Computational Aspects of Lattice Theory

John Francis Buckle

A dissertation submitted for
the Degree of
Doctor of Philosophy

Department of Computer Science
University of Warwick

November 1989

Contents

1) Introduction.

1.1) Introduction. 1
1.2) General Lattice Theory. 2
1.3) Notational Conventions. 5
1.4) Thesis Organisation. 6

Part One. Mathematical Aspects of Lattice Theory

2) A Study of Meet and Join Respecting Pre-orders and Congruences on Finite Lattices.

2.1) Introduction. 10
2.2) Preliminary Definitions. 11
2.3) Characterisation of Lattice Pre-orders. 13
2.4) Lattice Congruences. 15
2.5) Distributive Quotients of Lattice Congruences. 17
2.6) Arbitrary Quotient Lattices. 20

3) A Study of Replaceability on Finite Lattices.

3.1) Introduction and Motivation. 26
3.2) Computational Equivalence in Finite Lattices in Terms of Join-Irreducibles. 28
3.3) Examples of Replaceability in Modular Lattices. 31
3.4) Computational Equivalence and Replaceability in Terms of Covering Edges 34
3.5) Algorithm for Determining Replaceability. 39
3.6) Alternative Characterisation of Approximate Replaceability Triples. 40
3.7) Saturated Elements in Distributive Lattices. 42

Part Two. Computational Aspects of Lattice Theory

4) Technical Aspects of Computation Within Distributive Lattices.

4.1) Introduction. 47
4.2) Implementation of Free Distributive Lattice Functions on Computers. 48
4.3) Planar Monotone Computation. 55
4.4) Generation of Hasse Diagrams for Distributive Lattices. 58
4.5) Hasse Diagrams of FDL(4) and FDL(5). 63
4.6) Hasse Diagram of the Closure Lattice μ(FDL(5)). 71

Part Three. Computer Aided Mathematical Environments for Lattice Theory

5) DEST - A Definitive Environment for Set Theory.
- 5.1) Introduction. 82
- 5.2) Definitive Based Computer Aided Mathematical Environments. 85
- 5.3) Comparison Between Functional and Definitive Notions for Implementing Sets. 87
- 5.4) Aspects of DEST. 89

6) Foundations of Data Types and Operations in DEST.
- 6.1) Introduction. 94
- 6.2) Generators and Predicates. 95
- 6.3) Structured Data Types in DEST. 97
- 6.4) Mathematical Basis for Tuples. 99
- 6.5) Specification of Maps and Relations. 101
- 6.6) Labels. 103
- 6.7) Operations on Infinite Sets. 104

7) Pecan - a Definitive Environment for Lattice Theory.
- 7.1) Introduction. 106
- 7.2) Additional Data Types for Lattices. 106
- 7.3) Construction and Representation of Lattices. 110
- 7.4) Operations on Lattices. 116

8) Conclusions. 121

References. 125

Appendix 1: User Manual for DEST.

Appendix 2: User Manual for Pecan.

Appendix 3: Prime - Programming Archive Report.

Table of Figures

Figure 1.1	4	Figure 4.5	61
Figure 2.1	12	Figure 4.6	61
Figure 2.2	13	Figure 4.7	67
Figure 2.3	19	Figure 4.8	68
Figure 2.4	21	Figure 4.9	69
Figure 2.5	22	Figure 4.10	70
Figure 2.6	24	Figure 4.11	76
Figure 2.7	25	Figure 4.12	80
Figure 3.1	32	Figure 5.1	90
Figure 3.2	33	Figure 5.2	91
Figure 3.3	33	Figure 7.1	112
Figure 3.4	34	Figure 7.2	113
Figure 3.5	35	Figure 7.3	114
Figure 4.1	50	Figure 7.4	115
Figure 4.2	53	Figure 8.1	123
Figure 4.3	56		
Figure 4.4	57		

Table of Algorithms

Algorithm 3.1	40
Algorithm 4.1	54
Algorithm 4.2	55
Algorithm 4.3	59
Algorithm 4.4	65
Algorithm 4.5	73
Algorithm 7.1	116
Algorithm 7.2	119

Acknowledgements

I am indebted to my supervisor Dr. W. M. Beynon for his help and guidance throughout my time at the University of Warwick. I am also grateful for the help of Dr. P. E. Dunne during his stay at Warwick at the start of my study. I would also like to thank Nick Holloway, Jeff Smith and Greg Ryan plus all the support staff at both Warwick and Sydney for keeping the machines running and helping me format this thesis.

I would have never started without the inspiration and support of several people including, the three teachers who impressed me the most: Messrs Hiesemann, Katz and Sargent, my undergraduate friends especially Sally Smith, and my girl friend Kerren Davey. None of this would have been possible without the support of my family and of course especially my mother.

This research was supported by the Science and Engineering Research Council of Great Britain.

Declaration

This dissertation is the result of work carried out in the Department of Computer Science at the University of Warwick between October 1983 and September 1989.

None of the work presented in this dissertation has appeared in any journals or papers. Parts of chapters two and three appeared in Theory of Computation Report No. 72, produced internally by the department.

Dedicated to the memory of

Mr. Donald Stanley Buckle, M.Sc, B.Sc

Engineer, mathematician and father.

Summary

The use of computers to produce a user-friendly safe environment is an important area of research in computer science. This dissertation investigates how computers can be used to create an interactive environment for lattice theory. The dissertation is divided into three parts. Chapters two and three discuss mathematical aspects of lattice theory, chapter four describes methods of representing and displaying distributive lattices and chapters five, six and seven describe a definitive based environment for lattice theory.

Chapter two investigates lattice congruences and pre-orders and demonstrates that any lattice congruence or pre-order can be determined by sets of join-irreducibles. By this correspondence it is shown that lattice operations in a quotient lattice can be calculated by set operations on the join-irreducibles that determine the congruence. This alternative characterisation is used in chapter three to obtain closed forms for all replacements of the form "*h can replace g when computing an element f*", and hence extends the results of Beynon and Dunne into general lattices. Chapter four investigates methods of representing and displaying distributive lattices. Techniques for generating Hasse diagrams of distributive lattices are discussed and two methods for performing calculations on free distributive lattices and their respective advantages are given. Chapters five and six compare procedural and functional based notations with computer environments based on definitive notations for creating an interactive environment for studying set theory. Chapter seven introduces a definitive based language called Pecan for creating an interactive environment for lattice theory. The results of chapters two and three are applied so that quotients, congruences and homomorphic images of lattices can be calculated efficiently.

Chapter One

Introduction

1.1. Introduction

The study of abstract mathematical systems and the development of tools to study these systems provides greater insight into the concrete structures they represent because they provide a different perspective of the structure, allowing them to be seen in a wider context. Hence there is normally great advantage in creating and investigating abstract modules when researching into concrete applications since it removes specific details and allows the inherent structure to be seen. This thesis investigates computational aspects of finite lattices and tools for performing calculations on them, thereby generating an environment for investigating boolean functions and network complexity in an abstract setting.

Network complexity was shown to be a reasonable model of computation by Fisher and Pippenger [29] who demonstrated that any function computable by a deterministic Turing machine in n steps could be realised as a boolean network in O(n log n) gates. However the difficulty of obtaining tight lower bounds on the size of unrestricted circuits has lead to an increase in the investigation of complexity in monotone boolean circuits. It is a well known fact that monotone boolean networks can be identified with elements of free distributive lattices and this algebraic setting is often used as a basis for investigating monotone networks. However the problems associated with monotone boolean functions and free distributive lattices can easily be generalised to finite distributive lattices and lattices in general. Hence solutions in the general setting give new insight into the original problem. For example Beynon in [4] introduced the notion of computational equivalence and replaceability in an abstract mathematical setting and presented a characterisation of replacement rules in distributive lattices, providing an alternative derivation of Dunne's work which appeared in [26, 27].

To be able to use any abstract structure it is necessary that tools are developed to investigate the structure. Such tools in the case of lattice theory include methods of performing computation in lattices and methods for defining and displaying them. Operations on lattices include the calculation of expressions involving meets and joins, calculation of quotients and images of lattices and the use of congruences and homomorphisms between lattices. Since most people's mental image of a lattice is of a Hasse diagram representing the partial order it is based on, Hasse diagrams provide an intuitive method for expressing, defining and displaying results.

To use the tools listed above to investigate abstract lattices it would be beneficial to incorporate them in a computer aided environment where they can be used in a controlled fashion. In such an environment it would be useful to be able to experiment not only with values but also with functional relationships between objects, thereby creating a dynamic environment in which hypotheses can be examined under several different examples with ease.

1.2. General Lattice Theory

Basic lattice theoretic definitions and notations used throughout this thesis will be presented in this section. For a fuller treatment of lattice theory see Grätzer [30]. Readers familiar with lattice theory may ignore this section.

A *poset* $(P, \geq)$ is a partially ordered set consisting of a non-void set P and a reflexive, antisymmetric and transitive relation $\geq$. If for all $x, y \in P$ either $x \geq y$ or $y \geq x$ then the partial order is said to be a *total order* or a *chain*. Any two elements $x, y \in P$ are said to be *comparable* if either $x \geq y$ or $y \geq x$, otherwise they are said to be *incomparable*. A poset is a *lattice* if for every finite non-void subset there exists a least upper bound and a greatest lower bound. Lattices can be equivalently defined as an algebra $(L, \wedge, \vee)$ consisting of two binary operators *meet* $\wedge$ and *join* $\vee$ which are idempotent, commutative, associative and obey the absorption identities:

$$a \wedge (a \vee b) = a, \quad a \vee (a \wedge b) = a$$

If Φ is a "statement" about posets (or lattices), the *dual* of Φ is the statement obtained by replacing all occurrences of the partial order by its dual partial order (or by exchanging the operators meet and join). The *principle of duality* states that if a statement Φ is true for all posets (or lattices) then so is its dual.

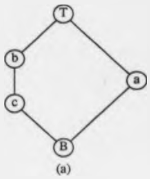
An element a is said to *cover* an element b (denoted as $a \succ b$) if $a > b$ and there does not exist an element x such that $a > x > b$. A *Hasse diagram* of a poset is a diagram depicting the elements of the poset and the covering relations between elements. The elements of the poset are represented as points and a line is drawn between two points if one covers the other; if a covers b then the point representing a is drawn high than b.

A map $\phi : L \rightarrow M$ between two lattices is a *meet-homomorphism* if $(a \wedge b)\phi = a\phi \wedge b\phi$. A *join-homomorphism* is defined dually. A *lattice homomorphism* is a map that is both a join and meet homomorphism. An *isomorphism* between lattices is a lattice homomorphism that is one-one and onto.

A *sublattice* $(K, \wedge, \vee)$ of a lattice $(L, \wedge, \vee)$ is a non void subset K of L where for all $a,b \in K$: $a \wedge b \in K$ and $a \vee b \in K$, (where $\vee$ and $\wedge$ are taken in L). The sublattice *generated* by a subset H of L is the intersection of all sublattices containing H. A subset K of a lattice L is *convex* if for all $a,b \in K$ and $c \in L$ such that $a \leq c \leq b$ imply $c \in K$. For all $a,b \in L, a \leq b$ the *interval* $[a,b]$ is the convex sublattice $\{x \mid a \leq x \leq b\}$.

An equivalence relation Φ is called a congruence relation on a lattice L if for all $a,a',x \in L$ such that $a \equiv a' (\Phi)$ implies $a \wedge x \equiv a' \wedge x (\Phi)$ and $a \vee x \equiv a' \vee x (\Phi)$. For all elements $a \in L$ the congruence class $[a]\Phi$ containing a is a convex sublattice of L. The *quotient lattice* L/Φ is the lattice consistings of the congruence classes of Φ and the operators $[a]\Phi \wedge [b]\Phi = [a \wedge b]\Phi$ and $[a]\Phi \vee [b]\Phi = [a \vee b]\Phi$. The map $\phi : L \rightarrow L/\Phi$ mapping a onto $[a]\Phi$ is called the natural homomorphism. A lattice K is a *homomorphic image* of a lattice L if there is a homomorphism from L onto K. The *Homomorphism theorem* states that every homomorphic image of a lattice is isomorphic to a suitable quotient lattice of the lattice.

A lattice M is said to be a *modular* lattice if for all $a,b,c \in M$ such that $b \geq a$ then $(b \wedge c) \vee a = b \wedge (a \vee c)$. A lattice is said to be *distributive* if meet distributes over join and vice versa. A lattice is modular if and only if it contains no sublattice isomorphic to fig. 1.1a. A lattice is distributive if and only if it contains no sublattice isomorphic to either 1.1a or 1.1b. A lattice F is *freely generated* by a subset X if F is generated by X and any map of the subset X to a lattice L extends to a homomorphism of F onto L. When a lattice F is freely generated by one of its subsets it is referred to as a *free lattice*.



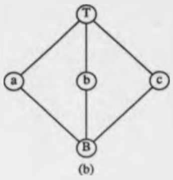


figure 1.1

Join and Meet Irreducibles

A non minimal element p of a lattice L is called a *join-irreducible* if for all $x,y \in L$ such that $x \vee y = p$ implies either $x = p$ or $y = p$. *Meet-irreducibles* are defined dually. If a is an element of a finite lattice L the representation

$$a = p_1 \vee p_2 \vee \cdots \vee p_k$$

of a as a join of join-irreducibles is called a *finite decomposition* of a. A decomposition is called *irredundant* if the join of a subset of the irreducibles is not equal to a. Any element other than the minimal and maximal elements of a finite lattice can be expressed as an irredundant join of join-irreducibles or as an irredundant meet of meet-irreducibles. In a distributive lattice irredundant representations are unique.

In free distributive lattices join-irreducibles are sometimes referred to as *monoms* and meet-irreducibles as *clauses*. A join-irreducible p is called a *prime implicant* of a function f if p is maximal subject to $p \le f$. *Prime clauses* are defined dually. The representation of an element as a join of join-irreducibles or a meet of meet-irreducibles is referred to as the *disjunctive normal form* and *conjunctive normal form* respectively.

Since elements can be expressed as an irredundant join of join-irreducibles it follows that if s,t are elements of a finite lattice L such that $s \nleq t$ then there exists a join-irreducible p such that $p \le s$ but $p \nleq t$. Moreover if $s > t$ it is possible to choose p so that the element it covers is less than t.

Lemma 1.2.1

If L is a finite lattice and $s,t \in L$ such that $s > t$ then there exists elements $x,y \in L$, x a join-irreducible, where $x \succ y$ and $s \geq x$, $t \not\geq x$ but $t \geq y$.

Proof.

Since $s > t$ it follows that there exists a join-irreducible p such that $p \leq s$ and $p \not\leq t$. Let $n = p \wedge t$ and define m such that $p \geq m \succ n$. If m is not a join-irreducible then there exists another join-irreducible p' such that $p' \leq m$ but $p' \not\leq n$. Since $p' \vee n = m \not\leq t$ it follows that $p' \not\leq t$, and since the lattice is finite the above argument can be repeated to show that there exists a join-irreducible x such that $x \leq s$ but $x \not\leq t$, and $x \succ y \leq t$.

□

1.3. Notational Conventions

For clarity the following notational conventions will be used through out the thesis unless otherwise stated.

Diagrams and algorithms will be numbered sequentially from the beginning of the chapter (eg. fig. 4.5 is the fifth diagram in chapter four). Theorems, propositions etc. will be numbered sequentially from the start of the subsection they appear in (eg. proposition 4.2.2.1 is the first statement in subsection two of section two in chapter four).

Roman capitals will be used to denote sets of elements, lower case letters for individual elements. Lattice elements beginning with the letter p (eg. p, p', p_i) denote join-irreducibles, elements beginning with a q denote meet-irreducibles. When the context is clear a lattice $(L,\vee,\wedge)$ will be referred to by just its base set L.

Where possible partial and pre-order relations will be denoted by an asymmetric symbol, the dual order is denoted by the reverse of that symbol. Equivalence and congruence relations will be denoted by symmetric symbols.

When the range of an index variable can be determined by context the range will not be explicitly stated.

1.4. Thesis Organisation

The thesis is divided into three parts. Chapters two and three concern the characterisation of lattice pre-orders, congruences and quotients and derives a characterisation of computational replaceability on finite lattices. Chapter four describes two methods by which distributive lattices can be manipulated on computers and demonstrates how these methods can be used for performing calculations, circuit building and creating Hasse diagrams. Chapters five, six and seven outline a programming environment for analysing lattices based on definitive notations using the results of chapters two and three and the methods of chapter four.

1.4.1. Chapter 2

A relation * between join-irreducibles in finite lattices is defined and its association with lattice pre-orders is demonstrated. It is shown that any lattice pre-order determines two sets of join-irreducibles closed under the relation * and that relations in the pre-order can be calculated from the two sets. The converse that any two sets of join-irreducibles closed under * determine a lattice pre-order is also demonstrated. Necessary and sufficient conditions for a congruence to have a distributive quotient lattice are presented, and it is shown how arbitrary finite lattices can be reconstructed from a quotient lattice and the relationship of the join-irreducibles determining the quotient.

1.4.2. Chapter 3

Computational equivalence and replaceability on finite lattices is considered and two derivations of closed forms for all replacements of the form "*g is replaceable by h in an expression computing f*" is given. The first derivation uses the results of chapter two to classify all possible replacements in the style of [4]. The second method relies on a more graphical approach based on covering edges. An alternative characterisation of approximate replaceability triples given by Dunne [28] is presented in which valid approximate replaceability triples $<f,D,C>$ are given as intervals in $FDL(n)^3$. Finally the notion of saturated elements is introduced and it is shown that these elements are the minimum one-replaceable elements $\mu(x)$ as defined by Beynon [4].

1.4.3. Chapter 4

Methods of implementing and manipulating elements of free distributive lattices are discussed in chapter four. Two methods are proposed, the first method being suitable for performing calculations on free distributive lattices where the number of variables is small (up to around FDL(8)), the second method uses dynamic memory and can handle calculations in lattices of "arbitrary" size. Techniques for the automatic display of Hasse diagrams of distributive lattices is discussed and it is shown how the first method of implementing free distributive lattices above can be combined with the techniques described to construct the Hasse diagrams of FDL(4), FDL(5) and the closure lattice μ(FDL(5)). Technical aspects of the algorithm for constructing planar monotone circuits given in [10] is also described.

1.4.4. Chapter 5

The notion of mathematical environments based on definitive principles is discussed and compared with other environments based on procedural and functional principles. It is argued that a definitive system provides a natural environment of interaction and experimentation which is lacking in the other two. The foundation of a definition based environment for set theory (DEST) is described with reference to creating an environment for finite lattices.

1.4.5. Chapter 6

The data types of DEST are described and their structure is given. It is demonstrated that by arranging the data types in a hierarchy and providing suitable conversion operators between types that an object oriented approach with polymorphic operators can be developed. It is argued that an underlying algebra of character string values combined with pattern-matching produce a suitable algebra of values for any environment dealing with sets and especially applicable for defining partial orders. By using a system of character string variables it is demonstrated that ordered tuples, maps and relations can be specified concisely.

1.4.6. Chapter 7

Additional data types for handling finite lattices are introduced and methods for specifying and storing lattices in a definitive environment are discussed. The results of chapters two and three are used to provide an efficient method to specify and calculate congruences, computational equivalence and quotient lattices.

Part One

Mathematical Aspects of Lattice Theory

Chapter Two

A Study of Meet and Join Respecting Pre-Orders and Congruences on Finite Lattices.

2.1. Introduction

The motivation for this chapter is that by expressing congruences and pre-orders using sets of join-irreducibles the alternative characterisation given provides an efficient method of specifying and implementing congruences and pre-orders on computers. That is instead of having to record a large set of ordered pairs it is more convenient to represent pre-orders and congruences by small sets of join-irreducibles. Also the notions of "Computational Equivalence" and "Computational Replaceability" on finite lattices discussed in chapter three determines lattice congruences and pre-orders respectively which in turn are determined by sets of join-irreducibles. Hence a study of general congruences and lattice pre-orders and how join-irreducibles are related to them gives a useful foundation to the techniques used in chapter three.

Theorem 2.3.1 proves that every pre-order which respects the lattice operations on a finite lattice ("lattice pre-orders") determines two sets of join-irreducibles which are closed under a relation * between join and meet-irreducibles. The converse to the theorem, that any two sets of join-irreducibles closed under * determine a pre-order which respects the lattice operations, is proved in Theorem 2.3.2. This result is used in proving the general result of computational equivalence and replaceability on finite lattices in chapter three. Corollary 2.3.3 proves that this correspondence between lattice pre-orders and sets of join-irreducibles gives rise to an anti-isomorphism between them. This result is a generalisation of Lemma 2.1 proved in [4] which deals with finitely generated distributive lattices. By these results it is possible to identify congruences with sets of join-irreducibles closed under * and hence study congruences via this identification.

Section two defines a correspondence $\tilde{}$ between join and meet-irreducibles and demonstrates some basic properties of this correspondence. Section three contains the main results of this chapter stating the correspondence between lattice pre-orders and sets of join-irreducibles. Section four is a small section detailing the characterisation of lattice congruences. Section five studies congruences which produce distributive quotients, describing the corresponding set of join-irreducibles. Section six describes how the quotient lattice of a congruence is determined by the poset of join-irreducibles associated with the congruence.

All results in this section are stated in terms of join-irreducibles. By duality all the results can be stated using meet-irreducibles instead.

2.2. Preliminary Definitions

A lattice pre-order $\lessdot$ on a finite lattice L is a reflexive and transitive relation such that, for all $a,b,c \in L$,

$$a \lessdot b \Rightarrow a \vee c \lessdot b \vee c \;\&\; a \wedge c \lessdot b \wedge c$$

A lattice congruence is a symmetric lattice pre-order.

For any join-irreducible p and meet-irreducible q:

$$\tilde{p} = \{\, x \mid x \text{ is maximal subject to } x \ngeq p \,\}$$
$$\tilde{q} = \{\, x \mid x \text{ is minimal subject to } x \nleq q \,\}$$

Obviously $\tilde{p}$ is a set of meet-irreducibles and $\tilde{q}$ is a set of join-irreducibles. Also for any element x, $p \leq x$ if and only if $\exists q \in \tilde{p} : x \leq q$ and dually. If F is a set of irreducibles, then $\tilde{F}$ will be used to denote $\bigcup_{f \in F} \tilde{f}$, and $\tilde{\tilde{F}} = \bigcup_{g \in \tilde{F}} \tilde{g} = \{ f \in \tilde{g} \mid g \in \tilde{F} \}$. Let F be a set of join-irreducibles, define a sequence of sets of join-irreducibles $F_0, F_1, \ldots, F_k$ via $F_0 = F$, $F_{i+1} = F_i \cup \tilde{\tilde{F_i}}$. Let $F^* = F_k$ where $F_{k+1} = F_k$. (This is bound to occur since there are only a finite number of join-irreducibles and the sets are non decreasing.) If $F = \{f\}$ it would be convenient to write f^* for F^*. F^* will be referred to as the $\tilde{}$ (tilde) closure of F.

The $\tilde{}$ closure of a set F could have been alternatively defined by saying that it is the smallest set F' that contains F and $\forall f \in F' : \tilde{\tilde{f}} \subseteq F'$. It should also be observed that the union and intersection of two $\tilde{}$ closed sets is also $\tilde{}$ closed.

For any set of join-irreducibles F, let $F[x] = \{y \in F \mid y \leq x\}$.

In distributive lattices the ¯ function is a one-one correspondence between join and meet-irreducibles. Figure 2.1 shows that ¯ is not symmetric (treating ¯ as a binary relation between irreducibles) in arbitrary lattices since $b \in \breve{c}$ (treating b as a join-irreducible) however $c \notin \breve{b}$. The next proposition states some basic properties of ¯.

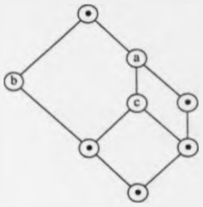


figure 2.1

Proposition 2.2.1

If L is a finite lattice then

i) For all join-irreducibles $p \in L$, $p \in \breve{p}$ and dually for meet-irreducibles.

ii) For all join-irreducibles $p \in L$, $|\breve{p}| = 1$ iff $\{p\} = \breve{p}$ and dually for meet-irreducibles.

iii) If L is a modular lattice and p a join-irreducible and q a meet-irreducible in L, then $p \in \breve{q}$ if and only if $q \in \breve{p}$.

Proof.

(i) Follows immediately from the definition of ¯.

(ii) Suppose $\breve{p} = \{q\}$ where p is a join-irreducible and let $p' \in \breve{q}$. If $p' \ngeq p$ then $L \setminus (\{x \mid x \geq p\} \cup \{x \mid x \leq q\})$ is non empty, contradicting $|\breve{p}| = 1$. Hence $p' \geq p$, but this implies $p = p'$ by definition of $\breve{q}$. The converse follows by a similar argument.

(iii) Suppose $p \in \bar{q}$, then p is a minimal element $\not\leq q$, hence $p \vdash p \wedge q$. To prove $q \in \bar{p}$ it will suffice to show that $p \vee q \vdash q$, hence q is a maximal element $\not\geq p$. Choose x such that $p \vee q \vdash x$ and $x \geq q$. Then by modularity $x = x \wedge (q \vee p) = q \vee (x \wedge p) = q$. Hence $x = q$ and $p \vee q \vdash q$.

The case for $q \in \bar{p}$ is similar.

□

Also while it is obvious that the $^-$ closure always exists the number of "iterations" required is not constant, for example the modular lattices described in figure 2.2 require iterations proportional to the height of the lattice.

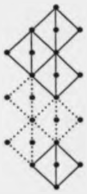


figure 2.2

2.3. Characterisation of Lattice Pre-orders

Theorem 2.3.1

If $\lhd$ is a pre-order on a finite lattice L, which respects the lattice operations then there exists two sets of join-irreducibles U, D such that $U = U^*$ and $D = D^*$ and for all $a,b \in L$

$$a \lhd b \text{ iff } U[a] \supseteq U[b] \;\&\; D[a] \subseteq D[b] \tag{1}$$

Proof.

Let D be the set of all join-irreducibles p such that $p \ntriangleleft x$ where $p \vdash x$. Similarly let U be the set of all join-irreducibles p such that $x \ntriangleleft p$.

Let $p \in D$ and $p' \in \hat{\beta}$ and let x, x' be the elements covered by p and p' respectively. Since $p' \in \hat{\beta}$ there exists a meet irreducible q such that $p' \in \hat{q}$ and $q \in \beta$, hence $((p' \vee q) \wedge p) \vee x = p$ while $((x' \vee q) \wedge p) \vee x = x$. However $p \not\lessdot x$ and $\lessdot$ respects the lattice operations, so $p' \not\lessdot x'$ and hence $p' \in D$ and $D = D^*$. Similarly $U = U^*$.

Let $a, b \in L$ such that the right hand side of (1) is invalid. Therefore there exists a join-irreducible p such that *either* $p \in U[b]$ and $p \notin U[a]$ *or* $p \in D[a]$ and $p \notin D[b]$. Let x be the element covered by p. In the former case $(b \wedge p) \vee x = p$ and $(a \wedge p) \vee x = x$ and $p \in U$ hence it follows that $a \not\lessdot b$. In the latter case $(a \wedge p) \vee x = p$ and $(b \wedge p) \vee x = x$ hence $a \not\lessdot b$. Thus $a \lessdot b \Rightarrow U[a] \supseteq U[b]$ and $D[a] \subseteq D[b]$.

For the converse, let $a, b \in L$ such that $a \not\lessdot b$. Let $c = a \wedge b$. Either $a \not\lessdot c$ or $c \not\lessdot b$ otherwise $a \lessdot c \lessdot b$.

Suppose $a \not\lessdot c$, the case for $c \not\lessdot b$ is similar. Since $a \not\lessdot c$ and $a > c$ there exists a covering pair a', c' such that $a \geq a' \succ c' \geq c$ and $a' \not\lessdot c'$. By lemma 1.2.1 there exists a join-irreducible p and an element x covered by p such that $a' \geq p$, $c' \ngeq p$ but $c' \geq x$. Since $p \vee c' = a'$ and $x \vee c' = c'$ it follows that $p \not\lessdot x$ and hence $p \in D$. Therefore $p \in D[a'] \subseteq D[a]$ and since $p \nleq c = a \wedge b$ but $p \leq a$ it follows that $p \notin D[b]$ and $D[a] \not\subseteq D[b]$.

□

Theorem 2.3.2

If U and D are two sets of join-irreducibles from a finite lattice L such that $U = U^*$ and $D = D^*$ then the relation $\lessdot$ between the elements of L defined by (1) above is a pre-order which respects the operations of the lattice.

Proof.

Obviously $\lessdot$ is a pre-order since it is reflexive and transitive, hence it will suffice to prove that $\lessdot$ respects the operations of the lattice.

$a \lessdot b \Rightarrow a \wedge c \lessdot b \wedge c$: Since $D[a \wedge c] = D[a] \cap D[c]$ and $D[a] \subseteq D[b]$ it follows that $D[a \wedge c] \subseteq D[c] \cap D[b] = D[b \wedge c]$. A similar argument shows that $U[b \wedge c] \subseteq U[a \wedge c]$.

$a \lessdot b \Rightarrow a \vee c \lessdot b \vee c$: It will suffice to show that if $a \vee c \not\lessdot b \vee c$, then $a \not\lessdot b$. Suppose $\exists p \in D$ such that $p \leq a \vee c$ but $p \nleq b \vee c$, then $\exists q \in \beta$ such that $q \geq b \vee c$ and $q \ngeq a \vee c$. Since $q \ngeq a \vee c$ and $q \geq c$ it follows that $q \ngeq a$. As $b \leq q$, $\exists t \in \hat{q}$ such that $a \geq t$ and $b \ngeq t$. Thus $D[a] \not\subseteq D[b]$ since $t \in \hat{\beta} \subseteq p^* \subseteq D$. The argument for $p \leq b \vee c$ and $p \nleq a \vee c$ follows a similar method using the U set.

□

Cor. 2.3.3

Let L be a finite lattice. The correspondence in Theorem 2.3.1 and Theorem 2.3.2 is an anti-isomorphism between lattice pre-orders on L ordered by inclusion and ordered pairs of sets of join-irreducibles closed under * ordered by the relation:

$$(D_1,U_1) \subseteq (D_2,U_2) \text{ iff } D_1 \subseteq D_2 \ \& \ U_1 \subseteq U_2$$

Proof.

Let D_1,U_1 and D_2,U_2 be two pairs of sets of join-irreducibles closed under * which determine the pre-orders $\trianglelefteq_1$ and $\trianglelefteq_2$.

Suppose $\trianglelefteq_1 \supseteq \trianglelefteq_2$. If $p \in D_1$, where $p \vdash x$, then $p \ntrianglelefteq_1 x$, hence $p \ntrianglelefteq_2 x$ and so $p \in D_2$. Similarly $U_2 \supseteq U_1$.

Suppose $(D_1,U_1) \subseteq (D_2,U_2)$. If $x,y \in L$ such that $x \trianglelefteq_2 y$ then $D_2[x] \subseteq D_2[y]$ and $U_2[x] \supseteq U_2[y]$. So $\forall d \in D_1 : d \leq x \Rightarrow d \leq y$ since $d \in D_2$, also $\forall u \in U_1 : u \leq y \Rightarrow u \leq x$ since $u \in U_2$. Hence $x \trianglelefteq_1 y$.

Moreover the correspondence is injective since if $p \in D_2 \backslash D_1$ (resp. $p \in U_2 \backslash U_1$) where $p \vdash x$ then $p \ntrianglelefteq_2 x$ (resp. $x \ntrianglelefteq_2 p$) but $p \trianglelefteq_1 x$ (resp. $x \trianglelefteq_1 p$).

□

2.4. Lattice Congruences

Lattice congruences play an important role in lattice theory and lattices of lattice congruences have been studied extensively.

Gratzer and Schmidt in [31] discuss the relationship between ideals and congruence relations. They gave an extensive study of congruence relations on distributive lattices and stated various properties of congruence relations which characterise the distributivity of the lattice. Further they gave necessary and sufficient conditions for the ideals of the lattice to be in one-one correspondence with the congruence relations of the lattice and also gave necessary and sufficient conditions for the lattice of congruence relations to be a boolean algebra.

Urosu in [45] also discusses the connection between standard ideals (see [32]) and congruence relations and gave a new proof for the one-one correspondence between standard ideals and congruence relations in complemented modular lattices.

Sivak in [42] considers congruence preserving extensions of a lattice L_1 into a super-algebra L_2 where the assignment $\Phi \rightarrow \Phi \cap (L_1, L_1)$ is an isomorphism between the congruence lattice of L_1 and the congruence lattice of L_2. It is shown that all lattices have such an extension and that the extended lattice is atomistic. It is also demonstrated that any congruence Φ on the extended lattice can be expressed in the form con(0 , p) where 0 is the least element of the original lattice and p is a distributive element.

Most of the work in [31] and [45] is based on complemented modular lattices. The results in this and subsequent sections give a more computational aspect to describing congruences on finite lattices and their quotients by expressing the results in terms of join-irreducibles rather than by using the more abstract terms of standard ideals and congruence preserving extensions.

The results for pre-orders which respect the lattice operations can be used directly for lattice congruences since a lattice congruence is a symmetric pre-order. The theorem can be restated using only one set of join-irreducibles.

Cor. 2.4.1

Every lattice congruence Φ on a finite lattice determines a set of join-irreducibles closed under * and every set of join-irreducibles closed under * determines a lattice congruence. The correspondence produced between congruences and sets of join-irreducibles closed under * is an anti-isomorphism.

Proof.

The pre-order $\lessdot$ in Theorem 2.3.1 is symmetric if and only if D = U.

The proof that the correspondence is an anti-isomorphism is similar to the proof of Cor 2.3.3

□

Theorem 2.4.2

If Θ and Φ are two congruences on a finite lattice L characterised by sets of join-irreducibles T and F respectively, then the congruence $\Theta \cap \Phi$ is characterised by the set $T \cup F$ and the congruence $\Theta \cup \Phi$ is characterised by the set $T \cap F$.

Proof.

Let $g,h \in L$ and Π be the congruence characterised by the set of join-irreducibles $P = F \cup T$ and Ψ be the congruence characterised by the set of join-irreducibles $S = F \cap T$. From the observation in section two that the union and intersection of $^{-}$ closed sets is $^{-}$ closed the definitions of Π and Ψ make sense.

If $g \equiv h\ (\Theta \cap \Phi)$ then $T[g] = T[h]$ and $F[g] = F[h]$, hence $P[g] = P[h]$ so $g \equiv h\ (\Pi)$.

If $g \not\equiv h\ (\Theta \cap \Phi)$ then either $g \not\equiv h\ (\Theta)$ or $g \not\equiv h (\Phi)$. Without loss of generality assume the former, hence $T[g] \neq T[h]$. Again without loss of generality assume that there exists a join-irreducible $p \in T[g]$ such that $p \in T[h]$, hence $p \notin F[h]$ since $p \not\leq h$. So $p \notin P[h]$ but $p \in P[g]$ hence $g \not\equiv h\ (\Pi)$.

If $g \equiv h\ (\Theta \cup \Phi)$ then there exists elements $a_0 = g, a_1, \ldots, a_k = h$ such that $a_i \equiv a_{i+1}\ (\Phi)$ or $a_i \equiv a_{i+1}\ (\Theta)$. If $p \in S[a_i]$ then $p \in S[a_{i+1}]$ since $p \in F \cap T$ and $a_i \equiv a_{i+1}\ (\Phi)$ or $a_i \equiv a_{i+1}\ (\Theta)$. Hence $S[g] = S[h]$ and $g \equiv h\ (\Psi)$.

If $g \not\equiv h\ (\Theta \cup \Phi)$ then there exists a pair a,b of elements such that $a \succ b$, $b \geq g \wedge h$ and either $g \geq a$ or $h \geq a$ where $a \not\equiv b\ (\Phi)$ and $a \not\equiv b\ (\Theta)$ (if no such pair exists then g would be equivalent to h). Without loss of generality assume that $g \geq a$. By lemma 1.2.1 there exists a join-irreducible p and an element x covered by p such that $a \geq p$, $b \not\geq p$ but $b \geq x$. Since $b \vee p = a$ and $b \vee x = b$ it follows that $p \not\equiv x\ (\Phi)$ and $p \not\equiv x\ (\Theta)$, hence $p \in F$ and $p \in T$. Since $g \geq a \geq p$ it follows that $p \in S[g]$, moreover since $b \not\geq p$ it follows that $h \not\geq p$ otherwise $p \leq g \wedge h \leq b$. Hence $p \notin S[h]$ and $g \not\equiv h\ (\Psi)$.

□

2.5. Distributive Quotients of Lattice Congruences

In this section necessary and sufficient conditions for the quotient of a lattice congruence to be distributive are given. The following lemma and proposition describe the effects of quotients on the join-irreducibles associated with the congruence.

Lemma 2.5.1

Let L be a finite lattice and Φ a congruence on L determined by a set P of join-irreducibles and let $p \in P$. If $q \in \bar{p}$ then $[q]\Phi$ is a meet-irreducible not greater than $[p]\Phi$ in the quotient lattice L/Φ.

Proof.

Let $p' = [p]\Phi$, $q' = [q]\Phi$, $s = p \vee q \succ q$ and $s' = [s]\Phi$. Since $s \geq p$ and $q \ngeq p$ it follows that $s' \neq q'$. Assume for a contradiction that q' is not a meet-irreducible and let t be the smallest element such that $s' \wedge t' = q'$ and $t' \neq q'$ where $t' = [t]\Phi$. Let $u = s \wedge t \equiv q\ (\Phi)$. Hence $(q \vee t) \wedge s = s$ since q is a meet-irreducible, but $(u \vee t) \wedge s = u$, contradicting $u \equiv q\ (\Phi)$. Hence q' is a meet-irreducible.

Since $s \not\equiv q\ (\Phi)$ and $[s]\Phi = [q \vee p]\Phi = [q]\Phi \vee [p]\Phi$ it follows that $q' \ngeq p'$.

□

Proposition 2.5.2

Let P be a set of join-irreducibles closed under $^{-}$ of a finite lattice L and Φ be the congruence determined by P. If p' is a join-irreducible and q' a meet-irreducible in the quotient lattice $L' = L/\Phi$ and p,q be the minimal, maximal elements respectively such that $p' = [p]\Phi$ and $q' = [q]\Phi$ then

i) p is a join-irreducible in P

ii) q is a meet-irreducible

iii) if $q' \in \bar{p}'$ then $q \in \bar{p}$

iv) if $m \in P$ and $n \in \bar{m}$ then $n' \in \bar{m}'$ where $n' = [n]\Phi$ and $m' = [m]\Phi$.

Proof.

(i) Since p' is a join-irreducible in the quotient it follows that for all elements x,y covered by p, $P[x] = P[y]$. However p is distinct from the elements it covers so $p \in P$ and hence is a join-irreducible.

(ii) Let X be the set of elements covering q. If $|X| = 1$ the q is a meet-irreducible. If $|X| > 1$ then let x,y be two distinct elements from X. Since q' is a meet-irreducible it follows that $P[x] = P[y]$, moreover if $z \in P[x]$ then $z \leq x \wedge y = q$ hence $P[x] = P[q]$, contradicting q being the maximal element such that $q' = [q]\Phi$.

(iii) By (ii) q is a meet-irreducible and since $q' \ngeq p'$ it follows that $q \ngeq p$, hence it has to be shown that there does not exist a meet-irreducible greater than q and not greater than p. Let s be a maximal meet-irreducible subject to $s \geq q$ and $s \ngeq p$. Therefore $s \wedge p < p$ and so $[s \wedge p]\Phi < p'$ which implies $[s]\Phi \ngeq p'$. However $[s]\Phi \geq q'$ and $q' \in \hat{p}'$ hence $[s]\Phi = q'$ and $s = q$ since q is the maximal element in the congruence class q'.

