case: τ free from X, $\tau \cup \dot{\tau}(XN)' = (\tau N)'$

case: $\tau = X, \tau \cup \dot{\tau}(XN)' = (\tau N)'$

ii Assume that the lemma is true for subterms ρ, and σ.

case: $\tau = \rho\sigma$,

$(\tau N)' = (\rho\sigma N)' \subseteq (\rho N)' \cup \rho\,(\sigma N)'$

$\subseteq \rho \cup \dot{\rho}\,(XN) \cup \rho\sigma \cup \rho\dot{\sigma}\,(XN)'$, induction Hyp

$\subseteq (\rho \cup \rho\sigma) \cup (\dot{\rho} \cup \rho\dot{\sigma})\,(XN)'$

$\subseteq \tau \cup \dot{\tau}\,(XN)'$

case: $\tau = [\rho, \sigma]$

$(\tau N)' = ([\rho, \sigma]N)' = (\rho N)' \cup (\sigma N)'$

$\subseteq \sigma \cup \rho \cup (\dot{\rho} \cup \dot{\sigma})\,(XN)'$, Induction Hyp

$= \tau \cup \dot{\tau}\,(XN)'$

case: $\tau = \rho \cup \sigma$

$(\tau N)' = (\rho N)' \cap (\sigma N)'$

$\subseteq (\rho \cup \dot{\rho}\,(XN)') \cap (\sigma \cup \dot{\sigma}\,(XN)')$ Induction Hyp

$= (\rho \cap \sigma) \cup (\dot{\rho}\,(XN)' \cap \dot{\sigma}\,(XN)') \cup (\sigma \cap \dot{\rho}\,(XN)' \cup (\rho \cap \dot{\sigma}\,(XN)')$

$\subseteq (\rho \cap \sigma) \cup (\dot{\rho}\,(XN)' \cup \dot{\sigma}\,(XN)')$

$\subseteq \tau \cup \dot{\tau}$

iii The conclusion is that the lemma is true for all simple terms.

5.3.2 If τ is simple in X then τ and $\dot{\tau}$ are syntactically continuous in X.

The proof follows simply from the formation rules for simple terms since no term with any X_i free is complemented in forming a derivative.

5.3.3

if τ is good relative to X, f and Φ

$$\Phi,\ X \subseteq f \vdash (\tau N)' = \tau_{\circ} \cup \dot{\tau}^{\circ}\,(XN)'$$

Again the proof proceeds by induction on the formation rules for simple terms.

i case: τ is free from X, trivial.

case: $\tau = X$, trivial.

ii Assume that the lemma is true for subterms ρ and σ.

case: $\tau = \rho\sigma$,

$(\tau N)' = (\rho\sigma N)' = (\rho N)' \cup_{\rho} (\sigma N)'$, goodness of τ

Also $(\tau N)' \supseteq (\rho\ (f/X)\ \sigma N)' \supseteq \rho\ (f/X)\ (\sigma N)'$, since $X \subseteq f$ and monotonicity of ρ.

so $(\tau N)' = (\rho N)' \cup \rho\ (\sigma N)' \cup \rho\ (f/X)\ (\sigma N)'$

But $\rho\ (\sigma N)' \subseteq \rho\ (f/X)\ (\sigma N)'$, $X \subseteq f$, monotonicity of ρ

so $(\tau N)' = (\rho N)' \cup \rho\ (f/X)\ (\sigma N)'$

$= (\rho \underset{\circ}{\cup} \rho\ (f/X)\ \sigma) \cup (\overset{\circ}{\rho} \underset{\circ}{\cup} \rho\ (f/X)\ \overset{\circ}{\sigma})\ (XN)'$, Induct Hyp

$= \tau \underset{\circ}{\cup} \overset{c}{\tau}\ (XN)'$

case: $\tau = [\rho, \sigma]$

$(\tau N)' = (\rho N)' \cup (\sigma N)'$

$= (\rho \underset{\circ}{\cup} \sigma) \cup (\overset{\circ}{\rho} \underset{\circ}{\cup} \overset{\circ}{\sigma})\ (XN)'$, Induction Hyp

$= \tau \underset{\circ}{\cup} \overset{\circ}{\tau}\ (XN)'$

case: $\tau = \rho \cup \sigma$

$(\tau N)' = (\rho N)' \cap (\sigma N)'$

$= (\rho \underset{\circ}{\cup} \overset{\circ}{\rho}\ (XN)') \cap (\sigma \underset{\circ}{\cup} \overset{\circ}{\sigma}\ (XN)')$

$= \rho \underset{\circ}{\cap} \sigma \cup (\overset{\circ}{\rho}\ (XN)' \cap (\sigma \underset{\circ}{\cup} \overset{\circ}{\sigma}\ (XN)')) \cup (\overset{\circ}{\sigma}\ (XN)' \cap (\rho \underset{\circ}{\cup} \overset{\circ}{\rho}\ (XN)'))$

But τ is good relative to X and f and so we may deduce that:

$$\overset{\circ}{\rho}\ (XN)' \subseteq \sigma \underset{\circ}{\cup} \overset{\circ}{\sigma}\ (XN)'$$

$$\overset{\circ}{\sigma}\ (XN)' \subseteq \rho \underset{\circ}{\cup} \overset{\circ}{\rho}\ (XN)'$$

and therefore:

$$(\tau N)' = \rho \underset{\circ}{\cap} \sigma \cup (\overset{\circ}{\rho} \underset{\circ}{\cup} \overset{\circ}{\sigma})\ (XN)' = \tau \underset{\circ}{\cup} \overset{\circ}{\tau}\ (XN)'$$

iii The conclusion is that the lemma is true for all terms τ, good relative to X, f and Φ.

5.4 Termination Theorems

Let $f = \mu X(\tau)$.

If τ is simple in X, then:

5.4.1 $\vdash \iota\ (\overset{\circ}{\tau} \cup [\underset{\circ}{\tau}, E]) \subseteq fN$

If, in addition, τ is good relative to X, f and Φ then:

5.4.2 $\Phi \vdash \iota\ (\overset{\circ}{\tau} \cup [\underset{\circ}{\tau}, E]) = fN$

Proofs

5.4.1 The proof is by fixpoint induction.

$((\overset{\circ}{\tau} \cup [\underset{\circ}{\tau}, E])\ (fN)')' \subseteq (\overset{\circ}{\tau}\ (fN)' \cup \underset{\circ}{\tau})'$ since $fN \subseteq N$

But from 5.3.1, for the case that $X = f$,
$(fN)' = (\tau\ (f/X)\ N)' \subseteq \overset{\circ}{\tau}\ (fN)' \cup \underset{\circ}{\tau} \subseteq \overset{\circ}{\tau}\ (fN)' \cup \underset{\circ}{\tau}$

Therefore:
$((\overset{\circ}{\tau} \cup [\underset{\circ}{\tau}, E])\ (fN)')' \subseteq fN$, and hence, by fixpoint induction:
$\iota\ (\overset{\circ}{\tau} \cup [\underset{\circ}{\tau}, E]) \subseteq fN$.

5.4.2 The proof is by Scott Induction on $P(X, Y) \equiv \{XN = Y, X \subseteq f\}$ with $\mathfrak{F}(X) = \tau(X)$, $\mathfrak{G}(Y) = (\overset{\circ}{\tau} \cup \underset{\circ}{\tau} Y')'$

i $P(\Omega, \Omega)$ is clearly true.

ii Assume $P(X, Y)$.

a $\tau N = (\tau_\circ \cup \overset{\circ}{\tau}\ (XN)')'$, Induction Hyp, lemma 5.3.3

$= (\tau_\circ \cup \overset{\circ}{\tau}\ Y')'$.

b $\tau \subseteq \tau\ (f/X)$ Induction Hyp, monotonicity of τ since $f = \mu X\ (\tau)$.

$\subseteq f$

iii The conclusion is:

$$\mu X\ (\tau)\ N = \mu Y\ (\tau_\circ \cup \overset{\circ}{\tau}\ Y')'$$

or $fN = \iota\ ([\tau_\circ, E] \cup \overset{\circ}{\tau})$ using 4.3.26

5.5 More about Goodness

An important subset of terms which are good relative to X, f and Φ is the relational form of conditional expressions [McCarthy: A Basis for a Mathematical Theory of Computation]. Consider a conditional expression of the form $(p1 \rightarrow e1, p2 \rightarrow e2, \ldots, p_n \rightarrow e_n)$, whose value is the value of the e_i corresponding to the first p_i that is true. The corresponding relational form is:

$$\tau = [P_1, E]\ \tau_1 \cup [P_1' \cap P_2, E]\ \tau_2 \ldots \cup [P_1' \cap P_2' \cap \ldots P_{n-1}' \cap P_n, E]\ \tau_n$$

Assuming that we have first normalised the conditional expression so that the terms τ_i are union free, τ is made up of subterms of the form $\rho \cup \sigma$ with $\rho = [A, E]\ \rho_1$, $\sigma = [B, E]\ \sigma_1$ and $A \cap B = \Omega$. The derivatives of such ρ and σ are:

$$\overset{\circ}{\rho} = [A, E]\ \overset{\circ}{\rho}_1, \rho_\circ = A' \cup [A, E]\ \rho_{\circ 1}$$

$$\overset{\circ}{\sigma} = [B, E]\ \overset{\circ}{\sigma}_1, \sigma_\circ = B' \cup [B, E]\ \sigma_{\circ 1}$$

Thus $\overset{\circ}{\rho}\ N = [A, E]\ \overset{\circ}{\rho}_1\ N \subseteq A \subseteq B'$ since $A \cap B = \Omega$. Hence $\overset{\circ}{\rho}\ N \subseteq \sigma_\circ$ and $\overset{\circ}{\rho} \subseteq \sigma_\circ$ $\mho$ which satisfies the second criterion for goodness.

We have deliberately chosen this form for conditional expressions rather than the equivalent form of $\tau = [P_1, \tau_1] \cup [P_1' \cap P_2, \tau_2] \ldots$ since it allows us to use simpler

derivatives than those in Hitchcock and Park [1972] and to still obtain the desired properties of conditional expressions.

If, in addition, all the components of the subterms τ_i are single valued, then the subterms τ_i must be single valued. Further, no evaluations of the τ_i can proceed in parallel and so the function f, given by τ, is single-valued and the first criterion for goodness will be satisfied.

The definition of goodness allows a limited amount of non-determinism. The first goodness criterion will be satisfied by union free terms given by the following production rules:

<good term>::= <basic term>|[<good term>, <good term>]
<basic term>::= <free from X>|X|<det term> X
<free from X>::= <det term>|A|B ....
<det term>::= P|Q ...
where P, Q ... are single valued and A, B ... may be non-deterministic.

The second goodness criterion allows a certain amount of parallelism in evaluation, ie the domains of subterms involved can overlap under certain conditions. Notice however that the definition of the domain of a non-deterministic program is such that the program is considered to terminate for a given argument if there is at least one terminating computation sequence from that argument, not that all possible computation sequences from that argument terminate.

The property of goodness is certainly undecidable when Φ is, for example, the axioms of arithmetic.

Consider the term (G $\cup$ H) X which is simple in X and has the form $\rho\sigma$. Let G and H be the relational form of two functions g (x) and h (x). Clearly G $\cup$ H is single valued only if g (x) and h (x) are equivalent functions, a property which is well known to be undecidable in the arithmetic domain.

5.6 Extension to Multiple Recursions

5.6.1 Definitions

A term τ is simple in $X_1 \ldots X_n$ if

i τ has no free occurrence of any X_i, $1 \leq i \leq n$.

ii $\tau = X_i$, $1 \leq i \leq n$

iii $\tau = \rho\sigma$

iv $\tau = [\rho, \sigma]$

v $\tau = \rho \cup \sigma$

where ρ and σ are simple in $X_1 \ldots X_n$.

The i'th partial derivative $\overset{\bullet}{\tau}(i)$ and the co-derivative of a term τ simple in $X_1 \ldots X_n$ are obtained as follows:

i if τ has no free occurrences of $X_1 \ldots X_n$ then $\overset{\bullet}{\tau}(i) = \Omega$, $1 \leq i \leq n$ and $\overset{\bullet}{\tau} = (\tau N)'$

ii if $\tau = X_i$ then $\overset{\bullet}{\tau}(i) = E$, $\overset{\bullet}{\tau}(j) = \Omega$, $j \neq i$ and $\overset{\bullet}{\tau} = \Omega$

iii if $\tau = \rho\sigma$ then $\overset{\bullet}{\tau}(i) = \overset{\bullet}{\rho}(i) \cup \rho\, \overset{\bullet}{\sigma}(i)$ and $\overset{\bullet}{\tau} = \overset{\bullet}{\rho} \cup \rho\, \overset{\bullet}{\sigma}$

iv if $\tau = [\rho, \sigma]$ then $\overset{\bullet}{\tau}(i) = \overset{\bullet}{\rho}(i) \cup \overset{\bullet}{\sigma}(i)$ and $\overset{\bullet}{\tau} = \overset{\bullet}{\rho} \cup \overset{\bullet}{\sigma}$

v if $\tau = \rho \cup \sigma$ then $\overset{\bullet}{\tau}(i) = \overset{\bullet}{\rho}(i) \cup \overset{\bullet}{\sigma}(i)$ and $\overset{\bullet}{\tau} = \overset{\bullet}{\rho} \cap \overset{\bullet}{\sigma}$

Let $\overset{\circ}{\tau}(i)$ abbreviate $\overset{\bullet}{\tau}(i)\ (f_1/X_i, \ldots f_n/X_n)$
$\overset{\circ}{\tau}$ abbreviate $\overset{\bullet}{\tau}\ (f_1/X_1 \ldots f_n/X_n)$

A term τ is <u>good</u> relative to $X_1 \ldots X_n \quad f_1 \ldots f_n$ if τ is simple in $X_1 \ldots X_n$ and

i for any subterm $\rho\sigma$ of τ in which some X_i occurs free, $\rho\ (f_1/X_1, \ldots f_n/X_n)$ is single valued.

ii for any subterm $\rho \cup \sigma$ of τ

$$\overset{\circ}{\sigma}(i) \subseteq \underset{\circ}{\rho}\, \mho \cup \overset{\circ}{\rho}(i), 1 \leq i \leq n$$

$$\overset{\circ}{\rho}(i) \subseteq \underset{\circ}{\sigma}\, \mho \cup \overset{\circ}{\sigma}(i), 1 \leq i \leq n$$

5.6.2 <u>Lemmas</u>

If τ is simple in $X_1 \ldots X_n$, then $\underset{\circ}{\tau}$ and $\overset{\circ}{\tau}(i)$ are syntactically continuous in $X_1 \ldots X_n$.

If τ is simple in $X_1 \ldots X_n$ then:

$$(\tau N)' \subseteq \underset{\cdot}{\tau}\, \mho \cup \underset{i}{\cdot}\, \dot{\tau}(i)\ (X_i\ N)'$$

If τ is good relative to $X_1 \ldots X_n$, $f_1 \ldots f_n$ then:

$$\{X_i \subseteq f_i \mid 1 \leq i \leq n\} \vdash (\tau N)' = \underset{\circ}{\tau} \cup \underset{i}{\cup}\, \overset{\circ}{\tau}(i)\ (X_i N)'$$

The proofs of all the above lemmas proceed in a straight forward manner by induction on the formation rules for simple terms.

5.6.3 <u>Theorems</u>

Let $f_i = \mu_i\ X_1 \ldots X_n\ (\tau_1, \ldots \tau_n)\ 1 \leq i \leq n$

If each τ_i is simple in $X_1 \ldots X_n$ then

$\iota_k\ (\Sigma) \subseteq f_k\ N$

where Σ is an $n \times n$ matrix whose elements are given by:

$$(\Sigma)_{ij} = \overset{\circ}{\tau}_i(j) \text{ if } i \neq j, 1 \leq i, j \leq n$$

$$= \overset{\circ}{\tau}_i(i) \cup [\underset{\circ}{\tau}_i, E] \text{ if } i = j$$

If, in addition, each τ_i is good relative to $X_1 \ldots$

X_n, $f_1 \ldots f_n$, then:

$$\iota_k\ (\Sigma) = f_k\ N$$

The proofs of the above theorems are essentially similar to the single recursive case.

5.7 Examples

5.7.1 Descending Recursions

Many recursion equations have the form
$f\ (x, y) = (x \notin \bigcap_{i=1}^{n} \mathrm{dom}\ (S_i) \rightarrow h\ (x, y), k\ (f\ (S_1(x), j_1\ (x, y)), \ldots\ldots\ldots f\ (S_n\ (x), j_n\ (x, y)), x, y))$
where $h, k, j_1 \ldots j_n$ are known to be total.

In the relational form
$f = \mu X\ ([E_1\ \bigcup_{i=1}^{n} (S_i N)', E]\ H \cup [E_1\ \bigcap_{i=1}^{n} (S_i N), [E_1 S_1, J_1]\ X$
$\ldots\ldots\ldots\ldots [E_1\ S_n,\ J_n]\ X, E]\ K) = \mu X\ (\tau\ (X))$

Given that $KN = HN = J_i\ N = N, i = 1, 2 \ldots n$, then the derivatives are:

$$\overset{\circ}{\tau} = [\bigcap_i S_i N | N] \cap \bigcup_i [(S_i N)' | N] = \Omega$$
$$\overset{\circ}{\tau} = \bigcup_i [E_1\ S_i, J_i] \subseteq \bigcup_i [S_i | \mho] = [\bigcup_i S_i | \mho].$$

Hence from 5.4.1, 4.3.10, 4.3.11,
$$fN \supseteq \iota\ (\overset{\circ}{\tau} \cup [\underset{\circ}{\tau}, E]) \supseteq \iota\ ([\bigcup_i S_i | \mho] = [\iota\ (\bigcup_i S_i) | N]$$

A sufficient condition for termination is then provided by 4.3.15 ie that $S_i \subseteq R^\tau$ and $\iota\ (R) = N$ for $i = 1, 2, \ldots n$.

The familiar cases of such recursion equations are those of arithmetic, with R = pred, and LISP, with R = car ∪ cdr. The conditions $S_i \subseteq R^\tau$ then amount to $S_i \subseteq >$ and $S_i \subseteq$ is-superlist of respectively.

Primitive recursion is the special arithmetic case where $n = 1$, $J_1 = E$ and $S_1 =$ pred.

5.7.2 Bounded upward recursion

Any equation scheme of the form:

$$f\ (x, y) = ((x \in \iota\ (s)' \cup \mathrm{dom}\ (s)') \to g\ (x, y),$$
$$k\ (f\ (s\ (x), j\ (x, y)), x, y))$$

is total, provided that k, j and g are total.

The relational form is given by:

$$F = \mu X\ (\tau\ (X)) = \mu X\ ([E_1\ (\iota\ (S) \cap SN)', E]\ G \cup$$
$$[E_1\ (\iota\ (S) \cap SN), E]\ [[E_1\ S, J]\ X, E]\ K$$

The derivatives are:

$$\overset{\circ}{\tau} = [\iota\ (S) \cap SN | N] \cap ([(\iota\ (S) \cap SN)' | N] \cup$$
$$[E_1\ (\iota\ (S) \cap SN), E]\ [\ (SN)' |\ N]\ ) = \Omega$$
$$\overset{\circ}{\tau} = [E_1\ (\iota\ (S) \cap SN), E]\ [E_1\ S, J] \subseteq [[\iota\ (S), S] | \overset{\circ}{\tau}]$$

Hence $N \subseteq [\iota\ ([\iota\ (S), S]) | N] \subseteq \iota\ (\overset{\circ}{\tau}) \subseteq fN$
using 4.3.23, 4.3.20, 4.3.11, 4.3.10, 5.4.1

We can use this to establish the totality of an arithmetic function which counts up to some limit.
eg $f\ (x, y) = x \geq 10 \to y, f\ (x + 1, x + y)$

Let $S = [\ \underline{10}'\ , \underline{succ}\ ]$, $\underline{10}$ is the tuple $<10, \wedge>$ and $\underline{succ} = \{<x, x + 1> | x \geq 0\}$, $\iota\ (S) = \underline{succ}^*\ \underline{10} = \underline{\leq 10}$
using 4.3.9, $\iota\ (S)' = \underline{\geq 10}$ and so f is of the above form, and hence is total.

5.7.3 McCarthy's 91 Function

This is the function defined by:

$$f\ (x) = (x > 100 \to x - 10, f\ (f\ (x + 11)))$$

or, in a relational form

$f = \mu X\ ([\underline{> 100}, R^{10}] \cup [\underline{\leq 100}, E]\ R^{-11}\ X\ X)$
with $R = \underline{pred}$.

Applying 5.4.1 we have that

$\iota\ ([\underline{\leq 100}, E]\ R^{-11}\ (E \cup f)) \subseteq fN$

Let $g = [> \underline{100}, R^{10}] \cup [\leq \underline{100}, E]\ \underline{91}$
where $\underline{91} = \{<a, b> | a \geq 0 \ \& \ b = 91\}$

Then by fixpoint induction $f \subseteq g$ and so by 4.3.10
$\iota([\leq \underline{100}, E]\ R^{-11}\ (E \cup g)) \subseteq fN$
ie $\iota([\leq \underline{100}, E]\ R^{-11} \cup [\leq \underline{100} \cap > \underline{90}, R^{-1}] \cup [\leq \underline{90}, \underline{91}]) \subseteq fN$

Each of the three terms involved in the initial part is contained in $[\leq \underline{100}, <]$, and, as in example 5.7.2, $\iota([\leq \underline{100}, <]) = N$, taking $S = [\underline{100}', <]$.

Hence $fN = N$, from 4.3.10, and $f = g$ since g is single valued.

5.7.4 Ackermann's Function

We consider the following form of Ackermann's function.
f (x, y) =
if x = o then y + 1
if y = o then f (x - 1, 1)
else f (x - 1, f (x, y - 1))

The relational form for f is given by:

$$f = \mu X\ ([(RN)'|R^{-1}] \cup [R|A]\ X \cup [E_1\ R, [E|R]X]\ X]$$

where $R = \underline{pred}$
$A = \{<0, 1>\}$

The derivatives $\overset{\circ}{\tau}$ and $\overset{\circ}{\tau}$ are:

$$\overset{\circ}{\tau} = \Omega$$
$$\overset{\circ}{\tau} = [R|A] \cup [E|R] \cup [E_1\ R, [E|R]\ f]$$

But $\overset{\circ}{\tau} \subseteq [E|R] \cup [R|\mho]$
and $\iota(\overset{\circ}{\tau}) = N$ from 4.3.21, 4.3.10 and the axiom that $\iota(R) = N$

Hence Ackermann's function is total.

5.7.5 We continue with the example first introduced in 3.3.2 This was the pair of recursion equations:

$s\ (x_1, x_2) =$
if $x_1 = 0$ then 0
else $t\ (x_1, x_2, x_2)$
$t\ (x_1, x_2, x_3) =$
if $x_3 = 0$ then $s\ (x_1 - 1, x_2)$
else $t\ (x_1, x_2, x_3 - 1) + 1$

These were abstracted to the schema.

$G = \mu_1\ X_1\ X_2\ (A \cup BX_2, CX_1 \cup DX_2\ F)$
$H = \mu_2\ X_1\ X_2\ (A \cup BX_2, CX_1 \cup DX_2\ F)$

The interpretation which gives the recursion equations can be expressed entirely in terms of the predecessor relation R.

$A = [[(RN)', E]|N]$
$B = [[RN, E]|[E, E]]$
$C = [R|E|(RN)']$
$D = [E|E'|R]$
$F = R^{-1}$

We have among the axioms for the integers that $R^{-1} R = E$ and that $\iota(R) = N$

From 5.6.3 the domains of G and H are given by the initial part of the matrix:

$$\begin{Bmatrix} \Omega & B & \\ C & D \cup & [(CN)' \cap (DN)', E] \end{Bmatrix}$$

Using 4.5.27

$HN = \iota\ (D \cup CB \cup [(CN)' \cap (DN)', E])$
$= \iota\ (\ldots [E|E|R] \cup [R\ [RN, E]|[E, E]|(RN)']$
$\cup [[(RN)'|N|(RN)'], E])$
$= \iota\ ([[RN, E]|E|R] \cup [R\ [RN, E]|[E, E]|(RN)']) \cap$
$([RN|N|N] \cup [N|N|RN])$
using 4.3.27.
$\supseteq \iota\ ([E|E|R] \cup [R|E|E]) \cap [RN|N|N]$
using 4.3.10.

$= [\iota\ (R)\,|N|\,\iota\ (R)] \cap [RN|N|N]$ using 4.3.20

$= [RN|N|N]$ since $\iota\ (R) = N$.

Again using 4.5.27

$GN \supseteq ([[RN, E]|[E, E]]\ ([RN|N|N])')'$

$= [N|N|N]$

These expressions for the domains of G and H are sufficient to show the correctness of the programs in 3.3.2.

6 INTERPRETATIONS

6.1 Many Sortedness

We extend our concept of an m → n relation on a domain D to relations between tuples whose elements come from domains of different sorts. We will pursue this in a less formal fashion that that of chapter 2, assuming a non-empty interpretation in giving the semantics of terms.

6.1.1 Many Sorted Relation Variables

We use the following notations to specify the type of a many sorted relation R, on domains S_1, S_2 ... S_m, T_1 ... T_n.

R: $\langle S_1, S_2 \ldots S_m \rangle \rightarrow \langle T_1, T_2 \ldots T_n \rangle$

or $\langle S_1, S_2 \ldots S_m \rangle \overset{R}{\rightarrow} \langle T_1, T_2 \ldots T_n \rangle$

R is some relation between tuples from $S_1 \times S_2 \times \ldots S_m$ and $T_1 \times T_2 \ldots T_n$ whose elements are denoted by:

$$\langle\langle s_1, s_2 \cdots s_m \rangle, \langle t_1, t_2 \cdots t_n \rangle\rangle$$

eg $\underbrace{\langle \text{lists} \rangle \rightarrow \langle \text{atoms} \rangle}_{\text{head}}$, $\underbrace{\langle \text{lists} \rangle \rightarrow \langle \text{lists} \rangle}_{\text{tail}}$

6.1.2 Many Sorted Relation Constants

$$\langle S_1, \ldots S_m \rangle \overset{\upsilon}{\rightarrow} \langle T_1, \ldots T_n \rangle = \{\langle\langle s_1, \ldots s_m \rangle, \langle t_1, \ldots t_n \rangle\rangle | s_i \in S_i,\ 1 \le i \le m\ \&\ t_j \in T_j,\ 1 \le j \le n\}$$

$$\langle S_1, \ldots S_m \rangle \overset{\Omega}{\rightarrow} \langle T_1, \ldots T_n \rangle = \phi$$

$$\langle S_1, \ldots S_m \rangle \overset{E}{\rightarrow} \langle S_1, \ldots S_m \rangle = \{\langle\langle s_i, \ldots s_m \rangle, \langle s_1, \ldots s_m \rangle\rangle | s_i \in S_i,\ 1 \le i \le m\}$$

$$\langle S_1, \ldots S_m \rangle \overset{E_i}{\rightarrow} \langle S_i \rangle = \{\langle\langle s_1, \ldots s_m \rangle, s_i \rangle | s_j \in S_j,\ 1 \le j \le m\}$$

$$\langle S_1, \ldots S_m \rangle \overset{N}{\rightarrow} o = \{\langle\langle s_1 \ldots s_m \rangle, \wedge\rangle | s_i \in S_i,\ 1 \le i \le m\}$$

6.1.3 Many Sorted Operations

The basic operations between many sorted relations are defined as obvious extensions to the basic operations defined previously.
eg if A is of type $\langle S_1, S_2 \ldots S_m \rangle \rightarrow \langle T_1, \ldots T_n \rangle$, and B is of type $\langle U_1, U_2 \ldots U_p \rangle \rightarrow \langle V_1, \ldots V_q \rangle$ then [A, B] is of type $\langle S_1, \ldots S_m \rangle \rightarrow \langle T_1, \ldots T_n, V_1 \ldots V_q \rangle$
iff $m = p \;\&\; S_i = U_i,\ 1 \leq i \leq m$.