(iv) By lemma 2.5.1 n' is a meet-irreducible not greater than m'. Moreover n is the maximal element in the congruence class n'. Let r' be a meet-irreducible such that $r' \geq n'$ and $r' \ngeq m'$ and let r be the maximal element in the congruence class of r', hence $r \geq n$. By hypothesis $r' \in \hat{m}'$ so by (iii) $r \in \hat{m}$ however $n \in \hat{m}$ and $r \geq n$ so $r = n$ and $r' = n'$.

□

Defn. A join-irreducible p bisects a modular diamond sublattice M (fig 2.3) if $p \leq T$ and $p \nleq B$.

A join-irreducible p bisects a pentagon sublattice S if $p \leq T$ and $p \nleq a$ and $p \nleq c$.

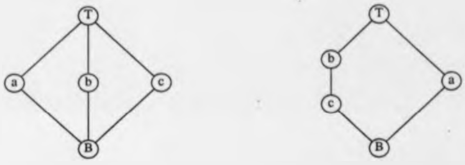


figure 2.3

Lemma 2.5.3

If p is a join-irreducible which bisects a modular diamond sublattice or a pentagon sublattice then $|\hat{p}| > 1$.

Proof.

Diamond Since $a \wedge b = b \wedge c = a \wedge c = B$ (fig 2.3) it follows that there exists distinct $x, y \in \{a,b,c\}$ such that $p \nleq x$ and $p \nleq y$. Let $q_x, q_y \in \hat{p}$ where $q_x \geq x$ and $q_y \geq y$. Since $x \vee y = T \geq p$ it follows that $q_y \ngeq x$ and $q_x \ngeq y$, hence $|\hat{p}| > 1$.

Pentagon Let $q_a, q_c \in \bar{p}$ such that $q_a \geq a$ and $q_c \geq c$. Since $a \vee c = T \geq p$ it follows that $q_a \neq q_c$, hence $|\bar{p}| > 1$.

□

Theorem 2.5.4

Let L be a finite lattice and P_1 be the set of join-irreducibles where $\forall p \in P_1 : |\bar{p}| = 1$. If $P \subseteq P_1$ then the equivalence relation determined by P is a congruence whose quotient is distributive. Moreover all congruences whose quotients are distributive are of this form.

Proof.

If there exists a non-distributive sublattice in the quotient, then there exists a join-irreducible p' in the quotient such that $|\bar{p}'| > 1$. Let p be the minimal element in the congruence class for p' and q'_1, q'_2 be distinct members of $\bar{p}'$ and q_1, q_2 be the maximal elements in the congruence classes respectively. Hence by Lemma 2.5.2.i, $p \in P$ and by Lemma 2.5.2.iii $|\bar{p}| > 1$ contradicting the hypothesis of P.

Suppose P' is a set of join-irreducibles closed under $^-$ and that $p \in P'$ such that $|\bar{p}| > 1$. Let q_1, q_2 be distinct elements of $\bar{p}$. Hence $(q_1 \vee q_2) \wedge p = p$ and $(q_1 \wedge p) \vee (q_2 \wedge p) \leq a$ where $p \succ a$. However a is not congruent to p and the congruence determined by P' respects the lattice operations hence the quotient is not distributive.

□

2.6. Arbitrary Quotient Lattices

In distributive lattices the quotient lattice produced by factoring over a congruence relation Φ is described by the partially ordered set of join-irreducibles determining Φ, (see [4] Lemma 2.1). However non-distributive quotients of arbitrary finite lattices can not be described in this way as can be seen in figures 2.4a and 2.4c, here the identity congruence in both lattices is determined by the poset of join-irreducibles shown in figure 2.4b.

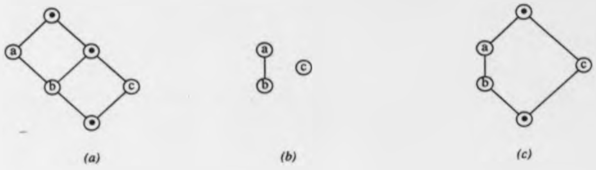


figure 2.4

Hence it is clear that knowledge of the poset of join-irreducibles is not sufficient. Moreover, as can be seen in figure 2.5, knowledge of the join-irreducibles and how they behave under $^-$ is also not sufficient and that details of how the poset behaves under $^-$ is required.

By Theorem 2.3.1 and Corollary 2.4.1 the congruence classes of a congruence Φ determined by a set of join-irreducibles P are in one-one correspondence with sets of decreasing join-irreducibles of the form $P[a]$, $a \in L$. Hence to characterise the quotient lattice of the congruence it will suffice to identify the appropriate decreasing sets of join-irreducibles of P.

Defn. Let P be a poset of join-irreducibles from a finite lattice L and $X \subseteq P$. X is said to be **hereditary with respect to P if $X = P[\vee X]$.**

Example.

Let $P = \{a,b,c,d,e,f\}$. In fig 2.5b there are eight hereditary sets with respect to P, namely $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{f\}$, $\{a,b,c,d,e,f\}$ and ϕ. Fig 2.5a has three more hereditary sets $\{a,b\}$, $\{c,d\}$, $\{e,f\}$.

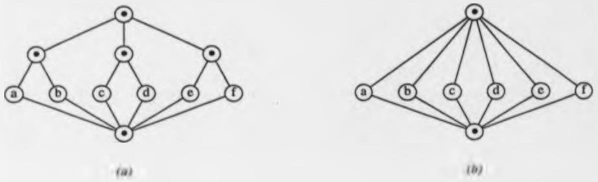


figure 2.5

The proof of Theorem 2.3.2 demonstrated that the intersection of hereditary sets is hereditary, and hence it is possible to define a lattice structure on hereditary sets.

Defns. Let $H(P) = \{X \subseteq P \mid X \text{ is hereditary wrt } P\}$ where P is a set of join-irreducibles closed under * and for all $X,Y \in H(P)$ let

$X \wedge_H Y = X \cap Y$

$X \vee_H Y =$

where $\lceil Z \rceil = \bigcap\{W \in H(P) \mid Z \subseteq W\}$

Theorem 2.6.2

If Φ is a congruence on a finite lattice L determined by a poset P of join-irreducibles closed under * then the quotient lattice $L' = L/\Phi = \langle L', \wedge_\Phi, \vee_\Phi \rangle$ is isomorphic to $\langle H(P), \wedge_H, \vee_H \rangle$.

Proof.

Let σ be a map $\sigma : L' \to H(P)$ defined by $\sigma(\Phi(a)) = P[a]$ where $a \in L$. By Theorem 2.3.1, Corollary 2.4.1 and the remark above this is a lattice isomorphism.

□

It is possible to give an alternative definition of hereditary with respect to a set P of join-irreducibles in terms of P and $\hat{P}$.

Proposition 2.6.3

If P is a set of join-irreducibles closed under $\bar{}$ and $X \subseteq P$ then X is hereditary wrt P if and only if for any $p \in P$

$$(\forall q \in \bar{p} : \exists x \in X : x \not\leq q) \Rightarrow p \in X \qquad (1)$$

Proof.

It will suffice to show that for hereditary sets (1) does not increase membership and that (1) can detect non-hereditary sets.

Let X be a hereditary set wrt P and let $p \in P \backslash X$. Hence $p \not\leq \vee X$ so there exists $q \in \bar{p}$ such that $q \geq \vee X \geq x$ for all $x \in X$, therefore the prerequisite for (1) is not satisfied.

Let $Y \subseteq P$ and $p \in P[\vee Y]$ and $q \in \bar{p}$. Since $p \leq \vee Y$ it follows that $q \not\geq \vee Y$ and so there exists $y \in Y$ such that $q \not\geq y$, hence the prerequisite of (1) is satisfied.

□

Since $x \not\leq q \Leftrightarrow \exists y \in P : x \geq y \ \& \ y \in \bar{q}$ the above condition (1) can be restated as

$$(\forall q \in \bar{p} : \exists x \in X : \exists y \in P : x \geq y \ \& \ y \in \bar{q}) \Rightarrow p \in X \qquad (2)$$

and hence

$$(\exists x \in X : p \leq x) \vee (\forall q \in \bar{p} : \exists x \in X : x \in \bar{q}) \Rightarrow p \in X \qquad (3)$$

If the underlying lattice is distributive then $\bar{p} = \{p\}$, hence (2) becomes $(\exists x \in X : x \geq p)$ which is the special case of [4]. The next corollary follows immediately from the definition given by (3).

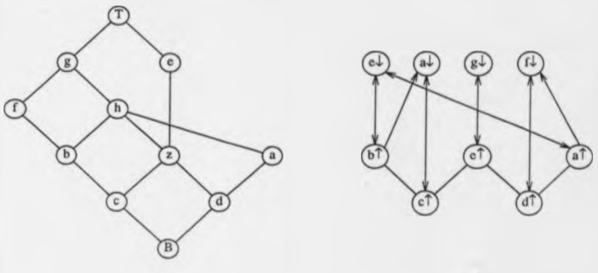
Cor. 2.6.4

The structure of a finite lattice is determined by the poset of join-irreducibles and the $\bar{}$ correspondence.

□

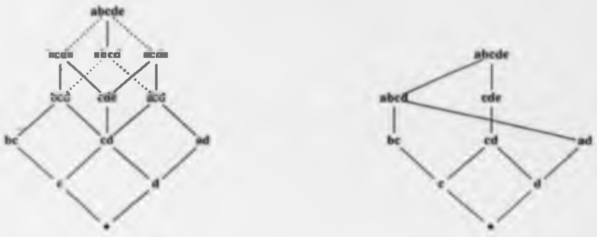
Example.

Let L be the lattice shown in figure 2.6a and let Φ be the equivalence relation determined by the set $P = \{a,b,c,d,e\}$ of join-irreducibles where $x \equiv y\,(\Phi) \Leftrightarrow P[x] = P[y]$. Figure 2.6b shows the poset P of join-irreducibles a,b,c,d,e (drawn here as $a\uparrow, \ldots, e\uparrow$) and the set $\hat{P} = \{a,e,f,g\}$ (drawn here as $a\downarrow, \ldots, g\downarrow$). Arrows connecting the sets show the $^-$ relationship between elements (eg $\bar{a}\uparrow = \{c,f\}$, $\bar{f}\downarrow = \{d\}$). As can be seen P is closed under $^=$ and hence Φ is a congruence. Figure 2.7a shows the lattice 2^P, the distributive lattice generated P, where dotted lines link non hereditary sets to their hereditary closure. For example, { a, c, d } is not hereditary since $e \not\geq a$ and $a \not\geq c$ so b must be present because $\bar{b} = \{a, e\}$. Figure 2.7b shows the final quotient lattice derived from figure 2.7a.



(a) *(b)*

figure 2.6



(a) *(b)*

figure 2.7

Chapter Three

A Study of Replaceability on Finite Lattices

3.1. Introduction

The use of replacement techniques in proving lower bounds on network size has appeared in many papers (eg. [37, 36, 27, 47]). In [27] Dunne gave closed forms for particular kinds of replacements on montone networks and used these forms to develop lower bounds on circuit size for threshold and related functions. The notions of computational equivalence and replaceability were defined in a general algebraic setting by Beynon in [4] which proceeded to give a detailed study of replaceability on finite distributive lattices generalising the results which appeared in [26] and [27]. In [9] computational equivalence and replaceability were discussed in more general algebras including general lattices, semi-lattices and integer semi-groups (Z_m, .). While these subjects have little or no direct computational application, they help to put the ideas of [4, 26, 27] into a wider perspective. Aspects of computational equivalence in semi-groups was previously studied in connection with syntactic monoids by Shyr [41] in reference to minimal congruences on monoids. Computational equivalence on Dyke languages was studied by Buckle [16] where the equivalence classes were identified and a connection between computational equivalence and parsing was discussed.

Using the terminology and wording of [4], let A be an Ω–algebra, and $f \in A$. A pre-order relation $\sqsubseteq_f$ associated with f is defined by h $\sqsubseteq_f$ g ("h may replace g in computing f") if:

"given an Ω–word ω, and elements $a_1, a_2, \ldots, a_n$ in A:

if $\omega(g, a_1, a_2, \ldots, a_n) = f$ then $\omega(h, a_1, a_2, \ldots, a_n) = f$".

The elements g and h are computationally equivalent modulo f ("g $\square_f$ h") if and only if g $\sqsubseteq_f$ h and h $\sqsubseteq_f$ g. The relation $\sqsubseteq_f$ defines a partial order on the equivalence classes of $\square_f$. The replaceability pre-order is identified by the following two lemmas whose proofs are in [4].

Lemma 3.1.1

If $f \in A$, then $\sqsubset_f$ respects the operations in Ω on A: if $\omega \in \Omega$ has arity k and $h_i \sqsubset_f g_i$ then

$$\omega(h_1,h_2,\ldots,h_k) \sqsubset_f \omega(g_1,g_2,\ldots,g_k)$$

Lemma 3.1.2

$\sqsubset_f$ is the unique maximal pre-order relation on A respecting the operations in Ω such that f is minimal (ie. if $g \sqsubset_f f$ then $g = f$).

$\square_f$ is the unique maximal Ω–congruence on A such that no element is equivalent to f.

The specification of arbitrary Ω–words in the definition above can be restricted to Ω–words in which the indeterminate appears only once.

Lemma 3.1.3

If $f,g,h \in A$, an Ω–algebra and $\omega(x,a_1,a_2,\ldots,a_n)$ is an Ω–word such that $\omega(h,a_1,a_2,\ldots,a_n) = f$ and $\omega(g,a_1,a_2,\ldots,a_n) \neq f$ then there exists an Ω–word $\omega'(x,a_1,a_2,\ldots,a_n)$ such that $\omega'(h,a_1,a_2,\ldots,a_n) = f$ and $\omega'(g,a_1,a_2,\ldots,a_n) \neq f$ and the indeterminate x occurs once in $\omega'(x,a_1,a_2,\ldots,a_n)$.

Proof.

Suppose the indeterminate x occurs k times in $\omega(x,a_1,a_2,\ldots,a_n)$. Let ω_i be the Ω–words created by assigning h to the first i occurrences of x in $\omega(x,a_1,a_2,\ldots,a_n)$ and g to the rest. Since $\omega(h,a_1,a_2,\ldots,a_n) = f$ and $\omega(g,a_1,a_2,\ldots,a_n) \neq f$ it follows there exists an i such that $\omega_i = f$ and $\omega_{i+1} \neq f$. Hence let $\omega'(x,a_1,a_2,\ldots,a_n)$ be the word derived from $\omega(x,a_1,a_2,\ldots,a_n)$ with the first i occurrences of x set h, the last k-i-1 set to g and the i^{th} being the indeterminate.

$\square$

This chapter is concerned with replaceability in general finite lattices. To study replaceability in this general setting it is necessary to extend the definitions of prime clause and prime implicant and the concept of duality between them. The generalisations of the results in [4] and [27] give a wider picture to how replaceability fits into the setting of finite modular and general lattices and hence into free distributive and distributive lattices which are special cases of the above.

Section two investigates replaceability in finite lattices in terms of join and meet irreducibles, using the extended concept of duality between join and meet irreducibles defined in chapter two, deriving the generalised result of the closed form for replaceability given in [4] and [27]. Section three gives some examples of replaceability in modular lattices and states some of the differences between replaceability in distributive and non distributive lattices. Section four approaches the problem of replaceability from a more geometric point of view and obtains the same results as section two by considering pairs of covering edges in the Hasse diagram of a lattice. Section five gives an algorithm for determining if two elements of a general lattice are replaceable using the technique developed in section two. Section six gives an alternative description of approximate replaceability in finite free distributive lattices as described by Dunne in [28]. Section seven introduces the notion of saturated elements in distributive lattices and shows their connection with the μ and λ functions as described in [4].

3.2. Computational Equivalence in Finite Lattices in Terms of Join-Irreducibles

In this section, computational equivalence and replaceability in finite lattices is described in terms of sets of join-irreducibles. The results in this section generalise theorems relating to distributive lattices proved in [4], in particular Cor. 3.4. It is first necessary to define sets of join and meet-irreducibles analogous to prime implicants and prime clauses in free distributive lattices. Maximal join-irreducibles less than an element and minimal meet-irreducibles greater than an element naturally determine the element however they fail to capture the computational nature of an element. For example in the non-modular lattice of five elements given in figure 3.5 it is obvious that $a \square_1 c$ even though a is a maximal join-irreducible less than 1. Hence the following definition is based on the computational aspect of an element rather than the join-irreducibles and meet-irreducibles it contains.

Defns. If L is a finite lattice and $f \in L$, then P(f) and Q(f) are defined as

$$P(f) = \{\ p \text{ is a join-irreducible} \mid \exists\, u<f : p \text{ is minimal subject to } u \vee p = f\ \}$$
$$Q(f) = \{\ q \text{ is a meet-irreducible} \mid \exists\, u>f : q \text{ is maximal subject to } u \wedge q = f\ \}$$

By definition, for all $p \in P(f)$ and for all $x < p$ it follows that $x \not\square_f p$, because p is minimal such that $p \vee u = f$ for some $u \in L$. Similarly, for all $q \in Q(f)$, $\forall\, x>q : x \not\square_f q$ The choice of u in the definition of P(f) and Q(f) can be restricted to the elements covered by and covering f respectively. That is, if p is a join-irreducible and x is any element less than p such that $p \vee u = f$ but $x \vee u < f$ for some $u < f$, and u' is any element covered by f such that $u' \geq x \vee u$ then $u' \vee p = f$ since $u' \geq u$ and $u \vee p = f$ but $u' \vee x = u'$. Hence the definitions of $P(f)$ and $Q(f)$ can be given as

$$P(f) = \{ \ p \text{ is a join-irreducible} \mid \exists\ u \text{ covered by } f : p \text{ is minimal subject to } p \not\leq u \ \}$$
$$Q(f) = \{ \ q \text{ is a meet-irreducible} \mid \exists\ u \text{ covering } f : q \text{ is maximal subject to } q \not\geq u \ \}$$

Lemma 3.2.1

If $f \in L$, a finite lattice, then $f = \bigvee P(f) = \bigwedge Q(f)$

Proof.

Choose P such that $\bigvee P = f$ is an irredundant representation for f as a join of join-irreducibles. The lemma is proved by defining a sequence of sets of join-irreducibles $P_0 = P, P_1, \ldots, P_k$ such that $f = \bigvee P_i$ irredundantly for $i \geq 0$, and $P_k \subseteq P(f)$.

Suppose that $P_0, P_1, \ldots, P_i$ have been defined, and that $P_i \not\subseteq P(f)$. Then for some p in P_i, $\exists z < p$ such that $z \vee \bigvee (P_i \setminus \{p\}) = f$. Let $z = \bigvee P'$ be an irredundant representation for z as a join of join-irreducibles, and define P_{i+1} to be a subset of $P' \bigcup P_i \setminus \{p\}$ such that $f = \bigvee P_{i+1}$ irredundantly. Only a finite sequence of subsets $P_0, P_1, \ldots, P_k$ can be generated in this way, since chains of join-irreducibles in a finite lattice have finite length.

A dual argument is used to prove $f = \bigwedge Q(f)$.

□

Define P_f to be the set of maximal elements of $P(f)$, and Q_f to be the set of minimal elements of $Q(f)$. Obviously $\bigvee P_f = \bigvee P(f) = f$. The definition of the set P_f coincides with the definition of prime implicants of a function when the lattice is free distributive. If the lattice is modular, then P_f can be alternatively defined as

$$P_f' = \{ \ p \mid p \text{ is join-irreducible and maximal subject to } \exists\, u<f : u \vee p = f \}$$

To see that the definitions are equivalent it will suffice to show that no element less than $p \in P_f'$ can replace p. Suppose $x \leq p$ and $x \sqsubseteq_f p$; since $p \in P_f'$ there exists $u < f$ such that $u \vee p = f$. Hence $x \vee u = f$ because $x \sqsubseteq_f p$. Therefore $p = p \wedge f = p \wedge (x \vee u) = (p \wedge u) \vee x$ by modularity ($x \leq p$). However $p \wedge u < p$ and p is a join-irreducible, hence $x = p$.

Let $\lessdot_f$ be the pre-order defined on a finite lattice L by:

$$g \lessdot_f h \iff P_f^*[g] \supseteq P_f^*[h] \text{ and } \hat{Q}_f^*[g] \subseteq \hat{Q}_f^*[h]$$

where $f,g,h \in L$. P_f^* is the $^-$ closure of the set P_f described in the last chapter, ($\hat{Q}_f^*$ is the $^-$ closure of the set $\hat{Q}_f$). By theorem 2.3.2 in the chapter two $\lessdot_f$ respects the lattice operations.

Theorem 3.2.2

If $f \in L$, a finite lattice, then $\lessdot_f = \sqsubset_f$

Proof.

Since $\lessdot_f$ is a lattice pre-order it will suffice by lemma 3.1.2 to show that $\lessdot_f \supseteq \sqsubset_f$ and that f is minimal in that $g \lessdot_f f \iff g = f$.

Suppose that $g \lessdot_f f$. If $p \in P_f$, then $p \in P_f^*[f] \subseteq P_f^*[g]$, hence $p \le g$. Thus $f \le g$ since $f = \bigvee P_f$. Let $q \in Q_f$, hence $\forall p \in \hat{q}: p \not\le f$ so $p \in \hat{Q}_f^*[f] \supseteq \hat{Q}_f^*[g]$. Hence $\forall p \in \hat{q}: p \not\le g$ therefore $q \ge g$, thus $\forall q \in Q_f : q \ge g$ and so $g \le f$.

To complete the proof it will be enough to show that $h \not\lessdot_f g$ implies $h \not\sqsubset_f g$. Suppose then that $h \not\lessdot_f g$. There are two possible cases:

Case 1: $\exists p \in P_f^*$ such that $g \ge p$ and $h \not\ge p$

Case 2: $\exists p \in \hat{Q}_f^*$ such that $g \not\ge p$ and $h \ge p$

Case 1. Since $p \in P_f^*$ there exists a sequence of join and meet-irreducibles $p = p_0, q_1, p_2, q_3, \ldots, p_l \in P_f$ such that $p_j \in \hat{q}_{j+1}$ and $q_j \in \hat{p}_{j+1}$, for $1 \le j < l$. Define $w(x)$ to be the lattice word

$$w(x) = (\cdots(((x \wedge p) \vee q_1) \wedge p_2) \vee \cdots \vee q_{l-1}) \wedge p_l$$

From the definition of P_f^* and the argument below, which shows $\omega(g, a_1, a_2, \ldots, a_n) = p_l$ and $\omega(h, a_1, a_2, \ldots, a_n) < p_l$, it follows that $h \not\sqsubset_f g$. Since $g \ge p$ and $p \not\le q_1$ it follows that $(g \wedge p) \vee q_1 > q_1$. Moreover q_1 is maximal subject to $q_1 \not\ge p_2$ because $q_1 \in \hat{p}_2$, hence $((g \wedge p) \vee q_1) \wedge p_2 = p_2$. Thus

$$w(g) = (\cdots((p_2 \vee q_3) \wedge p_4) \vee \cdots \vee q_{l-1}) \wedge p_l$$

It follows by induction that $\omega(g, a_1, a_2, \ldots, a_n) = p_l$. On the other hand $(h \wedge p) \vee q_1 = q_1$ and $(q_1 \wedge p_2) < q_3$ so $(q_1 \wedge p_2) \vee q_3 = q_3$ and

$$w(h) = (\cdots(q_3 \wedge p_4) \vee \cdots \vee q_{i-1}) \wedge p_i$$

hence by induction $w(h) = q_{i-1} \wedge p_i < p_i$

Case 2. In this case let

$$v(x) = (\cdots(((x \wedge p) \vee q_1) \wedge p_2) \vee \cdots) \vee q_{i-1}$$

where $q_1, p_2, \cdots q_{i-1} \in Q_f$ is a sequence of irreducibles such that $p_j \in \hat{q}_{j+1}$ and $q_j \in \hat{p}_{j+1}$, for $1 \leq j < i$. By adapting the argument above it can be shown that $v(h) > q_i$ and $v(g) = q_i$ hence $g \not\sqsubseteq_f h$.

$\square$

As an immediate corollary to Theorem 3.2.2:

Cor. 3.2.3

$g \square_f h$ iff $P_f^*[g] = P_f^*[h]$ and $\hat{Q}_f^*[g] = \hat{Q}_f^*[h]$.

3.3. Examples of Replaceability in Modular Lattices

In this section a few examples of the replaceability pre-order in finite modular lattices are given and some of the differences between replaceability in distributive lattices and finite modular lattices are stated.

Figure 3.1 shows the Hasse diagram for the free modular lattice on three variables (FML(3)).

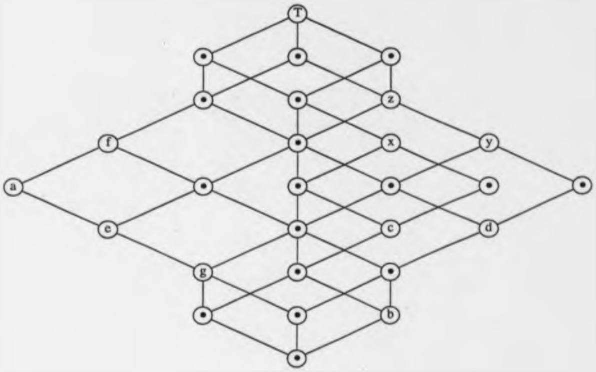


figure 3.1

As can be seen, $P_f = \{a,b\}$, $Q_f = \{f\}$, $P_f^* = \{a,b\}$ and $\dot{Q}_f^* = \{c,d,e\}$. By theorem 3.2.2, $e \not\sqsubset_f g$ since $\dot{Q}_f^*[e] = \{e\}$ while $\dot{Q}_f^*[g] = \{\}$ and by following the proof it is possible to obtain a lattice word which will map g onto f and e not onto f, viz $w(x) = ((x \vee y) \wedge c) \vee f$. The full quotient lattice for FML(3) over $\Box_f$ is given in figure 3.2.

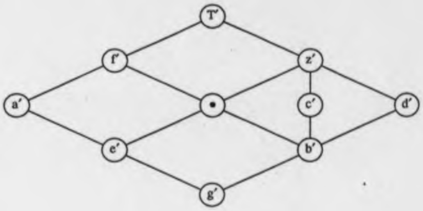


figure 3.2

In distributive lattices the element f is the unique minimal element in the $\sqsubseteq_f$ order, however in non distributive lattices this is not the case as seen in figure 3.3:

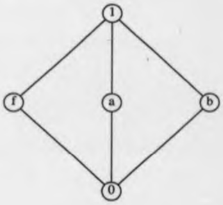


figure 3.3

Here $P_f^* = \{f,b,a\}$ and $\bar{Q}_f^* = \{f,b,a\}$ so every element is computationally distinct and the $\sqsubseteq_f$ order is trivial, for example $((a \wedge a) \vee b) \wedge f = f$ while $((f \wedge a) \vee b) \wedge f = 0$.

Theorem 4.3 of [4] states that computational equivalence in distributive lattices is a retract onto an interval in the lattice associated with the smallest and largest elements that contributes trivially towards a computation. Figure 3.4 shows that computational equivalence in modular lattices is not generally associated with a retract.

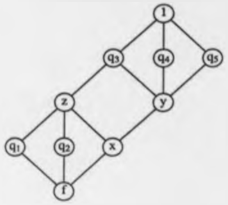


figure 3.4

In this case the only equivalences are $z \ \square_f \ q_3$ and $x \ \square_f \ y$ hence the quotient lattice is not a retract.

3.4. Computational Equivalence and Replaceability in Terms of Covering Edges

In this section, an alternative characterisation of $\square_f$ and $\sqsubseteq_f$ is described. If a,b are two elements of a finite lattice, then $a \succ b$ (a covers b) if $\forall x \le a : x > b \Rightarrow x = a$. When two elements have a covering relationship between them, they are connected by an edge in the Hasse Diagram. In this section it is proved that elements of a finite lattice are computationally equivalent or replaceable if they are connected by a path of special covering edges.

The first part of this section defines a relation between covering edges in the lattice from which a pre-order $\lessdot$ is defined. This pre-order is then shown to be equivalent to replaceability. This leads in particular to an alternative definition of the sets P_f^* and $\check{Q}_f^*$ of the previous section.

Defn. If L is a finite lattice, $a,b \in L$, and $a \succ b$, then the pair (a,b) is called a **covering edge** and is denote by $\langle a,b \rangle$.

A lattice word w is an **alternating word** if it can be expressed as $w(x) = (\cdots((x \wedge z_1) \vee z_2) \wedge \cdots) \wedge z_n$ where $n \ge 0$ and $z_1, z_2, \ldots, z_n$ are lattice elements. (This definition includes alternating words beginning with $\vee$ since it is possible to set $z_1 = 1$.)

If $\langle a,b \rangle$ and $\langle c,d \rangle$ are covering edges, and w is an alternating word such that w(a)=c and w(b)=d, then $\langle a,b \rangle$ **reduces to** $\langle c,d \rangle$ (denoted by $\langle a,b \rangle \twoheadrightarrow \langle c,d \rangle$) **via w.**

Obviously $\twoheadrightarrow$ is a reflexive and transitive relation, however it can be easily shown that in a general lattice it is not symmetric. For instance, in figure 3.5: $\langle b,0\rangle \twoheadrightarrow \langle a,c\rangle$ but $\langle a,c\rangle \not\twoheadrightarrow \langle b,0\rangle$.

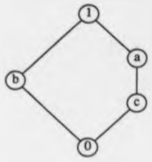


figure 3.5

In a modular lattice L two intervals [b,a] and [d,c] are called **similar** if they can be expressed as $[x, x\vee y]$ and $[x\wedge y, y]$ for some $x,y \in L$. Likewise two intervals [f,e] and [h,g] are called **projective** if there is a sequence of similar intervals connecting them. It is a well known fact that projectivity is an equivalence relation between intervals in a modular lattice (see [39].)

If $\langle a,b\rangle$ and $\langle c,d\rangle$ are covering edges in a modular lattice and $\langle a,b\rangle$ can be reduced to $\langle c,d\rangle$ by an alternating word $w(x)$ comprising of a single operation (ie. $w(x)=x\vee y$ or $w(x)=x\wedge y$ for some element $y \in M$) then the intervals [b,a] and [d,c] can be easily seen to be similar. Hence the definition of projectivity and $\twoheadrightarrow$ for covering edges coincides in modular lattices.

Proposition 3.4.1

In a modular lattice M, the relation $\twoheadrightarrow$ defines an equivalence relation on the covering edges of M.

□

Lemma 3.4.2

Suppose that L is a finite lattice, $c \in L$ and $\langle a,b\rangle, \langle g,h\rangle$ are coverings edge in L. Then:

(1) If there exists x,y such that $a\vee c \geq x \succ y \geq b\vee c$ and $\langle x,y\rangle \twoheadrightarrow \langle g,h\rangle$ then $\langle a,b\rangle \twoheadrightarrow \langle g,h\rangle$.

(2) If there exists x,y such that $a\wedge c \geq x \succ y \geq b\wedge c$ and $\langle x,y\rangle \twoheadrightarrow \langle g,h\rangle$ then $\langle a,b\rangle \twoheadrightarrow \langle g,h\rangle$.

Proof.

(1): Let w be the alternating word $w(z)=((z\vee y)\wedge x)$. Since $a\vee c\geq x$ and $y\geq c$ it follows that $a\vee y\geq x$ and hence $w(a)=x$ and $w(b)=y$. Therefore $<a,b> \twoheadrightarrow <x,y>$ and by transitivity $<a,b> \twoheadrightarrow <g,h>$.

(2) Dually.

□

Defn. Let L be a finite lattice and let $a,b,f \in L$ such that $a \vdash b$. If $<a,b> \twoheadrightarrow <f,z>$ for some $z \in L$ such that $f \vdash z$ then $<a,b>$ is a **non upper-replaceable-edge with respect to f** (ie. b can't replace a). If there doesn't exist such an alternating word then $<a,b>$ is an **upper-replaceable-edge with respect to f.**

If $<a,b> \twoheadrightarrow <y,f>$ for some $y \in L$ such that $y \vdash f$ then $<a,b>$ is a **non lower-replaceable-edge with respect to f.** In the remainder of this section only replaceable edges with respect to f are considered, and the "wrt f" clause is omitted.

Define a pre-order $\lhd_f$ on L such that $a \lhd_f b$ if $\exists n\geq 0$ and a sequence of elements $x_0\,(=a), x_1, x_2, \ldots, x_n\,(=b)$ where *either*

$$x_i \vdash x_{i+1} \text{ and } <x_i,x_{i+1}> \text{ is a lower-replaceable-edge}$$

or

$$x_{i+1} \vdash x_i \text{ and } <x_{i+1},x_i> \text{ is an upper-replaceable-edge}$$

Lemma 3.4.3

If L is a finite lattice and $f \in L$, then the pre-order $\lhd_f$ respects the lattice operations.

Proof.

Suppose $x \lhd_f y$. It will be shown that $\forall a$: $x\wedge a \lhd_f y\wedge a$ and $x\vee a \lhd_f y\vee a$.

Assume $x\wedge a \neq y\wedge a$. Since $x \lhd_f y$ there exists a sequence of elements $z_0\,(=x), z_1, \ldots, z_k\,(=y)$ such that *either* $z_i \vdash z_{i+1}$ and $<z_i,z_{i+1}> \not\twoheadrightarrow <y,f>$ for all $y \vdash f$ *or* $z_{i+1} \vdash z_i$ and $<z_{i+1},z_i> \not\twoheadrightarrow <f,z>$ for all z covered by f.

Consider the sequence $z_0\wedge a, z_1\wedge a, \ldots, z_k\wedge a$. If there exists an *i* such that $z_i\wedge a = z_{i+1}\wedge a$ then remove $z_{i+1}\wedge a$ from the sequence. If $z_i\wedge a > z_{i+1}\wedge a$ but does not cover it then introduce new elements $y_{i1}, y_{i2}, \ldots, y_{ik}$ so that there is a covering chain from $z_i\wedge a$ to $z_{i+1}\wedge a$. By lemma 3.4.2 every edge between $z_i\wedge a$ and $z_{i+1}\wedge a$ is lower-replaceable since $<z_i,z_{i+1}>$ is lower-replaceable. Similarly introduce new elements if necessary for $z_{i+1}\wedge a > z_i\wedge a$. Hence there exists a sequence of elements from $x\wedge a$ to $y\wedge a$ such that $x\wedge a \lessdot_f y\wedge a$

The case for $\vee$ is similar.

□

Theorem 3.4.4

Let L be a finite lattice and let $a,b \in L$, then $a \lessdot_f b$ if and only if $a \sqsubset_f b$.

Proof.

By lemma 3.1.2 and lemma 3.4.3 above it will suffice to show that $\lessdot_f$ contains $\sqsubset_f$ and f is minimal under $\lessdot_f$ (ie. $g \lessdot_f f \Rightarrow g=f$).

By definition of $\lessdot_f$, f is minimal since no edge adjacent to f is lower or upper-replaceable.

Let $a,b \in L$ be such that $a \not\lessdot_f b$; it will be shown that $a \not\sqsubset_f b$. Consider a sequence of elements $a=x_0,x_1,\ldots,x_k=a\wedge b$ and $b=y_0,y_1,\ldots,y_m=a\wedge b$, where $x_i \succ x_{i+1}$ and $y_i \succ y_{i+1}$. Since $a \not\lessdot_f b$ there exists *either* a non lower-replaceable-edge $<x_i,x_{i+1}>$ *or* a non upper-replaceable-edge $<y_i,y_{i+1}>$.

In the former case, since $<x_i,x_{i+1}> \twoheadrightarrow <y,f>$ for some $y \succ f$, there exists an alternating word $w(x)$ such that $w(x_i)=y$ and $w(x_{i+1})=f$. Hence the lattice word $v(x)=w(\,(x\wedge x_i)\vee x_{i+1}\,)$ maps a to y and b to f and $a \not\sqsubset_f b$.

The latter case is dealt with similarly.

□

Defn. Let L be a finite lattice and let $a,b \in L$ such that $a \succ b$. If $<a,b>$ is both an upper-replaceable and a lower-replaceable-edge then $<a,b>$ is called a **collapsible edge**.

Define an equivalence relation O_f by $a\ O_f\ b$ if $a=b$ or there exists a path of collapsible edges from a to b.

Theorem 3.4.5

Let L be a finite lattice and let $a,b \in L$, then $a \; O_f \; b \Leftrightarrow a \; \Box_f \; b$

Proof.

By lemma 3.1.2 it will suffice to show that O_f is a lattice congruence which leaves f solitary and contains $\Box_f$.

The proof that O_f is a congruence is similar to the proof of lemma 3.4.3, and uses the same construction for the new sequence from $x \wedge a$ to $y \wedge a$ and from $x \vee a$ to $y \vee a$.

Obviously O_f leaves f solitary since no edges adjacent to f is collapsible.

Lastly, the proof that $\prec_f$ contains $\sqsubset_f$ in lemma 3.4.4, can be adapted to prove that O_f contains $\Box_f$.

$\Box$

An alternative definition of the sets P_f^* and $\bar{Q}_f^*$ can now be given.

Defn. Let $f \in L$, a finite lattice, and let P be the set of all join-irreducibles.

$$P_f' = \{p \in P \mid \langle p,k \rangle \text{ is a covering edge and is non upper-replaceable}\}$$
$$Q_f' = \{p \in P \mid \langle p,k \rangle \text{ is a covering edge and is non lower-replaceable}\}$$

Theorem 3.4.6

Let f,g,h be elements of a finite lattice L, then

$$g \sqsubset_f h \Leftrightarrow P_f'[g] \supseteq P_f[h] \text{ and } Q_f[g] \subseteq Q_f'[h]$$

Proof.

$\Rightarrow$: Suppose $g \sqsubset_f h$: Since g can replace h there exists a sequence of lattice elements $x_0(=g), x_1, \ldots, x_n(=h)$ such that either $x_i \succ x_{i+1}$ and $\langle x_i, x_{i+1} \rangle \twoheadrightarrow \langle y,f \rangle$ or $x_{i+1} \succ x_i$ and $\langle x_{i+1}, x_i \rangle \twoheadrightarrow \langle f,z \rangle$ for some $y \succ f$ and $f \succ z$. It will be shown that for all i: $P_f'[x_i] \supseteq P_f'[x_{i+1}]$ and $Q_f'[x_i] \subseteq Q_f'[x_{i+1}]$ from which the result follows. Suppose $x_i \succ x_{i+1}$ then $P_f'[x_i] \supseteq P_f[x_{i+1}]$, hence only need to consider the Q_f' set. Let $p \in Q_f'[x_i]$, since $p \in Q_f'$ there exists $k \in L$ and an alternating word $w(x)$ such that $p \succ k$ and $w(p)=y$ and $w(k)=f$ for some $y \succ f$. Since $\langle x_i, x_{i+1} \rangle$ is a lower-replaceable-edge it follows that $x_i \sqsubset_f x_{i+1}$. Let $v(x)$ be the alternating word $v(x) = w(\, (k \vee (p \wedge x)\,)$ then $v(x_i) = y$. If $x_{i+1} \ngeq p$ then $v(x_{i+1}) = f$, contradicting $x_i \sqsubset_f x_{i+1}$, hence $x_{i+1} \geq p$ and $Q_f'[x_i] \subseteq Q_f'[x_{i+1}]$.