$[A, B] = \{\langle a, b \cap c \rangle | \langle a, b \rangle \in A \;\&\; \langle a, c \rangle \in B\}$.

6.2 Axioms for Data Structures

We will now give axiomatic definitions for some of the basic domains likely to occur in programs, and their associated operations.

Again type indications will be omitted when possible. The rules governing well formed terms will usually enable them to be restored.

6.2.1 Finite Sets

Our ultimate interest is in objects which can be represented in a machine, and so the "set theory" given here is more restrictive than any general set theory. We deal here with finite sets of objects which satisfy a predicate is-el. The basic operation, the removal of an element from such a set, is given the name sub. It is of type <is-set> → <is-set, is-el>, and is defined by the following axioms. These sets correspond to the powerset type of PASCAL, Wirth [1971b].

D(i) $\quad \langle s \rangle \xrightarrow{\phi} o \overset{D}{=} (\langle s \rangle \xrightarrow{sub} \langle s, e \rangle \langle s, \overset{N}{e} \rangle \rightarrow o)'$

D(ii) $\underset{\langle s\rangle \to \langle e\rangle}{\underline{\text{in}}} \overset{D}{=} \underset{\langle s\rangle \to \langle s,\ e\rangle}{\underline{\text{sub}}}\ E_2\ \langle s,\ e\rangle \to \langle e\rangle$

i $\phi\ \phi^{-1} \subseteq_{\langle s\rangle} \overset{E}{\to} \langle s\rangle$

ii $\iota\ (\underline{\text{sub}}\ E_1) = {}_{\langle s\rangle}N_{\to}\ o$ = is-set.

iii $\underline{\text{sub}}^{-1}\ \underline{\text{sub}}\ E_2 = E_1\ \underline{\text{sub}}\ E_2 \cup E_2$

iv $\underline{\text{sub}}\ \underline{\text{sub}}^{-1} \subseteq_{\langle s\rangle} \overset{E}{\to} \langle s\rangle$

v $(\underline{\text{sub}}^{-1}\ {}_{\langle s\rangle}N_{\to}\ o)' = \underset{\langle s,\ e\rangle \to o}{\underline{\text{in}}}$

We show in 6.3, that any model of these axioms is isomorphic to the set of finite subsets of elements from is-el with $\underset{\langle s\rangle \to \langle s,\ e\rangle}{\underline{\text{sub}}} = \{\langle a, \langle b, c\rangle\rangle \mid a \neq \phi\ \&\ c \notin b\ \&\ a = b \cup \{c\}\}$.
ie the axioms are complete relative to interpretations of is-el.

The more familiar set operations can now be defined in terms of the basic relation sub.

Define $\underset{\langle s,\ e\rangle \to \langle s\rangle}{\underline{\text{add}}} = \underline{\text{sub}}^{-1} \cup [\underset{\langle s,\ e\rangle \to o}{\underline{\text{in}}}, E_1]$
then:

$$\underset{\langle s,\ s\rangle \to \langle s\rangle}{\underline{\text{union}}} = \mu X\ ([E \mid \phi] \cup [E\ \underline{\text{sub}}]\ [[E_1, E_2]\ X, E_3]\ \underline{\text{add}})$$

$$\underset{\langle s,\ s\rangle \to \langle s\rangle}{\underline{\text{intersect}}} = \mu X\ (E_2\ [\phi, E] \cup [E \mid \underline{\text{sub}}]\ ([[E_1, E_3]\ \underline{\text{in}}, E_1, E_2]\ X \cup [[E_1, E_3]\ \underline{\text{in}}, [E_1, E_2]\ X, E_3]\ \underline{\text{add}}))$$

$$\underset{\langle s,\ s\rangle \to \langle s\rangle}{\underline{\text{difference}}} = \mu X\ (E_1\ [\phi, E] \cup [\underline{\text{sub}} \mid E]\ ([[E_3, E_2]\ \underline{\text{in}}, E_1, E_3]\ X \cup [[E_3, E_2]\ \underline{\text{in}}', [E_1, E_3]\ X, E_2]\ \underline{\text{add}}))$$

Notice that no complement operation is defined. If the domain is-el were infinite then this would result in infinite sets.

6.2.1.1 We can generalise our ideas about sets to describe finite sets which can include finite sets as elements. Consider the following axioms, sub is now a $2 \rightarrow 1$ relation on a mixed domain of sets and elements.

i $(\underline{\text{sub}}\ N)' = \phi \cup \underline{\text{is-el}}$

ii $\phi \cap \underline{\text{is-el}} = \Omega$

iii $\iota\ (\underline{\text{sub}}\ E_1 \cup \underline{\text{sub}}\ E_2) = N$

iv $\underline{\text{sub}}\ \underline{\text{sub}}^{-1} \subseteq E$

v $\underline{\text{sub}}^{-1}\ \underline{\text{sub}}\ E_2 = E_1\ \underline{\text{sub}}\ E_2 \cup E_2$

vi $(\underline{\text{sub}}^{-1}\ N)' = {}_{2 \rightarrow}\overset{\text{in}}{}\ 0$

vii $(\underline{\text{sub}}^{-1}\ N) \subseteq [\underline{\text{is-set}}|N]$

D(i) $\underline{\text{is-set}} = \phi \cup \underline{\text{sub}}\ N$

D(ii) ${}_{1 \rightarrow}\overset{\text{in}}{}\ 1 = \underline{\text{sub}}\ E_2$

We allow any object to be added to a set, and the induction rule now states that the domain of sets is well founded with respect to the operations of taking a subset and of taking an element.

The induction rule does not allow sets which are elements of themselves, using 4.3.17, and so paradoxes do not arise.

These objects bear the same resemblance to the finite sets of 6.2.1, as LISP S-expressions bear to linear lists.

6.2.2 Trees

The following objects are loosely based on those of the Vienna Definition Language, and are a generalisation of LISP S-expressions.

We consider first the single sorted case.

We suppose a finite number of constructor relations $\underline{mk-i_j}$ $i \geq 1$, where i gives the type of the relation as $i \to 1$, and j distinguishes relations of the same type.

These relations satisfy the following axiom schema.

i $\quad \underline{mk-i_j}\ \underline{mk-\ell_k}^{-1} = E, j = k, i = \ell$

$\quad = \Omega$ otherwise

ii $\quad \underline{mk-i_j}^{-1}\ \underline{mk-i_j} \subseteq E$

iii $\quad \iota\ (\bigcup_{i,\ j,\ k} \underline{mk-i_j}^{-1}\ E_k) = N$

Di $\quad \bigcap_{i,\ j} (\underline{mk} - i_j^{-1}\ N)' \overset{D}{=}$ is-eo, the set of elementary objects.

These easily extend to the many sorted case.

6.2.2.1 Arithmetic expressions

Consider the following definition of arithmetic expressions in the Vienna Notation, Walk et al [1969].

is-expr::= is-var ∨ is-unary ∨ is-binary

is-binary::= (<S1: is-expr>, <S2: is-bin-op>, <S3: is-expr>)

is-unary::= (<S4: is-un-op>, <S5: is-expr>)

is-un-op::= + ∨ -

is-bin-op::= + ∨ - ∨ * ∨ /

is-var::= x ∨ y ∨ z

We use $\langle e, \overline{b, e} \xrightarrow{mk-3} \langle e \rangle$ and $\langle u, \overline{e} \xrightarrow{mk-2} \langle e \rangle$ many sorted constructor relations, where e abbreviates is-expression, b, is-bin-op and u is-un-op, called mk-bin and mk-un respectively, and abbreviate, $\underline{mk-3}^{-1}\ E_1$ to S1 etc. Then we have that

i $(\underline{mk\text{-}un}^{-1}\ N)' \cap (\underline{mk\text{-}bin}^{-1}\ N)'' \overset{D}{=} \underline{is\text{-}var}$.

ii The induction axiom is given by:

$$\text{is-expr} = \iota_1\ (M)$$
$$\text{is-un-op} = \iota_2\ (M)$$
$$\text{is-bin-op} = \iota_3\ (M)$$

where M is the 3 × 3 matrix.

$$\begin{pmatrix} (\langle e \rangle \overset{S1}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle e \rangle \cup \langle e \rangle \overset{S3}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle e \rangle \cup \langle e \rangle \overset{S5}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle e \rangle) & \langle e \rangle \overset{S4}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle u \rangle & \langle e \rangle \overset{S2}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle b \rangle \\ \langle u \rangle \overset{\Omega}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle e \rangle & \langle u \rangle \overset{\Omega}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle u \rangle & \langle u \rangle \overset{\Omega}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle b \rangle \\ \langle b \rangle \overset{\Omega}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle e \rangle & \langle b \rangle \overset{\Omega}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle u \rangle & \langle b \rangle \overset{\Omega}{\rightarrow\!\!\!\!\!\!\rightarrow} \langle b \rangle \end{pmatrix}$$

By way of further explanation.

$\iota_3\ (M) = \iota\ (\langle b \rangle \overset{\Omega}{\rightarrow} \langle b \rangle)$ by 4.5.30
= is-bin-op by 4.3.12

$\iota_2\ (M) = \iota\ (\langle u \rangle \overset{\Omega}{\rightarrow} \langle u \rangle)$ = is-un-op similarly.

$\iota_1\ (M) = \mu X\ (((S1 \cup S3 \cup S5)\ X')' \cap (S4\ \text{is-un-op}')' \cap (S2\ \text{is-bin-op}')')$

ie, for a binary expression the S1 and S3 components must be expressions and the S2 component is a binary operator, recalling that

$(AX')'$ is the set all of whose predecessors under A are in X.

6.2.2.2 Operations on Trees

The basic operation on a tree is the ability to select a branch and modify it without affecting the other branches. This is analogous to the μ-operator of the Vienna Definition Language. We model this assignment in the following fashion.

Assume that we wish to modify the j'th component of a tree constructed by a mk - i relation. This is done by the term. $[\underline{mk - i}^{-1}|E]\ [E_1 \ldots E_{j-1}, E_{i+1}, E_{j+1}, \ldots E_i]\ \underline{mk - i}$

The modification to the tree is in the second argument. The tree is decomposed, into i components, the j'th one is changed, and the tree is reconstructed. The property that modification does not affect other branches follows immediately from the properties of selection and concatenation, in the same way as do those in 3.1.1

6.2.2.3 Structures

We have in mind the structures of PL/I or the record types of Pascal. No induction axiom is required in their definition for they are basically storage disciplines rather than recursively defined objects.

A structure whose components are of different types is defined using a single many sorted mk - i relation. The axioms are:

i, ii, iii as for 6.2.2.

iv $$\langle s \rangle \xrightarrow{mk\_n^{-1}} \langle T_1, \ldots T_n \rangle \; [is - T_1 \,|is - T_2 \,|\ldots. is - T_n] = \langle s \rangle \xrightarrow{N}{}_o = \underline{\text{is-structure-s}}$$

6.2.2.4 Arrays

Arrays, with fixed bounds, are a special case of structures, the components are all of the same type and the integers are used as selectors.

6.2.2.5 Limitations

The formalism is limited to operations of known and finite types. It is not possible to form tuples of an arbitrary length and hence to construct arrays of an arbitrary length. The arrays of such languages as APL cannot therefore be described.

The Vienna Definition Language uses objects which may be considered as trees with an infinite number of selectors, a finite number of which are non-empty. We cannot therefore describe these, but only that subset where the selectors used are all known in advance.

6.2.4 Lists

6.2.4.1 LISP S-expressions

These are a special case of trees using a single mk - 2 relation to construct binary trees from elements satisfying is-atom.

i $\quad \frac{mk - 2}{2 \to 1} \frac{mk - 2^{-1}}{1 \to 2} = E$

ii $\quad \underline{mk - 2^{-1}}\ \underline{mk - 2} \subseteq E$

iii $\quad \iota\ (\underline{mk - 2^{-1}}\ E_1 \cup \underline{mk - 2^{-1}}\ E_2) = N$
$= \underline{\text{is-s-expressions}}$

Di $\quad (\underline{mk - 2^{-1}}\ N)^{1} \overset{D}{=} \underline{\text{is-atom}}$

$\underline{mk - 2}$ is more usually known as $\underline{\text{cons}}$, $\underline{mk - 2^{-1}}\ E_1$ as $\underline{\text{car}}$, $\underline{mk - 2^{-1}}\ E_2$ as $\underline{\text{cdr}}$.