The case of $x_{i+1} \vdash x_i$ is dealt with similarly.

$\Leftarrow$: To show that $g \sqsubseteq_f h$ it will suffice to show that there exists a path of upper and lower-replaceable-edges from g to h. Let $c = g \wedge h$, and consider the path formed by two chains of elements $g = x_1, x_2, \ldots, x_k = c = y_j, \ldots, y_2, y_1 = h$ where $x_i \vdash x_{i+1}$ and $y_i \vdash y_{i+1}$. Hence $P_f'[h] = P_f'[c] \subseteq P_f[g]$ and $Q_f'[g] = Q_f'[c] \subseteq Q_f'[h]$. It will be shown that all edges $<x_i, x_{i+1}>$ are lower-replaceable and all the edges $<y_i, y_{i+1}>$ are upper-replaceable and hence $g \sqsubseteq_f h$.

Suppose for a contradiction that the edge $<y_i, y_{i+1}>$ is a non upper-replaceable edge. By lemma 1.2.1 there exists a join-irreducible p and an element x covered by p such that $y_i \geq p$, $y_{i+1} \ngeq p$ but $y_{i+1} \geq x$. Since $p \vee y_{i+1} = y_i$ and $x \vee y_{i+1} = y_{i+1}$ the edge $<p,x>$ is a non upper-replaceable and hence $p \in P_f'$ and $p \in P_f[y_i]$. However $p \notin P_f[y_{i+1}]$ hence $p \notin P_f[c]$, contradicting the fact that $P_f[y] = P_f[c]$.

The proof that every $<x_i, x_{i+1}>$ edge is lower-replaceable is similar.

□

3.5. Algorithm for Determining Replaceability

Algorithm 3.1 can be used to decide if two elements g,h are computational equivalent or replaceable with respect to a third element f in a finite lattice L and is based on theorem 3.2.2. The algorithm requires all the elements to be named and a nxn table of order relations. The algorithm calculates the sets P_f^* and $\hat{Q}_f^*$ in time polynomial in the size of L, to decide whether g and h are computationally equivalent or replaceable modulo f it is only necessary to determine which elements of P_f^* and $\hat{Q}_f^*$ are less than g and h.

The complexity of steps (1) and (2) is $O(n^2)$. Step (5) has complexity $O(p^3+q^3)$. Step (6) has complexity $O(\ nlp + nmq\ )$ where l and m are the number of elements covered and covering f respectively. Since it is only necessary for an irreducible to appear once in the calculation of P_f^*, step (7) has complexity $O(p^3+q^3)$. In a lattice (such as FDL(t)) where the number of irreducibles and the number of elements around f is small compared with the size of the lattice, the complexity of this algorithm is $O(n^2)$, but it may otherwise be $O(n^3)$. This algorithm is of course unreasonable when deciding replaceability for monotone functions since it requires the order relations in FDL(n), indeed Beynon showed in [4] that

Algorithm 3.1:

Algorithm for Calculating the sets P_f^* and $\hat{Q}_f^*$

```
Input: The lattice $L = \{y_1, y_2, \ldots, y_n\}$, together with an $n \times n$ table specifying all order relations, and an
       element $f$ in $L$.
1.     Topologically sort the elements of $L$ so that the elements are in a sequence $x_1, x_2, \ldots, x_n$ such
       that $x_i \geq x_j$ implies $i \geq j$.
2.     For each element $x_i$ check elements $x_1$ to $x_{i-1}$ to see if $x_i$ covers a unique element (and hence is
       a join-irreducible) and form a sequence $p_1, \ldots, p_p$ of join-irreducibles. Similarly check $x_i$ to see if
       it is a meet-irreducible and form a sequence $q_1, \ldots, q_q$.
3.     Find all the elements which f covers and is covered by.
4.     For all join-irreducibles $p_i$ and meet-irreducible $q_j$ set $\hat{p}_i = \hat{q}_j = \phi$.
5.     For $p_i = p_1$ to $p_p$
           For $q_j = q_q$ downto $q_1$
               If $q_j \not\geq p_i$ then
                   If $\forall p \in \hat{q}_j : p_i \not\geq p$ then $\hat{q}_j = \hat{q}_j \cup p_i$
                   If $\forall q \in \hat{p}_i : q_j \not\leq q$ then $\hat{p}_i = \hat{p}_i \cup q_j$
6.     For each covering edge $<p,k>$, where $p$ is a join-irreducible and $p \leq f$, determine whether there
       is an element $u$ covered by $f$ such that $k \leq u$ and $p \not\leq u$; determine $P(f)$ as the set of join-
       irreducibles $p$ for which such a $u$ exists. Similarly obtain the $Q(f)$ set.
7.     Starting with the set $P_0 = P(f)$, repeatedly compute $P_{i+1} = \hat{\hat{P}}_i$ until $P_{k+1} = P_k = P_f^*$ has been com-
       puted. Similarly compute $\hat{Q}_f^*$.
```

Theorem 3.5.2

[Beynon] The decision problem NONREP: *"Given monotone formulae representing f,g,h in FDL(n), is $h \not\leq_f g$?"* is NP-complete.

3.6. Alternative Characterisation of Approximate Replaceability Triples

Pseudo-complements are an important tool in obtaining bounds on circuit size for monotone boolean functions. Pseudo-complements were introduced by Berkowitz [2] where efficient pseudo-complements for slice functions were given. Further research can be found in [48, 27, 4]. In [49] Wegener introduced the concept of an "approximate" replacement. Dunne in [28] gave a formal definition for "approximate pseudo-complement" as:

Defn. Let f be a monotone boolean function with formal arguments $X = \{x_1, x_2, \ldots, x_n\}$. A monotone boolean function h is an approximate pseudo-complement for x_i in any standard circuit S computing f if and only if there exists an (n+1)-argument monotone boolean function R such that $R(f'(X),X) = f(X)$ where $f'(X)$ denotes the function computed by the standard circuit S with $\bar{x}_i$ replaced by h.

Noting that the function $R(f',X)$ can be expressed as $D(X) \vee (C(X) \wedge f'(X))$ for some pair of monotone boolean functions C and D, Dunne introduced the term of a "complementary triple" for any triple $\langle h,D,C \rangle$ which define an approximate replacement for $\bar{x}_i$ in standard circuits computing *f*. In [28] Dunne presented a characterisation of all approximate pseudo-complements by describing the intervals in which h, D and C must lie when one or two of the functions are fixed. In this section an alternative description of the range of triples is given by specifying the intervals in $FDL(n)^3$ for which $\langle h,D,C \rangle$ represent valid triples.

Notation. Let $P(f,h) = \vee\{p \in P_f \mid p \leq h\}$ and $Q(f,h) = \wedge\{q \in Q_f \mid h \leq q\}$.

Lemma 3.6.1

Let f and s be elements of FDL(n). If $P \subset P_f$ and $Q \subseteq Q_f$ then

$$P(f,s) = \vee P \text{ iff } \vee \bar{P} \leq s \leq \wedge \bar{P}$$
$$Q(f,s) = \wedge Q \text{ iff } \vee \bar{Q} \leq s \leq \wedge \bar{Q}$$

where $\bar{P} = P_f \setminus P$ and $\bar{Q} = Q_f \setminus Q$.

Proof.

If $s = \vee \bar{P}$ then $s \not\geq p$ for all $p \in P$, and since $s \geq p'$ for all $p' \in \bar{P}$ it follows that $P(f,s) = \vee P$.

If $s = \wedge \bar{P}$ then $s \not\geq p$ for all $p \in P$. Given that there are no order relations in P_f it follows that $p' \leq \wedge P$ for all $p' \in \bar{P}$ and hence $P(f,s) = \vee P$.

If $s \not\geq \vee \bar{P}$ then there exists $p \in \bar{P}$ such that $p \not\leq s$ hence $P(f,s) \not\leq \vee P$. Similarly if $s \not\leq \vee \bar{P}$ then there exists $p \in P$ such that $\bar{p} \geq s$ hence $p \leq s$ and $P(f,s) \not\geq \vee P$.

The case of $Q(f,s)$ is treated dually.

□

Theorem 3.6.2

$<h,D,C>$ is a complementary triple for the input $x \in X$ if and only if there exists $P_0 \subseteq P_{f^{x=0}}$, $P \subseteq P_f$, $Q_1 \subseteq Q_{f^{x=1}}$ and

$$< \vee P_0, \vee \overline{P_0} \vee \vee \bar{P}, \vee P > \; \leq \; <h,D,C> \; \leq \; < \wedge Q_1, f, \wedge \overline{Q_1} >$$

Proof.

Proof that all elements in the range given are complementary triples follows directly from Theorem 3 in [28].

Suppose $<h,D,C>$ is a complementary triple. Let $P_0 = P_{f^{x=0}}[h]$ and $P = P_f[C]$. By Theorem 3 (ii) in [28] D lies in the interval $[P(f^{x=0},h) \vee P(f,C)\ , f]$. Hence by lemma 1 D lies in the interval $[\vee \overline{P_0} \vee \vee \bar{P}\ , f]$.

Let $Q_1 = Q_{f^{x=1}}[h]$. Hence by Theorem 3 (iii) in [28] C lies in the interval $[P(f,D)\ , Q(f^{x=1},h)]$, hence by lemma 1 $C \leq \wedge \overline{Q_1}$.

$\Box$

3.7. Saturated Elements in Distributive Lattices

Beynon showed in [4] that if f,g,h are elements of a distributive lattice then

$$g \sqsubset_f h \text{ if and only if } P_f[g] \supseteq P_f[h] \text{ and } Q_f[g] \supseteq Q_f[h]$$

where P_f is the set of maximal join-irreducibles smaller than f and Q_f is the set of minimal meet-irreducibles larger than f. In Corollary 3.3 of [4] Beynon defined the elements $z(f)$, $u(f)$, $\mu(f)$ and $\lambda(f)$ as

$$z(f) = \wedge \hat{P}_f,\ \ u(f) = \vee \hat{Q}_f,\ \ \mu(f) = f \vee u(f) = \wedge \{ q \vee \hat{q} \mid q \in Q_f \},\ \ \lambda(f) = f \wedge z(f) = \vee \{ p \wedge \hat{p} \mid p \in P_f \}.$$

and showed for f,h elements of a distributive lattice that

$$0 \sqsubset_f h \text{ iff } h \in [0,z(f)] \text{ and } 1 \sqsubset_f h \text{ iff } h \in [u(f),1]$$
$$0 \Box_f h \text{ iff } h \in [0,\lambda(f)] \text{ and } 1 \Box_f h \text{ iff } h \in [\mu(f),1]$$

This section gives an alternative characterisation of the elements of the form $\lambda(f)$ and $\mu(f)$ which gives greater insight into the structure of the closure lattice $\lambda(D)$ and $\mu(D)$ for a distributive lattice D. This alternative characterisation is used as a means of enumerating the elements of $\mu(D)$ in the section 4.6.

Defn. Let $f \in D$, a distributive lattice, f is $\vee$**-saturated** if for all join-irreducibles $p \in D$ there exists

$p' \in P_f$ such that either $p \leq p'$ or $p \geq p'$. $\wedge$-saturated elements are defined dually.

A more intuitive description of $\vee$-saturated is that it is not possible to introduce new join-irreducibles without them either removing or being absorbed by present join-irreducibles. Hence for any $\vee$-saturated element f there does not exist an element $f' \neq f$ such that $P_{f'} \supseteq P_f$. It will be shown for any distributive lattice D that an element f is $\vee$-saturated if and only if $f \in \mu(D)$. Lemma 3.7.2 shows that the $\vee$-saturated elements of a distributive lattice form a $\wedge$-semi-lattice and proposition 3.7.3 shows that $\mu(q)$ is $\vee$-saturated for any meet-irreducible q, hence $\mu(D)$ is contained in the set of $\vee$-saturated elements. Proposition 3.7.4 proves the converse case that $\mu(D)$ contains the $\vee$-saturated elements of D.

Lemma 3.7.1

Let $f,g \in D$, a distributive lattice and $p \in P_f$. If $p \leq g$ then $p \in P_{f \wedge g}$.

Proof.

Certainly $p \leq f \wedge g$. Suppose there exists a join-irreducible p' such that $p \leq p' \leq f \wedge g$ then $p \leq p' \leq f$ hence $p = p'$ since $p \in P_f$.

□

Lemma 3.7.2

If f,g are $\vee$-saturated then so is $f \wedge g$.

Proof.

Let p be a join-irreducible. There are two cases to consider: either $p \leq f \wedge g$ or with loss of generality $p \nleq f$. If $p \leq f \wedge g$ then the criteria for saturation is satisfied since either $p \in P_{f \wedge g}$ or there exists $p' \in P_{f \wedge g}$ such that $p' \geq p$. Suppose then that $p \nleq f$. Since f is $\vee$-saturated and $p \nleq f$ there exists $f' \in P_f$ such that $p \geq f'$. If $f' \leq g$ then by lemma 3.7.1 it follows that $f' \in P_{f \wedge g}$, hence the criteria for saturation is satisfied. If $f' \nleq g$ then there exists $g' \in P_g$ such that $f' \geq g'$ since g is $\vee$-saturated. Hence $g' \leq f' \leq f$ so again by lemma 3.7.1 it follows that $g' \in P_{f \wedge g}$. Since $g' \leq f' \leq p$ the criteria for saturation is also satisfied in this case.

□

Proposition 3.7.3

If q is a meet-irreducible in a distributive lattice D then $\mu(q)$ is saturated.

Proof.

By definition $\mu(q) = q \vee \hat{q}$. Let p be a join-irreducible, then either $p \leq q \vee \hat{q}$ or $p \not\leq q$ and $p \not\leq \hat{q}$. In the former case there's nothing to prove since either $p \in P_{\mu(q)}$ or there exists a $p' \in P_{\mu(q)}$ such that $p < p'$ so the criteria for saturation is fulfilled. Hence suppose $p \not\leq q$ and $p \not\leq \hat{q}$. Since $p \not\leq q$ it follows that $p \geq \hat{q}$. So all that is required is to show that $\hat{q} \in P_{\mu(q)}$. Suppose there exists a join-irreducible p' such that $\hat{q} < p' \leq \mu(q)$, then $p' \leq q \vee \hat{q}$ and so $p' \leq q$ since p is a join-irreducible, therefore $p' \not\geq \hat{q}$ contradicting the choice of p'. Hence $\hat{q}$ is a maximal join-irreducible less than $\mu(q)$.

□

Proposition 3.7.4

If g is $\vee$-saturated then $g = \mu(h)$ where

$$h = \bigvee \{ \text{ p is a join-irreducible} \mid p \leq g \text{ and } p \in P_g \}$$

Proof.

Claim: $g \geq h \vee u(h) = \mu(h)$, where $u(h) = \bigvee \hat{Q}_h$. Obviously $g \geq h$ so it will suffice to show that $g \geq u(h)$. Let $\hat{q} \in \hat{Q}_h$, hence $\hat{q}$ is a join-irreducible. Since g is $\vee$-saturated it follows that there exists a join-irreducible $p \in P_g$ such that either $\hat{q} \leq p$ or $\hat{q} > p$. If $\hat{q} \leq p$ then $\hat{q} \leq g$. If $p \leq \hat{q}$ it follows that $\hat{p} \leq q$, however by definition of h it follows that $p \not\leq h$ so $\hat{p} \geq h$. Hence $q \geq \hat{p} \geq h$ and since $q \in Q_h$ it follows that $q = \hat{p}$ so $\hat{q} = p \leq g$. Therefore $g \geq \bigvee \hat{Q}_h = u(h)$.

Claim: $g \leq \mu(h)$. Let $p \in P_g$, then $\hat{p} \geq h$ since $p \not\leq h$. Let q be a meet-irreducible such that $\hat{p} \geq q \geq h$, hence $\hat{q} \leq p \leq g$. If $\hat{q} < p$ then $\hat{q} \leq h$ by the definition of h, however $q \geq h$ so $\hat{q} \not\leq h$. Therefore $p = \hat{q}$ and so $\hat{p} \in Q_h$ hence $\hat{Q}_h \supseteq P_g$.

Cor. 3.7.5

The $\vee$-saturated elements of a distributive lattice D are the elements of the closure lattice $\mu(D)$.

Proof.

By definition $\mu(x)$ is the meet of $\mu(q)$ for all $q \in Q_x$. Hence by lemma 3.7.2 and proposition 3.7.3 $\mu(x)$ is $\vee$-saturated. By 3.7.4 any $\vee$-saturated element is $\mu(x)$ for some element $x \in D$.

□

Part Two

Computational Aspects of Lattice Theory

Chapter Four

Technical Aspects of Computation Within Distributive Lattices

4.1. Introduction

For computers to be used for performing calculations in lattices it is necessary to develop methods for dealing with the technical aspects of their implementation. These technical aspects include the storage of lattice elements, their manipulation in expressions and the display of lattices using Hasse diagrams.

Due to the identity between free distributive lattices and monotone boolean functions particular interest lies in performing calculations in free distributive lattices, however their size restrains any attempt at using explicit multiplication tables to perform calculations. Hence it is necessary to use implicit systems based on algebraic rules in the general case. The usefulness of an implementation can be measured in the time and space it requires to store and manipulate elements. Normally these factors can be traded with each other, for example efficiency in performing conjunctions and disjunctions might be offset by large memory overheads.

Hasse diagrams are an important method for displaying small partially ordered systems in an intuitive and clear manner. Unfortunately though Hasse diagrams only display partial orders not lattices. In fact they are ideal for displaying posets since the axioms of a reflexive, transitive and anti-symmetric system are inherent in the diagram. However it is quite hard to prove that a Hasse diagram represents a lattice of any sort, let alone a modular or distributive lattice, since it is necessary to show that every pair of elements possesses a least upper and greatest lower bound. For this reason Hasse diagrams of large lattices are hard to draw by hand and computers are needed to handle the display. By allowing computers to generate the elements of a lattice as well as positioning them it is possible to obtain Hasse diagrams of lattices which can be verified by examining the algorithms used rather than the diagram itself.

This chapter illustrates methods for implementing lattice functions on computers with particular emphasis on free distributive lattices. Arbitrary distributive lattices are too general and tend to require explicit meet and join tables to represent them. Section two illustrates a method of representing and performing calculations on lattice elements from free distributive lattices and introduces the *prime* program which is an important tool in the analysis of finite free distributive lattices. Section three describes the principles behind the *pmc* program for constructing planar monotone circuits for functions in free distributive lattices. Section four addresses the issues of automatic construction of Hasse diagrams by computers. Three different techniques are described for producing diagrams of distributive lattices in two or three dimensions. Section five gives an algorithm for the generation of the elements of a free distributive lattice in disjunctive normal form using bit-strings. By using the methods of section four a planar diagram of FDL(4) and a layered view of a three dimensional diagram of FDL(5) are presented. Section six gives an algorithm for generating saturated elements as described in section 3.7 and by using the methods developed in section four and five displays the μ closure lattice of FDL(5).

4.2. Implementation of Free Distributive Lattice Functions on Computers

The properties of freeness and distributivity allow the elements of a free distributive lattice to be manipulated with greater ease on computers than elements from general lattices. For example in modular lattices elements don't have unique representations as joins of join-irreducibles or meets of meet-irreducibles, and in general distributive lattices it is not sufficient to multiply out conjunctions of joins of join-irreducibles when calculating the meet of two elements. Hence in non-distributive and non-free lattices it is normally necessary to resort to using a representation of the partial order (normally in the form of a Hasse diagram or multiplication table) to calculate the meet and join of elements, thereby restricting the size of lattices with which it is possible to operate. In free distributive lattices however elements can be represented and manipulated algebraically since the elements have unique representations as joins of join-irreducibles and dually and all the variables in the lattice are independent.

There are several ways of representing elements from free distributive lattices on computers. Elements can expressed in a general format of meets and joins or by giving their disjunctive/conjunctive normal form or as a bit-vector over all join-irreducibles in the lattice.

4.2.1. Using a Character Notation

The most straight forward method of representing elements is by storing them as arbitrary meet and joins of the generating variables written in postfix notation (or infix notation using brackets and precedence rules). This system has the advantage that it is quite general and that the conjunction and disjunction of two elements in this form can be obtained easily. However this system has an obvious problem that it is impossible to tell if two expressions represent the same function or perform any more general operations without first transforming the expression into conjunctive or disjunctive normal form.

4.2.2. Using Bit Vectors

Elements of a free distributive lattice can be identified by the join-irreducibles they contain or by the meet-irreducibles that contain them. Normally only the maximal join-irreducibles and minimal meet-irreducibles are used since the others are superfluous for identifying elements. However by recording all the join-irreducibles or meet-irreducibles the operations of calculating the meet and join of elements or the dual of an element can be performed much more quickly as the following definition and proposition show.

Defn. A **lattice bit-vector** of a free distributive lattice on n variables is a 2^n bit-vector where each bit represents a join-irreducible in FDL(n) (including the constant function 1). If V is a lattice bit-vector and p a join-irreducible then **V[p]** is the boolean value of the bit representing p.

The **para-dual** of a join-irreducible p is the dual of $\bar{p}$.

Since $\bar{p}$ is a meet-irreducible and both $\bar{}$ and duality are bijections on the irreducible elements of the lattice it follows that the para-dual of p is also a join-irreducible and that para-duality defines a bijection between join-irreducibles. The **dual** of a lattice bit-vector V is the vector V' where V'[p'] = V[p] and p' is the para-dual of p. The zero and one functions of a lattice are represented in a lattice bit-vector by the all zero and all one vectors respectively. As an example a lattice bit-vector for FDL(3) is given below, here the join-irreducibles are listed in lexicographical order of increasing implicant length (in this example the para-dual of a bit appears in the opposite position, hence the para-dual of this vector is its reversal).

1	a	b	c	ab	ac	bc	abc

figure 4.1

Since there are $2^n!$ possible lattice bit-vectors for FDL(n) it is necessary to select a standard ordering of the join-irreducibles so that lattice bit-vectors of different elements are compatible. From now on it will be assumed that some standard ordering for lattice bit-vectors in FDL(n) has been defined.

Proposition 4.2.2.1

Let g and h be elements of FDL(n) and V_g, V_h be the lattice bit-vector representation of g and h. If $V_g \wedge V_h$ represent the vector obtained by the bitwise-and of the two vectors and $V_g \vee V_h$ represent the bitwise-or of the two vectors then:

(i) the element $g \wedge h$ has the lattice bit-vector $V_g \wedge V_h$.

(ii) the element $g \vee h$ has the lattice bit-vector $V_g \vee V_h$.

(iii) the dual of g is represented by the para-dual of the complement of V_g

(iv) the rank of g (ie. the number of covering edges between g and the zero element) is the number of bits set in V_g.

Proof.

Parts (i) and (ii) follows immediately from the observation that for any join-irreducible p in a distributive lattice,

$$p \leq g \wedge h \text{ iff } p \leq g \text{ and } p \leq h$$

and,

$$p \leq g \vee h \text{ iff } p \leq g \text{ or } p \leq h.$$

Let g′ be the dual of g, so the prime implicants of g′ are the duals of the prime clauses of g and vice versa. If p is a join-irreducible that isn't set in V_g then $p \nleq g$ so $q = \bar{p} \geq g$. Hence the dual of q is less than dual of g. So the para-dual of p is set in $V_{g'}$. By a similar argument the para-duals of the join-irreducibles that are set in V_g are reset in $V_{g'}$. Therefore the lattice bit-vector of g′ is the para-dual of the complement of V_g.

Let k be the number of bits set in V_g and $p_1, \ldots, p_{m-k-1}, p_{m-k}$ be the missing join-irreducibles from V_g in non-decreasing order where $m=2^n$ and p_{m-k} is the constant function 1. The elements $g_i = g_{i-1} \vee p_i$ where $g_0 = g$ form a chain of elements of height at least m-k. Given that $g_{m-k} = 1$ and the function 1 has rank m it follows that the rank of g is at most $m-(m-k)=k$. By using (iii) and a similar argument the dual g' of g has rank at most m-k. However the rank of g' is m minus the rank of g since they are duals, so the rank of g = m - rank of $g' \geq m-(m-k) = k$. Hence the rank of g is k.

□

Since it is possible to calculate the dual of an element when it is presented as a lattice bit-vector it is also possible to calculate the conjunctive normal form of the element given as a vector using join-irreducibles. Hence only a single presentation of the vector is needed rather than two for both join-irreducibles and meet-irreducibles. This system has the obvious failing that the size of the vector grows exponentially with the number of generating variables. However even with this exponential increase it only takes 8 machine words to store elements from FDL(8) which compares well with the method of storing the normal forms of the elements.

The order in which join-irreducibles are arranged in a lattice bit-vector can be defined so that the operations of calculating the dual of a function or the bit-vector of an embedded image in FDL(n+1) of a function in FDL(n) can be performed efficiently on computers. Let $S(n) = (s_1, s_2, \ldots, s_{2^n})$ be an ordered sequence of join-irreducibles of FDL(n) defined recursively by the function

$$S(0) = (\,1\,), \quad S(n) = S(n-1) + \text{para\_dual}(\,S(n-1)\,)$$

where "+" represents the concatenation of ordered tuples and para_dual(S) represents the ordered sequence obtained by taking the para-dual (in FDL(n)) of the elements in the ordered set S. For example the join-irreducibles of FDL(3) would be ordered

$$S(3) = (1, a, ab, b, abc, bc, c, ac)$$

Hence in this arrangement the bit-vector of the dual of a function represented by a bit-vector V is the complement of the vector produced by splitting V in half and swapping the halves around, and the bit-vector of the embedded image in FDL(n+1) of a function in FDL(n) represented by the bit-vector V is obtained by concatenating V with the vector produced by splitting V in half and swapping the halves around.

Example

If $f = b \vee ac$ is an element of FDL(3) then the lattice bit-vector V representing f using the ordering S(3) is (0,0,1,1,1,1,0,1). Hence the lattice bit-vector of the dual of f is the complement of the vector (1,1,0,1)+(0,0,1,1) which is (0,0,1,0,1,1,0,0), which represents the function $ab \vee bc$. To find the bit-vector representation V′ of f in FDL(4) simply concatenate V with the vector produced by exchanging the halves of V, so V′=(0,0,1,1,1,1,0,1)+(1,1,0,1)+(0,0,1,1) which is consistent with the ordering of S(4) below.

$$S(4) = (1,a,ab,b,abc,bc,c,ac,abcd,bcd,cd,acd,d,ad,abd,bd)$$

Due to the ease with which elements can be combined with join-irreducibles and the high efficiency for small lattices (eg. it only uses one machine word for elements in FDL(5)) this system was used to calculate the data for sections four and five.

4.2.3. Using Normal Forms

Representing functions in disjunctive and conjunctive normal forms has the advantage that in a desk calculator environment the normal form of an arbitrary expression is often all that is desired. However the normal forms are duals of each other and it is often found that what is easy or concise in one form is hard or verbose in the other (for example calculating joins of two elements represented in disjunctive over conjunctive normal form). Also both forms normally have to be stored since there is quite a large overhead in converting from one to the other.

Of the three methods listed this is the most appropriate for use in a desk calculator environment since it can store elements from arbitrary large free distributive lattices quite concisely and still manipulate them easily. If for example each prime implicant and prime clause is stored as a bit-vector over all the generating variables then a function like T_8^4 would require 32 machine words to store both the disjunctive and conjunctive forms.

When both disjunctive and conjunctive normal forms are being used it is desirable to make sure that they are treated in exactly the same way so that the duality between meet and join can be exploited to the full. For example the operation of calculating the join of two functions given in CNF is the same as calculating the meet of functions given in DNF, also the algorithm for calculating the μ() and λ() of elements can be duplicated.

4.2.4. Implementation Methods in a Desk Calculator Environment

The ability to calculate normal forms and the z(), u(), λ(), μ() functions of elements in arbitrary free distributive lattices is a great aid in investigating the nature of these lattices. In [15] the author described the "Prime" desk calculator program in which free distributive lattices of up to 20 variables could be analysed and gave algorithms for the computation of z(), u(), λ() and μ() functions.

In *prime* both normal forms are stored as a list of bit-vectors over the generating variables representing the individual prime implicants and clauses. Hence up to 32 generating variables could have been represented by this system on most machines. Figure 4.2 gives a schematic diagram of the representation of the function abvcdevace.

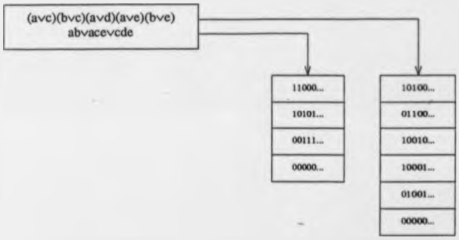


figure 4.2

Since the user normally only requires a few free variables, not all the free variables are required all the time and so only the first few are considered to be in use. The set of variables that are currently in use will be referred to as the current set of variables. Since the clauses and implicants are stored as bit-vectors the ¯ of a clause or implicant is obtained by complementing the bit-vector with the current set of variables.

By using the same system for both normal forms all the procedures involved in their calculation could be used twice because of the duality between meet and join, z() and u(), λ() and μ().

The calculation of the join of two expressions given in DNF simply involved the combining of the two lists of implicants to produce a list that was the size of the sum of the lists. To calculate the meet involved augmenting each implicant in the first expression with each implicant in the second to produce a list that was the size of the product of the sizes. In both the calculation of the join and the meet it was necessary to scan the resulting expressions to remove duplicates and redundancies.

Algorithm 4.1 calculates the z() and u() of elements by using the fact that the disjunctive form of u(x) for some element x is the join of $\acute{Q}_x$ while the conjunctive form of z(x) is the meet of $\acute{P}_x$, hence either of these can be obtained by calculating the necessary normal form and then complementing the list produced. If the DNF of z(x) or the CNF of u(x) is required then the previous result is converted rather than the answer derived directly.

Algorithm 4.1:

Algorithm to calculate u()/z() of an element.

```
uz_function( expression, return_type, function_type )
{
//      Input: expression - the argument value to u() or z().
//             return_type - either CNF or DNF.
//             function_type - either "u" or "z".

        if function_type = "u" then
                calculate the CNF of expression.
        else
                calculate the DNF of expression.

        for each monom/clause do
                complement monom/clause with the current set of variables.

        if (function_type = "u") = (return_type = DNF) then
                return the complemented expression.
        else
                return dual of the complemented expression.
}
```

Given that $\lambda(x) = x \wedge z(x)$ and $\mu(x) = x \vee u(x)$, algorithm 4.2 uses the uz_function procedure in the calculation of λ() and μ() functions. In the case of λ() both the argument x and z(x) are calculated in conjunctive normal form so that $z(x)$ is calculated directly by the previous algorithm and the conjunction of the two expressions can be obtained by combining the lists of clauses. The calculation of $\mu()$ follows a dual line.

Algorithm 4.2:

Algorithm to calculate $\mu()/\lambda()$ of an element.

```
μλ_function( expression, return_type, function_type )
{
//      Input: expression - the argument value to μ() or λ().
//             return_type - either CNF or DNF.
//             function_type - either "μ" or "λ".

        if function_type = "μ" then
               calculate the DNF of expression,
               calculate the DNF of u( expression ).
        else
               calculate the CNF of expression,
               calculate the CNF of z( expression ).

        Combine the two expressions together removing duplicates.
               // Ie. calculate the meet in the λ
               // case and join in the μ case.

        if (function_type = "μ") = (return_type = DNF) then
               return the combine expression.
        else
               return dual of the combine expression.
}
```

4.3. Planar Monotone Computation

In [10] Beynon and Buckle described a criterion for determining if a monotone boolean function is planar computable from a given sequence of inputs and outlined an algorithm for constructing planar monotone circuits when they exist. The criterion and the algorithm were based on the replaceability results of [4] and [26] and proceeded by constructing local sub-circuits which constantly "improves the input" until either the function had been computed or no further improvement could be done.

The operation and verification of the algorithm is greatly simplified by using special sub-circuits called $\vee$-*bridge pyramids* and $\wedge$-*bridge pyramids*. Bridge pyramids are k input, k-2 output planar circuits that are used to improve the middle k-2 inputs by introducing new prime implicants or prime clauses. By using bridge pyramids to construct boolean functions the number of *active* gates that need to be considered at any one time can be restricted to the number of inputs. As the construction proceeds the number of active gates reduces as computational inferior gates are superseded by their neighbours. An example of an ∨-bridge pyramid is given in figure 4.3, here it is shown how the prime implicant p is introduced to the middle gates while the prime clause q is left unaffected. Necessary and sufficient conditions for bridge pyramids to construct planar circuits can be found in [10] and will not be repeated here.

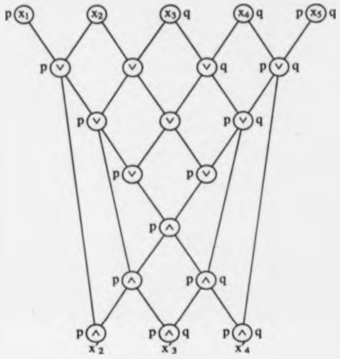


figure 4.3

The bridge pyramid can be seen as a two stage circuit in which the initial truncated pyramid unites separated components while the second pyramid restores the existing components. It is possible to extend bridge pyramids so that they unite several components that once. In this case the first stage should consist of several overlapping truncated pyramids and in the second stage the output replacing the input x_i should be the result of a pyramid whose base is the outer gates of the first pyramid affected by x_i. Figure 4.4 shows a schematic diagram of two components from a to b and c to d being united, with only one output pyramid drawn.

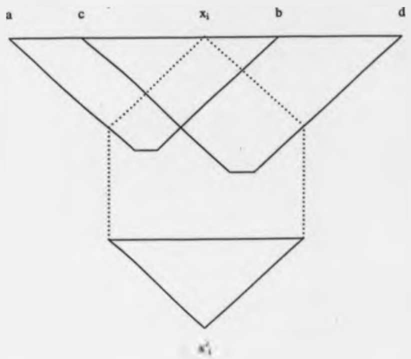


figure 4.4

The development of the criterion and algorithm was done experimentally by testing various constructing programs that used *prime* as a front end to generate the necessary disjunctive and conjunctive forms. All the constructing programs used a hybrid system of normal forms and bit-vectors to handle monotone functions, where partial functions were represented as bit-vectors over the prime implicants and clauses of the specified function. The progress of the algorithm as it runs is stored as two bit-tables of prime implicants and clauses against the active gates, where a bit is set if the gate is less than a prime clause or greater than a prime implicant. The operation of or'ing two gates involved performing a bitwise *or* on the implicant table and an *and* on the clause table, and'ing two gates has the dual effect. The algorithm terminates when one of the gates has a full line of bits set on both the implicant and clause tables or when no more constructions can be performed.

In [10] the term *persistent configuration* was used to refer to an arrangement of prime implicants and clauses in which it was impossible to unite any components without deleting some, hence never being able to construct a planar circuit. In the study of persistent configuration it is desirable to be able to construct functions which have specific prime implicants and clauses and the following proposition shows that this is possible in any distributive lattice.

Proposition 4.3.1

Let D be a finite distributive lattice and P and Q sets of non-comparable join-irreducibles and meet-irreducibles respectively such that $\vee P \leq \wedge Q$. Let

$$X = \bigcup_{q \in Q} \{ q' \mid q' \text{ is a maximal meet-irreducible} < q \}$$

$$Y = \bigcup_{p \in P} \{ p' \mid p' \text{ is a minimal join-irreducible} > p \}$$

If $f \in D$ then

$$\vee P \vee \vee \check{X} \leq f \leq \wedge Q \wedge \wedge \check{Y}$$

if and only if $P_f \supseteq P$ and $Q_f \supseteq Q$.

Proof.

Let f be any function such that $P_f \supseteq P$ and $Q_f \supseteq Q$. Since $f \not\leq x$ for all $x \in X$ it follows that $f \geq \check{x}$, hence $f \geq \vee P \vee \vee \check{X}$. Similarly $f \leq \wedge Q \wedge \wedge \check{Y}$. Let g be any function in the interval and let $p \in P_g$ such that $p \geq p' \in P$. Since $p \leq g \leq \wedge \check{Y}$ it follows that $p \leq \check{y}$ for all $y \in Y$, hence $p \not\geq y$. However Y contains the minimal join-irreducibles greater than p', so $p = p'$ and P is a set of maximal join-irreducibles smaller than g. A dual argument shows $Q_g \supseteq Q$.

□

The minimum and maximal elements of the interval are calculated by *prime* by algorithm 4.3.

4.4. Generation of Hasse Diagrams for Distributive Lattices

The process of drawing Hasse diagrams of large lattices can be divided into three stages. The initial stage is the calculation of the elements of the lattice including information of the partial order. From this stage it should be possible to find at what level each element must be placed and be able to determine covering relationships. The second step involves the calculation of the position of the elements of the lattice. At this point a virtual diagram should be derivable where the elements have been positioned using an appropriate notation however no fixed coordinates have been determined. For example the position of the elements

Algorithm 4.3:

Algorithm to calculate minimum/maximum range elements.

```
xy_function( expr_imp, expr_cla, return_type, function_type )
{
//      Input: expr_imp, - expression containing the implicants to be used
//             expr_cla, - expression containing the clauses to be used
//             return_type - either CNF or DNF.
//             function_type - either ''min'' or ''max''.

        if function_type = ''min'' then
                given = DNF of expr_imp
                extra = CNF of expr_cla
        else
                given = CNF of expr_cla
                extra = DNF of expr_imp

        for all monom/clause e ∈ extra do
                // Find the minimal monoms greater the e or the maximal
                // clauses less than e by removing variables from e.
                for all variables v ∈ e do
                        e' = e − v
                        // Add the complements to the list of given
                        // monoms/clauses (hence performing a join/meet).
                        given = given + complement of e'

        remove duplicates from given

        if (function_type = ''min'') = (return_type = DNF) then
                return given
        else
                return dual of given
}
```

might be specified relative to other points through a chain of dependencies or the elements placed on concentric rings. The last step is the display of the elements on a suitable terminal device. Here the positioning techniques used in the previous step must be converted into real coordinates.

4.4.1. Positioning of Elements

Algorithms for positioning lattice elements in Hasse diagrams should high light the natural structure of the lattice as much as possible. This involves on the immediate level the placing of pairs of covering elements close to each other and on a higher level the organisation of sub-lattices (for example the central core of the boolean sub-lattices in the free distributive lattices). Also elements in the same conjugacy class (ie. those elements which can be mapped onto one another by a permutation of the generating variables) should be displayed in a similar fashion.

The result of this stage should be a definitive representation of the elements indicating how they are to be presented relative to each other. This representation could be simply a left-right ordering of points of the same rank or a complex list of dependencies representing the lattice as a collection of sub-lattices. In the following such an arrangement is called an *ordering* of the elements.

Three methods are given below for obtaining a definitive presentation of the lattice. The first method identifies major sub-lattices and structures and builds the diagram around them. The second method relies on covering relationships between elements. The third method positions the elements according to their disjunctive normal form presentation. In practice a combination of all three methods would be used.

4.4.1.1. Construction via Sub-lattices

Distributive lattices can be easily divided into small boolean sublattices and chains and these can be displayed in a standard fashion hence emphasising the internal structure of the lattice. Unfortunately several of these smaller sub-lattices intersect and it is necessary to decided which sub-lattices get displayed clearly while others get distorted, thereby reducing the usefulness of this method.

In the case of free distributive lattices a similar technique can be used where the lattice is partitioned into three classes comprising of the central boolean sub-lattices, the elements comparable with a generating variable (not in the first class) and all the other elements. These three classes can be further sub-divided into smaller boolean lattices. This is a useful first step in defining a diagram since it distinguishes the three main classes of elements and suggests a general diagram of the form shown in figure 4.5.

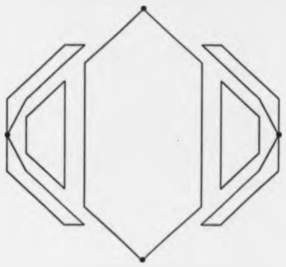


figure 4.5

4.4.1.2. Construction via Covering Edges

Since one the main features of a Hasse diagram is the display of the covering edges, a sensible method of arranging the elements would be by the position of the elements they cover or are covered by. This method requires an initial ordering of a non-trivial row (or perhaps an inner boolean sub-lattice which is quite easy to display) from which the ordering of subsequent levels of the lattice can be derived. However this method sometimes fails to resolve a set of elements all of whom should be placed at the same point. For example the position of the points a, b and c in figure 4.6 can not be determined from the present ordering of the upper level. In this case it is necessary to use a heuristic measure to order the set.

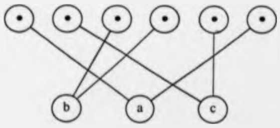


figure 4.6

4.4.1.3. Construction via Normal Form

The disjunctive or conjunctive normal form of an element can be used as a means to arrange the elements. In the disjunctive case every join-irreducible is given a weight and each element is attributed the value according to the sum of the weights of its join-irreducibles. Each level of the lattice is then ordered according to the weights of the elements.

To minimise the effect of individual weights on the ordering of the elements the elements should be partitioned into conjugacy classes before being ordered by weight. Once each class has been ordered the classes can then be split into two and be bracketed around each other. To emphasise the structure of the lattice and show the symmetry inside conjugacy classes the weights should be assigned so that the weight of a join-irreducible is much greater than the weight of any join-irreducible it contains and that the weights of all the join-irreducibles of the same rank should be balanced.

This method is especially suited when there are an odd number of variables and the weights are given as polar coordinates, such as in FDL(3) and FDL(5). In this case all join-irreducibles of the same rank can be assigned the same radius and equally spaced along a ring. By assigning weights in diminishing order as suggested above, elements of a conjugacy class appear as concentric rings around their dominate join-irreducible.

Obviously diagrams produced by this method will not be as clear as the method described in the last section since there is no direct connection between where an element is placed and the elements it covers.

4.4.2. Display of the Diagram

Once a general ordering has been generated, producing a diagram is normally quite straight forward and can be viewed as an arithmetic task. However a more sophisticated approach would be to use the ordering information so that modifications to the diagram can be redisplayed interactively.

To aid such a display system it is necessary that the display routine is given a *parametrised* diagram. In such a diagram notation the points are not given as absolute coordinates but by expressions in terms of other points. Hence the information needed to redisplay the diagram when a modification is done is available. For example a diagram might be specified by listing the major sub-lattices, each of which is then further divided until finally the individual points are expressed. Similarly if the ordering was determined by the covering relationships between elements then by moving one element the display of the whole lattice could be adjusted to balance the change.

Such a display system is much more appropriate for the display of lattices since it allows them to be used as highly sophisticated tools in the analysis of lattices since they are endowed with the structure of the lattice, not just its shape.

4.5. Hasse Diagrams of FDL(4) and FDL(5)

4.5.1. Construction of Free Distributive Lattices

Various algorithms have been published for the generation of the elements of a free distributive lattice, normally as a means of determining the size of the lattice. Dedekind in [25] first proposed the problem of determining the order of free distributive lattices and proved that FDL(4) has 166 non-constant elements. Later in 1940 Church [22] and in 1946 Ward [46] published respectively the sizes of FDL(5) and FDL(6) as 7579 and 7828352. Church calculated the order of FDL(5) by partitioning the elements into permutation classes and calculating the size of each class. All the calculations were performed by hand in 1936. Ward calculated the order of FDL(6) using a computer. He indicated that the method could be extended to FDL(7) but warned that this would be "prohibitively laborious".

In 1968 Czyzo and Mostowski [24] published an algorithm for constructing free distributive lattices and confirmed the results of Church and Ward. In their algorithm they represented elements of the lattice as bit-vectors of size 2^n. The bit-vector stored the values of the function under all the 2^n assignments to the free variables. The algorithm involved generating the elements of FDL(n-1) recursively which were then combined in pairs to produce the elements of FDL(n). As can be seen the arrangement of data in this algorithm is similar to that described in section three, however the terminology used in the previous section is better suited here since it eases the proof of Proposition 4.2.2.1. This algorithm has the unfortunate property that it requires large amounts of memory to run. Since it is necessary to store the points of FDL(n-1) to calculate FDL(n) it requires memory to store a number of bit-vectors which grow super exponentially with n. The algorithm took 22 hours to calculate the size of FDL(6). At the end of the paper

Czyzo and Mostowski hoped that one day the size of FDL(7) would be calculated using this system on a multi-processor computer.

In 1988 Kisielewicz [35] published a direct method of calculating the order of free distributive lattices using the fact that the order of the lattice equals the number of anti-chains in the boolean lattice on the same number of generators. Unfortunately the formula given involves a summation from 1 to 2^{2^n} over a product from 1 to 2^n.

Algorithm 4.4 enumerates the elements of FDL(n) in a form that could be used to construct a Hasse diagram. As well as calculating the names of the elements it is necessary to determine their ranks, order relations and their general location in the lattice (ie. is the element in one of the central boolean sub-lattices or directly comparable with a free variable).

The algorithm proceeds by recursively joining a lattice bit-vector representing a function whose disjunctive normal form contains only prime implicants of length less than j with join-irreducibles of length j. At each stage in the recursion all possible combinations of join-irreducibles of length j (including none at all) are used, hence every monotone function is produced. The algorithm is initiated by the call **generate(zero, 1)** where zero is the lattice bit-vector of the constant function 0.

If only the size of the lattice is required then it is possible to accelerate the algorithm by cutting off the search when the number of possible elements derivable from a lattice bit-vector is obvious. This occurs when the implicants in the next level of the recursion are all set, hence the number of possible functions that can be obtained is 2^x where x is the number of reset implicants in the current level. An implementation of an algorithm using this pruning technique based on a multi-user single processor system required 35 seconds of CPU time to calculate the size of FDL(6).

4.5.2. Hasse Diagram of FDL(4)

A simple analysis of FDL(4) reveals that it has 166 non-constant elements situated on 15 rows. Due to its small size and that, as in all free distributive lattices, only half of it needs to be drawn it is possible to order each row according to the covering relationships between elements. Such a diagram is given in figure 4.7. The only problem points are the elements i,j,k in the seventh and ninth rows, these elements have a

Algorithm 4.4:

Algorithm to Generate the Elements of a Free Distributive Lattice

```
generator( level, vector )
{
//      Input: level  - length of join-irreducibles to be included this level
//             vector         - the function under construction
//      Local:        S       - a queue of implicants

        if level = n+1 then output( vector )
        else {
               for all implicants p of length ''level'' do
                      if vector[p] = false then
                             add_to_queue( S, p )

               enumerate( vector, level, S )
}

enumerate( vector, level, S )
{
//      Generate all combinations of items from S
//      Input: vector         - the function to place implicants just selected
//             S       - queue of implicants that are reset in vector

        generator( level+1, vector )

        while S is non empty do
               implicant = get_from_queue( S )
               copy_vector = vector v implicant
               enumerate( copy_vector, level, S )
}
```

symmetrical relationship to the elements in the sixth and tenth rows respectively. However they can be resolved by ordering the eighth (middle) row first.

The diagram in figure 4.8 was produced by the method described in section 4.4.3 using the following weights:

a	-20000	ab	-2020	abc	-101	abcd	0
b	-10100	ac	-1000	abd	-50		
c	10000	ad	505	acd	51		
d	20100	bc	-500	bcd	100		
		bd	1015				
		cd	2000				

Some of the weights have a slight offset so that the problem points of the last diagram are resolved automatically.

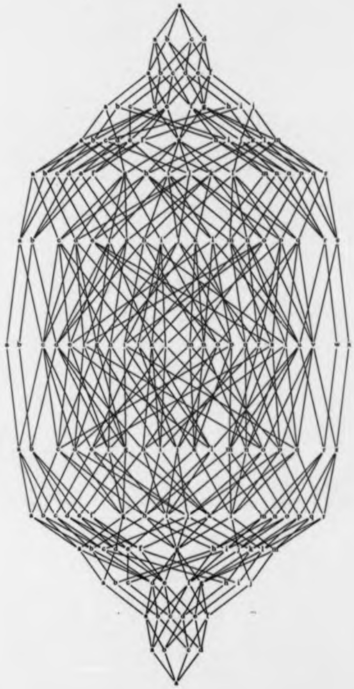


figure 4.7

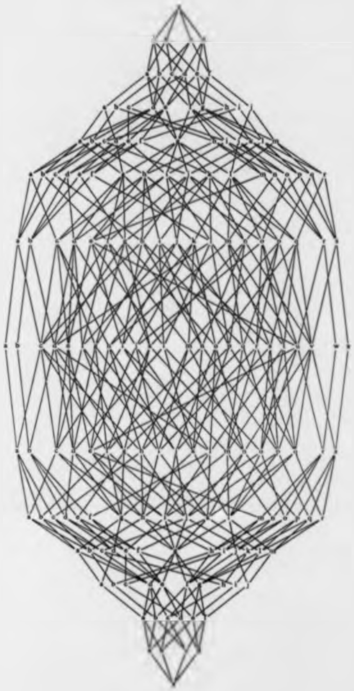


figure 4.8

4.5.3. Hasse Diagram of FDL(5)

The structure of FDL(5) is considerable more complex than that of FDL(4) and does not permit an easy construction of the Hasse diagram by studying covering edges. However since 5 is prime (or more precisely 5 divides ^nC_i for i between 1 and 4) it is possible to arrange all the monoms of FDL(5) symmetrically on a plane as shown in figure 4.9:

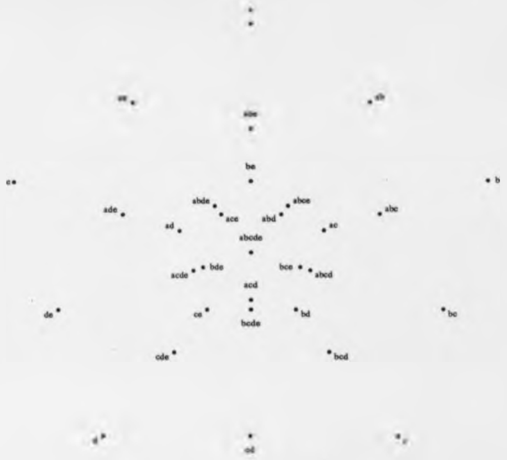


figure 4.9

Each monom is located at the centre of gravity of the variables it implies rather than having all monoms of the same length having the same radius. By doing this the number of elements in FDL(5) that are given the same weight is reduced. However after this adjustment there are still too many collisions caused by the fact that there are many lines of symmetry in the diagram. Hence to reduce the number of elements being given the same weight the radius of some of rings are adjusted to break the lines of symmetry. Therefore by assigning weights to the monoms in approximate accordance to the diagram it is possible to derive a three dimensional Hasse diagram of FDL(5) in which all the generating variables are placed symmetrically in the diagram, unlike the case of a two dimension diagram where some variables occur on the outside of the diagram, others on the inside etc.

Analysis of FDL(5) reveals that it has 7579 non-constant elements which are divided into 2109 elements in the central boolean sub-lattices, 1305 elements outside of them that are comparable with a generating variable and the other 4165 form a torus between levels 10 to 22. This gives rise to a vertical section view of FDL(5):

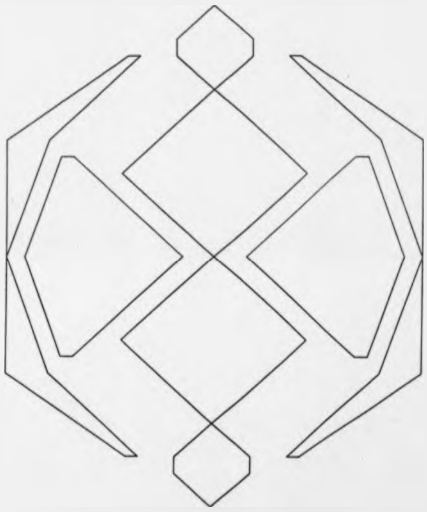


figure 4.10

A view of levels 1 to 16 of FDL(5) is given in diagram figure 4.11. Levels 17 to 31 are the reflection of these levels.

4.6. Hasse Diagram of the Closure Lattice $\mu(FDL(5))$

The closure lattice $\mu(FDL(4))$ was first given in [3] where some of the properties of these closure lattices were given. While it may be easy to calculate $\mu(x)$ for any point $x \in FDL(n)$ or determine if $x = \mu(y)$ for some y using the algorithm in section three there is no direct way of enumerating the elements of $\mu(FDL(n))$ using the definition of $\mu()$. In the case of FDL(4) the identification of elements of $\mu(FDL(4))$ is relatively straight forward since there are only 27 of them, however in larger lattices this direct approach is not possible. By using the equivalent definition of $\vee$-saturated however it is possible to enumerate the elements of $\mu(FDL(n))$ directly since the definition of saturation specifies a characteristic property of the elements rather than a function of the lattice.

Defn. Let V be a lattice bit-vector for FDL(n). A join-irreducible $p \in FDL(n)$ is **unaffected in V** if $V[p]$ is false and $V[p']$ is true for all join-irreducibles $p' < p$. A join-irreducible p is **maximal in V** if $V[p']$ is false for all join-irreducibles $p' > p$.

Let $f \in FDL(n)$ and let f_i be the functions whose prime implicants are the prime implicants of f of length i or less and let V_i be lattice bit-vectors representing f_i. If p is a join-irreducible of length k and is unaffected in V_k then f is not $\vee$-saturated. This follows since $V_k[p]$ is false, and the inclusion of longer prime implicants in $f_{k+1}, f_{k+2}, \ldots$ has no affect on $V[p]$ since $V_k[p']$ is true for all join-irreducibles $p' \leq p$. Hence $p \nleq f$ and there does not exist $p' \in P_f$ such that $p \geq p'$.

Lemma 4.6.1

If $g \in FDL(n)$ and V_g is a lattice bit-vector representing g then g is $\vee$-saturated if and only if for all join-irreducibles p for which $V_g[p]$ is false there exists a join-irreducible $p' < p$ such that p' is maximal in V_g.

Proof.

Obvious from the definition of $\vee$-saturated.

□

Algorithm 4.5 uses a similar method to enumerate the $\vee$-saturated elements in a free distributive lattice as algorithm 4.4. It proceeds by generating all possible derivations of a lattice bit-vector except that at each stage of the recursion all unaffected join-irreducibles of the current level in the bit-vector are set, thereby at the end of the recursion the hypothesis of lemma 4.6.1 is met. It should be noted that if p is a join-irreducible that is unaffected in V then the only action resulting in joining p with V is to make V[p] true, all other bits remain the same.

Let MON_i $(0 \le i \le n)$ be the set of join-irreducibles of length i and $MON_{i,j}$ $(1 \le j \le c_i = {}^nC_i)$ be an arbitrary ordering of the join-irreducibles of length i. The algorithm is initiated by the call **generator(zero, 1)** where zero is the lattice bit-vector of the zero element.

Proposition 4.6.2

The procedure *generator()* in algorithm 4.5 enumerates the $\vee$-saturated elements in FDL(n).

Proof.

The proof will be in three parts. First the proof that the algorithm lists only saturated elements, second the proof that all saturated elements are produced and finally that the elements are only listed once.

(1) Let g be an element listed by *generator()* whose lattice bit-vector is V_g and V_i be the lattice bit-vector given to *generator()* as a parameter when level = i (ie. V_i represents a function whose prime implicants have length less than i). Suppose p is a join-irreducible such that $V_g[p]$ is false and let k be the length of p. Since $V_g[p]$ is false another recursive call to *generator()* must have been made otherwise the p would have been included at line 7. Before either recursive call in lines 5 and 23 all join-irreducibles of length k are tested to see if they are unaffected in V. Since $V_g[p]$ is false it follows that there exists a join-irreducible p′ of length k+1 such that $p > p'$ and $V_k[p']$ is false. Since only unaffected join-irreducibles are included in lines 19-21 and 2-3 it follows that $V_{k+1}[p']$ is also false. If $V_g[p']$ is true then there exists a maximal join-irreducible less than p and hence the hypothesis of lemma 4.6.1 is fulfilled. If $V_g[p']$ is false then by repeating the argument it can be shown that there exists a join-irreducible p″ such that $p > p' > p''$

Algorithm 4.5:

Algorithm to Generate the Saturated Elements of FDL(n)

```
generator( V, level )
{
//      Input: level  - length of join-irreducibles to be included this level
//             V      - lattice bit-vector of the function under construction

1.      if level < n then
2.          for all p in $MON_{level}$ do
3.              if p is unaffected in V then $V = V \vee p$

4.          if $\exists p \in MON_{level+1}$ :  V[p] is false then
5.              generator( V, level+1 )
6.              enumerate( V, level, 1 )

7.      for all p in $MON_{level}$ do $V = V \vee p$
8.      output( V )
}

enumerate( V, level, offset )
{
//      Input: level  - length of join-irreducibles to be included this level
//             V      - lattice bit-vector of the function under construction
//             offset - start number of implicants to use

9.      found = false
10.     for i = offset to $c_{level}$ do
11.         if V[ $MON_{level,i}$ ] is false then
12.             found = true
13.             exit for loop

14.     if not found then return

15.     enumerate( V, level, i+1 )
16.     $V = V \vee MON_{level,i}$

17.     for j = 1 to i-1 do
18.         if $MON_{level,j}$ is unaffected in V then return

19.     for j = i+1 to $c_{level}$ do
20.         if $MON_{level,j}$ is unaffected in V then
21.             $V = V \vee MON_{level,j}$

22.     if $\exists p \in MON_{level+1}$ :  V[p] is false then
23.         generator( V, level+1 )
24.         enumerate( V, level, i+1 )
}
```

where p^{*} is a maximal join-irreducible in V_g.

(2) Let g be a saturated element whose longest prime implicant has length r and V_g its lattice bit-vector. For $1 \le k \le r$ let V_k be the lattice bit-vector of the function comprising of the prime implicants of g whose length is less than k. It will be shown by induction that there is a call generate(V_k, k) for $1 \le k \le r$ and hence g will be the result of generate(V_r, r).

Given that the zero function is the initial call to *generator()* the base case is obviously satisfied.

Assume by induction that there is call to *generator()* of the form generate(V_k, k) where $k < r \le n$. If $p \in MON_k$ and p is unaffected in V_k then $V_g[p]$ must be true since g is $\vee$-saturated. Hence lines 2-3 only include join-irreducibles which are less than g. Moreover since V_k and V_g agree up to join-irreducibles of length k-1 and that g has prime implicants of length greater than k it follows that the test in line 4 is true and that lines 5 and 6 are executed.

Let $P = \{p_1, p_2, \ldots, p_m\} \subseteq MON_k$ be the set of join-irreducibles for which $V_k[p_i]$ is false after line 3 and let $t_i = V_g[p_i]$. Without loss of generality assume that the indices in P are in the same order as MON_k. If t_i is false for all i then the call to *generator()* in line 5 will be of the form generate(V_{k+1}, k+1). Hence assume that some of the t_i are true.

Let T be the subset of P for which t_i is true and $p_s \in P \setminus T$. Since $p_s \le g$ and g is $\vee$-saturated it follows that there exists $p' \in P_g$ such that $p_s > p'$. Hence p_s is not unaffected in V_k joined with all the join-irreducibles in T. Hence by following the recursive path produced by taking *enumerator()* on line 15 if t_i is false and *enumerator()* on line 24 otherwise it is clear the V_{k+1} will be produced as soon as the last p_i from T is joined of V at line 16. Hence there is a call to *generator()* of the form generator(V_{k+1}, k+1).

(3) It will be shown for each call of generator(V, r) that the pair (V,r) is unique. Hence the inclusion of join-irreducibles of length r produces a unique output. The proof will be by induction on r.

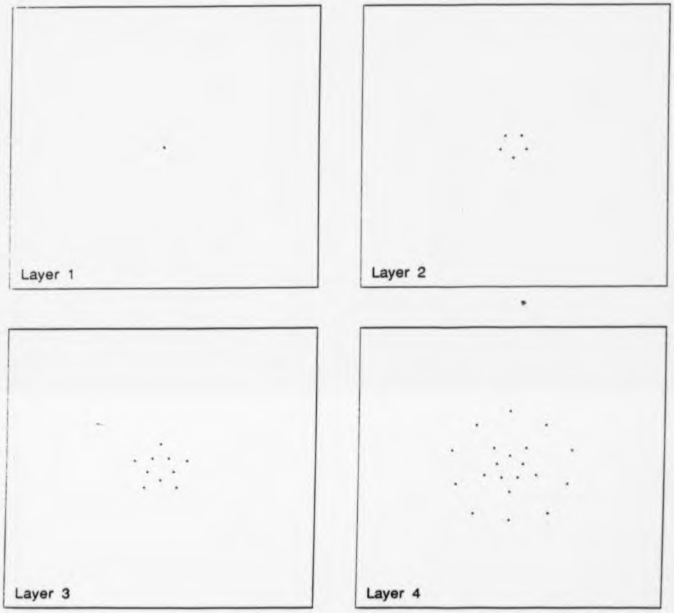
Given that there is only one insistence of *generator()* at level 1 the base case is trivially satisfied.

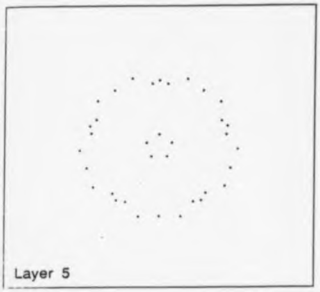
By induction assume the (V,r) is a unique pair of a lattice bit-vector and a level. Lines 5,6,15,16 form a binary counter, lines 19-21 make sure that any combinations produced by the counter are consistent with saturation and lines 17-18 make sure the lines 19-21 do not produce repeats. Hence the binary counter will produce distinct lattice bit-vectors from V. Since V is unique up to level r-1, the new bit-vectors are unique up to level r.

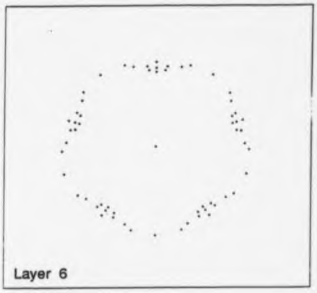
□

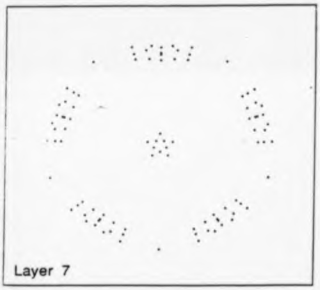
Once the saturated points have been obtained a three dimensional diagram of μ(FDL(5)) can be easily obtained by topologically sorting the points and selecting the coordinates from the Hasse diagram of FDL(5). Since μ(FDL(5)) is considerably less complex than FDL(5) it is possible to transform the three dimensional diagram into a planar one by slicing the rings which compose the diagram of FDL(5). The resulting picture is shown in figure 4.12. In this diagram points in red would lie in the central boolean lattices, points in blue are comparable with a free variable and other points are in green.

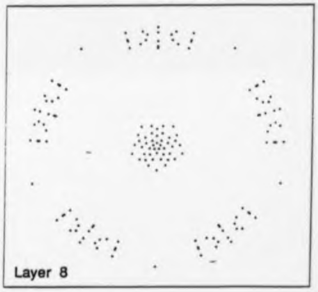
As noted in [3] the sublattice K generated by $\mu(x_1), \mu(x_2), \ldots, \mu(x_{n-1})$ is isomorphic to FDL(n-1) and the map $\alpha : K \to FDL(n-1)$ mapping w to [illegible] is an isomorphism. To identify the points in μ(FDL(5)) which belong to the sublattice generated by the four outer images of the generating variables it is simply necessary to determine which points in μ(FDL(5)) are height invariant under α. These points have been high-lighted in figure 4.12.

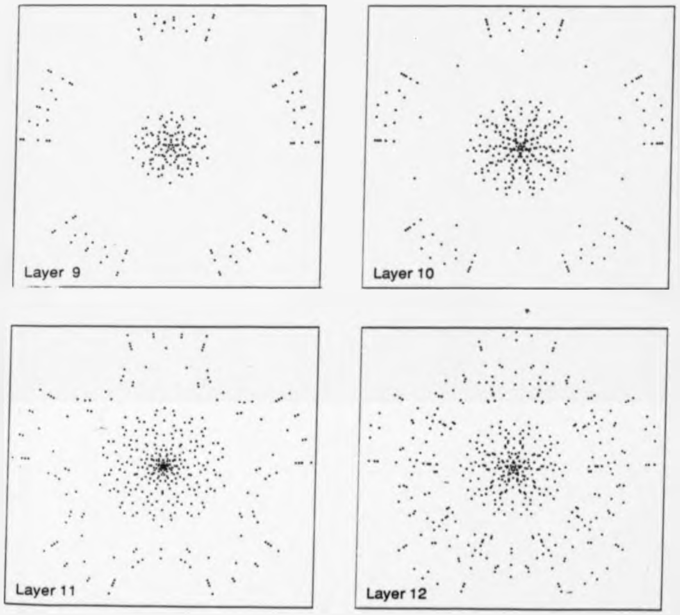

Layer 1
Layer 2
Layer 3
Layer 4

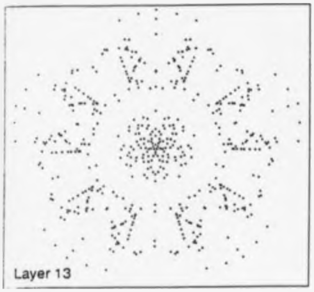

Layer 5

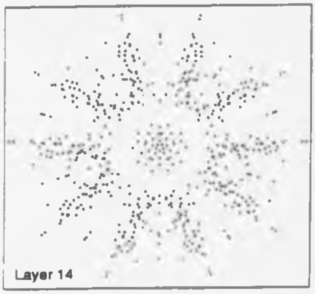

Layer 6

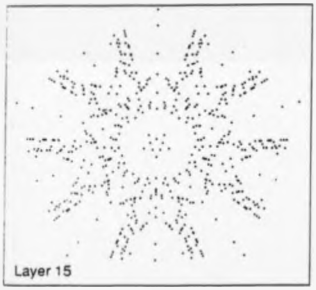

Layer 7

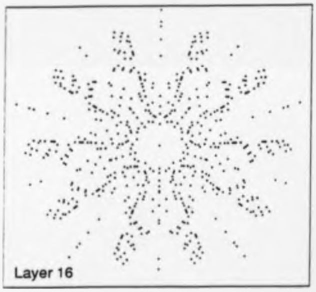

Layer 8

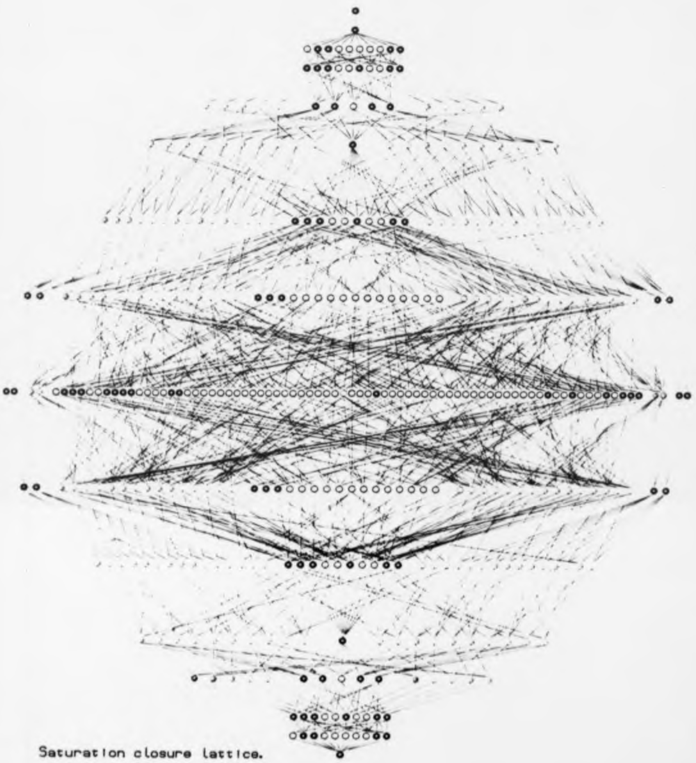

Layer 9
Layer 10
Layer 11
Layer 12


Layer 13


Layer 14


Layer 15


Layer 16



Saturation closure lattice.

Part Three

Computer Aided Mathematical Environments for Lattice Theory

Chapter Five

DEST - A Definitive Environment for Set Theory.

5.1. Introduction

Definition based programming languages are notations in which the variables of the language can be *defined* implicitly by formulae involving other variables. Using terminology based on [5] a **definitive notation** is specified by an underlying algebra comprising of a set of data types Δ, a set of values A and a family of operators Σ mapping between the data types. The variables of the language, whose types are in Δ, are defined by expressions in terms of variables and explicit values from A using the operators in Σ. A simple example of a system that uses definitive programming is a spread-sheet stripped of its tabular interface where the data types are **real, integer, character** etc.

The main principle behind definitive programming is that the user and the computer should perform a dialogue producing a network of definitions interactively, redefining and adjusting old definitions where necessary. The definitions produced by the dialogue between the user and the computer form a graph of dependencies between the variables. This graph is directed and acyclic since no variable can be defined recursively because this would lead to an infinite loop when the variable is evaluated.

Definitive notations were introduced in [5]. The language ARCA (see [6]) was designed for the display and manipulation of combinatorial diagrams such as Cayley diagrams of groups. In this notation it was possible to embed the structure of the group into the definitions. Similarly the language DoNaLD (see [8]), used definitive principles to specify two dimensional line drawings. In DoNaLD there are basic data types that can be used to represent points, lines and subdrawings. In both examples the language is used as a medium for a dialogue between the user and the computer whereby the user can change the values in their definitions while maintaining the functional relationships elsewhere. This reduces the cognitive load on the user of having to recall all the functional relationships present in the system when small changes are performed.

In [11] Beynon and Cartwright outlined a definitive programming approach to the implementation of Computer Aided Design software. The paper did not specify a particular CAD system using definitive notations but proposed a general-purpose programming model based on definitive principles. Also the relationship between a definitive programming approach to CAD and the study of CAD from an AI perspective was discussed.

Beynon also pointed out in [11] the limitations of *pure* definitive notations where the restriction to directed acyclic graphs causes problems when definitive programming is used, for example, in a CAD environment. Here it is common for several objects to be part of a constraint loop, where the adjustment of one object implies the readjustment of the rest. While it is normally possible to circumvent this problem by introducing auxiliary variables and binding all the definitions onto them, this has the undesirable effect that the user is now required to recall details which they would normally wish to be hidden.

In response to this Beynon proposed in [11] an enhancement to pure definitive notations where the computer played a more active role by maintaining constraints. Here the dialogue consists of several "intelligent views" of guarded actions by which the computer could monitor or maintain constraints. In this new model the machine is separated into three units consisting of a store D of variable *definitions*, a store A of guarded *actions* and a program store P containing *entities*, each entity being a block of definitions and actions. Computation consists of the execution of all actions in parallel whose guards are true. In this model the computer can act to maintain constraints since the actions allow it to act autonomously.

The "extended definitive notation idiom" has also been suggested as a means for the specification of concurrent programs (see [7, 12, 13]). In [12] a notation for concurrent systems called LSD is described which is mainly oriented towards design rather than simulation. Beynon in [7] introduced the Abstract Definitive Machine which, being based on extended definitive notation, can handle synchronisation and the multi-agent environment exhibited when dealing with simulation. Here the store P of entities consists of sets of actions and definitions that persist over the same period of time.

Definitive notations and user environments based on definitive notations provide a useful and intuitive way to explore and handle complex problems, as can be seen by the popularity of spread-sheet like programs in commerce. Since they allow the user to unload much of the burden of recalling the functional relationships between variables while maintaining a dynamic perspective, definitive notations provide an important foundation for constructing user environments where relationships and values are changing. Hence in systems where there are complex functional relationships between variables, or where the values of the variables are changing requiring re-evaluation of other variables, or where the user wishes to experiment, definitive environments provide a natural method to implement the system.

This chapter gives details of a definition based environment for the manipulation of and experimentation with mathematical sets called DEST. The main design features of DEST is that it provides an interactive environment for experimentation on sets, including infinite and recursively defined sets, it includes special data types for handling maps, relations, partial orders etc., and that the data types can be treated as objects in an object oriented programming sense and the language, data types and operators can be specified mathematically.

The motivating factors behind the design are to illustrate the usefulness of definitive notations in a human-computer interactive environment, to act as a foundation for the construction of an environment for lattice theory and to provide an environment for the teaching of set theory.

While spread-sheets are quite common, other examples of software based on definitive notations are scarce. Even though several uses of definitive notations have been cited above and the future of definitive notations is promising in areas of CAD and specification, no practical systems will be available in the near future since there are fundamental issues that still need to be resolved. Current implementations of ARCA and DoNaLD have been instructive in developing new ideas and techniques as well as demonstrating the power of definitive notations. A definition based mathematical environment will hopefully demonstrate a further area where definitive notations can be exploited.

In designing an environment for the analysis of lattices it is useful to identify areas that are self-contained. By examining the implementation of sets first and then implementing a second language for lattices as a super-set, issues that are firmly set based are treated separately and are not confused with the implementation of lattices, leading to a more coherent design.

Many specification languages are highly mathematical in nature and require the engineers using them to understand and be able to use discrete mathematics and set theory. This is posing a problem for many software houses since a proportion of their software engineers have had no teaching in modern mathematics. Hence an environment for teaching set theory will have uses not just in schools but in commerce as well.

A full description of DEST can be found in the user manual [17], this chapter and the following only illuminate on the mathematical aspects of the specification of DEST and should not be taken as a full description of the syntax or semantics. Section two introduces definition based computer aided mathematical environments and lists their basic requirements and features in illustrating and investigating abstract mathematics. Section three discusses other methods of implementing a set environment, highlighting the differences between definitive, functional and procedural techniques. Section four describes the basic features of DEST and details the hierarchical data typing and specification.

5.2. Definitive Based Computer Aided Mathematical Environments

Computers are being employed as instruments of guidance, control and inspiration. Computer aided design and manufacture are becoming prevalent in industry to hasten the processes of conception to manufacture, and research into expert systems is a major area of artificial intelligence with applications from mining to medicine. In all these cases the computer is being used to produce an environment to aid the user.

In these cases the computer is used to produce an environment where it can guide, monitor and control the actions of the user. By using the computer's ability to record and recall rules, exceptions and restrictions they become expert counsellors. By investigating possible outcomes the computer can guide the user, by enforcing the rules of the system the computer can monitor and control the design process.

One of the first uses of computers though was to produce a mathematical environment for performing arithmetic calculations, removing the burden of repetitive work and the errors that that tends to produce. With the increasing sophistication of programming languages computers have found many more uses in abstract mathematics, from educational software to theorem proving programs in logic, introducing computers as an aid for mathematicians by producing a mathematical environment under the control of a computer. Butler and Cannon [21] pointed out that mathematical computation has been one of the major application areas of computers, and that this has lead to the design of specialised programming languages. Schwartz *et al* [40] referred to these programming languages as "very-high-level" languages, listing LISP, APL, SNOBOL, SETL and PROLOG as examples in this class. The purpose of this class of languages according to Schwartz *et al* is to reduce the cost of programming by allowing direct manipulation of large composite objects, as opposed to integers, reals etc. While DEST and Pecan have compositive types like maps and lattices they can not be considered as very-high-level programming languages since their field of operation is limited to a specific area of mathematics. In this way DEST and Pecan are more similar to the algebraic language Cayley [21, 20] since they provide the user with an environment for seeking examples and testing hypotheses.

Computer aided environments based on definitive notations also introduce interaction and experimentation as well as the support listed above. Since it is possible to change functional relationships as well as values, definitive environments give good support for experimenting which is not directly possible in procedural or functional notations. In procedural notations it is necessary for the user to re-evaluate all dependent variables when a variable's value is altered, while in functional notations it is not possible to redefine functional relationships.

Computer aided mathematical environments also have the advantage that their scope can be clearly specified (even if it extends into computationally infeasible areas) because of the axiomatic treatment of mathematics. Hence CAMEs can be based on the axioms of the mathematical system making sure that all the objects of the language (eg. data types, operators etc.) are consistent. By specifying the environment axiomatically the data types and operators will naturally have an abstract specification, leading to easier implementation of the environment. While it is suggested that all the objects of the language should be consistent with and expressible by axioms, it is not intended that a working implementation should, for example, use sets to implement the natural numbers.

5.3. Comparison Between Functional and Definitive Notations for Implementing Sets

Since functional languages allow operations on lists of objects it is normally straight forward to implement an environment for set theory by writing simple functions to perform set union, intersection etc. Moreover these simple functions extend naturally to handle sets of infinite size. Hence an important question to answer is what advantages would a definition based environment have over a functional notation.

The main difference between the two approaches is that functional notations require the environment to be static while definitive notations have no such constraint. The environment in functional notations is static in that while the user can evaluate a function at random points, and hence experiment with values, it is not possible to redefine functions and hence experiment with relationships.

It is possible to side-step the problem of not being able to redefine functional relationships in functional notations by creating a secondary environment around the functional environment to act as a user interface. For example by allowing the user to edit a file containing the definitions of the functions it is possible to create a dynamic functional environment, however in doing so the environment is moving more towards a definition based system than functional.

The effects of a static environment in functional notations extend to the accessibility of values in expressions. Since it is only possible to define functions and not to perform assignments, intermediate values can not be stored and reused. For instance in a functional notation if a function returns a set it is not possible to use the values of the set directly, but a second function has to be applied to remove the required elements from the set. In definite notations direct assignment is permitted, and in DEST in particular a system of labels allows access to elements directly. Hence definitive notations reflect the thought processes of the user more closely.

While functional notations are equipped with lazy evaluation and it is possible to declare functions that return infinite sets, this does not necessarily imply that these notations can return sensible results when evaluating operations involving infinite sets. For example if P and N are functions returning the set of positive and negative integers then it would require an extremely intelligent interpreter to realise that their intersection is finite. Hence even though definitive notations do not use lazy evaluation, they are not necessary lacking in their ability to perform operations on infinite sets.

Cayley - a language for discrete algebraic structures

Cayley [21, 20] is a knowledge based system designed for solving hard problems in related areas of algebra, number theory and combinatorial theory. The system includes a very-high-level programming language, a large database containing mathematical knowledge and an inference engine to aid program synthesis, database retrieval and program optimisation. The authors of Cayley have high hopes for the language saying "the outcome of the current Cayley project will be a revolutionary integration of knowledge - algorithmic, deductive and factual - in a system which will act as a powerful and effective research assistant for modern algebra." The system is based on procedural principles where the user writes algorithms in a procedural notation and examines specific structures containing values/results of previous calculations. The Cayley system is an extremely sophisticated environment allowing the user to study many different computational domains and to answer questions not only about individual elements but also about the structure as a whole. Currently the system has no predefined operations for lattices and its main area of operation is fields and groups.

SETL - a language for finite set theory

SETL [40] is a Set Language designed in the mid-70's by Schwartz, Dewar, Dubinsky and Schonberg. The object of SETL was to produce a very-high-level language in which finite sets and maps are provided as basic objects of the language. The language has a rich set of operators and many programs can be written in one-line of code. However the underlining set of values in SETL consists of numbers and character strings with computer generated atoms. The atoms in SETL are created by a special command and the only operation possible on these atoms is a test for equality. Hence even though SETL has many useful operators included, the values to which they can be applied is not as extensive as DEST where, for example, it is possible for the user to define atoms and to perform pattern matching operations on them.