6.2.4.2 Linear Lists

These may be modelled as a restricted form of binary trees whose right hand components are always atoms. A special object $\underline{\text{nil}}$ is distinguished to signal the end of a list, and denotes the null list.

i $\quad \frac{mk - 2}{\langle \ell, a \rangle \to \langle \ell \rangle} \frac{mk - 2^{-1}}{\langle \ell \rangle \to \langle \ell, a \rangle} =$
$[{}_{\langle \ell \rangle} \overset{E}{\to} {}_{\langle \ell \rangle} | {}_{\langle a \rangle} \overset{E}{\to} {}_{\langle a \rangle}]$

ii $\quad \underline{mk - 2^{-1}}\ \underline{mk - 2} \subseteq {}_{\langle \ell \rangle} \overset{E}{\to} {}_{\langle \ell \rangle}$

iii $\quad \underline{\text{nil}}\ \underline{\text{nil}}^{-1} \subseteq {}_{\langle \ell \rangle} \overset{E}{\to} {}_{\langle \ell \rangle}$

iv $\quad \iota\ (\underline{mk - 2^{-1}}\ E_1) = N = \underline{\text{is-linear-list}}$

Di $\quad (\underline{mk - 2^{-1}}\ N)' \overset{D}{=} \frac{\underline{\text{nil}}}{\langle \ell \rangle \to o}$

$\underline{mk - 2}$ is known as $\underline{\text{cons}}$, $\underline{mk - 2^{-1}}\ E_1$ as $\underline{\text{tail}}$, $\underline{mk - 2^{-1}}\ E_2$ as $\underline{\text{head}}$

6.2.4.3 Linear Lists with no repeated elements

We achieve lists with no repeated elements by restricting the $\underline{mk - 2}$ operation so that

atoms are only added to lists if they have not been added before.

i $\underset{\langle \ell,\ a\rangle \to \langle \ell\rangle}{\underline{mk-2}}\ \underset{\langle \ell\rangle \to \langle \ell,\ a\rangle}{\underline{mk-2^{-1}}} = [\underset{\langle \ell,\ a\rangle \to o}{\underline{\text{list in}'}},\ [_{\langle \ell\rangle} \overset{E}{\to} \langle \ell\rangle |_{\langle a\rangle} \overset{E}{\to} \langle a\rangle]]$

ii, iii, iv, Di as above.

Dii $\underset{\langle \ell\rangle \to \langle a\rangle}{\underline{(mk-2^{-1}E_1)}}^{*}\ \underline{mk-2^{-1}E_2}^{D} =$

6.2.4.4 Ordered Linear Lists

Again we have the axioms for linear lists with the mk - 2 operation restricted so that the atom which is added is greater than the one at the head of the list being added to.

i $\underline{mk-2}\ \underline{mk-2^{-1}} = [_{\langle \ell,\ \overset{<<}{a}\rangle \to o}, [_{\langle \ell\rangle} \overset{E}{\to} \langle \ell\rangle |_{\langle a\rangle} \overset{E}{\to} \langle a\rangle]]$

ii, iii, iv, v, Di as above.

Dii $\underset{\langle \ell,\ \overset{<<}{a}\rangle \to o}{} = [_{\langle \ell\rangle} \underset{o}{\text{nil}} |_{\langle a\rangle \to o} \underset{o}{\text{is-atom}}] \cup [\underset{\langle 1\rangle \to \langle a\rangle}{\underline{mk-2^{-1}}} E_2 |_{\langle a\rangle} \overset{E}{\to} \langle a\rangle]\ \underset{\langle a,\ a\rangle \to o}{\overset{<}{}}$

where $\langle a\rangle \overset{\leq}{\to} \langle a\rangle$ is the ordering relation between atoms.

6.2.5 Constructed Domains

We have already seen 6.2.2.3, an example of the construction of a new domain from other domains. We now give two further examples.

6.2.5.1 Discriminated Union

These objects correspond to those with the

now obsolete CELL attribute of PL/I or to the union type of PASCAL.

We use many sorted conv relations to "convert" from a domain to the discriminated union domain. These relation relations obey the following axiom

i $\quad \langle s_i \rangle \overset{conv}{\rightarrow} \langle T \rangle \; \langle T \rangle \overset{conv^{-1}}{\rightarrow} \langle s_j \rangle \subseteq \langle s_i \rangle \overset{E}{\rightarrow} \langle s_i \rangle$
if $i = j$
$= \Omega$ otherwise.

This ensures that we are able to test unambiguously for the original sort of an individual element. Again this is basically a storage discipline.

6.2.5.2 Cartesian Product

We already have in our many sorted formalism the ability to form direct products of domains and to select components from those domains.

6.2.6 Integers

Finally we give the axioms for the integers.

i $\quad \overset{R}{1 \rightarrow 1} \; \overset{R^{-1}}{1 \rightarrow 1} \subseteq E$

ii $\quad R^{-1} R = E$

iii $\quad (RN)' \overset{D}{=} \underline{\text{is zero}}$

iv $\quad \underline{\text{is-zero}} \; \underline{\text{is-zero}}^{-1} \subseteq E$

v $\quad \iota (R) = N = \underline{\text{is-integer}}$

6.2.7 Representations

We will say nothing here concerning the modelling and representation of the objects defined axiomatically above but will instead refer the reader to Hoare [1972b].

6.3 Appendix

We show that any model of the axioms of 6.2.1 for sets is isomorphic to the set of finite subsets of elements from is-el with sub = $\{\langle a, \langle b, c\rangle\rangle \mid a \neq \phi \;\&\; \text{is-el}(c) \;\&\; c \notin b \;\&\; a = b \cup \{c\}\}$.

Let R be any relation which satisfies the axioms for sub and define $\psi(x) = \{c \mid \langle x, c\rangle \in RE_2\}$ as a mapping from the domain associated with R to the set of finite subsets of elements from is-el.

We show that ψ is an isomorphism, ie that ψ is single valued, total, onto, and preserves R, and that ψ^{-1} is single valued.

A ψ is single valued, by definition.

B ψ is a homomorphism with disjoint union compatible with R^{-1}.
ie $(\forall xyz)\ (\langle x, \langle y, z\rangle\rangle \in R \leftrightarrow \psi(x) = \psi(y) \cup \{z\} \;\&\; z \notin \psi(y))$

Proof: from iii $\{\langle\langle y, z\rangle, w\rangle \mid (\exists x)\ ((x, \langle y, z\rangle\rangle \in R \;\&\; w = \psi(x))\}$
$= \{\langle\langle y, z\rangle, w\rangle \mid w = \psi(y) \vee w = \{z\}\}$
(identifying $\psi(x)$ with RE_2)

from iv for each y, z there exists at most one x such that $\langle x, \langle y, z\rangle\rangle \in R$

from v D(ii) if $z \notin \psi(y)$ there is exactly one such x

Hence $(\forall xyz)$ $(\langle x, \langle y, z\rangle\rangle \in R \leftrightarrow \psi(x) = \psi(y) \cup \{z\}$ & $z \notin \psi(y))$

C ψ is total. This follows from ii and B.

D ψ is onto.

We show by induction on the number of elements in w that $(\forall w)$ $(\exists x)$ $w = \psi(x)$.
if $\omega = \{\}$ then $x = \phi$, from 1, D(i)

Assume $(\exists y)$ $w = \psi(y)$
then $(\exists x)$ $w \cup \{z\} = \psi(x)$ & $z \notin w$
since $(\exists x)$ $\psi(x) = \psi(y) \cup \{z\}$ & $z \notin \omega$ from induction hypothesis, B, v, D(ii)
with $\langle x, \langle y, z\rangle\rangle \in R$.

E ψ^{-1} is single valued ie $\psi(x) = \psi(x^1) \rightarrow x = x^1$

Suppose $\psi(x) = \psi(x^1)$, then either $\psi(x) = \phi$ or $\psi(x) = \{y_1, \ldots y_n\}$ since ψ is total.
if $\psi(x) = \phi$ then $x = x^1 = \phi$ from D(i), i.
if $\psi(x) = y \cup \{z\}$ & $z \notin y$ for some y, z
then $y = \psi(w)$ since ψ is onto
ie $\psi(x) = \psi(x^1) = \psi(w) \cup \{z\}$ & $z \notin \psi(w)$
$\rightarrow \langle x^1, \langle w, z\rangle\rangle \in R$ & $\langle x, \langle w, z\rangle\rangle \in R$ from B
$\rightarrow x = x^1$ from iv

6.4 Extensions to Hoares Axioms

The relational formalism may become notationally very clumsy when talking about complicated programs, and it may be more convenient to switch to the first order predicate calculus. This is done in a manner which generalises the development of Hoare [1969] in describing program semantics:

6.4.1 Non-Constructive Definition

We define an interpretation for a relation in the

following manner.
$S = \{\langle a, b\rangle | P(a, b)\}$.

The predicate P (a, b) is a formula in the first order predicate calculus whose domain, functions and predicate letters are known.

We find it convenient to express the predicate P in such a way that the domain of S is made explicit ie
$S = \{\langle a, b\rangle | Q(a) \& R(a, b)\}$ and $Q(a) \rightarrow (\exists b)\ R(a,b)$

These relations correspond precisely to the minimal valid predicates of Manna and Pnueli [1970].

6.4.2 Operations between non-constructive relations

We assume that we have the relations
$R = \{\langle a, b\rangle | P(a) \& Q(a, b)\}$ and $P(a) \rightarrow (\exists b)\ Q(a,b)$

$S = \{\langle a, b\rangle | T(a) \& U(a, b)\}$ and $T(a) \rightarrow (\exists b)\ U(a,b)$

The remainder of this section gives expressions for the basic operations between R and S. These are special cases of the operations defined before, and are shown in 6.4.3 to be a generalisation of Hoare [1969] .

6.4.2.1 Composition

If $P(a) \& Q(a, b) \rightarrow T(b)$
$P(a) \& Q(a, b) \& U(b, c) \rightarrow V(a, c)$
then R; $S = \{\langle a, c\rangle | P(a) \& V(a, c)\}$ and
$P(a) \rightarrow (\exists c)\ V(a, c)$.

6.4.2.2 Concatenation

If $O(a) = P(a) \& T(a)$
$P(a) \& T(a) \& Q(a, b) \& U(a, c) \& d =$

$b \cap c \rightarrow V(a, d)$
then $[R, S] = \{\langle a, d\rangle | O(a) \& V(a, d)\}$
and $O(a) \rightarrow (\exists d)\ V(a, d)$

6.4.2.3 Union

$R \cup S = \{\langle a, b\rangle | (P(a) \vee T(a)) \& ((P(a) \& Q(a, b)) \vee (U(a, b) \& T(a)))\}$

6.4.3 Hoares Axioms

If we had chosen to define relations by predicates on the input and output tuples separately, ie in the manner $S = \{\langle a, b\rangle | P(a) \& R(b)\}$, and did not explicitly define the domain of S, then operations between such relations model the axioms of Hoare [1969].

6.4.3.1 Notation

The notation $P \{Q\} R$ is taken to mean, "if the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion."

Let S be the relation $\{\langle a, b\rangle | P(a) \& R(b)\}$, then if the program Q, restricted to inputs satisfying P, is to satisfy R on termination, then $[P, E]\ Q \subseteq S$, and similarly, if the program Q is restricted to outputs satisfying R, then $S \subseteq Q\ [R, E]$

ie the notation $P \{Q\} R$ is represented by the formula $[P, E]\ Q \subseteq Q\ [R, E]$.

6.4.3.2 Rule of Composition

The rule of composition is:

If $\vdash P \{Q_1\} R_1$ and $\vdash R_1 \{Q_2\} R$ then

$$\vdash P \{Q_1; Q_2\} R$$

In the relational formalism.

$$[P, E] Q_1 \subseteq Q_1 [R_1, E], [R_1, E] Q_2 \subseteq Q_2 [R, E]$$

$$\vdash [P, E] Q_1 Q_2 \subseteq Q_1 [R_1, E] Q_2 \subseteq Q_1 Q_2 [R, E]$$

6.4.3.3 Rule of Iteration

The rule of iteration is:

if $\vdash P \& B \{S\} P$ then $\vdash P \{$while B do $S\} \neg B \& P$

The while loop is represented by the term

$L = \mu X ([B', E] \cup [B, E] S X)$

We wish to show that

$[P, E] [B, E] S \subseteq S [P, E] \vdash [P, E] L \subseteq L [B', E] [P, E]$

The proof proceeds by Scott Induction on P(X).

$P(X) \equiv [P, E] X \subseteq X [B', E] [P, E]$.

i P (Ω) is true

ii Assume P (X),

$[P, E] ([B', E] \cup [B, E] SX) \subseteq [B', E] [P, E] \cup [B, E] S [P, E] X$

$\subseteq [B', E] [P, E] \cup [B, E] SX [B', E] [P, E]$

$\subseteq ([B', E] \cup [B, E] SX) [B', E] [P,E]$

iii Hence $[P, E] L \subseteq L [B', E] [P, E]$

7 REPRESENTATION OF DATA

One may start to write a program and specify its domain and basic operations in an axiomatic manner. Development proceeds by finding satisfactory representations for this domain in terms of more specific domains and their operations, until finally we have domains which are representable in our target language. We must distinguish between the general notion of finding a representation such that any program will work, from finding a representation such that a particular program will work. The two may well be different. Compiler writers would be interested in the former whilst the latter is of use in the development of programs.

7.1 Representations

Equality between elements of a domain is a basic predicate which we assume in the set theoretic definition of the formalism When we progress, in the development of a program, from one model of the domain to another, there may be several possible representations of a single element from the first domain in the new domain, and we must ensure that this notion of equality is preserved, i.e. that any two representations of the same original element must be considered equal.

We show how to go, by a simple substitution process, from a program σ, written in a language L_ϕ, to an equivalent program $\hat{\sigma}$ written in another language L_ψ using representations of the basic operations of L_ϕ, and we state a representation theorem which enables such representations to be validated.

The interpretations of the new program $\hat{\sigma}$ will, in general, be inefficient since they are essentially non-deterministic making copious use of an equivalence relation, in order to preserve the notion of equality mentioned earlier, and we introduce the concept

of good representations which make a minimal use of such equivalence relations.

7.1.1 Representation Theorem

7.1.1.1 Operations modulo equivalence classes

If we have a domain N_ψ and an equivalence relation R on N_ψ which relates different representations of the same object from a domain N_ϕ, then we define N_ψ mod R to be the domain whose elements are equivalence classes of objects from N_ψ.

$\bar{x} \in N_\psi \bmod R \iff x \in R\ N_\psi$

$\iff x \in N_\psi$ since R is total.

where $\bar{x}$ is the equivalence class which contains x.