5.4. Aspects of DEST

This section illustrates some of the features of DEST, giving details of the set of underlying values and how they relate to the data types and operators. A precise specification of the mathematical structure of the data types and operators is given in the next chapter. As was stated in the introduction this chapter is not intended to give a full description of the syntax or semantics of DEST, this can be found in the user manual [17].

5.4.1. Values

The underlying set of values A consist of the boolean constants **true** and **false** plus **literals** consisting of all the atomic values in use. Literals include the integer numbers and symbolic names for the user's primitive elements (ie. elements which are not sets themselves). When an explicit query is made on a variable's value the value will be presented as sets of sets (ordered and unordered) etc. of literals and truth values. In DEST all symbolic names must begin with an underscore so as not to confuse them with variables.

As well as being used to distinguish different atomic values, the names given to literals can be used in expressions to specify structure or order in sets. It is possible to specify a basic character-pattern that can then be used as a predicate to produce structured sets. For example if L is a set containing literals _L1, _L2,..., _L10 then the expression

T := { (_L$1, _L$2) in L * L | $1 ≥ $2 }

will define T as a total order on L. Here $1 and $2 are being used as pattern-matching variables that parse the literal's name according to the template given in the predicate. Bounded ranges of integers can be expressed by using a similar syntax as in the functional language Miranda of specifying the bounds separated by two dots, the result is unordered when expressed as a set (ie. { · · · }) or ordered when written as a tuple (ie. (· · ·)). For example the set L above could have been defined by

L := { _L$1 | $1 in { 1..10 } }

Both of these examples specify their result as a subset of a larger set, in the first example as a subset of LxL and in the second as a subset of the set of all literals beginning with "_L". However semantically the two examples differ in the process of how the subset is generated. In the first example the elements of the subset are generated by the expression on the left hand side of the specification symbol "|" and are quantified by the predicate on the right. In the second example the elements are generated by the predicate and are then transformed by the expression. The use of string variables and patterns in generating sets is expanded in the next chapter.

5.4.2. Data Types

The set of data types Δ contains types for handling sets, ordered pairs and tuples, sets of ordered pairs, ordered sets, maps and relations. For the types to reflect the natural structure of the objects they are representing it is necessary for them to be hierarchically ordered so that, for example, any variable of type map can also be considered as a set-of-pairs. Hence the more specialised types are derived as special types from the more general, as illustrated in figure 5.1.

set
order
pairs
relation
map
tuple
pair

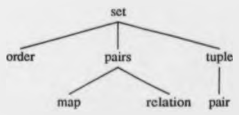


figure 5.1

The edges of the graph represent built-in coercion operators between two types that either forget part of or restructure the data held in a variable. Types higher in the tree represent more general types, types lower in the tree represent more specialised types. There is a special data type **literal** for representing literal values in expressions. The literal data type can be considered as the most primitive data type in DEST since their values can not be transformed into any other type and they are not part of the hierarchy given in figure 5.1.

Even though the axioms of extension, specification and union are worded using both sets and elements, Halmos points out in [33] that "What may be surprising is not so much as that sets can occur as elements, but that for mathematical purposes no other elements need ever be considered." While from a purely mathematical point of view it is only necessary to have a data type for sets, it is semantically useful to have a universal data type **element** to handle the members of a set. The element data type is a *named union* of the standard types consisting of a field for each type plus a tag entry to record the type currently being held.

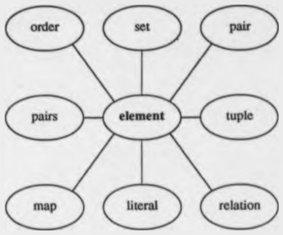


figure 5.2

Element variables allow generic definitions to be written so that similar structures over different types can be examined. Without element variables it would be necessary to write individual definitions to examine for example posets based on literals, sets, relations etc. By writing generic element definitions the definitions can be used in any context, the DEST interpreter will select the appropriate types and operators that the context requires.

While the variables of DEST are typed they do not need to be declared before use, their first defining assignment is used to specify their type. This is normally done implicitly by determining the type of the expression, however the user can specify the resulting type of an expression by casting the expression to the appropriate type.

5.4.3. Operators

The operators Σ contain the standard operators associated with set theory: membership, union, intersection, difference, product, image and inverse image of maps, relations and orderings etc. The exact function of the operators depends on the types of the arguments since variables are treated as objects of various types and the operators are applied appropriately. For example the operation of union between two relations is different than the operation between two equivalent sets of pairs.

Subsets of a set can be defined by using the specification operator |. The operator takes as its arguments an *iterating control variable e* of type **element**, a *range set S* and a *boolean condition P*. The operator works by iterating through all elements of S and returns the set containing those elements that comply with P.

$$T := \{ e \text{ in } S \mid P(e) \}$$

Similarly the elements of a set can be iterated through by using the iteration statement **for**. The iteration operator takes as arguments an iterating-control variable *e*, a range set *S* and statements $S_1, S_2, \ldots, S_t$.

$$\textbf{for } e \textbf{ in } S \textbf{ do } S_1, \ldots, S_t \textbf{ end}$$

The statements $S_1, S_2, \ldots, S_t$ will be executed for every element in the set produced by evaluating S, with the value substituted for the control-variable on each iteration.

It should be noted that the iterating-control variable in these statements is local to the statement and supersedes any other variable of that name. These methods of specification and iteration using a control variable should be compared with the approach used in functional languages. In functional languages where sets are represented as lists the normal method for examining the set is to use a recursive program based on head and tail operations. Here the recursion is replaced by iteration using an explicit variable.

An element can be extracted from a set by using the **elem()** operator. When applied to a non-empty set S elem() returns an element x such that $x \in S$. Other than its uses as an operator, elem() is used in specifying the operations associated with ordered pairs and tuples.

Set inclusion and equality are performed by the operators <=, =>, <, >, = and <>. These should be read as set inclusion, strict set inclusion, set equality and set inequality. As can be seen those are the same symbols that are used for the standard ordering on the integers, however this causes no problems since it is easy to determine by context what type of comparison is desired. In the case of ordered sets and relation orderings different symbols are used in order to differentiate between set comparisons and relation orderings.

Chapter Six

Foundations of Data Types and Operations in DEST

6.1. Introduction

The power of using a definition based paradigm will only be realised if the language provides efficient, concise and expressive methods of defining objects. DEST would be of little use if the only way to define relations and maps for example was by explicitly listing every pair that composed the relation or map. What is required is a notation that allows structured sets like these to be defined where the ordered pairs can be obtained implicitly from the domain/range sets.

In answer to the above DEST introduces a system of character string variables and pattern-matching functions which combine to produce a notation which is used extensively in DEST to define structured and infinite sets. The basic principle behind the use of pattern-matching is that the user should supply literals with names that reflect their use and relationship to each other. This basic requirement stems from the observation that it is normally hard to prescribed order out of random names.

Section two introduces the use of string variables and pattern-matching that is used to define tuples, maps and relations. Section three lists the basic properties of structured data types and how they are related to each other. Section four gives an abstract data type description for order pairs and tuples and illustrates how the standard Kuratowski formalisation of ordered tuples is not sufficient in this case. Section five shows how pattern-matching can be used to defined maps and relations. Section six introduces a second system to reference values called labels. Labels can complement the naming scheme of literals by providing a "user friendly" name rather than a logical one, or can enhance the referencing power by providing an alternative scheme. Section seven gives details on permitted uses of infinite sets and how they can be defined.

6.2. Pattern-Variables, Generators and Predicates

Many programming languages that have been designed to aid users in processing general information have been structured around character strings and pattern matching rather than numerical operations. The standard command interpreters on the UNIX† operating system (*sh* [14], *csh* [34]) use string variables extensively to manipulate commands, file names and parameters. In these interpreters evaluation of expressions and pattern matching are performed by auxiliary commands which make up the extremely popular UNIX environment. Similarly the versatile pattern matching and processing language *awk* [1] uses string based variables which are automatically transformed into numerical values where the context requires it. This produces a powerful environment where users can develop data processing programs in minutes rather than hours if a ''standard'' programming language was used.

Since sets are defined over arbitrary domains and not just numbers of some description it is natural for literals and the variables in DEST that deal with them to be based on character representations rather than numerical values. However it is not sufficient to use some form of enumerated typing as in Pascal to incorporate character strings into the language since these systems simply replace one total order for another using different names. What is required is a naming system which as well as allowing the user to use meaningful names also has the expressive power to handle partial orders, numerical calculations, predicates etc.

To this end DEST incorporates a system of *pattern-variables* that exist locally inside definitions and can be used in predicates, in expressions and to create new literals. Pattern-variables begin with a dollar symbol followed by a digit (for clarity it will be assumed that no more than ten pattern-variables will ever be needed in scope at any one time) and are bound to a set of literals. When evaluated the pattern-variables will be repeatedly matched against the literals in the set to produce all possible matchings, these matchings can then be used elsewhere in expressions. Two examples of their use were previously given in section 5.4.1.

Formally a *simple-pattern* is a sequence $T_1V_1T_2 \cdots V_{k-1}T_k$ where T_i's are arbitrary strings of alphanumeric characters plus underscore and V_i's are pattern-variables, a *pattern* is an n-tuple

† UNIX is a trademark of Bell Laboratories.

$P = (P_1, P_2, \ldots, P_n)$ of simple-patterns and patterns. A **generator** is an expression:

pattern **in** *tuple_expression*

where *tuple_expression* is an n-tuple $E = (E_1, E_2, \ldots, E_n)$ such that the simple patterns in P correspond to sets in E and patterns in P correspond to tuples of the same arity in E. The generator can be used to assign values to the pattern-variables which are either quantified by a predicate or used directly in an expression. The former method will produce a subset of the tuple-expression while the latter can create new literals. The two methods have the following syntax respectively:

{ *generator* | *quantifier* }
{ *expression* | *generator* }

In cases where both sides of the bar appear to be generators (ie. because they both use **in**) then the left expression is taken to be the generator, this follows the normal interpretation used in mathematics.

It should be noted that pattern-variables, generators and expressions involving pattern-variables can only refer to literals and can not be used to reference variables. To be able to would permit definitions to be defined whose actual definition would change over evaluation as well as its value. Also the use of the "such that" bar with generators should not be confused with its use in section 5.4.3. While both can be used to specify subsets of a set, in section 5.4.3 the superset could contain any type of elements while with generators it is required that the elements be literals.

The notion of generators and quantifiers used here was strongly influenced by the functional language Miranda which uses pattern-matching and generating expressions (called ZF-expressions) to create lists and the set language SETL which uses a combined generator/quantifier expression. For example in SETL generating expressions have the general form of

{ *expression* : *generator* | *conditional* }

However in SETL it is not possible to use pattern matching and hence the generator, expression and conditional has to be applied directly to elements. In Miranda though the use of pattern-matching is restricted to matching types and entire objects and can not be used as in DEST to *parse* a literal's name.

Pattern-variables and generators are used through-out DEST as a means of defining maps, relations, ordered sets, tuples etc. because they offer an extremely versatile way of linking literals to expressions and back to literals. They provide a system that as well as being semantically sound is also clear and easy to use. The use of patterns is further enchanced when labels are introduced in section 6.6.

6.3. Structured Data Types in DEST

The basic data type in DEST is the set, all other data types except literal being defined as combinations of ordered and unordered sets. For example the type **order** is an ordered pair consisting of a set field and a **relation** field, where the type **relation** is a set of **pair** variables etc. To gain access to this structure all the ''structured'' types in DEST are accompanied with *member functions* to retrieve the data. The difference between member fields and member functions is that the functions return a value normally based on all the fields of the type, and it is not possible to assign to a member function. For example the type **map** has a member function **.domain** which returns the domain of the map.

6.3.1. Ordered Tuples and Pairs

Ordered tuples are represented by variables of type **tuple** and **pair**. Variables of type pair have associated with them two member functions **.first** and **.second** which return the first and second element of the ordered pair respectively. Tuple variables have member functions **.head** and **.tail** which when applied to the ordered set $(a_1, a_2, \ldots, a_n)$ return the first element a_1 and the n-1 tuple $(a_2, \ldots, a_n)$. The tail of an ordered singleton is the empty set. In general any 2-tuple is automatically promoted into a pair if the context requires such a change of type. For example the definition var := (a,b) would declare var as a pair variable if the type of var is unknown. If var is intended to be a tuple variable then the definition should be *cast* as such by saying var := (tuple) (a,b).

Tuples can be transformed into sets, removing the ordering, by casting the tuple into a set variable: set_var = (set) tuple_var. The exact process is defined by the following recursive equation.

```
(set) tuple_var   =   if tuple_var = ∅ then ∅
                      else tuple_var.head ∪ (set) tuple_var.tail
```

6.3.2. Sets of Ordered Pairs - Maps and Relations

Variables of type **map** consist of a set of pairs from the product $\text{set}_A \times \text{set}_B$. There are two member functions, **.domain** returns the set of elements used in set_A and **.range** returns the set of elements used in set_B. They are defined as:

map_var.domain ≡ { s.first | s ∈ map_var }

map_var.range ≡ { s.second | s ∈ map_var }

Variables of type **relation** consist of a set of pairs from the product $\text{set}_A \times \text{set}_A$. Relations have only one member function **.domain** which returns the set of elements used in set_A.

rel_var.domain ≡ { s.first | s ∈ rel_var } ∪ { s.second | s ∈ rel_var }

Variables of type **pairs** have the same structure and member functions as the types **map** and **relation**. They are included as a type in DEST to re-enforce the fact that maps and relations are sets of ordered pairs.

6.3.3. Ordered Sets

Pre-orders, partial orders and total orders are represented by variables of type **order**. A variable of type **order** is an ordered pair (base_set, order_relation) consisting of a base set and a relation from base_set × base_set. It has two member functions to access its components, .set returns the set of elements used and .order returns the relation.

order_var.set ≡ base_set

order_var.order ≡ order_relation ∩ base_set×base_set

6.3.4. Coercion Operators Between Data Types

Values can be transferred between variables of different types by *casting* an expression into the destined type:

new_type_variable := (*type name*) *old_type_expression*

Type conversions from a more structured type (ie. a type printed lower in the hierarchy tree above) to a less structured type are performed automatically *(automatic coercion)* according to context. For example the expression relation_variable.range is interpreted as ((pairs) relation_variable).range . Automatic coercion is always defined and normally results in some data being forgotten. Type conversions in the opposite direction *(reverse coercion)* is sometimes undefined and may require extra information to be given by the user.

Automatic Coercions

set of pairs → set :	No data loss, lose use of .domain, .range
map → set of pairs :	No data loss, lose use of map operations
relation → set of pairs :	No data loss, lose use of relation operations
pair → tuple :	No data loss, lose use of .second
order → set :	Order loses .order and is just left with .set
tuple → set :	Always defined, returns the set with the ordering removed.

Reverse Coercions

set → set of pairs :	Undefined if any element of the set is not a pair or 2-tuple.
set of pairs → map :	Undefined if there exists $p_1, p_2 \in$ pair_variable such that p_1.first $= p_2$.first and p_1.second $\neq p_2$.second.
set of pairs → relation :	Always defined.
set → order :	This requires a pair of objects (base_set , order_relation).
tuple → pair :	Undefined if tuple.tail.tail $\neq \varnothing$
set → tuple :	Undefined if the set is not a singleton.

6.4. Mathematical Basis for Tuples

Ordered pairs can be defined by sets using the standard Kuratowski formalisation:

$$(elem_1, elem_2) = \{ \{elem_1\}, \{elem_1, elem_2\} \}$$

While tuples can be defined using a similar technique, it is more suitable to recorded them as a series of ordered pairs with the length included:

$$(elem_1, elem_2, \ldots, elem_n) = (n, (elem_1, (elem_2, (\cdots (elem_n, \varnothing) \cdots))))$$

To ease the definition of the tail member function for tuple variables the empty set is included in all tuples so that all the elements of the tuple appear as the first component of a pair. While the Kuratowski method for representing ordered n-tuples works when $n \geq 2$ or when no operations like head and tail are being performed, it fails for the "ordered" 1-tuple. More specifically if (a,a) is an ordered pair then the normal Kuratowski representation is $\{\{a\}\}$, hence the tail of this pair can be interpreted as either $\{\{a\}\}$, $\{a\}$, a or $\varnothing$: all of which are wrong. Therefore ordered tuples have to include a length member so as to handle single and empty ordered tuples. Luckily the natural numbers and the successor function are easily defined by sets.

Since pair variables are defined using the Kuratowski formalisation the set union of a pair $(a,b) = \{\{a\},\{a,b\}\}$ is the set $\{a,b\}$ and similarly the intersection of a pair is $\{a\}$ (see [44], footnote on page 64), hence the definitions of the member functions for pairs and tuples can be expressed as:

```
pair_variable.first      =  elem( ∩ pair_variable)
pair_variable.second     =  if singleton( pair_variable )
                            then pair_variable.first
                            else elem( ∪ pair_variable \ ∩ pair_variable )

tuple_variable.head      =  if tuple_variable = ∅ then undefined
                            else tuple_variable.second.first
tuple_variable.tail      =  if tuple_variable = ∅ then undefined
                            else if tuple_variable.first = "1" then ∅
                            else (tuple_variable.first - 1 , tuple_variable.second.second)
```

Even though the member functions of tuples are defined in terms of pair member functions (the abstract data type representation of tuples are defined using pairs) the pair member functions are not directly available to variables of type tuple. However the tuple member functions head and tail can be applied to variables of type pair since they are treated as tuple variables.

Tuples can be defined using generators as well as by explicitly listing its members. For example the following will assign the first ten square numbers:

tuple_var := ($1*$1 | $1 in (1..10))

The elements of a tuple can be obtained by placing a colon and an index after the tuple's name, for example tuple_var:4 will be "16" in the example above. This is an early example of labels to be introduced in section six.

It should be noted that while tuples and pairs are defined in terms of sets it is not possible to defined a tuple or pair value by explicitly giving the set representation. For example if tuple_var has been declared as being of type **tuple** then the following will produce a type error

tuple_var := { 2, { a, { b, ∅ } } }

because the right hand side is a set expression.

6.5. Specification of Maps and Relations

The methods of defining maps and relations are identical since they are both sets of ordered pairs, it is only in usage that they differ. There are three ways in which sets of pairs can be defined. Firstly the ordered pairs that make up the map or relation can be explicitly listed. Secondly the pairs can be calculated from an expression using a generator. Lastly a sequence of guarded-expressions can be given that specify a means of obtaining the corresponding result(s) from elements in the domain or range.

By explicitly listing the pairs that make up the map or relation the map/relation is described exactly and all the information necessary for evaluating inverse images of maps or transitive closures of relations etc. is provided. Obviously though this method can only be used on small domains.

To specify maps and relations on large finite domains generators can be used. Either the resulting set of pairs can be defined as a subset of a cross product quantified by a suitable predicate (first method described in section two) or by an expression bound to a generator (second method in section two). In the first method all pairs in the cross product will be considered resulting in many evaluations of the predicate, hence to avoid unnecessary computation when calculating the image of maps this method is best left for defining relations. In the second method it is possible to restrict the generator to the domain and hence is more economic when defining functions.

```
rel := { ($1,$2) in (1..12)*(1..12)  |  $1 mod $2 = 0 }
lat := { (_L$1_$2, _L$3_$4) in L*L | $1 ≥ $3 and $2 ≥ $4  }
fib := { ($1, if $1 > 1 then fib($1−1)+fib($1−2) else 1 ) |  $1 in (0..100) }
```

The first example illustrates how the first method can be used to define a relation to express "is a multiple of". The second example defines a partial order on a set L×L by examining the names of the literals. Here it is assumed that the elements of L have been assigned names of the form _L10_3 etc. The last example defines the fibonacci function using recursion. It should be noted that this definition does not contradict the condition of non self-referencing definitions since at no point is any set in *fib* defined in terms of itself and hence no infinite referencing loops occur.

The use of generators can be extended to handle more complicated functions and relations and infinite domains by introducing guards on the expressions. Guards are similar to if-then-else however they can only occur inside a set or tuple definition and it is possible to omit the else clause. If the guard is true then the expression following the guard is evaluated, otherwise the expression following the ? is evaluated if provided.

```
fib := { [$1 > 1] → ($1, fib($1−1)+fib($1−2)) ? ($1,1)  |  $1 in (0..100) }
abs := { [$1 < 0] → ($1,−$1) ? ($1,$1),  [$2 ≥ 0] → ($2,$2),  [$2 ≥ 0] → (−$2,$2)  |  ($1,$2) in Z*Z }
```

The first example is a repeat of the fibonacci function this time using guards rather than the if-then-else. In this case the domain of the generator is finite so the function will be evaluated fully. In the second example an absolute value function is defined. Here there are four expressions giving both the function and its inverse.

As can be seen from the example of the absolute function the only way to be able to find the inverse of a function when the domain is infinite is to supply the necessary expressions to calculate it. If the inverse function is not provided when the domain is infinite then it is impossible for DEST to determine any inverse images since it can not enumerate the function.

The use of guards for defining functions and relations on infinite domains was inspired by the standard mathematical notation used to defined these objects. The notation for defining the absolute function in DEST should be compared with a purely mathematical definition:

$$abs := \{ (x,y) \in Z{*}Z \mid (x < 0 \text{ and } x = -y) \text{ or } (x \geq 0 \text{ and } x = y) \}$$

as can be seen the mathematical definition is also expressing a relation, which can easily be interpreted as a sequence of guarded expressions.

6.6. Labels

Labels provide a parallel system for referencing elements which can be used to complement the use of the literal's name or to enhance the power of generators. Labels act as a local naming scheme in a set that can act on elements of any type. Also labels are produced automatically for cross products by combining the labels present in the product's arguments. Hence by labelling elements appropriately the interpreter will produce a suitable labelling for the new set. The elements of a set's definition can be referenced by two labelling systems called enumeration labels and set labels.

Enumeration Labels: By requesting an enumeration of a set, all the elements of the set will be associated with a unique integer. For unordered sets the number associated with each element will be time dependent and the element will only have that value while the definition and its associated environment remains constant. For ordered sets the number associated with each element will be in accordance with the element's position in the set, the element at the head having label "1".

Set Labels: Set labels are names given by the user to the elements of a set's definition, either when the definition is first given or later by attaching a name to an element using an enumeration label. Unlike enumeration labels, set labels retain their meaning over time. Set labels can be used in patterns and hence provide the same means of specifying structured sets as literals. For example if S is defined as

```
S   := { e1, s1, _a, _b, s2 :: { _a, s1 } }
```

then the enumeration labels might be S:1 = e1, S:2 = s1 etc. The only set label defined so far is S:s2. Note that an underscore is not required for labels since their names are always prefixed by a colon.

Labels provide a means of accessing the value of the elements of a set and not a way to change them, that is it should be noted that labels are not *l-values* and can not have expressions assigned to them. To be able to change an element via a label would usurp the definition of the set and the elements.

6.6.1. Creation of Labels by Operators

The operator cross product will create new labels for the elements of sets defined as a product from the labels of the arguments. Let C := A × B and let e1 be a variable in the definition of A and e2 a variable in the definition of B where e1 has set label :lab1 and e2 has set label :lab2. The element representing (e1,e2) in C will have set label :lab1_lab2. If either e1 or e2 are labelless then (e1,e2) will be labelless. For example if B is a set of elements with set labels :L0, :L1, :L2, ... then the definitions

```
S   := B*B
O1  := { (:L$1, :L$2) in B*B | $1 ≥ $2 }
O2  := { :L$1_L$2 in S | $1 ≥ $2 }
```

will define O1 and O2 as a set of pairs which defines a total order on B. The definition of O1 uses labels in generators and the definition of O2 uses the creation of labels by the product operator to act as a pattern.

6.6.2. Recursive Definition of Labels

Labels can be applied to elements that appear in a recursive definition. Since a label is local to a set and appears only in conjunction with the set's name it is possible for a set to have several instances of the same label at different levels. For example a recursive map could be defined as:

```
rec := { [$1 = 1]→(1,{t::_a}) ? ($1,{t::_a,T::rec($1−1)}) | $1 in Z }
```

Hence rec(3) equals {t::_a, T::{ t::_a, T::{ t::_a }}}, so rec(3):t is the literal _a and rec(3):T is the set {t::_a, T::{ t::_a }}. Since rec(3):T is a set which contains labels the above process can be repeated, for example rec(3):T:T:t etc.

6.7. Operations on Infinite Sets

DEST has two predefined infinite sets, the set **Z** consisting of the integers and **N** of the natural numbers (including zero). As has been shown earlier infinite sets of literals can be defined by basing a generator on one on these sets:

```
B  := { _L$1 | $1 in N }
```

Furthermore it is possible to define an order on infinite sets using pattern matching with generators. For example the definition

O := (relation) { (_LS1,_LS2) in B*B | S1 ≥ S2 }

will define O to be a total order on the set B. This order can be used to compare literals as in any finite order because by using pattern-matching it is possible to determine if an ordered pair is a member of the relation without enumerating the expression.

DEST will not attempt to evaluate any expression that is considered to involve an infinite set since this may force the interpreter into an infinite loop. The only operator that can be applied to infinite sets is the membership operator in since by pattern-matching it is possible to determine membership without enumerating the set. As can be seen even with this restriction it is possible to determine maps and relations on infinite domains.

DEST is unable to determine whether an expression that involves infinite sets is infinite, for example the intersection to two infinite sets need not be infinite. This problem is common to most computer interpreters and the normal solution is to classify any set derived from an infinite set as unenumerable. However there are two exceptions to this, when an infinite set is either intersected with a finite set or equivalently specified as a subset of a finite set (by the specification operator or by a generator) then the result is considered finite.

Chapter Seven

Pecan - a Definitive Environment for Lattice Theory

7.1. Introduction

Pecan is a definition based language designed for interactive analysis of lattices. As indicated in earlier chapters Pecan contains the language DEST as a subset and incorporates additional data types and operators to handle lattices. While DEST can handle infinite sets to a limited degree, Pecan is restricted to finite lattices so that the results of chapters two, three and four can be applied. Many of the techniques introduced in chapters two, three and four require that a complete enumeration of the lattice is given as well as tables representing the $^{-}$-relation between join-irreducibles and meet-irreducibles. Hence the user is required to *open* a lattice to indicate that this information should be calculated. The process of opening a lattice introduces two levels of definitions in Pecan. On one level the user can define an algebra and on a second level the user can define expressions based on the algebra. This can be contrasted with a spread-sheet where the algebra is fixed.

This chapter does not give a complete description of Pecan, this can be found in the user manual [18]. Section two introduces the additional data types of **lattice**, **congruence** and **homomorphism** included in Pecan and lists their member functions and methods of definition. Section three details how lattices are constructed and represented in Pecan. Section four lists the operations permissible on lattices and shows how the results of the earlier chapters are used.

7.2. Additional Data Types for Lattices

Pecan has three extra data types not included in DEST for handling lattices, congruences and homomorphisms. The data types **congruence** and **homomorphism** are extensions of the DEST types relation and map and are used in defining quotients. The type **lattice** is an extension of the type order and includes new member functions for accessing the components of the algebra.

7.2.1. Lattices

As stated in section 1.2 lattices can be defined either as a special type of poset or as a special type of algebra. To reflect these two methods of definition Pecan allows variables of type lattice to be defined either by giving a poset expression or by specifying a set with two operators.

L1 := (lattice) P
L2 := (lattice) (S, M, J)

In the example L1 is defined by giving a partial order and L2 is defined by giving an algebra. In the second case S should be a set and M and J maps from $S{\times}S \rightarrow S$ representing the operators meet and join.

Since lattice variables can originate from two different sources, variables of type lattice have four member functions reflecting the two methods available. The member functions .set and .order of orders are inherited by lattice variables as well as two new functions called .join and .meet. The operations of join and meet on a lattice L can either be expressed using the infix operators "\L/" and "/L\" or by using the member functions, for example the expression "x \L/ y" is equivalent to "L.join(x,y)".

To be able to implement the operators meet and join efficiently and to be able to calculate quotients etc., large amounts of information are required to be calculated for each lattice variable. Hence while lattices can be defined and used in definitions at any time, the elements of a lattice can only be accessed after the lattice has been explicitly *opened* for use by the user. In opening a lattice a full enumeration of the lattice is performed and all join-irreducibles and meet-irreducibles in the lattice are found plus how they are related by the ˉ-relation. To aid brevity the infix operators of meet and join of the most recently opened lattice may be referred to as "/\" and "\/".

To stop massive recalculations caused by accidentally redefining a lattice expression, all opened lattices are tagged so that in the event of a redefinition the user is warned and is given the option to cancel the operation. In this way Pecan is arranged as a two layer definitive language where the user defines lattices and then performs calculations in and on them. When the user wishes to investigate another system of lattices these lattices must be re-calculated by opening the definitions.

Once the enumeration of the lattice has been performed the elements of the lattice are given enumeration labels (see section 6.6) according to the order in which the elements appear in a topological sort of the lattice, the minimal element being given label :0. Enumeration labels are used by operators like quotient and cross product in generating set labels for elements of the newly created lattice. To reduce memory overheads the elements of a derived lattice are not calculated explicitly but are referenced by a combination of the set and enumeration labels. Hence only labels are recorded in the derived lattice instead of sets of elements reducing the amount of information needed to be stored.

Several standard lattices are included in Pecan. Chains of arbitrary length can obtained by using the natural and integer number posets N and Z. Free distributive lattices can be obtained from the function FDL(n) where n is the number of generating variables. All lattices of five or less variables are also defined. These standard lattices can be combined to define arbitrary lattices using a cut and paste technique. More information about this process is given in section three.

7.2.2. Congruences

Congruences can either be defined by coercing an expression of type relation or by specifying a set of join-irreducibles to determine a congruence by corollary 2.4.1. If the former method is used then upon evaluation of the congruence all join-irreducibles not related to the element they cover are located and the $^{-}$-closure of this set is used to determine the congruence. In the case that the relation expression is a congruence then the resulting congruence is equal to the relation. If however the relation was not a congruence then obviously there may be little connection between the two. (The only thing that can be said is that they agree on a subset of the join-irreducibles.) If the second method is used then the $^{-}$-closure of the set of join-irreducibles is calculated and this set is then used to determine the congruence.

Congruence variables have a new member function .Join which returns the set of join-irreducibles determining the congruence. Since it is required that the $^{-}$-closure of the set of join-irreducibles is calculated, any definition of a congruence has to be bound to a definition of a lattice. This is expressed by casting a congruence or relation expression, stating the lattice to which the congruence is to be bound:

C := (congruence on L) R

In this case C is a congruence on L generated by the relation R. The congruence class of an element e is given by the expression ''[e]C''. To change the base lattice of a congruence simply re-cast it onto the new lattice. Since the relation/congruence expression is bound to a lattice, all occurrences of the operators \/ and /\ will be taken to apply to that lattice rather than the most recently opened lattice.

7.2.3. Homomorphisms

Homomorphisms can be defined in much the same way as maps, however like congruences it is necessary to specify the lattices between which the homomorphism acts:

H := (homomorphism L1 to L2) M

If the homomorphism is defined by using an expression bound to a generator as described in section 6.5 then all occurrences of the operators /\ and \/ in the expression will be taken to be in the range lattice. If the expression has guards then occurrences of the operators in the guards will be taken to be in the domain lattice. In this way it is possible for the same map definition to be used between several different lattices.

7.2.4. Quotient Lattices

Lattices, congruences and homomorphisms are all connected by quotient lattices in the Homomorphism Theorem (see section 1.2 or [30] p 26) which states that every homomorphic image of a lattice L is isomorphic to a suitable quotient lattice of L. More precisely if $\phi: L \rightarrow L_1$ is a homomorphism from L onto L_1 and Φ is the congruence relation on L defined by

$$x \equiv y\ (\Phi) \text{ if and only if } x\phi = y\phi \qquad (1)$$

then $L/\Phi \cong L_1$ and the map $\psi: [x]\Phi \rightarrow x\phi$ is an isomorphism.

If L is a lattice variable and C a congruence variable bound to L then the quotient lattice L/C can be defined as follows:

Q := L / C

The elements of Q will be presented as intervals $[e_1,e_2]$ of L, however as mentioned above they will not be explicitly stored as sets of elements of L but referenced via enumeration labels. Each element of Q will be assigned an enumeration label according to its topological order in Q and a set label of the form :#n_1_n_2 where n_1,n_2 are the enumeration labels of the minimal and maximal elements in the congruence class in L.

The natural homomorphism $H: L \rightarrow Q$ can be defined as follows:

```
H := (homomorphism L to Q) { ($1,[$1]C) | $1 in L }
```

This definition however does not specify an inverse transform and is hence very inefficient in calculating the pre-image of an element in Q. Since all of the information necessary to calculate both the image and pre-image of the natural homomorphism is given by the set and enumeration labels of L and Q the natural homomorphism can be defined just by saying:

```
H := (homomorphism L to Q) natural
```

If L1 and L2 are two isomorphic lattices then the function isomorphism() returns an isomorphism from L1 to L2. This function works by recursively sorting the elements of both lattices trying to find a match. Naturally this function may take some time. If L1 and L2 are not isomorphic then an empty map (causing an error if used) is returned.

The kernel of a homomorphism (that is the congruence relation defined by (1) above) can be obtained by the function kernel():

```
C1 := kernel( H1 )
```

The congruence C1 is bound to the domain of H1. If L, L1 are lattices and H is a homomorphism from L to L1 then the homomorphism theorem can be demonstrated by the following definitions and queries:

```
IH      :=   H(L)                              // Calculate image of H.
L2      :=   sublattice( IH, L1 )
H2      :=   embedding( L2, L1 )
C       :=   kernel(H)                         // Obtain suitable congruence
Q       :=   L/C                               // and quotient.
H1      :=   (homomorphism L to Q) natural
I       :=   isomorphism( Q, L2)

print Q = L1                                  // True if H is onto.
print Q = L2                                  // Isomorphic lattices.
print I@H1 = H2@H                             // Identical maps.
```

7.3. Construction and Representation of Lattices

While lattices can be defined in terms of quotients, images, products etc. of other lattices and partial orders it is necessary at some point to explicitly define a lattice without reference to any other variable. As was indicated in the previous section lattices can be defined by expressing a partial order or an algebra where the corresponding sets of ordered pairs (order relation in the case of a poset and the meet and join operators in the case of an algebra) are given in full. Even in a small lattice the definition of the partial order by this method would be extremely tedious and likely to be error prone. The use of generators and pattern matching allows larger lattices to be defined, however lattices defined this way tend to have an extremely uniform structure (eg. total orders or boolean algebras) and hence restrict their use.

To allow larger and more complex lattices to be defined, Pecan includes several *basic* and *pre-defined* lattices which can be used to construct general lattices. The basic lattices consist of all the lattices with five or less elements, the eight element lattice 2^3, the free boolean lattice on two variables FBL(2) and the free distributive and the free modular lattices on three variables, FDL(3) and FML(3). The pre-defined lattices consist of the free distributive lattices on eight or less variables. The difference between basic and pre-defined lattices is that basic lattices are stored explicitly while the pre-defined lattices are represented and manipulated algebraically.

7.3.1. Construction of Lattices from Basic Lattices.

Arbitrary lattices can be constructed by combining several basic lattices to represent an ordered set, which is then coerced into a lattice. The general principal behind this method is that the user draws a Hasse diagram of the lattice and then identifies overlapping sublattices of the sort given in the basic set. The Hasse diagram is finally reconstructed by identifying overlapping elements of the basic lattices producing a connected diagram. This process only creates an ordered set, this set has then got to be converted into a partial order and finally a lattice. However it does produce concise and easy method of constructing arbitrary lattices.

The method described above deviates from the general aspects of a definitive mathematical environment because the process is more procedural and does not have a mathematical base. However as was stated above it is necessary at some point to define objects without reference to anything else, and at this point definitive and procedural notations coincide. Moreover the system described here is intuitive and is based strongly on most users mental image of a lattice. While it might be more mathematically sound to express arbitrary lattices as quotients of free lattices it does however produce an extremely awkward and unintuitive method of doing so.

To isolate the construction process from the main definitive environment the lattice construction is performed in a separate environment, hence only the resulting ordered set produced by the construction will be visible to the definitive environment. A suitable alternative environment can simply be a text editor editing a file of construction definitions. In this way the user can redefine and reuse lattices without confusing the construction process with the main environment.

A lattice construction consists of a block of declarations where identifiers are assigned to basic lattices, followed by a block of element identifications where elements in different lattices are identified and finally followed by a block of order relations where individual elements can be ordered. To make the last two stages easier all the elements of a basic lattice are given set labels so that the elements can be identified. These set labels are carried through to the final lattice and provide an efficient method of labelling the elements. The non-totally ordered basic lattices of five or less elements with their set labels are given in figure 7.1.

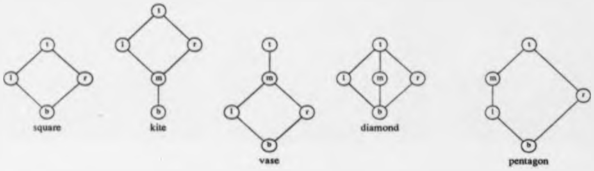


figure 7.1

The set labels given by default can be overridden when the basic lattice is first used in the declaration section by specifying alternative labels. An example of this is given later.

As well as being able to use basic lattices in the declaration block it is also possible to use cross products of basic lattices. In this case the set labels of the elements of the lattices are concatenated together. In this way the basic lattices 2^3 and FBL(2) are equivalent to the declaration chain(2) × square and square × square.

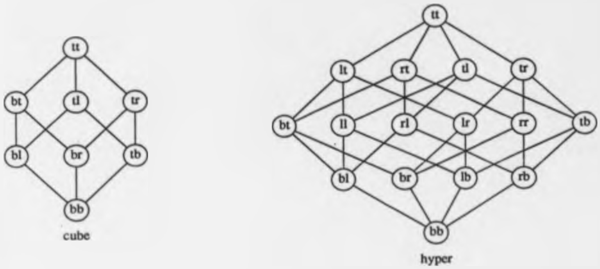


figure 7.2

The lattice FML(3) is given as a basic lattice, however as an example it can be constructed by the following definition.

```
FML3 := (
        B := cube ; M := diamond ; T := cube
        X := square( "", "x", "mx", "" ) ;              // Relabel elements
        Y := square( "", "my", "y", "" ) ;
        Z := square( "", "mz", "z", "" ) ;

        M:t := T:bb ; M:b := B:tt        // Join lattices together
        M:l := X:mx ; M:m := Y:my ; M:r := Z:mz

        T:bl > X:t > X:b > B:bt                  // Add remaining edges
        T:br > Y:t > Y:b > B:tl
        T:tb > Z:t > Z:b > B:tr
)
```

There are obviously several methods of constructing FML(3), the above was chosen here to demonstrate all three stages of the construction. The final Hasse diagram with labels is given in figure 7.3. As can be seen several of the elements have unusual labels, these can be changed by the user using the normal procedure (eg. L:T_bb := :mt).

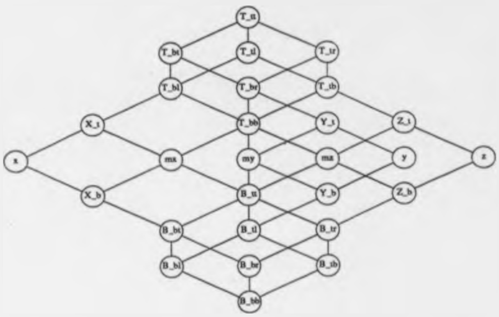


figure 7.3

7.3.2. Pre-defined Lattices

The pre-defined lattices consist of the free distributive lattices on three to eight variables. These lattices can not be represented in the normal fashion due to their size, however they can still be manipulated in the same way by using algebraic identities. The elements of the lattice are represented in disjunctive normal form by set labels, for example the element $x_1x_2x_4 \vee x_2x_3 \vee x_3x_4$ would be represented by the label "abd_bc_cd". Many of the operations associated with normal lattices can be applied to the pre-defined lattices, including quotients, homomorphisms, products etc. However operations which would result in enumerating the entire lattice or explicitly constructing a lattice too large for Pecan will be stopped.

The pre-defined free distributive lattices are obtained from the function FDL(). It should be noted that the basic lattice isomorphic to FDL(3) is called FDL3, not FDL(3).

7.3.3. Internal Representation of Lattices

As was stated earlier when a lattice is opened a full enumeration of the lattice is made and all join and meet-irreducibles are located. The lattice data structure can be represented by figure 7.4.

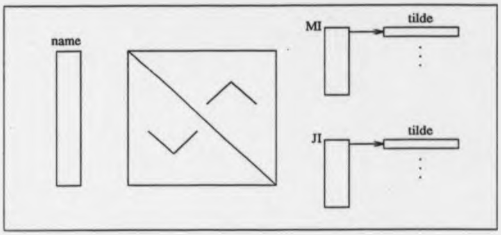


figure 7.4

The name field lists the set labels of the elements. These are either given by the user or if the lattice is derived from another by the enumeration labels of the base lattice. The operators and order information is recorded in the lower and upper triangular matrix. This obviously is the major component of the structure in terms of memory usage and where it is possible to calculate the meet and join of elements without using a table (for example quotient and product lattices) this table is omitted. The MI and JI fields list the meet and join-irreducibles of the lattice and also for each irreducible there is a pointer to a list of irreducibles related to it by ~. All the fields are indexed by the enumeration order of the elements in the lattice.

Derived lattices such as quotient lattices, product lattices etc. have an extra field linking the elements of the derived lattice to the elements in the base lattice. In the case of quotient lattices the extra field represents the natural homomorphism and lists for each element in the quotient the enumeration labels of the maximal and minimal elements in the interval mapped to that element.

7.4. Operations on Lattices

This section describes how the operations of the last section can be implemented using the characterisation of congruences described in chapter two.

7.4.1. Quotients of Lattices

By theorem 2.3.1, corollary 2.4.1 and theorem 2.6.2 the elements of a quotient lattice L/C, where the congruence C is determined by a set of join-irreducibles P, are in one-one correspondence with *hereditary* subsets of P. As stated in the definition of hereditary subsets in section 2.6, a subset X of a set of join-irreducibles P is called hereditary if $X = P[\vee X]$. Hence to enumerate the elements of the quotient lattice it is sufficient to enumerate all decreasing sets of the above type. The algorithm 7.1 enumerates through all subsets of a set of join-irreducibles, making sure only to output hereditary sets and not to output the same set twice. In the algorithm it is assumed that the elements of P are enumerated in some arbitrary order.

Algorithm 7.1:

Algorithm to Enumerate all Hereditary Subsets of a set P

```
enumerate( X, i )
{
//      Input: X - a hereditary set of join-irreducibles
//             i - index of the next join-irreducible to be included
//      Global:      P - a set { $p_1, p_2, \ldots, p_n$ } of join-irreducibles

        if  i > |P| then
               output( X )
        else
               enumerate( X, i+1 )
               $s = \vee X \vee p_i$
               $Y = P[s]$
               $Z = Y \setminus X$
               if $\forall j \in [1..i-1] : p_j \in Z$ then
                      enumerate( Y, i+1 )
}
```

As can be seen the algorithm is based on the algorithm 4.4 to enumerate saturated elements. In this case the second recursive call is only made if the inclusion of the new join-irreducible p_i does not force the inclusion of earlier join-irreducibles. Hence no hereditary set is printed twice.

Theorem 2.6.2 also indicates how the meet and join operators can be calculated. The meet of two elements $x,y \in L/C$ represented by the hereditary sets h_x,h_y is the intersection of the sets $h_x \cap h_y$. The join of the two elements is the intersection of all hereditary sets containing the union of h_x and h_y (equivalently the minimal hereditary set containing $h_x \cup h_y$).

To determine the natural homomorphism between a lattice and a quotient it is necessary to find the minimal and maximal elements that are mapped to each element of the quotient.

Proposition 7.4.1.1

If L is a finite lattice and P a set of join-irreducibles closed under $\overline{}$ and X a hereditary subset of P then the minimal element $s \in L$ such that $P[s]=X$ is $\vee X$. The maximal element $t \in L$ such that $P[t]=X$ is $\wedge\{q \in \overline{P\setminus X} \mid q \geq \vee X\}$.

Proof.

Obviously $\vee X$ is the minimal element containing the join-irreducibles X. Let $w=\wedge\{q \in \overline{P\setminus X} \mid q \geq \vee X\}$. By construction $w \geq \vee X$ hence $P[w] \supseteq X$. Moreover if $p \in P\setminus X$ then $p \not\leq \vee X$ since X is hereditary, hence there exists $q \in \bar{p}$ such that $q \geq \vee X$ therefore $q \geq w$ so $p \not\leq w$ and hence $P[w]=X$. To show that w is the maximal element with this property it is necessary to show that any element greater than w is also greater than a join-irreducible in $P\setminus X$. Let $v > w$, hence it follows that there exists a meet-irreducible $q \in \overline{P\setminus X}$ such that $q \not\geq v$ but $q \geq w$. Hence there exists a join-irreducible $p' \in \bar{q}$ such that $v \geq p'$ but $w \not\geq p'$. Since P is closed under $\overline{}$ it follows that $p' \in P$. Hence $P[v] \neq P[w]$.

□

7.4.2. Kernel of a Homomorphism

If ϕ is a homomorphism between lattices then the kernel Φ of ϕ is the equivalence relation on the domain set defined by

$$x \equiv y\ (\Phi) \text{ if and only if } x\phi = y\phi$$

It can be easily verified that this equivalence relation is a congruence on the domain lattice. If the domain is finite then the set of join-irreducibles determining the congruence can be identified as the join-irreducibles that are not mapped to the same element that they cover.

Proposition 7.4.2.2

If ϕ is a homomorphism between finite lattices then the kernel of ϕ is determined by the set of join-irreducibles

$$\{ p \mid p \text{ is a join-irreducible and } p\phi \neq x\phi \text{ where } p \vdash x \} \qquad (2)$$

Proof.

Let P be the set described by (2) above and let P′ be the set of join-irreducibles that determine the kernel of ϕ. Let p be a join-irreducible and x the element it covers. Since $P'[p] \neq P'[x]$ if and only if $p \in P'$ it follows that if $p \in P$ then $p \in P'$, hence $P \subseteq P'$. Moreover if $p \in P'$ then p and x are not equivalent under the kernel of ϕ and so $p\phi \neq x\phi$, hence $p \in P$ and $P' \subseteq P$.

□

7.4.3. Lattice of Congruences

It is a well known fact that for any lattice L the set of all congruences on L ordered by inclusion forms a distributive lattice (for example [23] p 75). By corollary 2.4.1 and theorem 2.4.2 it is possible to identify all the congruences of a finite lattice and calculate the meet and join of two congruences.

Algorithm 7.2 gives an algorithm for enumerating all $^-$-closed subsets of a set of join-irreducibles. Hence by enumerating all closed subsets of join-irreducibles of a lattice it is possible to determine the lattice of congruences.

Theorem 2.4.2 states that the meet of two congruences is the congruence determined by the union of the closed sets of join-irreducibles and the join of two congruences is the intersection of the closed sets. Hence it is possible to calculate meet and join in the lattice of congruences directly from the closed subsets and it is not necessary to store a full operator table. This fact is extremely useful since even a small lattice can have a large number of congruences.

Algorithm 7.2:

Algorithm to Enumerate all ~-Closed Subsets of a set P

```
enumerate( X, i )
{
//      Input: X - a ~-tilde closed set of join-irreducibles
//             i - index of the next join-irreducible to be included
//      Global:       P - a set { $p_1, p_2, \ldots, p_n$ } of join-irreducibles

        if  i > |P| then
                output( X )
        else
                enumerate( X, i+1 )
                $S = p_i^*$
                $Y = X \cup S$
                if $\forall j \in [1..i-1] : p_j \notin S$ then
                        enumerate( Y, i+1 )
}
```

Conclusion

Chapter Eight

Conclusions

With the ever increasing power and availability of computers much research is being spent on using computers to produce safe, user friendly environments for investigating and researching into complex dynamic systems such as design and manufacturing. This thesis has addressed the issues relating to the implementation of a mathematical environment for investigating lattice theory based on definitive notations. Areas covered have included the efficient representation of lattice congruences, calculation of quotient lattices, representation of arbitrary and free distributive lattices and methods of defining partial orders, lattices and maps between them.

Chapter two presented an alternative characterisation of lattice pre-orders and congruences. It was shown that any congruence on a finite lattice can be determined by a set of join-irreducibles and any two elements can be tested for equivalence under the congruence by comparing the elements with the set of join-irreducibles. Hence it is possible to determine if any two elements are equivalent without recording the full set of ordered pairs normally associated with equivalence relations. Also by determining congruences this way dual definitions of the intersection and union of two congruences are produced and can be calculated easily, whereas the union of the sets of ordered pairs determining two congruences does not in general produce a congruence.

Chapter three demonstrated an application of the characterisation of pre-orders given in chapter two to the study of computation equivalence and replaceability. It also gave alternative characterisations for approximate replaceability triples classified by Dunne in [28] and the $\mu()$ and $\lambda()$ functions defined by Beynon in [4]. The new characterisation of the $\mu()$ and $\lambda()$ functions permitted these functions to be enumerated directly rather than having to enumerate the whole lattice. This fact was used to calculate the Hasse diagram of the closure lattice $\mu(FDL(5))$.

Chapter four addressed the issues of manipulating and displaying distributive lattices. Two methods of representing elements of free distributive lattices were given and algorithms for calculating meet, join, $\mu()$ and $\lambda()$ of elements listed. Both methods allowed free distributive lattices to be manipulated algebraically, however the first method was able to perform lattice operations more quickly at the expense of exponential increase in memory while the second method allowed for "arbitrary" size lattices to be implemented. These methods were combined to implement the algorithm given in [10] for constructing planar monotone circuits. Methods for displaying Hasse diagrams of arbitrary distributive lattices was discussed and by using the first method of manipulating free distributive lattices the elements of FDL(4) and FDL(5) were enumerated and displayed.

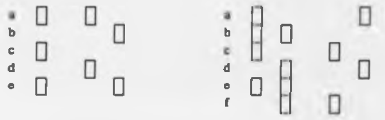
Chapters five and six discussed the advantages of designing a user environment for manipulating mathematical sets based on definitive notations. Issues that were addressed covered the ability of the system to record and change functional relationships between variables, ease of recalling relationships between variables and values and the ability to experiment with relationships and values. It was argued that a definition based system supports such operations well and provides an intuitive method for doing so. Chapter six concentrated on methods of defining sets, maps, relations etc. in such a notation and proposed that an underlying algebra of character strings complemented with labels using pattern matching gave a sufficiently rich algebra to defined these structures. It was pointed out that most programming languages use an underlying algebra based on the natural numbers and that this ordering tends to usurp any non-total order that the user might want to use.

Chapter seven combined the results of the previous chapters to demonstrate how a definitive environment for manipulating arbitrary lattices can be defined. The methods of chapter six for defining partial orders were extended to define lattices and complemented with special lattice construction operators that used the labelling system suggested. The ability to determine lattice congruences by sets of join-irreducibles demonstrated in chapter two was used to define and manipulate congruences in the environment. Moreover the ability to use the set of join-irreducibles to identify the congruence classes, determine the quotient and calculate the meet and join of congruences allows the environment to handle quotient lattices and lattice of congruence relations which would otherwise be too large to handle.

By combining the characterisation of congruences given in chapter two with the definitive environment specified in chapter seven a dynamic environment is created in which the definitions are used to evaluate expressions. In comparison a procedural environment would require every lattice, congruence, quotient lattice etc. to be fully calculated and stored, where as in the definitive environment specified here very little has to be recorded since most of the data can be obtained by projecting backwards through the definitions. This leads to an efficient system were the expressive power of definitive notations is used to aid the evaluation of expressions.

Open Problems

Section 4.3 introduced the term persistent configuration in reference to an arrangement of prime implicants and clauses of a monotone function f that indicated that the function could not be computed by a planar monotone circuit. It can be easily shown that there exists only two persistent configurations on five or six variables and these displayed pictorially in figure 8.1.



(a) ace × (a∨d)∧(b∨e) (b) abce∨bdef × (c∨e)∧(a∨d)

figure 8.1

While it is possible to create larger persistent configurations on more variables which do not contain either of these configurations, no function has been found that is not planar computable and does not contain either configuration displayed above. This leads to the following conjecture:

Conjecture 8.1

Every non planar computable monotone function either contains three prime implicants/clauses in the configuration of figure 8.1a or four prime implicants/clauses in the configuration of figure 8.1b.

Obviously there exists functions that are not computable on a plane which are computable on surfaces of higher genus. Knowing the exact connection between the "size" of the persistent configuration and the surfaces on which a function can be computed would be illuminating to general circuit construction as well as just monotone circuits. The algorithm given for generating planar circuits had an obvious upper bound of $\frac{1}{8}n^4 + O(n^3)$. The largest functions found that are planar computable have size $O(n^2)$, it is suggested that the upper bound is this value.

All the results relating to chapters two and three required the lattice to be finite so that all elements could be expressed as a join of join-irreducibles. For these results to be extended to infinite lattices it is necessary to find an alternative characterisation of the elements of the lattice. In the case of distributive lattices this can be resolved by using prime ideals (see [30] p 74). However in arbitrary lattices prime ideals can not be used in this way and an extension of theorem 2.3.1 looks uncertain.

References

1. Aho, A., Kernighan, B., and Weinberger, P.: *Awk - A Pattern Scanning and Processing Language: User's Manual.*

2. Berkowitz, S.: "On Some Relationships Between Monotone and Non-Monotone Circuit Complexity,"
Ph D Thesis (mentioned in "On the Complexity of Slice Functions", I. Wegener, TCS Vol 38, 1985), University of Toronto, 1982

3. Beynon, M.: "Replaceability and Computational Equivalence in Finite Distributive Lattices,"
Univ. of Warwick T.C. Report, No. 61, 1984

4. Beynon, M.: "Replaceability and Computational Equivalence for Monotone Boolean Functions,"
Acta Informatica, vol. 22, pp. 433-449, 1985

5. Beynon, M.: "Definitive Principles for Interaction,"
Proc hci'85, pp. 23-24, CUP, 1985

6. Beynon, M.: "ARCA: a Notation for Displaying and Manipulating Combinatorial Diagrams,"
Univ. of Warwick T.C. Report, No. 78, 1986

7. Beynon, M.: "Definitive Programming for Parallelism,"
Univ. of Warwick T.C. Report, No. 132, 1988

8. Beynon, M., Angier, D., Bissell, T., and Hunt, S.: "DoNaLD: a Line Drawing System Based on Definitive Principles,"
Univ. of Warwick T.C. Report, No. 86, 1986

9. Beynon, M. and Buckle, J.: "Computational Equivalence and Replaceability in Finite Algebras,"
Univ. of Warwick T.C. Report, No. 72, 1985

10. Beynon, M. and Buckle, J.: "On the Planar Monotone Computation of Boolean Functions,"
Theoretical Computer Science, vol. 53, pp. 267-279, 1987

11. Beynon, M. and Cartwright, A.: "A Definitive Programming Approach to the Implementation of CAD Software,"
Intelligent CAD Systems 2: Implementation Issues, Springer Verlag, 1988

12. Beynon, M., Norris, M., and Slade, M.: "Definitions for Modelling and Simulation of Concurrent Systems,"
Applied Simulation and Modelling, Proc IASTED ASM'88, pp. 94-98, Acta Press, 1988

13. Beynon, M., Slade, M., and Yung, Y.: "Parallel Computation in Definitive Models,"
Proc Conpar '88, 1988

14. Bourne, S.: "Unix Time-Sharing System: The Unix Shell,"
Bell Sys. Tech. J., vol. 57(6), pp. 1971-1990, 1978

15. Buckle, J.: *Prime - Desk Calculator for Finite Free Distributive Lattices*,
(Included), Programming Archive Report, 1984

16. Buckle, J.: *Computational Equivalence in Dyke Languages*,
(Unpublished manuscript), 1985

17. Buckle, J.: *DEST - User Manual*,
(Included), Programming Archive Report, 1989

18. Buckle, J.: *Pecan - User Manual*,
(Included), Programming Archive Report, 1989

19. Buckle, J.: *Displaying Large Free Distributive Lattices*, Programming Archive Report, 1989.

20. Butler, G. and Cannon, J.: "The Cayley V4 - The User Language," *Proc of the 1988 International Symposium on Symbolic and Algebraic Computation*, Rome, 1988

21. Butler, G. and Cannon, J.: "The Design of Cayley - A Language for Modern Algebra," *Technical Report No 334*, 1988

22. Church, R.: "Numerical Analysis of Certain Free Distributive Structures," *Duke Mathematical Journal*, vol. 6, pp. 732-734, 1940

23. Crawley, P. and Dilworth, R.: *Algebraic Theory of Lattices*, Prentice-Hall, New Jersey, 1973

24. Czyzo, E. and Mostowski, A.: "Algorithm for the Generation of Free Distributive Lattices," *Bull. of the Academie Polonaise des Science*, vol. 16, pp. 593-595, 1968

25. Dedekind, R.: "Ueber Zerlegungen von Zahlen durch ihre Grössten Gemeinsamen Teiler," *Festschrift Hoch. Braunschweig u. ges. Werke*, vol. 2, pp. 103-148, 1897

26. Dunne, P.: "Some Results on Replacement Rules in Monotone Boolean Networks," *Univ. of Warwick T.C. Report, No. 64*, Jan 1984

27. Dunne, P.: "Techniques for the Analysis of Monotone Boolean Networks," *Univ. of Warwick T.C. Report, No. 69*, Sept 1984

28. Dunne, P.: "Approximate Replacement Rules and Pseudo-Complementation," *Univ. of Liverpool, Internal Report*, 1985

29. Fisher, M. and Pippenger, N.: "Relations Among Complexity Measures," *JACM*, vol. 26, pp. 361-381, 1979

30. Gratzer, G.: *Lattice Theory: First Concepts and Distributive Lattices*, W H Freeman and Company, San Francisco, 1971

31. Gratzer, G. and Schmidt, E.T.: "Ideals and Congruence Relations in Lattices," *Acta Math Acad Sci Hungar*, vol. 9, pp. 137-175, 1958

32. Gratzer, G. and Schmidt, E.T.: "Standard Ideals in Lattices," *Acta Math Acad Sci Hungar*, vol. 12, pp. 17-86, 1961

33. Halmos, P.: *Naive Set Theory*, Springer-Verlag, New York, 1960

34. Joy, W.: "An Introduction to C shell," *Unix Manual*

35. Kisielewicz, A.: "A Solution of Dedekind's Problem on the Number of Isotone Boolean Functions," *Journal für die Reine und Angewandte Mathematik*, vol. 389, pp. 139-144, 1988

36. Melhorn, K. and Galil, Z.: "Monotone Switching Networks and Boolean Matrix Product," *Computing*, vol. 16, pp. 99-111, 1976

37. Paterson, M.: "Complexity of Monotone Networks for Boolean Matrix Product," *Theoretical Computer Science*, vol. 1, pp. 13-20, 1975

38. *Miranda System Manual (version 1.009)*, Research Software Limited, 1987

39. Rutherford, D.E.: *Introduction to Lattice Theory*, Oliver and Boyn, Edinburgh, 1965

40. Schwartz, J., Dewar, R., Dubinsky, E., and Schonberg, E.: *Programming with Sets, an Introduction to SETL,*
Springer-Verlag, New York, 1986

41. Shyr, H.: "Free Monoids and Languages,"
Lecture Notes, Dept. of Maths, Soochow Univ.,, Taipei, Taiwan, 1979

42. Sivak, Bohuslav: "Congruences on Finite Lattices,"
Math Slovaca, vol. 32, pp. 283-290, 1982

43. Slade, M.: "Laden - Lattices and Definitive Notations,"
Third Year Project, 1987

44. Stewart, I. and Tall, D.: *The Foundations of Mathematics,*
OUP, Oxford, 1977

45. Urosu, Carmencita: "On the Connections between Congruence Relations and the Neutral Ideals of Latticies,"
Bull. Stiint Tehn. Inst Politehn. "Traian Vuia" Timisoara 22(36), vol. 22, pp. 366-368, 1977

46. Ward, M.: "Note of the Order of Free Distributive Ltiices,"
Bull. American Mathematical Society, vol. 52, p. 423, 1946

47. Wegener, L: "On the Complexity of Slice Functions,"
Univ. of Frankfurt, Internal Report, 1983

48. Wegener, I.: "On the Complexity of Slice Functions,"
Theoretical Computer Science, vol. 38 (1), pp. 55-68, 1985

49. Wegener, I.: "More on the Complexity of Slice Functions,"
Univ. of Frankfurt, Internal Report, 1985

Appendix One

DEST - User Manual

DEST - A Definitive Environment for Set Theory

User Manual

John Buckle

DEST is an interactive interpreted language designed to allow users to learn, experiment and investigate the ideas of set theory. The language is based on definitive principles whereby variables in DEST store expressions ("definitions") not values. Hence complex functional relationships between variables can be stored and maintained, and the user can experiment with different values without having to recalculate intermediate expressions.

DEST is a subset of the language Pecan used for investigating finite lattices. This document contains the minimal user specification for DEST, local versions of the language may contain extra features. The version described here can be used on any standard terminal running on a UNIX† or similar operating system.

† UNIX is a trademark of Bell Laboratories.

1. Introduction

DEST is an interactive interpreted language designed to allow users to learn, experiment and investigate the ideas of set theory. The language is based on definitive principles whereby variables in DEST store expressions ("definitions") not values. So a session with DEST consists of a dialogue between the user and the computer where the user can enter a sequence of definitions followed by an examination of the resulting expressions. Hence via the dialogue a network of definitions is produced that is maintained by the computer in a dynamic environment. Therefore complex functional relationships between variables can be stored and maintained, and the user can experiment with different values without having to recalculate intermediate expressions.

As well as having data types for handling sets and elements, DEST can also handle ordered pairs and tuples, maps, relations and ordered sets. These higher order or *structured* data types are expressed as combinations of ordered and unordered sets, and special coercion operators exist to transform data between different data types. DEST is equipped with the standard operations common to set theory, such as union, intersection, product, map image and inverse image etc., and how they behave depends on the types of the arguments. For example the union of two relation variables in DEST is treated differently to the union of the equivalent set variables.

DEST also allows a partial implementation of infinite sets, maps and relations. Since these sets are infinite and it is not possible to list every element in them the only operations possible on infinite sets are actions based on set membership. This is because DEST will not perform any action which it believes will cause it to enter into an infinite loop. Even with this constraint it is possible to use maps and relations based on infinite domains.

To aid in formulating definitions of maps, relations and infinite sets DEST uses a system of character variables with pattern matching operators and predicates. Here it is possible to obtain a subset of a set by locating all the elements containing a certain pattern, or to generate new sets containing elements with special patterns, or in the case of an infinite set define a rule that determines if an element is a member of the set.

One of the most important rules in DEST is not to define a variable in terms of itself. A common error for beginners is to type something like the following, expecting it to add a new element to a set:

```
        s := s + { _a }
```

While in most programming languages this simply "increases" the value of `s`, in DEST it creates a self-referencing loop. This is because in DEST expressions are stored in variables, not values. Hence the expression `s + { _a }` is stored, when this is evaluated the interpreter will plunge into an ever decreasing pit trying to find the value of `s`. (Obviously it's not that bad, as soon as DEST sees the same variable twice when it tries to evaluate anything it will complain, however it is important that the point that DEST stores expressions not values is clearly understood.)

2. Getting Started With Sets

The basic data types used in DEST are the types set and literal. Literals consist of the boolean constants true and false, the integer numbers and all the symbolic names of the user's most primitive objects (ie. objects that are not sets themselves). Variable names in DEST can be any string of characters consisting of letters, numbers and underscore beginning with a letter. So that DEST can tell the difference between a variable and a symbolic name, all symbolic names must begin with an underscore. Simple sets consisting of just literals and other variables can be defined by placing the elements of the set between braces { · · · }. For example the following definitions define i to be the intersection of the sets s and t, and define u to be the union.

```
s := { _a, _b, _c, {_a, _b} }
t := { _b, _c, s }

i := s & t
u := s + t

print i, u
{ _b, _c }
{ _a, _b, _c, { _a, _b }, { _a, _b, _c, { _a, _b } } }

s := { _a }

print i, u
{}
{ _a, _b, _c, { _a } }
```

As can be seen from the example the elements of a set's definition do not have to be literals, for example the definition of t uses the variable s. Also there is no requirement for the elements of a set to be all of the same type. The intersection of two sets is denoted by the symbol & ("elements in this *and* that set"), union is denoted by + and the operation of set difference by -. The empty set is represented by two braces with nothing between them {}. The procedure print simply evaluates the arguments given to it and prints them. If only one expression is to be evaluated then the word print can be omitted.

3. Ordered Pairs and Tuples

Ordered pairs and ordered tuples are defined in DEST in the same way as sets except that curved brackets are used instead of braces. Hence to defined a variable p to hold an ordered pair whose first element is _a and second is _b or define alpha to hold the first ten letters of the alphabet in that order then enter

```
p := ( _a, _b )
alpha := ( _a,_b,_c,_d,_e,_f,_g,_h,_i,_j )
```

Since p was defined as an ordered tuple consisting of two elements DEST automatically declared p to be of type pair, similarly alpha was declared to be of type tuple. Also since curved brackets are used in controlling the evaluation of expressions it is not possible to define a 1-tuple directly. The type pair is simply a special case of tuple that has some extra functions associated with it and is used in conjunction with the types `map` and `relation`. In fact ordered pairs and two element tuples are interchangeable, DEST will convert data between these two types when the context requires such a change.

In the above examples the elements of the ordered tuples consisted of literals, however as in sets they can hold any type of expression, and it is not required that all the elements have the same type.

```
a := { _f, _g }
b := ( a, _f )
t := ( a, b, _x, _y )
```

In this example `b` is the ordered pair `({_f,_g},_f)`, (not the 3-tuple `(_f,_g,_f)`), similarly `t` is a 4-tuple consisting of a set, a pair and two literals.

An ordered pair or tuple can be unordered, producing a set consisting of the elements in no particular order, by *casting* the tuple into a set,

```
s := (set) t
```

will define `s` to be the unordered set produced from `t`, (while `t` is defined by the above expression, `s` will be `{_x,_y,{_f,_g},({_f,_g},_f)}`). The union, intersection and difference of tuples differ from that of plain sets. The union of two tuples is the concatenation of the tuples, producing a tuple whose length is the sum of its arguments. The intersection of two tuples is the longest prefix common to both tuples. The difference is the suffix of the first tuple remaining after finding the longest common prefix. While these definitions produce a non-commutative union, they introduce a rich algebra for tuples that parallels the set identity $A=(A\cap B)\cup(A\backslash B)$. The normal definitions of union etc. can be obtained by first removing the ordering on the tuples by casting the arguments into sets.

The cross product of two sets is denoted by the operator `*`. Given two sets `A` and `B` the product `A*B` is the set of all the ordered pairs `(x,y)` where `x` is an element of `A` and `y` is an element of `B`. There is a special type in DEST for sets consisting of just ordered pairs called `pairs`. Cross product is an example of an operator that returns an expression of type `pairs`, and is mainly used in defining maps and relations.

4. Control in DEST

DEST incorporates several control constructs to aid in the definition of subsets, evaluation of expressions and the enumeration of results. The use of pattern-matching, pattern generators and labels is discussed in a later section. This section gives details of the control constructs associated directly with sets and elements.

The actual definition of a variable can be printed using the `print_def` command. This will display what the interpreter currently thinks the variable is defined to be.

An element can be tested for membership in a set by using the `in` operator which returns either the literal `true` or `false`, for example the test $x \in X$ is entered as `x in X`.

Subsets of a set can be defined by using the specification operator `|`. The operator takes as its arguments an iterating control variable, a range set and a boolean condition. The operator works by iterating through all the elements of the range set and returns the set containing those elements that comply with the condition. For example in the definition below if `S` is a set of integers then `T` would be the subset of integers divisible by 3,

```
T := { e in S | e mod 3 = 0 }
```

Similarly the elements of a set can be iterated through by using the iteration statement `for`. The iteration operator takes as arguments an iterating control variable, a range set and a sequence of statements. For example if `S` is a set of sets then the variables `S1, S2,...,` would be defined to be the individual elements of `S`,

```
for e in S do S$$ := e ; print e ; end
```

The use of `$$` in the above example is a means of generating a sequence of unique variable names inside a `for` loop. It initially starts with a value of one and is increment at every iteration. It should be noted that a `for` loop is a shorthand for producing several definitions or expressions, not a definition in itself. Hence in the above example the variable `S1` is defined by an expression which is taken to be independent of `S` (ie. `S1` will keep its old definition even if `S` is redefined).

The iterating control variable used in defining subsets and in `for` loops is local to the statement and supersedes any other variable of that name. At the end of the statement the control variable is removed from the symbol table.

Sometimes it is desirable for a variable to use the *current* value of an expression in a definition rather than maintaining the dynamic link normally given by DEST. To use the current value the expression should be given as an argument to the function `eval()` which will evaluate the expression and substitute the value. `Eval()` will always return a constant expression except when the expression involves an iterating control variable. In this case the control variable is not substituted, however all other variables will be replaced by their current value.

Integer expressions can be calculated in DEST using the normal arithmetic symbols of `+, -, *, div, mod`. In this case the arguments and results of the expressions are considered to be literals. Integer ranges can be defined in sets by using a similar notation to that of Pascal and Miranda by specifying the bounds separated by two dots, `1..10`. Integer ranges in tuples are ordered with the lowest integer coming first. Arithmetic progressions can also be defined by specifying the first two elements in the sequence followed by the upper bound, `1;4..20`. If the upper bound is less than the lower bound then an empty range is produced.

Expressions can be evaluated conditional by using the `if-then-else` expression. If the condition is true then the expression following `then` is evaluated, otherwise the expression following `else` is evaluated. In DEST the `else` is compulsory unlike most procedural languages. The conditional expression can be any expression returning either the literal `true` or `false` constructed from sub-expressions using `and, or` and `not`. Binary comparisons between sets (and between integers) are done by the operators `<=, =>, <, >, =, <>`. These should be read as set inclusion, strict set inclusion, set equality and set inequality. These are the same symbols that are used for the standard ordering on the integers, however this causes no problems since it easy to determine by context what type of comparison is desired.

5. Maps, Relations and Ordered Sets

Variables of type map and relation are treated as sets of ordered pairs with special operations to reflect their use. Order variables are considered as a pair of objects representing the underlying set and an order relation on that set. Since maps, relations and ordered sets tend to be quite large objects (each requiring a large array of ordered pairs) it is not normally possible to define them explicitly. To enable the user to define these objects concisely DEST uses a system of pattern-matching predicates so that sets of ordered pairs can be defined, from which maps, relations and ordered sets can be obtained. This section introduces the operations possible on maps, relations and ordered sets. Methods of defining these objects are introduced in sections seven and eight.

The image of an element x in a map m is given by m(x), the image of a set X in m is m{X}. The preimage of a element in the range of m is given by m^(y) and the preimage of a set is m^{Y}. The type of a preimage or the image of a set is always a set, the type of the image of an element depends, of course, on the image. The composition of two maps f and g is denoted by g@f ("the value of g at f"). The union and intersection of maps is treated as the union or intersection of the corresponding sets of ordered pairs.

If r is a relation then relations in r is examined by the operator ~r~. That is, if (x,y) ∈ r then x~r~y is true. The set of elements related to an element a (ie. {x | (a,x) ∈ r}) is denoted by r(a). The set of elements that relate to a is denoted by r^(a). The intersection of two relations is the relation produced by the intersection of the corresponding sets of ordered pairs. The union of two relations is the transitive closure of the union of the corresponding sets of pairs. If the transitive closure is not required then one of the relations should be cast first into pairs.

Order relations on an ordered set p are denoted by the operators <p<, <p=, =p=, <p>, =p>, >p>. The union and intersection of two ordered sets is the union or intersection of the base sets and of the relation. The relation given to an ordered set can represent any kind of ordering, for example it could be a pre-order, partial order or total order. The comparison a =p> b is equivalent to the condition that (a,b) is a member of the order relation.

6. Type Hierarchy

The data types of DEST are organised into a hierarchy so that they reflect the natural structure of the objects they are representing. For example any variable of type map can also be considered as a variable of type pairs. Hence the more specialised types are derived as special types from the more general, as illustrated in figure 1.

set
order
pairs
relation
map
tuple
pair

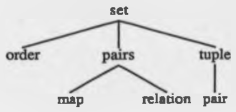


figure 1

The edges of the graph represent built-in coercion operators between two types that either forget part of or restructure the data held in a variable. Types higher in the tree represent more general types, types lower in the tree represent more specialised types. The data type `literal` can be considered as the most primitive data type in DEST since literals can not be transformed into any other type and are hence not part of the hierarchy given in figure 1.

While the variables of DEST are typed they do not need to be declared before use, their first defining assignment is used to specify their type. This is normally done implicitly by determining the type of the expression, however the user can specify the resulting type of an expression by casting the expression to the appropriate type.

While from a purely mathematical point of view it is only necessary to have a data type for sets, it is semantically useful to have a universal data type `element` to handle the members of a set. The `element` data type is a *named union* of the standard types consisting of a field for each type plus a tag entry to record the type currently being held.

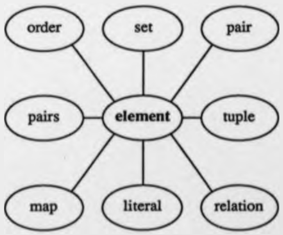


figure 2

Element variables are implicitly used in `for` loops and in subset specification since the iteration control variable is of type `element`. In these cases the control variable adopts the type of the element it is representing. Hence element variables allow generic definitions to be written so that similar structures over different types can be examined. For example without element variables it would be necessary to write individual definitions to examine posets based on literals, sets, relations etc. By writing generic element definitions the definitions can be used in any context, the DEST interpreter will select the appropriate types and operators that the context requires.

The basic data type in DEST is the set, all other data types except `literal` being defined as combinations of ordered and unordered sets. For example the type `order` is an ordered pair consisting of a `set` field and a `relation` field, where the type `relation` is a set of `pair` variables etc. To gain access to this structure all the "structured" types in DEST are accompanied with *member functions* to retrieve the data. The difference between member fields and member functions is that functions return a value normally based on all the fields of the type, and it is not possible to assign to a function. For example the type `map` has a member function `.domain` which returns the domain of the map.

Pairs and Tuples

Variables of type `pair` have associated with them two member functions `.first` and `.second` which return the first and second element of the ordered pair respectively. Tuple variables have member functions `.head` and `.tail` which when applied to the ordered set $(a_1, a_2, \ldots, a_n)$ return the first element a_1 and the n-1 tuple $(a_2, \ldots, a_n)$. The tail of an ordered singleton is the empty set.

Maps, Relations and Pairs

Variables of type `map` consist of a set of ordered pairs from the product $set_A \times set_B$. There are two member functions, `.domain` returns the set of elements used in set_A and `.range` returns the set of elements used in set_B.

Variables of type `relation` consist of a set of pairs from the product $set_A \times set_A$. Relations have only one member function `.domain` which returns the set of elements used in set_A.

Variables of type `pairs` have the same structure and member functions as the types `map` and `relation`. They are included as a type in DEST to re-enforce the fact that maps and relations are sets of ordered pairs.

Ordered Sets

A variable of type `order` is an ordered pair `(base_set, order_relation)` consisting of a base set and a relation on base_set × base_set. Order variables have two member functions to access their components, `.set` returns the set of elements used and `.order` returns the relation.

Type Conversions

Values can be transferred between variables of different types by *casting* an expression into the destined type:

new_type_variable := (*type name*) *old_type_expression*

Type conversions from a more structured type (ie. a type printed lower ir the hierarchy tree above) to a less structured type are performed automatically *(automatic coercion)* according to context. For example the expression `relation_variable.range` is interpreted as `((pairs) relation_variable).range`. Automatic coercion is always defined and normally results in some data being forgotten. Type conversions in the opposite direction *(reverse coercion)* is sometimes undefined and may require extra information to be given by the user.

Automatic Coercion

set of pairs $\rightarrow$ set :	no data loss, lose use of .domain, .range
map $\rightarrow$ set of pairs :	no data loss, lose use of map operations
relation $\rightarrow$ set of pairs :	no data loss, lose use of relation operations
pair $\rightarrow$ tuple :	no data loss, lose use of .second
order $\rightarrow$ set :	order loses .order and is just left with .set
tuple $\rightarrow$ set :	Always defined, returns the set with the ordering removed.

Reverse Coercion

set $\rightarrow$ set of pairs :	Undefined if any element of the set is not a pair or 2-tuple.
set of pairs $\rightarrow$ map :	Undefined if there exists $p_1, p_2 \in$ pair_variable such that p_1.first = p_2.first and p_1.second $\neq$ p_2.second.
set of pairs $\rightarrow$ relation :	Always defined.
set $\rightarrow$ order :	This requires a pair of objects (base_set , order_relation).
tuple $\rightarrow$ pair :	Undefined if tuple.tail.tail $\neq \varnothing$
set $\rightarrow$ tuple :	Undefined if the set is not a singleton.

7. Generators and Pattern-Matching

Since sets are defined over arbitrary domains and not just numbers of some description it is natural for literals and the variables in DEST that deal with them to be based on character representations rather than numerical values. DEST uses a naming system which as well as allowing the user to use meaningful names also has the expressive power to define maps and relations, handle ordered sets, perform numerical calculations and can be used predicates. DEST incorporates a system of *pattern-variables* that exist locally inside definitions and can be used in both predicates and expressions and to create new literals. Pattern-variables begin with a dollar symbol followed by a digit and are bound to a set of literals. When evaluated the pattern-variables will be repeatedly matched against the literals in the set to produce all possible matchings, these matchings can then be used elsewhere in expressions.

Formally a *simple-pattern* is a sequence $T_1 V_1 T_2 \cdots V_{k-1} T_k$ where T_i's are arbitrary strings of alphanumeric characters plus underscore and V_i's are pattern-variables, a *pattern* is an n-tuple $P = (P_1, P_2, \ldots, P_n)$ of simple-patterns and patterns. A generator is an expression:

pattern `in` *tuple_expression*

where *tuple_expression* is an n-tuple $E = (E_1, E_2, \ldots, E_n)$ such that the simple patterns in P correspond to sets in E and patterns in P correspond to tuples of the same arity in E. The generator can be used to assign values to the pattern-variables which are either quantified by a predicate or used directly in an expression. The former method will produce a subset of the tuple-expression while the latter can create new literals. The two methods have the following syntax respectively:

{ *generator* | *quantifier* } (1)

{ *expression* | *generator* } (2)

In cases where both sides of the bar appear to be generators (ie. because they both use `in`) then the left expression is taken to be the generator, this follows the normal interpretation used in mathematics. For example if `L` is a set containing the literals `_L1, _L2, ..., _L10` then the expression

```
      T = (relation) { ( _L$1, _L$2 ) in L*L | $1 >= $2 }
```

would define `T` to be a total order on `L`. Here `$1` and `$2` are being used as pattern-matching variables that parse the literal's name according to the template given in the predicate. The set `L` could have been defined by

```
      L = { _L$1 | $1 in { 1 .. 10 } }
```

Both of these examples specify their result as a subset of a larger set, in the first example as a subset of `L*L` and in the second as a subset of the set of all literals beginning with `_L`. However semantically the two examples differ in the process of how the subset is generated. In the first example the elements of the subset are generated by the expression on the left hand side of the specification symbol "|" and are quantified by the predicate on the right. In the second example the elements are generated by the predicate and are then transformed by the expression.