We define operations modulo these equivalence classes as follows.

$\langle \bar{x}, \bar{y} \rangle \in {}_m\tau_n \bmod R \iff \langle x, y \rangle \in R_m \tau R_n$

where $R_o = {}_o\underset{\rightarrow}{E}{}_o$

$R_1 = R$

$R_{n+1} = [R_n | R]$

This definition is meaningful because R is an equivalence relation. Consider the tuple $\langle \hat{x}, \hat{y} \rangle$ such that $\hat{x} \in \bar{x}$, $\hat{y} \in \bar{y}$.

Then $\langle \bar{x}, \bar{y} \rangle \in \tau \bmod R \Rightarrow \langle x, y \rangle \in R \tau R$

$\Rightarrow R\ \langle x, y \rangle\ R \subseteq RR \tau RR \subseteq R \tau R$

$\Rightarrow \langle \hat{x}, \hat{y} \rangle \in R \tau R$

$RR = R$ since R is an equivalence relation,

the extension for $m \to n$ relations and many sorted relations is straight forward.

7.1.1.2 Lemma

Let L_ψ be a language with constants ψ_i, given by a set of axioms $\underline{\psi}$, let Δ be an equivalence relation added to the language L_ψ, and let I_ψ be an interpretation of this language over a domain N_ψ which assigns an equivalence relation R to Δ. In addition let L_ϕ be another language with constants ϕ_i, given by a set of axioms $\underline{\Phi}$ and let I_ϕ be an interpretation of L_ϕ over the domain N_ψ mod R which assigns to the constants ϕ_i the relation $I_\psi(\tau_i)$ mod R, where τ_i are terms in the language L_ψ.

Then for all terms σ in L_ϕ,

$\langle \bar{x}, \bar{y} \rangle \in I_\phi(\sigma) \iff$

$\langle x, y \rangle \in I_\psi(\sigma(\Delta/E, E_i\Delta/E_i, \Delta\tau_i\Delta/\phi_i))$.

The proof, which will not be given here, proceeds by straightforward induction on the formation rules for terms in L_ϕ.

7.1.1.3 Representation Theorem

Let L_ψ, I_ψ, L_ϕ, I_ϕ and Δ be as in 7.1.1.2

Then I_ϕ satisfies the axioms $\underline{\Phi} \iff$

I_ψ satisfies the modified axioms

$$\underline{\Phi}(\Delta/E, E_i\Delta/E_i, \Delta\tau_i\Delta/\phi_i)$$

Proof:

The individual elements of the set of axioms $\underline{\Phi}$ are atomic formulae of the form $\alpha \subseteq \beta$.

I_ϕ satisfies $\alpha \subseteq \beta$

$$<=> I_{\phi} \ (\alpha) \subseteq I_{\phi} \ (\beta)$$

$$<=> I_{\psi} \ (\alpha(\Delta/E, E_i\Delta/E_i, \Delta \tau_i\Delta/\phi_i)) \subseteq I_{\psi} \ (\beta(\Delta/E, E_i\Delta/E_i, \Delta\tau_i \Delta/\phi_i))$$

using the previous lemma

$$<=> I_{\psi} \text{ satisfies } (\alpha \subseteq \beta) \ (\Delta/E, E_i\Delta/E_i, \Delta \tau_i\Delta/\phi_i))$$

The intended use of this theorem is to validate that terms $\Delta \tau_i\Delta$ in the language L_{ψ} are representations of constants ϕ_i in the original language L_{ϕ}.

7.1.2 Representations of programs

Having used the representation theorem to validate that terms $\Delta \tau_i\Delta$ in L_{ψ} are representations of constants ϕ_i in the original language L_{ϕ}, it is a straightforward consequence of the lemma that a program P in L_{ϕ}, modified by the substitution $(\Delta/E, E_i\Delta/E_i, \Delta \tau_i\Delta/\phi_i)$ is a program $\hat{P}$ in the language L_{ψ} such that any representation in N_{ψ} of the input to P is mapped to any representation of the output of P.

We identify as good terms those which have the property that $I_{\psi}(\Delta\tau_i\Delta) = I_{\psi}(\tau_i\Delta)$, where τ_i does not contain Δ, and it is clear that good terms which are combined by the operations of composition, concantenation and union result in good terms. This concept enables simplifications to be made to program $\hat{P}$ such that it will be more efficient. In many cases the resulting program $\hat{P}$ will be good, and if the final equivalence relation in $\hat{P}$ is removed the program $\hat{P}$ produces a representation of the result of the program P rather than all, which is usually all that is required.

Unfortunately this concept of goodness is not all that powerful, since it is possible to produce programs which are good from components which are not good (see the list union program used later as an example).

7.1.3 Example

We take as an example the language L_ϕ of finite sets defined by the set of axioms $\overline{\Phi}$ of 6.2.1 and the language L_ψ, linear lists with no repeated elements defined by the axioms Ψ of 6.2.4.3. L_ψ is extended by Δ.

Define a function f which maps lists into the finite set of elements in the list.

$$f = \mu X\ (\underline{nil}\ \phi^{-1} \cup \underline{cons}^{-1}\ [E, X, E_2]\ \underline{sub}^{-1})$$

and define the equivalence relation:

$$R = ff^{-1}$$

This is assigned to Δ in the interpretation I_ψ. The relation R makes equivalent all lists with the same set of elements. E is the equivalence relation on the domain of elements of lists. We will show, in 7.1.3.1, that R has the following properties.

1 $RR = R$
2 $R = R^{-1}$
3 $E \subseteq R$
4 $[R|E]\ \underline{cons} \subseteq \underline{cons}\ R$
5 $R\ \underline{cons}^{-1}\ E_2 = (\underline{cons}^{-1}\ E_1) * \underline{cons}^{-1}\ E_2 = \underline{listin}$

Using these properties we can show that I_ψ satisfies the modified axioms

$$\overline{\Phi}\ (\Delta/E,\ E_i\ \Delta/E_i,\ \Delta\ \underline{cons}^{-1}[\Delta\ |E]/\ \underline{sub})$$

Making the substitutions in the axioms of 6.2.1

i $(R\ \underline{cons}^{-1}\ N)'\ ((R\ \underline{cons}^{-1}\ N)')^{-1} \subseteq$
$R(\underline{cons}^{-1}\ N)'\ ((\underline{cons}^{-1}\ N)')^{-1} R^{-1}$
$\subseteq R\ \underline{nil}\ \underline{nil}^{-1}\ R \subseteq R$ using 1, 2, 6.2.4.3. (iii D (i))

ii $\iota\ (R\ \underline{cons}^{-1}\ E_1 R) \supseteq \iota(R\ \underline{cons}^{-1}\ E_1)$ using 1,4
$\supseteq R\ \iota\ (\underline{cons}^{-1}\ E_1)$, using 4.3.24,
$\supseteq N_\psi$ using 6.2.4.3 (iv)

iii $[R|E]\ \underline{cons}\ R\ \underline{cons}^{-1}\ E_2 = E_1\ R\ \underline{cons}^{-1}\ E_2 \cup E_2$
5, 6.2.4.3 (i)

iv $R\ \underline{cons}^{-1}\ [R|E]\ \underline{cons}\ R \subseteq R\ \underline{cons}^{-1}\ \underline{cons}\ R$ using 1,4
$\subseteq R$ using 6.2.4.3 (ii), 1.

v $\underset{\langle \ell, e\rangle \to o}{\underline{listin}} = (\underline{cons}\ N)' \subseteq ([R|E]\ \underline{cons}\ N)' \subseteq$
$[R|E]\ (\underline{cons}\ N)'$

6.2.4.3 (i)

$[R|E]\ (\underline{cons}\ N)' \subseteq [R|E]\ \underset{\langle \ell, e\rangle \to o}{\underline{listin}} \subseteq \underline{listin}$

6.2.4.3. (D (ii)), 4

Hence $([R|E]\ \underline{cons}\ N)' = \left(\underset{\langle \ell, e\rangle \to o}{R\ \underline{cons}^{-1}\ E_2}\right)$ (note the implicit conversion of the RHS to a relation of type $\langle \ell, e\rangle \to o$)

and hence from the representation theorem, that we have a representation in L_ψ of the language L_ϕ.

7.1.3.1 Properties of R.

We will first establish some properties of f.

i $f^{-1}\ f = E$

Proof.

$f^{-1} = \mu Y\ (\phi\ nil^{-1} \cup \underline{sub}\ [E_1\ Y, E_2]$

$\underline{cons})$

f^{-1} N = N from termination theorem and 6.2.1 (ii)

$\subseteq$) induction with $Xf^{-1} \subseteq E$

=) since E is single valued and f^{-1} is total.

ii f $\underline{in}$ = $\underline{listin}$

Induction on P (X, Y) = X $\underline{sub}^{-1}$ E_2 = Y

with $\mathcal{F}(X) = \phi\ \underline{nil}^{-1} \cup \underline{sub}\ [E_1\ X, E_2]\ \underline{cons}$

$\mathcal{G}(Y) = (\underline{cons}^{-1}\ E_1\ Y \cup E)\ \underline{cons}^{-1}\ E_2$

and using 6.2.1 (iii).

Now we establish the required properties of R

1 RR = $ff^{-1}\ ff^{-1} = ff^{-1}$ = R from (i)

2 $R = ff^{-1} = R^{-1}$

3 $E \subseteq ff^{-1}$ since f is total.

4 $\underline{cons}\ ff^{-1} = \underline{cons}\ (\underline{cons}^{-1}\ [f|E]\ \underline{sub}^{-1})\ f^{-1}$, fixpoint property of f.

$= [f|E]\ \underline{sub}^{-1}\ f^{-1}$

$= [f|E]\ \underline{sub}^{-1}\ \underline{sub}\ [f^{-1}\ E]\ \underline{cons}$,

fixpoint property of f^{-1}.

$\supseteq [f|E]\ [\underline{in}', E]\ [f^{-1}|E]\ \underline{cons}$

$\supseteq [f|E]\ [[f^{-1}|E]\ \underline{listin}', E]\ [f^{-1}|E]\ \underline{cons}$ using (ii)

$\supseteq [ff^{-1}|E]\ [\underline{listin}', E]\ \underline{cons}$

$\supseteq [ff^{-1}|E]\ \underline{cons}$ since we have lists with no repeated elements.

5 $R\ \underline{cons}^{-1}\ E_2 = ff^{-1}\ \underline{cons}^{-1}\ E_2$

$= f\ (\underline{sub}\ [f^{-1}|E]\ \underline{cons})\ \underline{cons}^{-1}\ E_2,$

fixpoint property of f

$= f\ \underline{sub}\ [f^{-1}|E]\ E_2$

$= f\ \underline{sub}\ E_2$ since f^{-1} is total.

$= f\ \underline{in}$, 6.2.1 D(ii)

$= \underline{listin}$ from (ii)

The mapping f is canonical, in that it maps equivalent lists to a unique representation.

7.1.4 Example of program representation

We showed in 7.1.3 that sets may be represented by lists with no repeated elements, and that R $\underline{cons}^{-1}$ [R|E] was a representation of $\underline{sub}$.

The following terms are good representations relative to ψ and R.

1 $\underset{\langle \ell, e \rangle \rightarrow o}{\underline{listin}}$ for $\underline{in}$ since [R|E] $\underline{listin}$ = $\underline{listin}$

2 $\underline{cons}$ R for $\underline{sub}^{-1}$ since [R|E] $\underline{cons}$ R = $\underline{cons}$ R

3 $\underline{nil}$ for ϕ since R $\underline{nil}$ = $\underline{nil}$

However R $\underline{cons}^{-1}$ [R|E] $\neq$ $\underline{cons}^{-1}$ [R|E]

If we consider the program $\underline{union}$ defined in 6.2.1 then I_ψ ($\underline{union}$ (Δ/E, $E_i\Delta|E_i, \Delta\ \underline{cons}^{-1}$[$\Delta$|E]/$\underline{sub}$)) is a representation of $\underline{union}$ in the language L_ψ. Let this be the program P, note that ϕ is redefined as I_ψ (Δ $\underline{nil}$).

$$P = \mu X\ ([R|R\ \underline{nil}] \cup [R|R\ \underline{cons}^{-1}\ [R|E]]\}\ [[E_1, E_2]$$

$$[R|R]\ \ X, E_3]\quad ([[R|E]\ \ \underline{listin}, R] \cup [R|E]\ \underline{cons}\ R)$$

if we define listunion to be:

$$\underline{listunion} = \mu Y\ ([E\ \underline{nil}] \cup [E\ \underline{cons}^{-1}]\ [\ [E_1, E_2]\ \ X, E_3]$$
$$([\underline{listin}, E_1] \cup \underline{cons}))$$

then using the fact that listin, cons, nil are good representations relative to L_ψ and R, it can be shown that

$$P = [E|R]\ \ \underline{listunion}\ R.$$

The original program union was single valued, furthermore listunion is total, hence

$$[E|R]\ \ \underline{listunion}\ R = \underline{listunion}\ R$$

and so the program listunion is a good representation of union relative to L_ψ and R.

Note that mechanical substitution did not take us all the way to the final program listunion. The program P was optimised as a separate process to produce listunion. Note also that this is a good representation of a program whose components were not all good representations.

The program listunion has the property that given any representations of two sets as lists, it produces a list which is a representation of the result of the union of the two original sets. This is usually what is desired.

PAGES 103 TO 105 HAVE BEEN INTENTIONALLY OMITTED

7.2 Simulation

We may have a representation for a domain which is sufficient for a particular program to work but which need not satisfy the axioms for the original domain. We state this formally by saying that a program **with** this representation simulates the original program, cf Milner [1971].

7.2.1 Simulation Theorem

Let f be a relation between the input domains of programs represented by $\mu X\ \mathcal{F}(X)$ and $\mu Y\ \mathcal{G}(Y)$, and let g be a relation between their output domains.