It should be noted that pattern-variables, generators and expressions involving pattern-variables can only refer to literals and can not be used to reference variables. To be able to would permit definitions to be defined whose actual definition would change over evaluation as well as its value. Also the use of the "such that" bar with generators should not be confused with its use in section four. While both can be used to specify subsets of a set, in section four the superset could contain any type of elements while with generators it is required that the elements be literals.

Pattern-variables and generators are used through-out DEST as a means of defining maps, relations, orders, tuples etc. because they offer an extremely versatile way of linking literals to expressions and back to literals. They provide a system that as well as being semantically sound is also clear and easy to use. The use of patterns is further enchanced when labels are introduced in section nine.

8. Using Pattern-Matching to Define Maps and Relations

The methods of defining maps and relations are identical since they are both sets of ordered pairs, it is only in usage that they differ. There are three ways in which sets of pairs can be defined. Firstly the ordered pairs that make up the map or relation can be explicitly listed. Secondly the pairs can be calculated from an expression using a generator. Lastly a sequence of guarded-expressions can be given that specify a means of obtaining the corresponding result(s) from elements in the domain or range.

By explicitly listing the pairs that make up the map or relation the map/relation is described exactly and all the information necessary for evaluating inverse images of maps or transitive closures of relations etc. is provided. Obviously though this method can only be used on small domains.

To specify maps and relations on large finite domains generators can be used. Either the resulting set of pairs can be defined as a subset of a cross product quantified by a suitable predicate or by an expression bound to a generator. In the first method given in section seven all the pairs in the cross product will be considered resulting in many evaluations of the predicate, hence to avoid unnecessary computation when calculating the image of maps this method is best left for defining relations. In the second method it is possible to restrict the generator to the domain and hence is more economic when defining functions.

```
rel := { ($1,$2) in {1..12}*{1..12} | $1 mod $2 = 0 }

lat := { (_L$1_$2,_L$3_$4) in L*L | $1>=$3 and $2>=$4 }

fib := { ($1, if $1>1 then fib($1-1)+fib($1-2) else 1)
                | $1 in (0..100)}
```

The first example illustrates how the first method can be used to define a relation to express "is a multiple of". The second example defines a partial order on a set L*L by examining the names of the literals. Here it is assumed that the elements of L have been assigned names of the form _L10_3 etc. The last example defines the fibonacci function using recursion. It should be noted that this definition does not contradict the condition of non self-refencing definitions since at no point is any set in fib defined in terms of itself and hence no infinite refencing loops occur.

The use of generators can be extended to handle more complicated functions and relations and infinite domains by introducing guards on the expressions. Guards are similar to if-then-else however they can only occur inside a set or tuple definition and it is possible to omit the else clause. If the guard is true then the expression following the guard is evaluated, otherwise the expression following the ? is evaluated.

```
fib := { [$1>1] -> ($1, fib($1-1)+fib($1-2)) ? ($1,1)
                | $1 in (0..100) }

abs := { [$1 <  0] -> ( $1,-$1) ? ($1,$1),
         [$2 >= 0] -> ( $2, $2),
         [$2 >= 0] -> (-$2, $2) | ($1,$2) in Z*Z }
```

The first example is a repeat of the fibonacci function this time using guards rather than the if-then-else. In this case the domain of the generator is finite so the function will be evaluated fully. In the second example an absolute value function is defined. Here there are four expressions giving both the function and its inverse.

As can be seen from the example of the absolute function the only way to be able to find the inverse of a function when the domain is infinite is to supply the necessary expressions to calculate it. If the inverse function is not provided when the domain is infinite then it is impossible for DEST to determine any inverse images since it can not enumerate the function.

9. Alternative Labelling System

Labels provide a parallel system for referencing elements which can be used to complement the use of the literal's name or to enhance the power of generators. Labels act as a local naming scheme in a set that can act on elements of any type. Also labels are produced automatically for cross products by combining the labels present in the product's arguments. Hence by labelling elements appropriately the interpreter will produce a suitable labelling for the new set. The elements of a set's definition can be referenced by two labelling systems called enumeration labels and set labels.

Enumeration Labels: By requesting an enumeration of a set, all the elements of the set will be associated with a unique integer. For unordered sets the number associated with each element will be time dependent and the element will only have that value while the definition and its associated environment remains constant. For ordered sets the number associated with each element will be in accordance with the element's position in the set, the element at the head having label "`:1`". :

Set Labels: Set labels are names given by the user to the elements of a set's definition, either when the definition is first given or later by attaching a name to an element using an enumeration label. Unlike enumeration labels, set labels retain their meaning over time. Set labels can be used in patterns and hence provide the same means of specifying structured sets as literals.

For example if `S` is defined as

```
S := { e1, s1, _a, _b, s2 :: { _a, s1 } }
```

then the enumeration labels might be `S:1 = e1`, `S:2 = s1` etc. The only set label defined so far is `S:s2`. Note that an underscore is not required for labels since their names are always prefixed by a colon.

Labels provide a means of accessing the value of the elements of a set and not a way to change them, that is it should be noted that labels are not *l-values* and can not have expressions assigned to them. To be able to change an element via a label would usurp the definition of the set and the elements.

The operator cross product will create new labels for the elements of sets defined as a product from the labels of the arguments. Let `C:=A*B` and let `e1` be a variable in the definition of `A` and `e2` a variable in the definition of `B` where `e1` has set label `:lab1` and `e2` has set label `:lab2`. The element representing `(e1,e2)` in `C` will have set label `:lab1_lab2`. If either `e1` or `e2` are labelless then `(e1,e2)` will be labelless. For example if `B` is a set of elements with set labels `:L0, :L1, :L2, ...` then the definitions

```
S  := { B * B }
O1 := { (:L$1, :L$2) in B*B | $1 >= $2 }
O2 := { :L$1_L$2 in S | $1 >= $2 }
```

will define O1 and O2 as a set of pairs which defines a total order on B. The definition of O1 uses labels in generators and the definition of O2 uses the creation of labels by the product operator to act as a pattern.

Labels can also be applied to elements that appear in a recursive definition. Since a label is local to a set and appears only in conjunction with a set it is possible for a set to have several instances of the same label at different levels. For example a recursive map could be defined as:

```
rec := { [$1=1] -> ( 1, {t::_a}) ?
                   ($1, {t::_a, T::rec($1-1)} ) | $1 in Z }
```

Hence `rec(3)` equals `{t::_a, T::{t::_a, T::{t::_a }}}`, so `rec(3):t` is the literal `_a` and `rec(3):T` is the set `{t::_a, T::{t::_a }}`. Since `rec(3):T` is a set which contains labels the above process can be repeated, for example `rec(3):T:T:t` etc.

10. Infinite Sets

DEST has two predefined infinite sets, the set Z consisting of the integers and N of the natural numbers (including zero). As has been shown earlier infinite sets of literals can be defined by basing a generator on one on these sets:

```
B :=  { _L$1 | $1 in N }
```

Furthermore it is possible to define an order on infinite sets using pattern matching with generators. For example the definition

```
O := (relation) { (_L$1, _L$2) in B*B | $1 >= $2 }
```

will define O to be a total order on the set B. This order can be used to compare literals as in any finite order because by using pattern-matching it is possible to determine if an ordered pair is a member of the relation without enumerating the expression.

DEST will not attempt to evaluate any expression that is considered to involve an infinite set since this may force the interpreter into an infinite loop. The only operator that can be applied to infinite sets is the membership operator `in`, since by pattern-matching it is possible to determine membership without enumerating the set. As can be seen even with this restriction it is possible to define and use maps and relations on infinite domains.

DEST is unable to determine whether an expression that involves infinite sets is infinite, for example the intersection to two infinite sets need not be infinite. This problem is common to most computer interpreters and the normal solution is to classify any set derived from an infinite set as unenumerable. However there are two exceptions to this, when an infinite set is either intersected with a finite set or equivalently specified as a subset of a finite set then the result is considered finite.

11. Syntax of DEST

```
dest :        expr
       |      print expr_list
       |      print_def variable
       |      variable := expr
       |      label   := variable label
       |      for variable in expr do dest_list end
       |

dest_list :   dest
       |      dest ; dest_list

expr :        { expr_list }
       |      ( tuple_list )
       |      { generator | expr }
       |      ( generator | expr )
       |      { expr_list | generator }
       |      ( expr_list | generator )
       |      eval ( expr )
       |      func_sym ( expr )
       |      variable ( expr )
       |      variable { expr }
       |      variable ^ ( expr )
       |      variable ^ { expr }
       |      variable . member
       |      expr operator expr
       |      expr label
       |      - expr
       |      not expr
       |      if expr then expr else expr
       |      ( type ) expr
       |      ( expr )
       |      variable
       |      literal
       |      label
       |      number
       |      pattern

expr_list :   ele_expr
       |      ele_expr , ele_expr expr_more
       |

tuple_list :  ele_expr , ele_expr expr_more
       |      integer_range
       |

expr_more : , ele_expr expr_more
```

```
ele_expr :  variable :: expr
      |     integer_range
      |     [ expr ] -> expr
      |     [ expr ] -> expr ? expr
      |     expr

generator : expr in expr

func_sym :  <predefined function>

operator :  +    -    *    &    @
      |     mod  div  and  in   or
      |     >    <    <=   =>
      |     =    <>   ||
      |     relation_order
      |     ordered_set_order

integer_range :   number .. number
      |     number ; number .. number

literal :   _ <alphanumeric> +
      |     true
      |     false

number :    <numeric> +
      |     - number

variable :  <alphabetic><alphanumeric>*

label :      : <alphanumeric> +

pattern :   <alphanumeric + $> +

alphabetic :      < a..z, A..Z>
alphanumeric :    < a..z, A..Z, 0..9, _>
```

12. Predefined Functions and Operators

Operator	Operation
+	set union, arithmetic addition
&	set intersection
-	set difference, arithmetic subtraction
*	cross product, arithmetic multiplication
@	composition of maps
mod	modulo arithmetic
div	arithmetic division
in	set membership, generator symbol
and	logical and
not	logical not
or	logical or
f(x)	map image of an element
f(X)	map image of a set
f^(x)	inverse image of an element
f^(X)	inverse image of a set
x..y	integer numbers from x to y
x;y..z	arithmetic progression x, y · · · z
<	arithmetic less than, strict set inclusion
>	arithmetic greater than, strict set inclusion
<=	arithmetic less than or equal to, set inclusion
=>	arithmetic greater than or equal to, set inclusion
=	equality
<>	not equal
||	not comparable
~r~	relation test on the relation r
<p<	
<p=	
>p>	
=p>	comparison test on the ordered set p
=p=	
<p>	
|p|	
::	definition of a set label
$1	pattern variable
$$	iteration count in for loops

Function	Argument Type	Operation
eval()	—	Evaluate operand and substitute value
type()	—	Return type of variable as a literal
elem()	set	Return an element from the operand if it is a set
union()	set	Return the set union of the operand
inter()	set	Return the set intersection of the operand
is_singleton()	set	True if the operand is a singleton set
is_empty()	set	True if the operand is the empty set
is_set()	—	
is_order()	—	
is_pairs()	—	
is_tuple()	—	
is_map()	—	True if operand is of that type or can
is_relation()	—	automatically coerced to that type
is_pair()	—	
is_literal()	—	
is_element()	—	
is_injective()	map	True if operand is an injective map
is_transitive()	pairs	True if operand is a transitive relation
is_reflexive()	pairs	True if operand is a reflexive relation
is_symmetric()	pairs	True if operand is a symmetric relation
is_antisym()	pairs	True if operand is an anti-symmetric relation
is_equivalence()	pairs	True if operand is an equivalence relation
is_partial()	pairs	True if operand is a partial order
is_preorder()	pairs	True if operand is a pre-order
is_total()	pairs	True if operand is a total order
transitive()	pairs	Return the transitive closure of the set
symmetric()	pairs	Return the symmetric closure of the set
reflexive()	pairs	Return the reflexive closure of the set
equivalence()	pairs	Return the equivalence closure of the set
partial()	pairs	Return the partial order closure of the set
total()	pairs	Return the total order closure of the set

Appendix Two

Pecan - User Manual

Pecan - A Definitive Environment for Lattice Theory

User Manual

John Buckle

Pecan[1] is an interactive interpreted language enabling the user to construct and investigate finite lattices. The language is based on a definitive notation where variables in Pecan store expressions rather than values. This creates a dynamic environment where the user can experiment with several different kinds of lattices while letting the computer maintain the functional relationships between variables.

Pecan contains the language DEST as a subset and it is assumed that the reader is familiar with the language. It is also assumed that the reader is familiar with lattice theory and understands the terms quotient lattice, homomorphism theorem etc. and is aware of the identification between congruences relations and join-irreducibles. This document contains the minimal specification of the language Pecan, local versions of the language may contain extra features.

[1] Pecans are small edible nuts of the walnut family.

1. Introduction

Pecan is a definitive based language designed for interactive analysis of lattices. Pecan contains the language DEST as a subset and incorporates additional data types and operators to handle lattices. While DEST can handle infinite sets to a limited degree, Pecan is restricted to finite lattices so that the identification between lattice congruences and sets of join-irreducibles can be applied[2]. Operations in Pecan are mainly based on the construction of lattices and the investigation of congruences. Congruences in Pecan can either be defined via a relation expression or by giving an appropriate set of join-irreducibles. Special operators exist for defining quotient lattices and natural homomorphisms between lattices and quotients, and the lattice of all congruence relations on a lattice can also be defined.

As well as using the methods available in DEST to define lattices, Pecan introduces a less formal, more intuitive method of constructing lattices using a cut and paste method. In this method the user constructs a Hasse diagram of the lattice from basic blocks which are combined to produce the final order. By this method it is relatively easy to construct complicated lattices.

To be able to work efficiently with lattices Pecan needs to enumerate the lattice and pre-calculate the table of meets and joins. Hence the user is required to *open* a lattice to indicate that this information should be calculated. The process of opening a lattice introduces two levels of definitions in Pecan. On one level the user can define an algebra and on a second level the user can define expressions based on the algebra.

2. New Data Types for Pecan

Pecan has three extra data types not included in DEST for handling lattices, congruences and homomorphisms. The data types `congruence` and `homomorphism` are extensions of the DEST types relation and map and are used in defining quotients. The type `lattice` is an extension of the type order and includes new member functions for accessing the components of the algebra.

Lattices

Abstractly lattices can be defined either as a special type of poset or as a special class of algebras. To reflect these two methods of definition Pecan allows variables of type `lattice` to be defined either by giving a poset or by specifying a set with two operators.

```
L1 := (lattice) P
L2 := (lattice) ( S, m, j )
```

In the example `L1` is defined by giving a partial order and `L2` is defined by giving an algebra. In the second case `S` should be a set and `m` and `j` maps from $S{\times}S \rightarrow S$ representing the operators meet and join.

Since lattices can originate from two different sources, variables of type `lattice` have four member functions reflecting the two methods available. The member functions

[2] Buckle, J. *Computational Aspects of Lattice Theory*

`.set` and `.order` of orders are inherited by lattice variables as well as two new functions called `.join` and `.meet`. The operations of join and meet on a lattice `L` can either be expressed using the infix operators `\L/` and `/L\` or by using the member functions, for example the expression "x `\L/` y" is equivalent to "`L.join(x, y)`".

To be able to implement the operators meet and join efficiently and to be able to calculate quotients etc., large amounts of information are required to be calculated for each lattice variable. Hence while lattices can be defined and used in definitions at any time, the elements of a lattice can only be accessed after the lattice has been explicitly *opened* for use by the user. In opening a lattice a full enumeration of the lattice is performed and all join-irreducibles and meet-irreducibles in the lattice are found plus how they are related by the ~-relation. To aid brevity the infix operators of meet and join of the most recently opened lattice may be referred to as `/\` and `\/`.

To stop massive recalculations caused by accidentally redefining a lattice expression, all opened lattices are tagged so that in the event of a redefinition the user is warned and is given the option to cancel the operation. In this way Pecan is arranged as a two layer definitive language where the user defines lattices and then performs calculations in and on them. When the user wishes to investigate another system of lattices these lattices must be re-calculated by opening the definitions.

Once the enumeration of the lattice has been performed the elements of the lattice are given enumeration labels according to the order in which the elements appear in a topological sort of the lattice, the minimal element being given label `:0`. Enumeration labels are used by operators like quotient and cross product in generating set labels for elements of the newly created lattice.

Several standard lattices are included in Pecan. Chains of arbitrary length can obtained by using the function `chain()` or by using the natural and integer number posets `N` and `Z`. Free distributive lattices can be obtained from the function FDL(n) where n is the number of generating variables. All lattices of five or less variables are also defined. These standard lattices can be combined to define arbitrary lattices using a cut and paste technique. More information about this process is given in section four.

The table of meet and join operators on a lattice can be printed using the `display` command. The command will list the element's enumerated and set labels followed by the product table and a list of meet and join irreducibles and how they are related under the ~-relationship. To save space the meet and join tables are printed together with meets occupying the upper triangular portion and joins occupying the lower triangular portion.

Congruences

Congruences can either be defined by coercing an expression of type relation or by specifying the set of join-irreducibles that determines the congruence. If the former method is used then upon evaluation of the congruence all join-irreducibles not related to the element they cover are located and the ~-closure of this set is used to determine the congruence. In the case that the relation expression is a congruence then the resulting congruence is equal to the relation. If however the relation was not a congruence then obviously there may be little connection between the two. (The only thing that can be said is that they agree on a subset of the join-irreducibles.) If the second method is used then the ~-closure of the set of join-irreducibles is calculated and this set is then used to determine the congruence.

Congruence variables have a new member function `.join` which returns the set of join-irreducibles determining the congruence. Since it is required that the —closureof the set of join-irreducibles determining the congruence is calculated, any definition of a congruence has to be bound to a definition of a lattice. This is expressed by casting a congruence or relation expression, stating the lattice to which the congruence is to be bound:

```
    C := (congruence on L) R
```

In this case `C` is a congruence on `L` generated by the relation `R`. The congruence class of an element `e` is given by the expression `[e]C`. To change the base lattice of a congruence simply re-cast it onto the new lattice. Since the relation/congruence expression is bound to the lattice `L`, all occurrences of the operators `\/` and `/\` will be taken to apply to `L` rather than the most recently opened lattice.

The lattice of congruence relations on a lattice can be obtained from the function `congruences()`. The elements of the lattice are sets of join-irreducibles determining the congruences on the base lattice

Homomorphisms

Homomorphisms can be defined in much the same way as maps, however like congruences it is necessary to specify the lattices between which the homomorphism acts:

```
    H := (homomorphism L1 to L2) M
```

If the homomorphism is defined by using an expression bound to a generator then all occurrences of the operators `/\` and `\/` in the expression will be taken to be in the range lattice. If the expression has guards then occurrences of the operators in the guards will be taken to be in the domain lattice. In this way it is possible for the same map definition to be used between several different lattices.

3. Quotient and Product Lattices

Lattices, congruences and homomorphisms are all connected by quotient lattices in the Homomorphism Theorem which states that every homomorphic image of a lattice L is isomorphic to a suitable quotient lattice of L. More precisely if $\phi: L \to L_1$ is a homomorphism from L onto L_1 and Φ is the congruence relation on L defined by

$$x \equiv y\,(\Phi) \text{ if and only if } x\phi = y\phi \qquad (1)$$

then $L/\Phi \cong L_1$ and the map $\psi: [x]\Phi \to x\phi$ is a isomorphism.

If `L` is a lattice variable and `C` congruence variable bound to `L` then the quotient lattice can be defined as follows:

```
    Q := L / C
```

The elements of `Q` will be presented as intervals $[e_1, e_2]$ of `L`. Each element of `Q` will be assigned an enumeration label according to its topological order in `Q` and a set label of the form $:\#n_1\_n_2$ where n_1, n_2 are the enumeration labels of the minimal and maximal elements in the congruence class in `L`.

The natural homomorphism $H: L \rightarrow Q$ can be defined as follows:

```
H := (homomorphism L to Q) {($1, [$1]C) | $1 in L }
```

This definition however does not specify an inverse transform and is hence very inefficient in calculating the pre-image of an element in `Q`. Since all of the information necessary to calculate both the image and pre-image of the natural homomorphism is given by the set and enumeration labels of `L` and `Q` the natural homomorphism can be defined just by saying:

```
H := (homomorphism L to Q) natural
```

If `L1` and `L2` are two isomorphic lattices then the function `isomorphism()` returns an isomorphism from `L1` to `L2`. If `L1` and `L2` are not isomorphic then an empty map (causing an error if used) is returned. The kernel of a homomorphism (that is the congruence relation defined by (1) above) can be obtained by the function `kernel()`

```
C1 := kernel( H1 )
```

The congruence `C1` is bound to the domain of `H1`. If `S` is a subset of a lattice `L` then the sublattice generated by `S` is obtained by the function `sublattice()` and the embedding of `S` into `L` is obtained from `embedding()`.

The product of two lattices `L` and `M` is given by `L*M`. Each element (x,y) in the product is given the set label $:\#n_1\_n_2$ where n_1, n_2 are the enumeration labels of the elements x and y in `L` and `M`. The projection homomorphism from `L*M` to `L` and `M` is defined by saying:

```
P     := L * M
left  := (homomorphism P to L) projection
right := (homomorphism P to M) projection
```

Example

If `L`, `L1` are lattices and `H` is a homomorphism from `L` to `L1` then the homomorphism theorem can be demonstrated by the following definitions and queries:

```
IH :=  H(L)                   // Calculate image of H.
L2 :=  sublattice(IH,L1)
H2 :=  embedding(L2, L1)
C  :=  kernel(H)              // Obtain suitable congruence
Q  :=  L/C                    // and quotient.
H1 :=  (homomorphism L to Q) natural
I  :=  isomorphism(Q,L2)

print Q = L1                  // True if H is onto.
print Q = L2                  // Isomorphic lattices.
print I@H1 = H2@H             // Identical maps.
```

Sets and Orders

Any set or order can be bound to a lattice to express the fact that all lattice operations are to be performed in that lattice. For example a set `P` can be bound to a lattice `M` by casting `P`

```
P := (set on M) { ... }
```

To bind `P` onto a new lattice then `P` has just to be recast, not redefined,

```
P := (set on L)
```

By casting sets onto lattices operators like `tilde()`, `meet()`, `join()` etc. know within which lattice to work. If the set is not bound to any lattice then the lattice has to be provided as an argument to the function.

Orders can also be bound to a lattice, in this case only the relation has to be provided since the base set is taken to be the lattice. While general orders do not need to be bound to a lattice, certain functions that return lattice pre-orders do need to be bound to be used. For example the function `preorder(X, Y)` returns the pre-order determined by the sets of join-irreducibles `X` and `Y`.

```
pre := (order on L) preorder(X,Y)
```

Hence the lattice `L` has two orders defined on it, the partial order `=L>` and the pre-order `=pre>`. Orders can be carried through to quotient lattices by using the natural homomorphism. For example the pre-order `pre` defined above can be defined on a quotient lattice `Q` of `L` by saying

```
preQ := (order on Q) natural(pre)
```

It is assumed that the pre-order `pre` respects the operations of lattice, if it dosn't then the resulting order is undefined.

Computational Equivalence and Replaceability

Special functions exist in Pecan to determine the computational equivalence congruence and replaceability pre-order. If `L` is a lattice and `f` an element in `L` then the functions `minimal(L, f)` and `maximal(L, f)` return the set of minimal meet-irreducibles greater than f and the set of maximal join-irreducibles less than f respectively. The set of elements contained in a set `P` that are less than an element `e` is given by `P(e)` . In this case the set `P` must be bound to a lattice. The sets of join-irreducibles that determine the replaceability pre-order are obtained by the functions `comp_rep_p(L, f)` and `comp_rep_q(L, f)` . Hence the replaceability pre-order and the computational congruence can be defined by

```
Pf  := comp_rep_p(L,f)
Qf  := comp_rep_q(L,f)
pre := (order on L) preorder(Pf,Qf)
equ := (congruence on L) ( Pf + Qf )
```

The pre-order and congruence can be defined directly however by using the functions `comp_rep(L, f)` and `comp_equ(L, f)` .

4. Constructing Lattices

While lattices can be defined in terms of quotients, images, products etc. of other lattices and partial orders it is necessary at some point to explicitly define a lattice without reference to any other variable. As was indicated in the section two, lattices can be defined by expressing a partial order or an algebra where the corresponding sets of ordered pairs (order relation in the case of a poset and the meet and join operators in the case of an algebra) are given in full. Even in a small lattice the definition of the partial order by this method would be extremely tedious and likely to be error prone. The use of generators and pattern matching allows larger lattices to be defined, however lattices defined this way tend to have an extremely uniform structure (eg. total orders or boolean algebras) and hence restrict their use.

To allow larger and more complex lattices to be defined Pecan includes several *basic* and *pre-defined* lattices which can be used to construct general lattices. The basic lattices consist of all the lattices with five or less elements, the eight element lattice 2^3, the free boolean lattice on two variables FBL(2) and the free distributive and the free modular lattices on three variables, FDL(3) and FML(3). The pre-defined lattices consist of the free distributive lattices on eight or less variables. The difference between basic and pre-defined lattices is that basic lattices are stored explicitly while the pre-defined lattices are represented and manipulated algebraically.

Construction of Lattices from Basic Lattices.

Arbitrary lattices can be constructed by combining several basic lattices to represent an ordered set, which is then coerced into a lattice. The general principal behind this method is that the user draws a Hasse diagram of the lattice then identifies overlapping sublattices of the sort given in the basic set. The Hasse diagram is finally reconstructed by identifying overlapping elements of the basic lattices producing a connected diagram. This process only creates an ordered set, this set has then got to be converted into a partial order and finally a lattice. However it does produce concise and easy method of constructing arbitrary lattices.

To isolate the construction process from the main definitive environment the lattice construction is performed in a separate environment, hence only the resulting ordered set produced by the construction will be visible to the definitive environment. All lattice constructions are stored in a separate file so that the user can redefine and reuse lattices from previous sessions with Pecan. New lattices are constructed by the command `construct` *<latticename>* which opens the file containing the user constructed lattices and allows the user to edit the file. When the lattice has been entered the lattice name can then be used as a basic lattice, including the construction of further lattices. To read in a pre-constructed lattice from a file or write the current definition of a lattice then the commands `read` *<latticename>* and `write` *<latticename>* should be used.

A lattice construction consists of a block of declarations where identifiers are assigned to basic lattices, followed by a block of element identifications where elements in different lattices are identified and finally followed by a block of order relations where individual elements can be ordered. To make the last two stages easier all the elements of a basic lattice are given set labels so that the elements can be identified. These set labels are carried through to the final lattice and provide an efficient method of labelling the elements.

The non-totally ordered basic lattices of five or less elements with their set labels are given in figure 1.

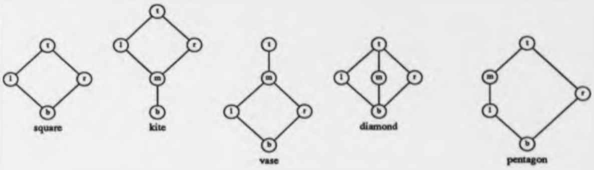


figure 1

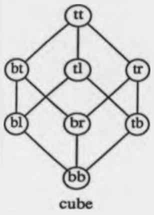
The set labels given by default to these lattices can be overridden when the basic lattice is first used in the declaration section by specifying alternative labels as arguments to the basic lattice. The alternative labels should be given in the order bottom, left, middle, right, top. For example the declaration

```
V := vase( "", "x", "z", "y", "" )
```

would leave `V:t` and `V:b` unaltered but change `V:l`, `V:m` and `V:r`.

Chains of elements can be declared with the `chain()` function. The elements are given enumeration labels starting at 0, and the top and bottom elements are given set labels `t` and `b`. Unless labels are given as arguments the middle elements in the chain will be given set labels of the form `C`*i*.

As well as being able to use basic lattices in the declaration block it is also possible to use cross products of basic lattices. In this case the set labels of the elements of the lattices are concatenated together. In this way the basic lattices 2^3 and FBL(2) are equivalent to the declaration `chain(2)*square` and `square*square`.



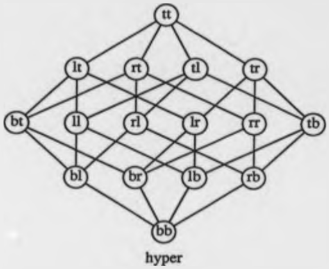


figure 2

As was said earlier set labels from the construction phase are carried forward into the Pecan environment. In the case where a set label of a basic lattice is not altered (eg. `V:t` from above) then the set label given to Pecan will be the concatenation of the variable name and the set label (eg. `:V_t` from above). If the set label of the basic lattice is changed then the label used by Pecan will be just the label given (eg. `:x` from above). If two elements are identified then the label of the element on the right will be used for both elements.

The lattice FML(3) is given as a basic lattice, however as an example it can be constructed as follows.

```
FML3 := {
// Define lattices
     B := cube ; M := diamond ; T := cube

     X := square("", "x"," mx", "" );
     Y := square("", "my", "y", "" );
     Z := square("", "mz", "z", "" );

// Identify common elements
     M:t := T:bb; M:b := B:tt
     M:l := X:mx; M:m := Y:my; M:r := Z:mz

// Add remaining order relations
     T:bl > X:t > X:b > B:bt
     T:br > Y:t > Y:b > B:tl
     T:tb > Z:t > Z:b > B:tr
}
```

There are obviously several methods of constructing FML(3), the above was chosen here to demonstrate all three stages of the construction. The final Hasse diagram with labels is given in figure 3. As can be seen several of the elements have unusual labels, these can be changed by the user using the normal procedure (eg. `L:T_bb:=:mt`). The set labels used for the lattice FDL(3) is given in figure 4.

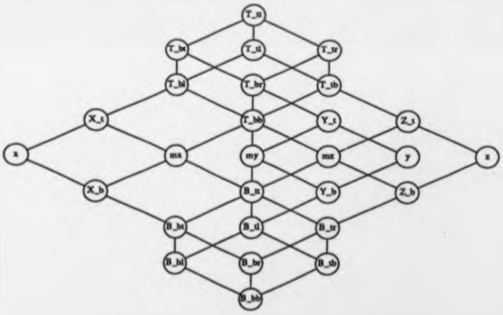


figure 3

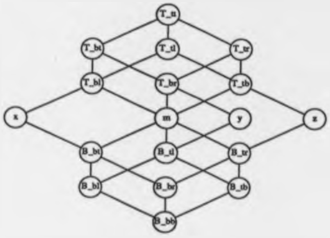


figure 4

Pre-defined Lattices

The pre-defined lattices consist of the free distributive lattices on three to eight variables. These lattices can not be represented in the normal fashion due to their size, however they can still be manipulated in the same way by using algebraic identities. The elements of the lattice are represented in disjunctive normal form by set labels, for example the element $x_1 x_2 x_4 \vee x_2 x_3 \vee x_3 x_4$ would be represented by the label `:abd_bc_cd`. Many of the operations associated with normal lattices can be applied to the pre-defined lattices, including quotients, homomorphisms, products etc. However operations which would result in enumerating the entire lattice or explicitly constructing a lattice too large for Pecan will be stopped.

The pre-defined free distributive lattices are obtained from the function `FDL()` . It should be noted that the basic lattice isomorphic to FDL(3) is called `FDL3`, not `FDL(3)` .

5. Predefined Functions and Operators

Operator	Operation
L/C	Quotient lattice of L over C
L*M	Cross product of two lattices
/\	meet of two elements
\/	join of two elements
/L\	meet of two elements in the lattice L
\L/	join of two elements in the lattice L
[*e*]C	congruence class of *e* in C
P(*e*)	elements $\le e$ in P

Function	Argument Type	Operation
kernel(h)	homo	Returns the kernel of h
congruences(L)	lattice	Returns the lattice of congruence relations
comp_rep(f)	element	Returns the replaceability pre-order $\sqsubset_f$
comp_equ(f)	element	Returns the computation equivalence congruence $\square_f$
square()	--	
pentagon()	--	
diamond()	--	
vase()	--	
kite()	--	
chain()	--	Returns a basic lattice
cube()	--	
hyper()	--	
FDL3()	--	
FML3()	--	
FDL()	--	Returns a pre-defined lattice
is_lattice()	--	
is_homo()	--	True if operand is of that type
is_congruence()	--	
is_distributive()	lattice	True if operand is an distributive lattice

Function	arg1	arg2	Operation
meet(L,X)	lattice	set	Returns the meet of X
join(L,X)	lattice	set	Returns the join of X
tilde(L,X)	lattice	set	Returns the ~of X
dual(L,x)	lattice	element	Returns the dual of x in a pre-defined lattice
sublattice(S,L)	set	lattice	Returns the lattice generated by S in L
embedding(K,L)	lattice	lattice	Returns the embedding $K \rightarrow L$
isomorphism(K,L)	lattice	lattice	Returns an isomorphism from K to L
is_join_irr(L,x)	lattice	element	True if operand is a join-irreducible
is_meet_irr(L,x)	lattice	element	True if operand is a meet-irreducible

minimal(L,f)	lattice	element	Returns the minimal meet-irreducible $\geq$ f
maximal(L,f)	lattice	element	Returns the maximal join-irreducible $\leq$ f
is_closed(L,X)	lattice	set	True if operand is closed under the ~-relation
closure(L,X)	lattice	set	Returns the tilde closure of the operand
preorder(X,Y)	set	set	Returns the pre-order determined by X and Y
comp_rep_p(L,f)	lattice	element	Returns the sets determining the replaceability
comp_rep_q(L,f)	lattice	element	pre-order $\sqsubset_f$ on L

Appendix Three

Prime Source Listings

NAME

prime – monotone boolean desk calculator

SYNOPSIS

prime

DESCRIPTION

Prime displays the disjunctive and conjunctive normal forms of monotone boolean expressions entered by the user and constructs planar monotone circuits. Arbitrary monotone expressions can be entered with the lower case letters "a" to "t" representing the free variables and the symbols "+" and "." representing boolean disjunction and conjunction. Conjunction has higher precedence than disjunction and the period symbol can be omitted, brackets can be used to force a particular evaluation order.

There are 14 expression variables denoted by the upper case letters "A" to "N" which can be assigned a value. The statment

F = a(b+c)+de

will set F to the value "a.b+a.c+d.e" and print the DNF and CNF of this expression. The statment

# F = a(b+c)+de

will assign F to the same value however will not print the result.

The special variable V stores the conjunction of the default set of free variables, initial set to "abcd". This value is used in the definitions of the threshold function T() and the functions u(), z(), U(), Z() (see below). To change the default set of variables V should be explicitly assigned the new *number* of default variables (hence the default variables always start from the variable "a").

The function **T** (*threshold_number* , *variable_range*) returns the threshold *threshold_number* function on *variable_range* . If *variable_range* is missing then the default variable range is used instead. For example to find the DNF and CNF of the threshold 3 function on variables b,d,e,f,g type

T(3,bdefg)

The functions **u** (*expression*), **z** (*expression*), **U** (*expression*) and **Z** (*expression*), return the unit, zero, μ and λ functions of the expression. Complements are taken with respect to the default set of variables unless the expressions involve new variables which are then included.

The functions **X** (*exp_1* , *exp_2*) and **Y** (*exp_1* , *exp_2*) return the minimum and maximum functions respectively that contain the prime implicants in *exp_1* and prime clauses in *exp_2* . The function **W** (*expression*) returns the dual of the expression.

Planar monotone circuits of functions can be constructed (if they exist) by giving the function as an argument to the pmc command:

pmc *expression*

The variables a,b,c... are assumed to be in left-right order and are initially referred as gates 1,2,3... *Prime* lists a sequence of gate operations either explicitly stating the individual gates or using the pyramid shorthand (see below and reference). When the verbose option is on a bit-table of gate values against prime implicants and clauses is displayed.

PYRAMIDS

The pmc command constructs the circuit using OR and AND pyramids. These consist of two pyramids slotted tip to tip resulting in a network that replaces the middle gates. For example a "v-pyramid from 1 to 5" is a truncated OR pyramid of 9 gates with an upside down AND pyramid of 6 gates slotted into the vacant apex of the OR pyramid. The three AND gates at the base of the pyramid now replace gates 2,3 and 4. The inputs to the outer AND gates are connected to the outputs of outer OR gates in order, starting with input of the new 2 gate joining the OR of gate 1 and the old gate 2 and ending with the input of the new 4 gate joining the OR of gate 4 and 5.

AUTHOR

John Buckle

SEE ALSO

For details on u(), z(), U() and Z() see:
BEYNON: Replaceability and computational equivalence for monotone boolean functions; (Acta Informatica, 22, 1985).

For details on planar monotone functions see:
BEYNON, BUCKLE: On the planar monotone computation of boolean functions; (Theoretical Computer Science, 53, 1987).

BUGS

This program is not designed to behave well when given wrong data, please do not expect sensible answers.

```
/**********************************************************************/
/*                                                                    */
/*              P R I M E   D E S K   C A L C U L A T O R             */
/*                                                                    */
/*              bnf.y - grammar for the YACC(1) parser.               */
/*                                                                    */
/**********************************************************************/

%{
#include "header.h"
%}

%union {
          int         INT ;
          unsigned  UNINT ;
          POINTER    NODES ;
          }

%token <INT>   IDENT   DIGITS   END   VAR   PMC   VERBOSE   ON   OFF   errSYM

%type  <UNINT>      idlist      Tidlist
%type  <NODES>      fact        exp        o_term     a_term
%type  <INT>        number

%start first

%%
first         :   prime   '\n'
                  { return( TRUE ) ; }

              |   error '\n'
                  { printf( ">> " ) ;   yyerrok ; }
                   first
                  { ; }

              |
                  { return( FALSE ) ; }
              ;

prime    :   exp
                  { treewalk( $1 ) ; }

              |   '@' VAR '=' exp
                  { setvariable( $2, $4, 0 ) ; }

              |   VAR '=' exp
                  { setvariable( $1, $3, 1 ) ; }

              |   'V' '=' number
                  { Noofvars = $3;  if (Noofvars > 20) Noofvars = 20 ; }

              |   PMC   exp
                  { pmc( $2 ) ; }

              |   VERBOSE ON
                  { verbose = TRUE ; }

              |   VERBOSE OFF
                  { verbose = FALSE ; }

              |

              ;
```

```
exp         :   o_term
            ;

o_term      :   a_term
            |   a_term '+' o_term
                        { $$  =  makenode( '+', $1, $3 ) ; }
            ;

a_term      :   fact
            |   fact a_term
                        { $$  =  makenode( '.', $1, $2 ) ; }
            |   fact '.' a_term
                        { $$  =  makenode( '.', $1, $3 ) ; }
            ;

fact        :   IDENT
                        { $$ = makenode( yylval.INT, PNULL, PNULL) ; }
            |   VAR
                        { $$ = makenode( yylval.INT, PNULL, PNULL) ; }
            |   'n' '(' exp ')'
                        { $$ = makenode( 'n' , $3, PNULL ) ; }
            |   'z' '(' exp ')'
                        { $$ = makenode( 'z' , $3, PNULL ) ; }
            |   'T' '(' number  Tidlist ')'
                        { $$ = makethreshold( $3 , $4 ) ; }
            |   'U' '(' exp ')'
                        { $$ = makenode( 'U' , $3, PNULL ) ; }
            |   'V'
                        { $$ = makenode( 'V', PNULL, PNULL ) ; }
            |   'W' '(' exp ')'
                        { $$ = makenode( 'W' , $3, PNULL ) ; }
            |   'X' '(' exp ',' exp ')'
                        { $$ = makenode( 'X' , $3, $5 ) ; }
            |   'Y' '(' exp ',' exp ')'
                        { $$ = makenode( 'Y' , $3, $5 ) ; }
            |   'Z' '(' exp ')'
                        { $$ = makenode( 'Z' , $3, PNULL ) ; }
            |   '('  o_term  ')'
                        { $$ = $2 ; }
            ;

number  :   DIGITS
                        { $$ = yylval.INT ; }
            ;

Tidlist :
                        { $$ = 0 ; }
            |   ',' idlist
                        { $$ = $2 ; }
            ;

idlist  :   IDENT
                        { $$ = 01 << $1 - 'a';}
            |   IDENT idlist
                        { $$ = ( $2 | (01 << $1 - 'a'));}
            ;

%%

POINTER
makenode( op, lhs, rhs )                    /* This routine is called during the
```

makenode

...makenode

```
POINTER    lhs, rhs ;                            /* parse of the input. Build a node to
int        op ;                                  /* hold the operation and pointers to
{                                                /* the subtrees.
POINTER pointer ;

           pointer = (POINTER) malloc( sizeof(TREE) ) ;
           pointer->OP = op ;
           pointer->LHS = lhs ;
           pointer->RHS = rhs ;

           return( pointer ) ;
}
```

makethreshold

```
POINTER
makethreshold( num, a )
int        num ;
NUMBER     a ;
{
POINTER makenode() ;
           return( makenode( 'T' , makenode( num, PNULL, PNULL ) ,
                                          makenode( (int) a, PNULL, PNULL ) ) ) ;
}
```

yyerror

```
yyerror( s )
char * s ;
{
           fprintf( stderr, "%s\n", s ) ;
}
```

```
/************************************************************************/
/*                                                                       */
/*          P M C - P L A N A R   M O N O T O N E   C O M P U T A T I O N  */
/*                                                                       */
/*          text - tokens for Lex(1) lexical analyser generator          */
/*                                                                       */
/************************************************************************/

%{
#include "header.h"
#include "y.tab.h"
%}

%%

quit            { return( 0 ) ; }
pmc             { return( PMC ) ; }
verbose         { return( VERBOSE ) ; }
on              { return( ON ) ; }
off             { return( OFF ) ; }

[\t ]           { ; }

[a-t]           { yylval.INT = yytext[0] ; return( IDENT ) ; }

[A-N]           { yylval.INT = yytext[0] ; return( VAR ) ;}

[TUVWXYZuz]     { return( yytext[0] ) ; }

[#,=+()$.]      { return( yytext[0] ) ; }

[0-9]+          { yylval.INT = atoi( yytext ) ; return( DIGITS ) ; }

[\n]            { return( '\n' ) ; }


%%
```

```
/**************************************************************************/
/*                                                                        */
/*                  P R I M E   D E S K   C A L C U L A T O R             */
/*                                                                        */
/*                  com.c - main arthimatic evaluation module             */
/*                                                                        */
/**************************************************************************/

#include "header.h"

FILE        * fp, * popen() ;

NUMBER      zero = 1,
            * variables[2][14] ;

char        * morefilter, * getenv() ;

int         Noofvars = 4 ;

main()                                                                      main
{
register i;

            verbose = TRUE ;

            for ( i = 0; i < 14; i++ )
                    { variables[ DUAL ][i] = variables[ PRIME ][i] = & zero ; }

            if ( (morefilter = getenv( "PAGER" )) == NULL )
                    morefilter = "/usr/ucb/more" ;

            do {
                    printf(">> ") ;
                    fflush( stdout ) ;
            } while ( yyparse() ) ;
}

setvariable( var, poi, printflag )             /* Set a variable to the value         setvariable
int         var, printflag ;                   /* of the expression.
POINTER poi ;
{
NUMBER      * primewalk() ;

            if ( var < 'A' | var > 'N' ) return;
            var -= 'A';

            variables[ PRIME ][var]         = primewalk( poi, PRIME ) ;
            variables[ DUAL  ][var]         = primewalk( poi, DUAL ) ;

            if ( ! printflag ) return;

            printvalues( variables[ PRIME ][ var ], variables[ DUAL ][ var ] ) ;
}

NUMBER      *
getvariables( from )                           /* Return the conjunction of           getvariables
int         from ;                             /* the current set of variables
{
NUMBER      * p ;
int         i ;
            if ( from == PRIME ) {
                    p = num_alloc( 2 ) ;
                    p[1] = p[0] = 0 ;
                    for ( i = 0 ; i < Noofvars ; i ++ )
```

```
                                p[0] = ( p[0] << 1 ) | 01 ;
                        }
          else          {
                        p = num_alloc( Noofvars + 1 ) ;
                        p[ Noofvars ] = 0 ;
                        for ( i = 0 ; i < Noofvars ; i ++ )
                                p[i] = 01 << i ;
                        }
          return( p ) ;
}

treewalk( pointer )                         /* This is called when input is           treewalk
POINTER pointer;                            /* complete and evaluation begins.
{
NUMBER    * pi, * pq, * primewalk() ;

          pi = primewalk( pointer, PRIME ) ;
          pq = primewalk( pointer, DUAL ) ;

          printvalues( pi, pq ) ;
}

          /* This routine returns a pointer to a string of unsigned                      */
          /* integers terminated by a zero marker. Variables are stored                  */
          /* as bit vectors, variable 'a' being bit 0 and variable 't'                   */
          /* being bit 19.                                                               */

NUMBER    *
primewalk( pointer, from )                                                          primewalk
POINTER   pointer ;
int       from ;
{
int       i, k, lenp1, lenp2 ;
NUMBER    * p1 , * p2, * p3, * threshold(), * getvariables(),
          * uzfunc(), * UZfunc(), * XYfunc() ;

          switch ( pointer->OP ) {
          case 'T' :            return( threshold( pointer, from ) ) ;
          case 'V' :            return( getvariables( from ) ) ;
          case 'W' :            return( primewalk( pointer->LHS, ! from ) ) ;
          case 'u' :            return( uzfunc( pointer, from, UFUNC ) ) ;
          case 'z' :            return( uzfunc( pointer, from, ZFUNC ) ) ;
          case 'X' :            return( XYfunc( pointer, from, XFUNC ) ) ;
          case 'Y' :            return( XYfunc( pointer, from, YFUNC ) ) ;
          case 'U' :            return( UZfunc( pointer, from, UFUNC ) ) ;
          case 'Z' :            return( UZfunc( pointer, from, ZFUNC ) ) ;
          }

          if ( islower( pointer->OP ) ){
                    /* We are at a leaf, start a new string                          */
                    p1          = num_alloc( 2 ) ;
                    p1[0]       = 01 << (pointer->OP - 'a') ;
                    p1[1]       = 0;
                    return( p1 ) ;
                    }

          if ( isupper( pointer->OP ) ){
                    /* We are at a function, insert value.                           */
                    k = pointer->OP - 'A' ;
                    if ( k > 14 | k < 0 ) k = 0 ;
                    lenp1       = veclen( variables[ from ][k] ) ;
                    p1          = num_alloc( lenp1 + 1 ) ;
                    for ( i     = 0 ; i <= lenp1 ; i++ )
                                p1[i] = variables[ from ][k][i] ;
```

...primewalk

```
                return( p1 ) ;
                }

                                                /* Calculate .l+ of the left and right
                                                /* subtrees recursively.
        p1 = primewalk( pointer->LHS, from ) ;
        p2 = primewalk( pointer->RHS, from ) ;
        lenp1 = veclen( p1 ) ;
        lenp2 = veclen( p2 ) ;

        if ( (pointer->OP == '.') == (from == PRIME) ){
                                        /* Must concatenate each word form the left
                                        /* with each word from the right.
                p3 = num_alloc( lenp1*lenp2+1 ) ;
                p3[ lenp1*lenp2 ] = 0 ;
                for ( i = 0 ; i < lenp1 ; i++ )
                        for ( k = 0 ; k < lenp2 ; k++ )
                                p3[i*lenp2+k] = p1[i] | p2[k] ;
                }
        else {
                                        /* Must string all words from both subtrees
                                        /* together.
                p3 = num_alloc( lenp1+lenp2+1 ) ;
                p3[ lenp1+lenp2 ] = 0 ;
                for ( i = 0 ; i < lenp1 ; i ++ )
                        p3[i] = p1[i] ;
                for ( k = 0 ; k < lenp2 ; k ++ )
                        p3[k+lenp1] = p2[k] ;
                }

        free( (char *) p1 ) ; free( (char *) p2 ) ;
        removewaste( p3 ) ;
        return( p3 ) ;
}
```

removewaste

```
removewaste( p )                                /* This removes redundant implicants
NUMBER   * p ;                                  /* from the function p.
{
int m, k, i ;
        m = veclen( p ) ;
        for ( i = 0 ; i < m-1 ; i++ )
                for ( k = i+1 ; k < m ; k++ ) {
                        if ( (p[k] & p[i]) == p[i] ){
                                p[k--] = p[--m] ;               /* p[i] <= p[k]
                                p[m] = 0 ;                      /* remove p[k]
                                }
                        else if ( (p[k] & p[i]) == p[k] ){
                                p[i] = p[k] ;                   /* p[i] > p[k]
                                p[k] = p[--m] ;                 /* remove p[i]
                                p[m] = 0 ;
                                k = i ;
                                }
                }
}
```

veclen

```
veclen( p )
register NUMBER     *p ;                                /* Return the number of clauses
{                                                       /* or implicants stored.
register int i = 0 ;
        while ( * p++ ) i ++ ;
        return(i) ;
}
```

..printvalues

printvalues

```
printvalues( p, q )                                  /* Open the output filter and
NUMBER   * p, * q ;                                  /* then call primeprint().
{
int          pipeflag = TRUE ;

             if ( (fp = popen( morefilter, "w" )) == NULL ) {
                     fp = stdout ;
                     pipeflag = FALSE ;
                     }

             fprintf(fp, "Prime implicants -\n" ) ;
             primeprint( p ) ;

             fprintf( fp, "Dual  implicants -\n" ) ;
             primeprint( q ) ;

             if ( pipeflag) pclose( fp ) ;
}
```

primeprint

```
primeprint( p )                              /* Prints the variables stored in the
NUMBER   *p ;                                /* NUMBER string p.
{
int x, c ;
                                             /* Get the implicants/clauses in a
                                             /* consistent order.
             quicksort( p, 0, veclen( p ) - 1 ) ;

             p -- ;
             while (*++p) {                  /* This loops until there are no words
                     c = 'a' ;               /* left.
                     x = * p ;
                     while (x) {             /* Loops until all variables have been
                                             /* printed from this word.
                             if ( x & 01 ) fprintf(fp,"%c",c) ;
                             c++ ;
                             x >>= 1 ;
                             }
                     fprintf(fp,"\n\t\t\t") ;
                     }
             fprintf(fp,"\n") ;
}
```

quicksort

```
quicksort( p, m, n )                         /* Standard quicksort algorithm.
NUMBER   * p ;
int          m, n ;
{
register i, j ;
unsigned q, k ;
             if ( m < n ){
                     i = m ;
                     j = n+1 ;
                     k = p[m] ;
                     while (i < j){
                             do i++; while ( p[i] < k && i<=j ) ;
                             do j--; while ( p[j] > k && i<=j ) ;
                             if ( i < j ) { q = p[i] ; p[i] = p[j] ; p[j] = q ; }
                             }
                     q = p[m] ; p[m] = p[j] ; p[j] = q ;
                     quicksort( p, m, j-1 ) ;
                     quicksort( p, j+1, n  ) ;
                     }
}
```

```
/**********************************************************************/
/*                                                                    */
/*               P R I M E   D E S K   C A L C U L A T O R            */
/*                                                                    */
/*               func.c - T(), u(), s(), U(), X(), Y(), Z()           */
/*                                                                    */
/**********************************************************************/

#include "header.h"

NUMBER    *
threshold( poi, from )                /* Return the prime implicant/clause      threshold
POINTER   poi ;                       /* representation of a threshold func.
int       from ;
{
int       i, j ;
NUMBER    * choose(), * ths, k ;
          j = poi->LHS->OP;                                /* Threshold number
          k = poi->RHS->OP;                                /* Identifers

          if (k == 0)
                    for ( i=0 ; i<Noofvars ; i++ ){  /* Default case, use
                              k <<= 1;                     /* currently defined
                              k++;                         /* variables.
                              }
          i = bitcount( k ) ;

          if ( j <= 0 || j > i ) {                         /* Handle zero and one
                    ths       = num_alloc( 1 ) ;           /* functions, (return
                    ths[0]    = 0 ;                        /* the zero function).
                    return( ths ) ;
                    }
          if ( from == DUAL ) j = i - j + 1 ;              /* Switch to dual case.
          return( choose( i, j, k ) ) ;
}

NUMBER *
choose( n, c, a )                     /* Return the list of combinations of c     choose
int       c, n ;                      /* objects out of n objects present in
NUMBER    a ;                         /* the word a.
{
int       k, m ;
NUMBER    *ths, y, *z, u ;

          if ( c == 1 ) {                                  /* End of recursion.
                    ths       = num_alloc( n+1 ) ;
                    ths[n]    = 0 ;
                    y         = 01 ;
                    while ( a ) {
                              if (a & 01) ths[--n] = y ;
                              y <<= 1 ;
                              a >>= 1 ;
                              }
                    return( ths ) ;
                    }

          k         = combinations( n, c ) ;               /* Find size of answer.
          ths       = num_alloc( k+1 ) ;
          ths[k]    = 0 ;

          y = 01 ; u = a ;                                 /* Get first variable.
          while ( !(u & 01) ) {
                    u >>= 1;
                    y <<= 1;
```

```
                }
        a ^= y;                                    /* Switch off that variable and
        z = choose( n-1, c-1, a );                 /* find all c-1 combinations.

        for (m = 0; z[m]; m++ )                    /* Transfer combinations across
                the[m] = z[m] | y ;                /* switching the variable on.
        free( (char *) z ) ;

        if ( n > c ) {                             /* Find all c combinations.
                z = choose( n-1, c, a );
                for ( k = 0; z[k]; k++ )
                        the[m++] = z[k];
                }
        free( (char *) z ) ;
        return( the );
}

combinations( n, c )                               /* Returns 'n' choose 'c'.        combinations
int             n, c ;
{
int             prod = 1, c2 ;

        if ( c == n-1 ) return( n ) ;              /* Handle big cases here.
        if ( c == n ) return( 1 ) ;

        for ( c2 = c ; c2 ; c2 — ) prod *= n — ;
        for ( ; c ; c — ) prod /= c ;
        return( prod ) ;
}

bitcount( u )                          /* Return the bitcount of the unsigned        bitcount
NUMBER   u ;                           /* number in u (ie. number of variables
{                                      /* set in the word).
register NUMBER x ;
register j, i = 0 ;
        x = u ;
        for( j = 24 ; j ; j— ){
                if (x & 01) i++;
                x >>= 1;
                }
        return( i );
}

NUMBER   *
getdual( func )                    /* Calculate the dual of func by reconstructing     getdual
NUMBER   * func ;                  /* a parse tree in DNF and then returning the
{                                  /* CNF of the tree.
POINTER  buildtree() ;
        return( primewalk( buildtree( func, '.' ), DUAL ) ) ;
}

POINTER
buildtree( p, c )                  /* Reconstruct a parse tree from the expression     buildtree
NUMBER   * p ;                     /* in p, c = '.' or '+' which is the operator
char c ;                           /* which is to go in the inner clauses.
{
POINTER clause() ;
char d ;
        d = (c=='.') ? '+' : '.' ;                    /* Get the other operator.

        if ( p[1] == 0 ) return( clause( 'a', p[0], c ) ) ;
        else        return( makenode( d, clause( 'a', p[0], c),
                                buildtree( p+1, c ) ) ) ;
}
```

```
POINTER
clause( ch, u, c)                   /* Reconstruct a '.' or '+' clause, returning            clause
char ch, c ;                        /* the parse tree.
NUMBER u ;
{
          while (!(u & 01) ){                         /* Get the variable.
                    u >>= 1;
                    ch ++;
                    }
          u >>= 1;                                    /* Delete the variable.

          if (u) return( makenode( c,
                    makenode( ch, PNULL, PNULL ), clause( ch+1, u, c ) ) );

          return( makenode( ch, PNULL, PNULL ) );
}

NUMBER *
uzfunc( poi, from, func )                             /* Calculate the unit and zero       uzfunc
POINTER poi;                                          /* functions of the function in
int from, func;                                       /* poi. The u and z functions
{                                                     /* are duals of each other.
int i;
unsigned com, *p1;
                                                                /* Get appropriate type
          if ( func == UFUNC )                                  /* of normal form.
                    p1 = primewalk( poi->LHS, DUAL );
          else      p1 = primewalk( poi->LHS, PRIME );

          com = 0 ;
          for (i=0 ; i < Noofvars ; i++){                       /* Get range of active
                    com <<= 1;                                  /* variables.
                    com ++;
                    }

          for (i = 0 ; p1[i] ; i++ )                            /* See if any more have
                    while ( p1[i] > com ){                      /* introduced.
                              com <<= 1;
                              com ++;
                              Noofvars ++;
                              }
                                                                /* Calculate the tilde
          for (i = 0 ; p1[i]; i++ )                             /* of each clause, and
                    p1[i] ^= com;                               /* interpret the func.
                                                                /* in the other form.
          if ( (func == UFUNC)  ==  (from == PRIME) )
                    return( p1 ) ;
          else      return( getdual( p1 ) ) ;
}

NUMBER *
UZfunc( poi, from , func )                            /* Calculates the lambda and mu      UZfunc
POINTER   poi ;                                       /* functions for the function
int       from, func ;                                /* in poi. Lambda and mu are
{                                                     /* dual functions.
int       i, k, lenp1, lenp2 ;
NUMBER    *p1, *p2, *p3 ;

          if ( func == UFUNC ) {              /* Calculate u( poi ) and poi itself
                                              /* in terms of prime implicants.
                    p1 = uzfunc( poi, PRIME, UFUNC ) ;
                    p2 = primewalk( poi->LHS, PRIME );
                    }
```

```
        else {                          /* Calculate z( poi ) and poi itself
                                        /* in terms of prime clauses.
                p1 = uzfunc( poi, DUAL, ZFUNC ) ;
                p2 = primewalk( poi->LHS, DUAL ) ;
                }

        lenp1 = veclen( p1 ) ;
        lenp2 = veclen( p2 ) ;

        p3 = num_alloc( lenp1 + lenp2 + 1 ) ;
        p3[lenp1+lenp2] = 0 ;
                                        /* Combine the two expressions together
        for (i=0;i<lenp1;i++)           /* to get the final result.
                p3[i] = p1[i];

        for (k=0;k<lenp2;k++)
                p3[i+k] = p2[k];

        removewaste( p3 );

        if ( (func == UFUNC)  ==  (from == PRIME) )
                return( p3 );
        else    return( getdual( p3 ) ) ;
}

NUMBER *
XYfunc( poi, from , func )                      /* Calculates the minimum and
POINTER   poi ;                                 /* maximum functions that have
int       from, func ;                          /* the implicants on the left
{                                               /* and clauses on the right.
NUMBER    * given, * given_poi,
          * extra, * extra_poi,                 /* Minimum function is obtained
          * result, * result_poi,               /* using PRIME and DIS, maximum
          com, com_var, c, u ;                  /* is obtained dually.
int       i, sum ;

        if ( func == XFUNC ) {                  /* Minimum function
                given = primewalk( poi->LHS, PRIME ) ;
                extra = primewalk( poi->RHS, DUAL ) ;
                }
        else {                                  /* Maximum function
                extra = primewalk( poi->LHS, PRIME ) ;
                given = primewalk( poi->RHS, DUAL ) ;
                }
                                                /* Find the current set of
        com = com_var = 0 ;                     /* active variables.
        for ( i = 0 ; i < Noofvars ; i ++ ) {
                com_var <<= 1 ; com_var ++ ;
                }
        for ( given_poi = given ; * given_poi ; given_poi ++ )
                com |= * given_poi ;
        for ( extra_poi = extra ; * extra_poi ; extra_poi ++ )
                com |= * extra_poi ;

        while ( com > com_var ) {               /* Final result is the combined
                com_var <<= 1 ;                 /* list of the `given' clauses
                com_var ++ ;                    /* plus the complements of the
                Noofvars ++ ;                   /* `extra' clauses, each with
                }                               /* one variable missing.

        sum = 0 ;                       /* Calculate upper bound on result size
        for ( extra_poi = extra ; * extra_poi ; extra_poi ++ )
                sum += bitcount( * extra_poi ) ;
        result = result_poi = num_alloc( num + veclen( given ) ) ;

                                /* Go through each result clause and add all
                                /* the new clauses with one variable missing
        for ( extra_poi = extra ; * extra_poi ; extra_poi ++ ) {
                if ( bitcount( * extra_poi ) == 1 ) continue ;
                u = 01 ;
                c = * extra_poi ;
                while ( u <= c ) {
                        if ( u & c ) * result_poi ++ = (c ^ u) ^ com_var ;
                        u <<= 1 ;
                        }
                }
                                /* Add all the given clauses and remove waste
        for ( given_poi = given ; * given_poi ; given_poi ++ )
                * result_poi ++ = * given_poi ;
        * result_poi = 0 ;

        removewaste( result ) ;

        if ( ( func == XFUNC ) == ( from == PRIME ) )
                return( result ) ;
        else    return( getdual( result ) ) ;
}

NUMBER  *
num_alloc( n )
int     n ;
{
        return( (NUMBER *) calloc( (unsigned) n, sizeof( NUMBER ) ) ) ;
}
```

```
/**********************************************************************-/
/*                                                                     */
/*        P M C - P L A N A R   M O N O T O N E   C O M P U T A T I O N */
/*                                                                     */
/*        pmctree.c - planar circuit generation module.                */
/*                                                                     */
/***********************************************************************/

#include "header.h"

#define    CON       1
#define    DIS       0
#define    PROPER    2
#define    LEFT      -1
#define    RIGHT     1

int        num,                        /* Number of free inputs
           num_ops,                    /* Number of gates used so far
           lenpf, lenqf,               /* Number of prime implicants/clauses
           activegates,                /* Number of gates still alive
           verbose,                    /* Whether to print full tables
           change ;

char  *    dead,                       /* Indicates if a gate is redundant
      **   PFX,                        /* Bit table of inputs and implicants
      ** QFX ;                         /* Bit table of inputs and clauses

pmc( tree )                            /* PMC - construct a circuit of the          pmc
POINTER    tree ;                      /* function specified in the parse tree
{
NUMBER     * pf, * qf ;
int        i ;

           pf = primewalk( tree, PRIME ) ;
           qf = primewalk( tree, DUAL  ) ;

           pmcinit( pf, qf ) ;         /* Initialise PFX, QFX, num etc.

           num_ops     = 0 ;
           change      = TRUE ;
           activegates = num ;

           while (change && activegates) {
                      change = FALSE ;
                      if ( finish() ) break ;
                      pyramid( PFX, QFX, lenpf, lenqf, DIS ) ;
                      pyramid( QFX, PFX, lenqf, lenpf, CON ) ;
                      if ( ! change )
                                for ( i = 0 ; i < num ; i++ ) improve( i ) ;
                      }
           if ( finish() )
                      printf( "Circuit constructed\n" ) ;
           else       printf( "Circuit does not exist\n" ) ;

           if ( ! verbose ) printveo() ;
}

pmcinit( pf, qf )                      /* Initialise the tables PFX, QFX so         pmcinit
NUMBER     * pf, * qf ;                /* PFX[i][j] is true if variable i is
{                                      /* in implicant j.
int        i, j, k ;
NUMBER     u = 0, v = 1 ;
```

```
        lenpf = veclen( pf );                       /* Get the number of implicants
        lenqf = veclen( qf );                       /* clauses and find the number
                                                    /* of free variables in use.
        for( i = 0 ; pf[i] ; i++ ) u |= pf[i] ;
        for( num = 0 ; u ; num++ ) u >>= 1 ;

        PFX = (char**) calloc( (unsigned) num, sizeof( char* ) ) ;
        QFX = (char**) calloc( (unsigned) num, sizeof( char* ) ) ;
        dead= (char* ) calloc( (unsigned) num, sizeof( char  ) ) ;

        for ( i = 0 ; i < num ; i++ ){
                PFX[ i ] = (char*) calloc( (unsigned) lenpf, sizeof( char ) ) ;
                QFX[ i ] = (char*) calloc( (unsigned) lenqf, sizeof( char ) ) ;
                dead[i]  = FALSE ;
                }

        for( i = 0 ; i < num ; i++ ){
                for ( j = 0 ; j < lenpf ; j++ )
                                PFX[i][j] |= ((pf[j] & v) != 0) ;
                for ( k = 0 ; k < lenqf ; k++ )
                                QFX[i][k] |= ((qf[k] & v) != 0) ;
                v <<= 1 ;
                }

        if (verbose) printvec() ;
}
```

improve

```
improve( middle )                       /* Find two neighbours for middle and then try
int             middle ;                /* to build a circuit around them. Only place a
{                                       /* gate if it is going to be constructive.
int             left, right, alive ;

        if ( dead[ middle ] ) return ;

        alive =         find_neighbour( middle, & right, RIGHT ) &&
                        find_neighbour( middle, & left, LEFT ) ;

        if ( alive ) {
                binaryop( middle, left,  PFX, QFX, lenpf, lenqf, DIS ) ;
                binaryop( middle, left,  QFX, PFX, lenqf, lenpf, CON ) ;
                binaryop( middle, right, PFX, QFX, lenpf, lenqf, DIS ) ;
                binaryop( middle, right, QFX, PFX, lenqf, lenpf, CON ) ;

                tripleop( middle, left, right, PFX, QFX, lenpf, lenqf, DIS ) ;
                tripleop( middle, left, right, QFX, PFX, lenqf, lenpf, CON ) ;
                }
}
```

find_neighbour

```
find_neighbour( middle, neigh, dir )                    /* Finds the next variable to
int             middle, * neigh, dir ;                  /* middle that is distinct from
{                                                       /* it. Returns false if middle
                                                        /* is now redundant.
        for ( * neigh = middle + dir ;
                        0 <= * neigh && * neigh < num ; * neigh += dir ) {
                if ( dead[ * neigh ] ) continue ;

                if ( Lesseq( middle, * neigh ) ){
                        dead[ middle ] = TRUE ;
                        printf("Killing variable %d\n", middle+1 ) ;
                        activegates -- ;
                        return( FALSE ) ;
                        }
                else if ( Lesseq( * neigh, middle ) ){
                        dead[ * neigh ] = TRUE ;
```

```
                                activegates -- ;
                                printf("Killing variable %d\n", * neigh+1 );
                                }
                        else break ;            /* Found a good neighbour.
                }
                return( TRUE ) ;
}
```

binaryop

```
binaryop( middle, other, FX1, FX2, len1, len2, op )
char **         FX1, ** FX2 ;                   /* Combine middle with other so that it
int             middle, other,                  /* gains some new implicants/clauses
                len1, len2, op ;                /* but doesn't lose anything.
{
int             i ;
char            op_char ;

                if ( other < 0 || other >= num ) return ;

                op_char = ( op == DIS ) ? 'v' : '^' ;

                if (            Included( FX1[middle], FX1[other],  len1 ) &&
                        ||      Included( FX2[other],  FX2[middle], len2 ) ) {
                        for ( i = 0 ; i < len2 ; i++ )
                                FX2[middle][i] |= FX2[other][i] ;
                        printf("%d = %d %c %d\n", middle+1, middle+1,
                                                op_char, other+1 ) ;
                        if (verbose) printvec() ;
                        change = TRUE;
                        num_ops ++ ;
                        return ;
                        }
                return ;
}
```

tripleop

```
tripleop( middle, left, right, FX1, FX2, len1, len2, op )
char **         FX1, ** FX2 ;
int             middle, left, right,
                len1, len2, op ;
{
int             i ;
                if (left < 0 || right >= num ) return ;

                if (            Con_Union( FX2[left], FX2[right], FX2[middle], len2 ) &&
                        ||      Inter_Con( FX1[left], FX1[right], FX1[middle], len1 ) ) {

                        print_triple( middle, left, right, op ) ;

                        for ( i=0; i < len1; i++ )
                                FX1[middle][i] |= (FX1[left][i] & FX1[right][i]) ;
                        if (verbose) printvec() ;
                        change = TRUE ;
                        return ;
                        }

                return ;
}
```

print_triple

```
print_triple( middle, left, right, op )
int             middle, left, right, op ;
{
                printf( "%d = ", middle + 1 ) ;
                num_ops ++ ;

                if (op == CON) {
```

```
                if ( Smalleq( left, middle ) )
                        printf( "%d v ", left + 1 ) ;
                else if ( Smalleq( middle, left ) )
                        printf( "%d v ", middle + 1 ) ;
                else {
                        printf( "(%d ^ %d) v ", left+1, middle+1 ) ;
                        num_ops ++ ;
                        }
                if ( Smalleq( right, middle ) )
                        printf( "%d\n", right + 1 ) ;
                else if ( Smalleq( middle, right ) )
                        printf( "%d\n", middle + 1 ) ;
                else {
                        printf( "(%d ^ %d)\n", middle+1, right+1 ) ;
                        num_ops ++ ;
                        }
                }
        else {
                if ( Smalleq( middle, left ) )
                        printf( "%d ^ ", left + 1 ) ;
                else if ( Smalleq( left, middle ) )
                        printf( "%d ^ ", middle + 1 ) ;
                else {
                        printf( "(%d v %d) ^ ", left+1, middle+1 ) ;
                        num_ops ++ ;
                        }
                if ( Smalleq( middle, right ) )
                        printf( "%d\n", right + 1 ) ;
                else if ( Smalleq( right, middle ) )
                        printf( "%d\n", middle + 1 ) ;
                else {
                        printf( "(%d v %d)\n", middle+1, right+1 ) ;
                        num_ops ++ ;
                        }
                }
}

Con_Union( first, second, third, length )                      /* True if the union of       Con_Union
char *  first, * second, * third ;                              /* 1st and 2nd contains
int     length ;                                                /* the 3rd.
{
int     i, contains ;
        contains = TRUE ;
        for (i=0; i<length; i++)
                switch ( first[i]+second[i]-third[i] ) {
                case 0 :
                case 1 :
                case 2 : break ;
                default: return( FALSE ) ;
                }
        return( contains ) ;
}

Inter_Con( first, second, third, length )                       /* True if the inter-         Inter_Con
char *  first, * second, * third ;                              /* section of the 1st
int     length ;                                                /* and 2nd is contained
{                                                               /* in the 3rd.
int i, contains ;
        contains = TRUE ;
        for (i=0; i<length; i++)
                switch ( third[i] - first[i]*second[i] ) {
                case 0 : break ;
                case 1 : contains = PROPER ;
                         break ;
                        default: return( FALSE ) ;
                        }
            return( contains ) ;
}
```

Lesseq

```
Lesseq( first, second )                 /* True if the first is computationally
int first, second ;                     /* less than or equal to the second.     */
{
            if ( Included( PFX[first], PFX[second], lenpf ) &&
                 Included( QFX[first], QFX[second], lenqf ) ) return( TRUE ) ;
            return ( FALSE ) ;
}
```

Smalleq

```
Smalleq( first, second )                /* True if the first is less than or
int first, second ;                     /* or equal to the second.
{
            if ( Included( PFX[first], PFX[second], lenpf ) &&
                 Included( QFX[second], QFX[first], lenqf ) ) return( TRUE ) ;
            return ( FALSE ) ;
}
```

finish

```
finish()                        /* Returns true if there is a variables whose
{                               /* PFX and QFX rows are all ones.
register i, j, ok ;
            ok = FALSE ;
            for ( i = 0 ; i < num && ! ok ; i++ ){
                        ok = TRUE ;
                        for ( j = 0 ; j < lenpf && ok ; j++ ) ok &= PFX[i][j] ;
                        for ( j = 0 ; j < lenqf && ok ; j++ ) ok &= QFX[i][j] ;
                        }
            return( ok ) ;
}
```

printvec

```
printvec()
{
register i, j ;
            printf( "Vectors are :-\n" ) ;
            for ( i = 0 ; i < num ; i++ ){
                        printf( "%3d - ", i+1 ) ;
                        for ( j = 0 ; j < lenpf ; j++ ) printf( "%u ", PFX[i][j] ) ;
                        printf( "  " ) ;
                        for ( j = 0 ; j < lenqf ; j++ ) printf( "%u ", QFX[i][j] ) ;
                        printf( "\n" ) ;
                        }
            fflush( stdout ) ;
}
```

Included

```
Included( set1, set2, length )
char *set1, *set2 ;
int  length ;
{
int contains = TRUE, i ;
            for ( i = 0 ; i < length ; i++ ){
                        if (set1[i] > set2[i]) return( FALSE ) ;
                        if (set1[i] < set2[i]) contains = PROPER ;
                        }
            return( contains ) ;
}
```

```
/***********************************************************************/
/*                                                                      */
/*        P M C - P L A N A R   M O N O T O N E   C O M P U T A T I O N */
/*                                                                      */
/*        pmc_pyr2.c -       pyramid construction for planar circuit    */
/*                           generation module.                         */
/*                                                                      */
/*        Proceeds by constructing AND/OR pyramids so that components   */
/*        can be bridged.                                               */
/*                                                                      */
/***********************************************************************/

#include "header.h"

#define CON          1
#define DIS          0

#define      R       0
#define      S       1
#define      A       2

pyramid( F1, F2, len1, len2, op )               /* Find all components in F1
char **     F1, ** F2 ;                         /* that are I blocked in F2 and
int         len1, len2, op ;                    /* place a op-pyramid over it.
{
char   * in_gap ;                      /* These variables are used to find the
int    * start,                        /* longest gap between components for
       *      gap,                     /* each variable.
              i, v, filled ;

              in_gap      = malloc( (unsigned) len1 ) ;
              start       = ( int * ) calloc( (unsigned) len1, sizeof( int ) ) ;
              gap         = ( int * ) calloc( (unsigned) num,  sizeof( int ) ) ;

              for ( v = 0 ; v < num  ; v++ ) gap[v] = 0 ;
              for ( i = 0 ; i < len1 ; i++ ) {
                          start[i] = 0 ;
                          in_gap[i] = R ;
                          }

              for ( v = 0 ; v < num ; v++ ) {
                          if ( dead[v] ) continue ;
                          for ( i = 0 ; i < len1 ; i++ )
                                      switch ( in_gap[i] ) {
                                      case R:    if ( F1[v][i] ) {
                                                             in_gap[i] = S ;
                                                             start[i] = v ;
                                                             }
                                                 break ;
                                      case S:    if ( F1[v][i] == 0 ) {
                                                             in_gap[i] = A ;
                                                             }
                                                 else        start[i] = v ;
                                                 break ;
                                      case A:    if ( F1[v][i] ) {
                                                             in_gap[i] = S ;
                                                             if ( notblock( F2, len2, start[i], v ))
                                                                        gap[ start[i] ] = v ;
                                                             start[i] = v ;
                                                             }
                                                 break ;
                                                 }
                          }
```

pyramid

```
        for ( filled = 0, v = 0 ; v < num ; v ++ )
                if ( gap[v] > filled ) {
                        filled = gap[v] ;
                        make_pyramid( v, gap[v], P1, P2, len1, len2, op ) ;
                        }
        free( (char *) gap ) ;
        free( (char *) m_gap ) ;
        free( (char *) start ) ;
}
```

notblock

```
notblock( P2, len2, start, end )                        /* Tests to see if there is a
char ** P2 ;                                            /* band of 1's totally within
int     len2, start, end ;                              /* the start/end markers.
{
int     state, v, i ;

        for ( i = 0 ; i < len2 ; i ++ ){
                state = R ;
                for ( v = start; v <= end; v ++ ){
                        if ( dead[v] ) continue ;
                        switch ( state ) {
                        case R:  if ( P2[v][i] == 0 ) state = S ;
                                         break ;
                        case S:  if ( P2[v][i] == 1 ) state = A ;
                                         break ;
                        case A:  if ( P2[v][i] == 0 ) return( FALSE ) ;
                        }
                }
        }
        return( TRUE ) ;
}
```

make_pyramid

```
make_pyramid( start, end, P1, P2, len1, len2, op )
int     start, end, len1, len2, op ;
char ** P1, ** P2 ;
{
char    op_char ;
int     i, v, alive, middle, * last ;

        change = TRUE ;

        op_char = ( op == CON ) ? '^' : 'v' ;

        for ( alive = 0, v = start + 1 ; v < end ; v ++ )
                if ( ! dead[v] ) {
                        alive ++ ;
                        middle = v ;
                        }
        if ( alive == 1 ) {
                tripleop( middle, start, end, P1, P2, len1, len2, op ) ;
                return ;
                }

        last    = ( int * ) calloc( (unsigned) len1, sizeof( int ) ) ;

        for ( i = 0 ; i < len1 ; i ++ )
                for ( v = end ; v >= start ; v -- )
                        if ( P1[v][i] && ! dead[v] ) {
                                last[i] = v ;
                                break ;
                                }

        for ( i = 0 ; i < len1 ; i ++ ) {
                if ( last[i] == 0 ) continue ;
                for ( v = start ; P1[v][i] == 0 || dead[v] ; v ++ ) ;
                for ( ; v <= last[i] ; v ++ ) P1[v][i] = 1 ;
                }

        printf( "%c -- pyramid from %d to %d\n", op_char, start+1, end+1 ) ;
        if (verbose) printvec() ;
}
```

```
/**************************************************************************/
/*                                                                        */
/*          PMC-PLANAR MONOTONE COMPUTATION                               */
/*                                                                        */
/*          header.h - external variables and functions                   */
/*                                                                        */
/**************************************************************************/

#include <stdio.h>
#include <ctype.h>
#define     TRUE        1
#define     FALSE       0

#define     PRIME       1
#define     DUAL        0

#define     UFUNC       1
#define     ZFUNC       0

#define     XFUNC       1
#define     YFUNC       0

#define     PNULL       (POINTER) NULL

typedef     struct tnode {
            char OP ;                       /* '+' '*' or lower case letter */
            struct tnode *LHS ;             /* Left sub tree */
            struct tnode *RHS ;             /* Right sub tree */
            } TREE, *POINTER ;

typedef     unsigned NUMBER ;

char        * malloc(), * calloc() ;
NUMBER      * num_alloc() ;

extern      int Noofvars ;
extern      int veclen() ;

POINTER     maketnode() ;
NUMBER      * primewalk() ;

extern
int             num,                        /* Number of free inputs
                num_ops,                    /* Number of gates used so far
                lenpf, lenqf,               /* Number of prime implicants/clauses
                activegates,                /* Number of gates still alive
                verbose,                    /* Whether to print full tables
                change ;

extern
char *          dead,                       /* Indicates if a gate is redunant
     **         PFX,                        /* Bit table of inputs and implicants
     ** QFX ;                               /* Bit table of inputs and clauses
```