If $f\ X\ g^{-1} = Y \vdash f\ \mathcal{F}(X)\ g^{-1} = \mathcal{G}(Y)$
then $f\ \mu X\ \mathcal{F}(X)\ g^{-1} = \mu Y\ \mathcal{G}(Y)$
and we say that $f\ \mu X\ \mathcal{F}(X)\ g^{-1}$ simulates $\mu Y\ \mathcal{G}(Y)$.

This is easily extended to multiple recursions.

If $\{f_i X_i\ g_i^{-1} = Y_i \mid 1 \le i \le m\}$
$\vdash \{f_i\ \mathcal{F}_i(\underline{X})\ g_i^{-1} = \mathcal{G}_i(\underline{Y}) \mid 1 \le i \le m\}$
then $f_i\ \mu_i \underline{X}\ \mathcal{F}(\underline{X})\ g_i^{-1} = \mu_i \underline{Y}\ \mathcal{G}(\underline{Y}),\ 1 \le i \le m.$

Proofs are a straight forward application of Scott Induction.

7.2.2 Example

We can pursue the previous example of union and

listunion in a simulation style. Here however the form of listunion must be 'guessed' at rather than be mechanically produced by substitutions.

If f is again the function which maps from lists to sets, we can easily show, using the simulation theorem and results from the previous section that:

$$[f^{-1} | f^{-1}] \text{ listunion } f = \text{union}$$

and since f is total and f^{-1} i = E

listunion $\subseteq$ [f|f] union f^{-1}

ie $\langle\langle x, y\rangle, z\rangle \in$ listunion $\Rightarrow \langle\langle f(x), f(y)\rangle, f(z)\rangle \in$ union

Furthermore listunion is total, hence again listunion acts on any representation of two sets to produce a representation of their union.

8 CHANGES TO CONTROL STRUCTURE - RECURSION REMOVAL

8.1 Introduction

It may be most natural to pose a problem or an initial solution, in a recursive manner and then to develop from this a flowchart program augmented by stacks. A result from Paterson and Hewitt [1970] states that there exist recursive program schema which cannot be represented by flowchart schema. It follows that, in general, we must use flowchart schema augmented by stacks to simulate recursive program schema.

8.2 Labelled Stacks

Compilers usually handle recursion in the following manner, Dijkstra [1960]. When a procedure is called, link information is stacked which enables the calling program to continue when control is returned from the called procedure. This link information contains a 'return address' which tells us the point from which execution is to continue, and also contains a way of restoring the environment to that which was current at the time of the procedure call.

We formalise this by using labelled stacks. A labelled stack is a conventional stack whose elements are state vectors. Return addresses are not stacked, rather, this information is kept by giving each stack operation a label. There is a corresponding unstack operation for each label which is used both to restore the state vector, and to switch control to the appropriate place.

Any augmented flowchart schema will only use a fixed number, n, of labels. This may be determined statically. The labelled stacks are defined by the following axiom schema.

$$\begin{array}{l} \text{stack}_i \; \text{unstack}_j = E, \quad i = j \\ \langle v, s\rangle \to s, \; s \to \langle v, s\rangle = \Omega \quad i \neq j \end{array} \tag{1}$$

where v is a state vector and s is a stack.

We will also use a degenerate form of these stacks as to count. Here no information, other than the label, is put onto the stack.

$$\begin{array}{ll} \underline{inc_i}\ \underline{dec_j} & = E,\ i = j \\ 1\to 1\quad 1\to 1 & = \Omega, i \neq j \end{array} \qquad (2)$$

At any one time the stack can be viewed as a stack of coloured counters. Labels may be identified with colours, some counters will have information written on them, if they have been put there by a stack operation, and some will be blank, if they have been put there by an inc operation.

We will use the following abbreviations:

$stack_i$ $unstack_i$ to s_i u_i

inc_i dec_i to i_i d_i.

8.3 Informal Introduction to the General Theorem

To introduce the general theorem we will first study two examples.

Example 1: Given a recursive schema represented by $f = \mu X (A \cup B \times C \times D)$, we can identify the relation A with the idea of a return instruction, ie that its invocation tells us that an evaluation of f has finished. The subterm B, commits us to the evaluation of the remainder of this term, which includes recursive calls to f, and again the final subterm D can be associated with a return. We can produce a pair of flowchart schema, the first of which calculates f, by either returning, having evaluated A, or by applying B, stacking a return address, and then invoking itself again. The second schema calculates the remainders of terms by inspecting markers on the stack and then using them to switch to evaluation of the appropriate remainders. These remainders too may involve recursive calls to f, and so markers may be stacked and control passed back to the first schema.

We produce terms:

$$(\tau)_\alpha = [A|E]\ Z \cup [B|i_1]\ Y$$

$$(\tau)_\beta = [E|d_0] \cup [E|d_1]\,[C|i_2]\,Y \cup [E|d_2]\,[D|E]\,Z$$

and define $f_\alpha = \mu_1\, YZ\, ((\tau)_\alpha, (\tau)_\beta)$

$f_\beta = \mu_2\, YZ\, ((\tau)_\alpha, (\tau)_\beta)$

The schemas define 2 → 2 relations. The first component of their state vector is the argument, and the second is a stack of markers. These schemas are related to the original schema by the theorem of 8.4.4 as follows:

$$[f|E]\, f_\beta = f_\alpha$$

Clearly $[E|i_0]\, f_\beta = E$, and so we obtain the following equality

$f = [E|\, i_0\,]\, f_\alpha$ where $\overset{i_0}{\circ\!\!\rightarrow\!\!1}$ produces a stack initialised to i_0.

Example 2: We study the schema corresponding to a tree traversal program.

$f = \mu X\, (A \cup [BX, CX]\, D)$

The concatenation operation, [,], is dealt with as follows. We arbitrarily decide to evaluate the left subterm first, and then the right subterm, which must be evaluated with the same argument as the left term. The sequence of operations to be carried out, together with the corresponding subterms is:

1. Stack the argument, $[E_1, s_1]$
2. Evaluate the left subterm which includes a recursive call to f, $[B|E]\, Y$
3. Unstack the argument and stack the result of the left subterm, $[E|u_1][E_2,[E_1,E_3]\, s_2]$
4. Evaluate the right subterm, $[C|E]\, Y$
5. Unstack the result of the left subterm and form the result vector, $[E|u_2][[E_2,E_1], E_3]$
6. Apply the remaining term, $[D|E]$

7 Evaluate the remaining stack, Z.

The resulting schema are:

$$(\tau)_\alpha = [A|E]\ Z \cup [E_1, S_1]\ [B|E]\ Y$$

$$(\tau)_\beta = [E|d_0] \cup [E|u_1]\ [E_2, [E_1, E_3]\ s_2]\ [C|E]\ Y \cup [E|u_2]\ [[E_2, E_1], E_3]\ [D|E]\ Z$$

and the theorem relating these to the original schema is again

$$[f|E]\ f_\beta = f_\alpha$$

8.4 The General Theorem

8.4.1 Unique Labels

The only problem remaining before embarking on the general theorem is that of ensuring uniqueness of labels. In general, we will consider terms τ of the form $\tau = \underset{i=1}{\overset{m}{\cup}} \tau_i$, where the terms τ_i are free from the union operation. The index i will uniquely identify each subterm. Within each subterm τ_i, we give each matching pair of [ ] brackets a unique 'block' number b, written as $[^b$, $]$ and give each of its subterms a further index x of value 0 for the left subterm, and 1 for the right subterm. Each occurrence of X within these subterms at the same block level is then given a fourth index y in turn. Two indices i, b thus serve to uniquely identify each concatenation operation, and four indices, i, b, x, y, identify each occurrence of X. This need to ensure unique labelling is the main reason why the following algorithms to derive terms $(\tau)_\alpha$ and $(\tau)_\beta$ initially look rather complex. The stack, unstack, increment and decrement operations will have unique labels depending on the above indices.

8.4.2 Definitions

Simplicity: A definition of the simplicity of a term in X was given in the section dealing with termination. We

find it more convenient to use the following equivalent definition.

A term τ is simple in a relation variable X if either

i τ contains no free occurrences of X

or

ii $\tau \equiv \tau_1 X$

or

iii $\tau \equiv \tau_1 X \tau_2$

or

iv $\tau \equiv [\tau_3, \tau_4] \tau_5$

or

v $\tau \equiv \tau_3 \cup \tau_4$

where τ_1 contains no free occurrences of X and τ_2, τ_3, τ_4, τ_5 are terms simple in X. There may be an implicit use of E to obtain terms in this form. This definition is ambiguous in the sense that a term may have more than one form eg $AX \equiv AXE$ and so is of the form ii or iii. This ambiguity is deliberately introduced to avoid unnecessary inefficiency in the derived schema, the associated algorithms are expressed in terms of conditional expressions, and so will act on the first permissible form.

Union Normal Form:

Any simple term τ can be written in union normal form as $\tau = \bigcup_{i=1}^{m} \tau_i$ where the terms τ_i do not contain unions, except possibly in terms not containing X free.

Derivatives:

τ_α is the α-resultant of a term τ where τ is simple in X, and is expressed in union normal form, if

$\tau_\alpha = \underset{i=1}{\overset{m}{\cup}} (\tau_i)_\alpha$

$(\tau_i)_\alpha = \alpha\ (\tau_i, i, 0, 0, 1)$ and (3)

$\alpha\ (\tau, i, b, x, y)$ = if τ contains no free occurrences of X then $[\tau|E]\ Z$

if $\tau = \tau_1\ X$ then $[\tau_1|E]\ Y$

if $\tau = \tau_1\ X\ \tau_2$ then $[\tau_1|i_{ibxy}]\ Y$

if $\tau = [^j\tau_3, \tau_4]\ \tau_5$ then $[E_1, s_{ij0}]\ \alpha\ (\tau_3, i, j, 0, 1)$ where τ_1 does not contain X free.

τ_β is the β resultant of a term τ, where τ is simple in X and is expressed in union normal form, if:

$\tau_\beta = [E|d_0] \cup \underset{i=1}{\overset{m}{\cup}} (\tau_i)_\beta$

$(\tau_i)_\beta = \beta\ (\tau_i, i, 0, 0, 1)$ and (4)

$\beta\ (\tau, i, b, x, y) =$

if τ contains no free occurrences of X then Ω

if $\tau = \tau_1\ X$ then Ω

if $\tau = \tau_1\ X\ \tau_2$ then $[E|d_{ibxy}]\ \alpha\ (\tau_2, i, b, x, y + 1) \cup \beta\ (\tau_2, i, b, x, y + 1)$

if $\tau = [^j\tau_3, \tau_4]\ \tau_5$ then $\beta\ (\tau_3, i, j, 0, 1) \cup [E|u_{ij0}]\ [E_2, [E_1, E_3]\ s_{ij1}]\ \alpha\ (\tau_4, i, j, 1, 1) \cup \beta\ (\tau_4, i, j, 1, 1) \cup [E|u_{ij1}]\ [[E_2, E_1], E_3]\ \alpha\ (\tau_5, i, b, x, y) \cup \beta\ (\tau_5, i, b, x, y)$

Example:

if $\tau = A \cup [^1BX, CX]\ D$ then

$\tau_\alpha = [A|E]\ Z \cup [E_1, s_{210}]\ [B|E]\ Y$

$\tau_\beta = [E|d_0] \cup [E|u_{210}]\ [E_2, [E_1, E_3]\ s_{211}]\ [C|E]\ Y \cup [E|u_{211}]\ [[E_2, E_1], E_3]\ [D|E]\ Z.$

Derived relations

Let $f_\alpha = \mu_1\ YZ\ (\tau_\alpha, \tau_\beta)$

$f_\beta = \mu_2\ YZ\ (\tau_\alpha, \tau_\beta)$

where $f = \mu X\ (\tau\ (X))$.

8.4.3 Lemmas

Definition

Let $f^{\wedge}_{\beta} = \mu\hat{Z}\ (\tau^{\wedge}_{\beta}\ (\hat{Z}))$

where $\tau^{\wedge}_{\beta} = \tau_{\beta}\ ([f|E]\ \hat{Z}/Y,\ \hat{Z}/Z)$

Abbreviation: Let $\mathring{\alpha}\ (\tau, i, b, x, y) = \alpha\ (\tau, i, b, x, y)\ ([f|E]\ f^{\wedge}_{\beta}/Y,\ f^{\wedge}_{\beta}/Z)$ and similarly $\mathring{\beta}$.

Lemma 1

From the indexing system used for labels and the definitions of α and β, there can only be at most one term σ commencing with $[E|u_{ijk}]$ or $[E|d_{ibxy}]$ in $\tau_{\hat{\beta}}$.

Lemma 2

$\mathring{\beta}\ (\tau, i, b, x, y) \subseteq f^{\wedge}_{\beta} \Rightarrow$

$\mathring{\alpha}(\tau, i, b, x, y) = [\tau\ (f/X)\ |E]\ f^{\wedge}_{\beta}$.

Proof:

The proof is by induction on the formation rules for union free simple terms.

i if τ contains no free occurrences of X

$\mathring{\beta}\ (\tau, i, b, x, y) = \Omega \subseteq f^{\wedge}_{\beta}$

$\mathring{\alpha}\ (\tau, i, b, x, y) = [\tau|E]\ f^{\wedge}_{\beta}$

ii if $\tau = \tau_1\ X$ where τ_1 contains no free occurrence of X

$\mathring{\beta}\ (\tau, i, b, x, y) = \Omega \subseteq f^{\wedge}_{\beta}$

$\mathring{\alpha}\ (\tau, i, b, x, y) = [\tau_1|E]\ [f|E]\ f^{\wedge}_{\beta} = [\tau(f/X)\ E]\ f^{\wedge}_{\beta}$.

iii if $\tau = \tau_1\ X\ \tau_2$ where τ_1 contains no free occurrences of X.

$\mathring{\beta}\ (\tau, i, b, x, y) = [E|d_{ibxy}]\ \mathring{\alpha}\ (\tau_2, i, b, x, y + 1) \cup \mathring{\beta}\ (\tau_2, i, b, x, y + 1) \subseteq f^{\wedge}_{\beta}$ (Given)

Hence $\mathring{\alpha}(\tau_2, i, b, x, y+1) =$
$[\tau_2(f/X)|E]\, f_{\hat{\beta}}$, induction hypothesis (5)

Also from lemma 1.
$\rho \cup [E|d_{ibxy}]\, \mathring{\alpha}(\tau_2, i, b, x, y+1) = f_{\hat{\beta}}$ (6)

where ρ contains no terms starting with $[E|d_{ibxy}]$

$\mathring{\alpha}(\tau, i, b, x, y) = [\tau_1|i_{ibxy}]\, [f|E]\, f_{\hat{\beta}}$
$= [\tau_1\, f|E]\, [\tau_2\, (f/X)|E]\, f_{\hat{\beta}}$ from (2), (5), (6)
$= [\tau\, (f/X)|E]\, f_{\hat{\beta}}$.

iv if $\tau = [{}^{j}\tau_3, \tau_4]\, \tau_5$ then
$\mathring{\beta}(\tau, i, b, x, y) = \mathring{\beta}(\tau_3, i, j, 0, 1) \cup [E|u_{ij0}]\, [E_2, [E_1, E_3]s_{ij1}]\, \mathring{\alpha}(\tau_4, i, j, 1, 1) \cup \mathring{\beta}(\tau_4, i, j, 1, 1) \cup [E|u_{ij1}]\, [[E_2, E_1], E_3]\, \mathring{\alpha}(\tau_5, i, b, x, y) \cup \mathring{\beta}(\tau_5, i, b, x, y)$.

$\subseteq f_{\hat{\beta}}$ (Given)

Hence from the induction hypothesis.
$\mathring{\alpha}(\tau_3, i, j, 0, 1) = [\tau_3\, (f/X)|E]\, f_{\hat{\beta}}$ (7)
$\mathring{\alpha}(\tau_4, i, j, 1, 1) = [\tau_4\, (f/X)|E]\, f_{\hat{\beta}}$ (8)
$\mathring{\alpha}(\tau_5, i, b, x, y) = [\tau_5\, (f/X)|E]\, f_{\hat{\beta}}$ (9)
$\mathring{\alpha}(\tau, i, b, x, y) = [E_1, s_{ij0}]\, \mathring{\alpha}(\tau_3, i, j, 0, 1)$
and again by use of lemma 1 we can show that:

$\mathring{\alpha}(\tau, i, b, x, y) = [E_1, s_{ij0}]\, [\tau_3\, (f/X)|E]\, f_{\hat{\beta}}$ from (7)
$= [\tau_3\, (f/X), s_{ij0}]\, [E|u_{ij0}]\, [E_2, [E_1, E_3]\, s_{ij1}]\, [\tau_4\, (f/X)|E]\, f_{\hat{\beta}}$
(lemma 1, and (8))
$= [\tau_3\, (f/X), s_{ij0}]\, [E|u_{ij0}]\, [E_2, [E_1, E_3]s_{ij1}]\, [\tau_4\, (f/X)|E]\, [E|u_{ij1}]\, [[E_2, E_1], E_3]\, [\tau_5\, (f/X)|E]\, f_{\hat{\beta}}$
(lemma 1, and (9))

$= [[\tau_3\ (f/X),\ \tau_4\ (f/X)]\ \tau_5|E]\ f_\beta^\wedge$
(using (1))

Hence the lemma is true for all union free simple terms.

Lemma 3

$\beta\ (\tau, i, b, x, y)\ ([X|E]\ f_\beta/Y,\ f_\beta/Z) \trianglelefteq f_\beta \Rightarrow \alpha\ (\tau, i, b, x, y)\ ([X|E]\ f_\beta/Y,\ f_\beta/Z) = [\tau|E]\ f_\beta$.

Proof: The proof, not given here, proceeds by induction on the formation rules for simple terms in essentially the same manner to the proof of Lemma 2.

8.4.4 Theorem

Let $f = \mu X\ (\tau\ (X))$ where τ is simple in X.

Let $f_\alpha = \mu_1\ YZ\ (\tau_\alpha, \tau_\beta)$
$f_\beta = \mu_2\ YZ\ (\tau_\alpha, \tau_\beta)$
then $[f|E]\ f_\beta = f_\alpha$. (10)

Proof:

We actually prove the following.
$[f|E]\ f_\beta^\wedge = f_\alpha$
$f_\beta^\wedge = f_\beta$

$\trianglerighteq$) The proof is by fixpoint induction
$f_\beta\ ([f|E]\ f_\beta^\wedge/Y,\ f_\beta^\wedge/Z) = \underset{i=1}{\overset{m}{\sqcup}}\ \overset{\circ}{\beta}\ (\tau, i, 0, 0, 1)$
from (4) and defn of $f_\beta^\wedge$.
$= f_\beta^\wedge$ (11)
$f_\alpha\ ([f|E]\ f_\beta^\wedge/Y,\ f_\beta^\wedge/Z) = \underset{i=1}{\overset{m}{\sqcup}}\ \overset{\circ}{\alpha}\ (\tau, i, 0, 0, 1)$
from (3)
$= \underset{i=1}{\overset{m}{\sqcup}}\ \lfloor \tau\ (f/X)|E \rfloor\ f_\beta^\wedge$ from lemma 2 and (11)
$= [f|E]\ f_\beta^\wedge$

Hence $f_\beta \trianglelefteq f_\beta^\wedge$
$f_\alpha \trianglelefteq [f|E]\ f_\beta^\wedge$.

$\subseteq$) We first show by Scott Induction that
$[f|E]\ f_\beta \subseteq f_\alpha$. (12)

Let $P(X) \equiv [X|E]\ f_\beta \subseteq f_\alpha$

a $P(\Omega)$ is true

b Assume $[X|E]\ f_\beta \subseteq f_\alpha$

$f_\beta = \underset{i}{\overset{m}{\underline{\cup}}}_1\ \beta\ (\tau_i, i, 0, 0, 1)\ (f_\alpha/Y, f_\beta/Z)$
(from (4))

From the assumption and monotonicity of terms produced by β.
$\underset{i}{\overset{m}{\underline{\cup}}}_1\ \beta\ (\tau_i, i, 0, 0, 1)\ ([X|E]\ f_\beta/Y, f_\beta/Z) \subseteq f_\beta$

Hence, using lemma 3
$\underset{i}{\overset{m}{\underline{\cup}}}_1\ \alpha\ (\tau_i, i, 0, 0, 1)\ ([X|E]\ f_\beta/Y, f_\beta/Z) = [\tau|E]\ f_\beta$
but $\underset{i}{\overset{m}{\underline{\cup}}}_1\ \alpha\ (\tau_i, i, 0, 0, 1)\ ([X|E]\ f_\beta/Y, f_\beta/Z)$
$\subseteq \underset{i}{\overset{m}{\underline{\cup}}}_1\ \alpha\ (\tau_i, i, 0, 0, 1)\ (f_\alpha/Y, f_\beta/Z)$ from inductive assumption and monotonicity of terms produced by α.

$= f_\alpha$ (from (3))

c) Hence $[f|E]\ f_\beta \subseteq f_\alpha$ by Scott Induction.

We can now use this result to show by fixpoint induction that

$\hat{f_\beta} \subseteq f_\beta$.
$\hat{f_\beta}\ (f_\beta/Z) = f_\beta\ ([f|E]\ f_\beta/Y, f_\beta/Z)$
$\subseteq f_\beta\ (f_\alpha/Y, f_\beta/Z)$ (using (12))
$\subseteq f_\beta$.

Hence by fixpoint induction $\hat{f_\beta} \subseteq f_\beta$.

8.4.5 Intended Use of the Theorem

The intended use of the theorem is for the initial term $[E|d_0]$ in the derived relation σ_β of 8.5.2 to be the test for the empty stack, and the corresponding operation $0 \overset{i}{\rightarrow} 1$ to create the empty stack. Clearly then

$f = [f|_0 \overset{E}{\rightarrow} 0] = [E|\underline{\text{empty}}^{-1}]\, \bar{f}_\alpha\, E_1$

8.5 Extension to Multiple Recursions

8.5.1 Introduction

The general theorem of 8.4 extends easily to multiple recursions. From each equation schema we derive a flowchart schema augmented by a stack, and from all the equation schema we derive a single augmented flowchart schema which evaluates the stack and so handles the flow of control. The only change we need to make is to include a further index which identifies the equation in which a concatenation block, or a reference to some X_i occurs.

8.5.2 Definitions

Consider a set of mutually recursive equation schema with solutions given by $f_i = \mu_i X_1 \ldots X_n\, (\sigma_1 \ldots \sigma_n)$, $1 \leq i \leq n$, where the terms σ_1 are simple in each X_i, $1 \leq i \leq n$.

Derivatives

$(\sigma_i)_\alpha$ is the α resultant of a term σ_i where σ_i is simple in $X_1 \ldots X_n$, and is in union normal form, if:

$$(\sigma_i)_\alpha = \bigcup_{j=1}^{m_i} (\tau_{ij})_\alpha$$

$(\tau_{ij})_\alpha = \alpha\,(\tau_{ij}, i, j, 0, 0, 1)$ and

$\alpha\,(\tau, i, j, b, x, y)$ = if τ contains no free occurrences of any X_k then $[\tau|E]$ Z

if $\tau = \tau_1 X_k$ then $[\tau_1|E]$ Y_k

if $\tau = \tau_1 X_k \tau_2$ then $[\tau_1 | i_{ijbxy}] Y_k$

if $\tau = [^{\ell}\tau_3, \tau_4] \tau_5$ then $[E_1, s_{ij\ell o}] \alpha (\tau_3, i, j, \ell, 0, 1)$.

where τ_1 does not contain X free.

$(\sigma_i)_\beta$ is the β-resultant of a term σ_i if:

$(\sigma_i)_\beta = \underset{j=1}{\overset{m_i}{U}} (\tau_{ij})_\beta$

$(\tau_{ij})_\beta = \beta (\tau_{ij}, i, j, 0, 0, 1)$ and

$\beta (\tau, i, j, b, x, y)$ = if τ contains no free occurrences of any $_k$ then Ω

if $\tau = \tau_1 X_k$ then Ω

if $\tau = \tau_1 X_k \tau_2$ then $[E|d_{ijbxy}] \alpha (\tau_2, i, j, b, x, y + 1) \cup \beta (\tau_2, i, j, b, x, y + 1)$.

if $\tau = [^{\ell}\tau_3, \tau_4] \tau_5$ then

$\beta (\tau_3, i, j, \ell, 0, 1) \cup [E|U_{ij\ell o}] [E_2, [E_1, E_3] s_{ij\ell 1}] \alpha (\tau_4, i, j, \ell, 1, 1) \cup \beta (\tau_4, i, j, \ell, 1, 1) \cup [E|u_{ij\ell 1}] [[E_2, E_1], E_3] \alpha (\tau_5, i, j, b, x, y) \cup \beta (\tau_5, i, j, b, x, y)$.

Derived relations

Define $\sigma_\beta = [E|d_o] \cup \underset{i=1}{\overset{n}{U}} (\sigma_i)_\beta$

Let $(f_i)_\alpha = \mu_1 Y_1 \ldots Y_n Z ((\sigma_1)_\alpha, \ldots (\sigma_n)_\alpha, \sigma_\beta)$, $1 \leq 1 \leq n$

and $f_\beta = \mu_{n+1} Y_1 \ldots Y_n Z ((\sigma_1)_\alpha, \ldots (\sigma_n)_\alpha, \sigma_\beta)$

where $f_i = \mu_i X_1 \ldots X_n (\sigma_1, \ldots \sigma_n)$. $1 \leq i \leq n$.

8.5.3 Lemmas

Lemma 1. Clearly there is still at most one term σ commencing with $[E|u_{ij\ell k}]$ or $[E/d_{ijbxy}]$ in (σ_β), hence $\sigma \subseteq f_\beta \Rightarrow \rho \cup \sigma = f_\beta$ where ρ contains no terms commencing with $[E|u_{ij\ell k}]$ or $[E|d_{ijbxy}]$.

Lemma 2

Let $f\hat{}_\beta = \mu\hat{z} (\sigma\hat{}_\beta (\hat{z}))$

where $\sigma_\beta^\wedge = \sigma_\beta\ (\{[f_i\,|E]\ f_\beta^\wedge/Y_i\,|1 \le i \le n\},\ f_\beta^\wedge/Z)$
then $\beta\ (\tau, i, j, b, x, y)\ (\{[f_i\,|E]\ f_\beta^\wedge/Y_i\,|1 \le i \le n\}, f_\beta^\wedge/Z\ )$
$\subseteq f_\beta \quad => \quad \alpha\ (\tau, i, j, b, x, y)\ (\{[f_i\,|E]\ f_\beta^\wedge/Y_i\,|1 \le i \le n\},\ f_\beta^\wedge/Z) = [\tau\ (\{f_i/X_i\}\,|1 \le i \le n\})\,|E]\ f_\beta^\wedge$

The proof is similar that of lemma 2 in section 7.4.3 and proceeds by induction on the formation rules for simple terms.

Lemma 3

$\beta\ (\tau, i, j, b, x, y)\ (\{[X_i\,|E]\ f_\beta/Y_i\,|1 \le i \le n\},\ f_\beta/Z)$
$\subseteq f_\beta => \alpha\ (\tau, i, j, b, x, y)\ (\{[X_i\,|E]\ f_\beta/Y_i\,|1 \le i \le n\},\ f_\beta/Z) = [\tau/E]\ f_\beta.$

Again the proof is straightforward by induction on the formation rules for simple terms.

8.5.4 Theorems

Let $f_i = \mu_i\ X_1 \ldots X_n\ (\sigma_1, \ldots \sigma_n), 1 \le i \le n$ be solutions to a set of recursion equations, where the terms $\sigma_1 \ldots \sigma_n$ are simple in all X_k, $1 \le k \le n$.

Let $(f_i)_\alpha = \mu_i\ Y_1 \ldots Y_n\ Z\ ((\sigma_1)_\alpha \ldots (\sigma_n)_\alpha, \sigma_\beta)\ 1 \le i \le n$
and $f_\beta = \mu_{n+1}\ Y_1 \ldots Y_n\ Z\ ((\sigma_1)_\alpha \ldots (\sigma_n)_\alpha, \sigma_\beta)$
then $[f_i\,|E]\ f_\beta = (f_i)_\alpha, 1 \le i \le n$
where $(\sigma_1)_\alpha \ldots (\sigma_n)_\alpha$ and σ_β are augmented flowchart schemas.

Proof:

The proof will not be given here, but proceeds in essentially the same manner as the proof of 7.4.4, by actually showing that:

$[f_i\,|E]\ f_\beta^\wedge = (f_i)_\alpha, 1 \le i \le n$

$f_\beta^\wedge = f_\beta$

8.6 Example

8.6.1 Tree Traversal

We wish to produce a string from a binary tree by traversing its terminal nodes from left to right and concatenating them together in order. Our problem is stated in a recursive form, and our target language does not contain recursion.

traverse (x) = if is-atom (x) then x else traverse (car (x)) $^{\frown}$ traverse (cdr (x)).

where $^{\frown}$ is the associative operation concatenate.

We abstract this recursive form to a schema:

$T = \mu X (A \cup [BX , CX] D)$ and apply the theorem of 8.4.4 to produce, after simplification of labels, the following flowchart schema.

$[T|E]\ \mu_2 YZ = \mu_1 YZ$

where $\mu_i YZ \equiv \mu_i YZ\ ([A|E]\ Z \cup [E_1\ B, s_2]\ Y, [E|d_1] \cup [E|u_2]\ [E_2\ C, [E_1, E_3]\ s_3]\ Y \cup [E|u_3]\ [[E_2, E_1]\ D|E]\ Z)$.

This can be further simplified. The operation D is associative and so we can keep a 'result so far', rather than stacking intermediate results.

Lemma.

if D is associative ie $[E \mid [F, G]\ D]\ D = [[E|F]\ D, G]D$

and $\mu X \equiv \mu X(A \cup [BX, CX]\ D)$ and

$\mu_i YZ \equiv \mu_i YZ\ ([E|A|E]\ [D|E]\ Z \cup [E|\ [E_1\ B, s_2]]Y, [E|E|d_1] \cup [E|E|u_2]\ [E|C|E]\ Y)$

then $[E|\mu X|E]\ [D|E]\ \mu_2 YZ = \mu_1 YZ$.

The proof is straight forward.

i Define $\hat{\mu Z}$ as before

ii Show that $[E|\mu X|E]\ [D|E]\ \hat{\mu Z} = \mu_1 YZ$

$\hat{\mu Z} = \mu_2 Y \hat{Z}$

iii $\subseteq$) by fixpoint induction

$\supseteq$) Scott Induction to show that $[E|\mu X|E]\ [D|E]\ \mu_2\ Y\hat{Z} \subseteq \mu_1 Y\hat{Z}$ and using this to show $\hat{\mu Z} \subseteq \mu_2 Y\hat{Z}$ by fixpoint induction.

We can now return to the original interpretation, and by noting that concatenation has an identity element, ie $nil^{\frown}x = x$, and that we can think of the marker detected by d_1 as the empty stack, we can produce the usual form of a tree traversal program. Knuth [1968] 2.3.1 p 317.

T (x) = Tr (nil, x, empty)
Tr (S, x, k) = if is-atom (x) then if is-empty (k)
then $S^{\frown}x$, else Tr ($S^{\frown}x$, cdr (hd (k)), tl (k))
else Tr (S, car (x), stack (x, k))
where hd = unstack E_1
tl = unstack E_2.

Further development of the program would now take place by finding a more machine oriented representation for trees and stacks.

8.6.2 Factorial

We can use the theorem of 8.4.4 to gain insight into an iterative form of the factorial program.

Let $F = \mu X(A \cup [B\ X, E]\ C)$ with the interpretation.
$A = \{<0, 1>\}$, $B = \{<a, b> | a > 0 \ \&\ b = a - 1\}$
$C = \{<<a_1, a_2>, b> | a_1, a_2 \geq 0 \ \&\ b = a_1 * a_2\}$

It is easily shown that F is total and correctly computes factorial.

Using 8.4.4 and 8.5.5 we obtain the following iterative form of F

$F = [E|i_o]\ \mu Y\ ([A|E] \cup [E_1\ B,\ s_2]\ Y)$
$\mu Z\ ([E|d_o] \cup [E|u_2].[C|E]\ Z)$

The next level in the development is the actual representation of the stack and unstack operations.

We choose to represent the stack by two integers, a_2 which is the value of its top element, and a_3 which is a marker for i_o (This is only possible here because in this particular case a preceding element on the stack can be obtained by knowing the top one).

The stack operations are given by.
$S_2 = \{<<a_1, a_2, a_3>, <b_2, b_3>|a_1 \geq 0 \ \&\ a_2 = a_1 + 1 \ \&$
$a_3 \geq a_2 \ \&\ b_2 = a_1 \ \&\ b_3 = a_3\}$
$u_2 = s_2$
$i_o = \{<a_2, a_3>|a_2 = a_3\}$

Assuming that the domain of the stack operations is given by $\underline{\text{is-valid-stack-op}} = \{<a_1, a_2, a_3>|a_1 \geq 0 \ \&\ a_2 = a_1 + 1 \ \&\ a_3 \geq a_2\}$
then the axioms for stacks are satisfied.
$s_2\ u_2 = E$
$u_2\ s_2 \subseteq E$
$s_2\ d_o = \Omega$
$i_o\ u_2 = \Omega$
This assumption holds provided that the input satisfies $\underline{\text{is-valid-stack}}$, since $[\underline{\text{is-valid-stack-op}}, E]$ $[E_1\ B, S_2] = [E_1\ B, S_2]\ [\underline{\text{is-valid-stack-op}}, E]$.

This gives the following program for F (x), which we will write in the more familiar functional form.
F (x) = Z (Y (x, x + 1, x + 1))
where Y (x, y, z) = if x = o then <1, y, z>
else Y (x - 1, y - 1, z)
Z (x, y, z) = if y = z then x
else Z (x * y, y + 1, z)

Clearly the function Y (x, x + 1, x + 1) always has

the result <1, 1, x + 1> and so
F (x) = Ƶ (1, 1, x + 1) which is a familiar iterative form for factorial.

9 CONCLUSIONS

The motivation for this thesis was to take an existing formalism, the relational calculus, and to explore its application to formal reasoning about programs, in particular that reasoning necessary to justify some techniques used in the stepwise development of programs. The relational calculus was a good tool with which to do this, providing a common framework in which to reason about the many facets of program proofs.

The development of a program starts with its specification as a relation between input and output values. Development proceeds by specifying a schema and subsidiary relations as its interpretation, this forms an initial solution to the specification. We showed in 3.3 examples of proofs of partial correctness of schemas. A proof of termination is needed to establish total correctness, this requires an induction rule on the domain of interpretation, which is related to the program schema by the derivatives of chapter 5. This is made straightforward because the relational calculus can describe induction rules and schemas in the same language. Having shown the correctness of this initial solution, the process is repeated for each of the subsidiary relations until a schema is obtained whose interpretation is related immediately to the target programming language.

In parallel with this refinement of control and function is a process of refining the data structures of the domains of interpretation, until we arrive at acceptable structures in the target programming language. Chapter 6 gives axioms for many commonly occuring data structures and chapter 7 shows, with an example in 7.1.3, how we can change an interpretation based on a certain data structure to another based on representations of the original data structure in the language of the new one.

We have also shown in Chapter 8 how to mechanically transform a recursive program schema to a set of schemas which are not recursive.

This justifies the technique of choosing a recursive schema as an initial solution to a problem and later refining it into a program which does not use recursion.

Although the relational calculus provides a convenient metalanguage in which to work, it is clumsy in actual application and we see the main use of the presented theorems being the justification of less opaque versions of them. There is a need to develop the relational calculus into a language with named selectors, rather than positional ones and with more familiar programming constructs than the relational constructs used here.

High level languages have many constructs, subroutines, loops, macros etc., which aid in the abstraction of operations and the flow of control, but few which allow the abstraction of data structures and which separate them from a particular representation, and we foresee a need for language development in this area.

We have left several areas unexplored. We have not attempted to formalise an important transition in program development, that from a non-deterministic form to a deterministic one involving back-tracking, Floyd [1967b] , we have not tried to apply the formalism to proofs about parallel programs nor have we tried to extend the formalism to deal with such constructs as functions of higher types, call by name parameter mechanisms or dynamic changes to control or data structures.

We foresee the development of interactive systems to aid program development , calling upon theorems presented above to aid in the justification of certain steps, or in some cases to mechanically carry out appropriate substitutions, derivations etc., and ultimately limited program synthesis.

10 REFERENCES

de Bakker J W (1971) Recursive procedures, Mathematical Centre Tracts 24, Mathematical Centre, Amsterdam.

de Bakker J W, and de Roever W P (1972) A calculus for recursive program schemes
Automata, Languages and Programming pp 167-195 (ed M Nivat) North Holland/American Elsevier.

Burstall R (1969) Proving Properties of Programs by Structural Induction,
Comp Journal 12, pp 41-48

Conway J H (1971) Regular Algebra and Finite Machines, Chapman and Hall.

Cooper D C (1969) Program scheme equivalence and second-order logic.
Machine Intelligence, Vol 4 pp 3-15 (eds B Meltzer and D Michie) Edinburgh University Press.

Darlington J and Burstall R M (1973) A System which automatically Improves programs. Experimental Programming Report No. 28.
School of Artificial Intelligence
University of Edinburgh.

Djikstra E W (1960) Recursive Programming
Num Math 2 pp 312-318

Djikstra E W (1969) Notes on Structured Programming
Report EWD 249. Technische Hogeschool, Eindhoven, Netherlands.

Floyd R (1967a), Assigning Meanings to Programs.
Proc Sym in Applied Math 19
Mathematical Aspects of Computer Science (Schwartz J T ed)
Amer Math Soc pp 19-32

Floyd R (1967b) Non-deterministic algorithms
JACM 14 pp 636-644.

Hitchcock P and Park D M R (1972) Induction Rules and termination proofs.
Automata, Languages and Programming pp 225-251 (ed M Nivat)
North Holland/American Elsevier.

Hoare C A R (1969) An Axiomatic Basis for Computer Programming
CACM 12 pp 576-583.

Hoare C A R (1971a) Proof of a Program: FIND
CACM, vol 14, no 1, pp. 39 - 45

Hoare C A R (1971b) Procedures and Parameters: An axiomatic approach, Symposium on Semantics of Algorithmic Languages (ed Engeler)
Springer-Verlag Lecture Notes in Mathematics 188.

Hoare C A R (1972a) An Axiomatic Definition of the Programming Language PASCAL - Second Draft. (unpublished notes)

Hoare C A R (1972b), Dijkstra E W, Dahl O - J, Structured Programming
Academic Press. New York 1972.

Jones C B (1972) Formal Development of Correct Algorithms: an Example Based on Early's Recogniser. in Proc. of an ACM Conference on Proving Assertions about Program, Las Cruces, New Mexico, Jan. 6/7, 1972, pp. 150 - 169.

Knuth D E (1968) The Art of Computer Programming vol 1 Fundamental Algorithms Addison-Wesley.

Keisler (1971) Model Theory for Infinitary Logic, ch 10
North Holland Publishing Company.

McCarthy J (1962) Towards a Mathematical Science of Computation
Proc IFIP Conference 1962
North-Holland.

Manna Z and Pnueli A, (1970) Formalisation of properties of functional programs, J ACM 17, pp 555-569.

Manna Z and Waldinger R J (1971) Towards Automatic Program Synthesis
CACM 14 no 3 pp 151-165.

Milner R (1971) An Algebraic Definition of Simulation Between Programs
Second International Joint Conference on Artificial Intelligence pp. 481 - 489
British Computer Society.

Milner R [1972] Implementation and Applications of Scott's Logic for computable Functions. in Proc. of an ACM Conference on Proving Assertions about Programs, Las Cruces, New Mexico, Jan. 6/7, 1972, pp. 1 - 6.

Park D M R (1970) Fixpoint induction and proofs of program semantics, Machine Intelligence, Vol 5 pp 59-78, (eds B Meltzer, D Michie)
Edinburgh University Press.

Paterson M S and Hewitt C E (1970) Comparative Schematology
Project MAC Conference on Concurrent Systems and Parallel Computation.
ACM pp 119-128.

de Roever W P (1973) Operational and Mathematical Semantics for Recursive Polyadic Program Schemata.
Mathematical Centre Report
Amsterdam (to appear).

Scott D, and de Bakker J W, (1969) A theory of programs, unpublished notes, IBM Seminar, Vienna.

Scott D (1972) Data Types as Lattices (unpublished notes).

Tarski A (1955) A lattice-theoretical fixpoint theorem and its applications

Pacific J of Maths 5 285-309.

Walk K et al (1969) Abstract Syntax and Interpretation of PL/I.

IBM Laboratory Vienna
TR.25.098

Wirth N (1971a) Program Development by Stepwise Refinement

CACM, volume 14, No 4. pp. 221 - 227

Wirth N (1971b) The Programming Language Pascal.

Acta Informatica 1, pp 35-63.