# Partial Order Databases

Darrell Raymond
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

## ABSTRACT

Order is a fundamental property of information that is not explicitly captured in model database models. The *partial order model* introduces the idea of partially ordered sets as the basic construct for modelling data. This model exhibits two novel properties. First, it is capable of describing structure without reference to data types. Second, it inherently separates the structure of data from the objects being structured. These two properties mean that the model naturally facilitates the use of multiple structures for data.

We investigate a collection of algebraic operators for manipulating ordered sets. An implementation of these operators is presented, based on the use of realizers as a data structure. An algorithm is provided for generating realizers for arbitrary finite partial orders.

The partial order model is useful for data domains that involve containment or dependency relationships. Text databases and software repositories are two examples of such domains. We show how the partial order model can be used to structure text and software data, and how it provides new insights in both areas. In particular, partial orders can be the foundation of better systems for handling both tables and makefiles.

Partial orders are prominent not only for data stored in databases, but for database system internals as well. Partial orders play key roles in dependency theory, object-oriented modelling, and the management of redundant data. Thus partial orders are a concept of broad importance both in understanding and implementing the fundamentals of almost any database system.

TABLE OF CONTENTS

# LIST OF ILLUSTRATIONS

## ACKNOWLEDGEMENTS

In the course of graduate studies, one suddenly realizes that a university is not an ivory tower, it's a chaotic zoo whose keeper is permanently out of town. To keep going after that is difficult, unless you have one or two good people who serve as role models. In my case, there were four. First was Evelyn Nelson, who despite suffering through my work in two mathematics courses, still insisted I go to graduate school. Next was Derick Wood, one of the few researchers I know who can span the range between the 'frontiers of abstract nonsense' (to use his phrase) and down-to-earth practical utility. Frank Safayeni taught me once again the joy of curiosity, something I had mislaid during my undergraduate education. Most of all I have to thank Frank Tompa, who has been my supervisor, employer, coach, critic, cheerleader, and friend, all in their proper seasons.

My writing was helped along by Glenn Paulley, Ken Salem, and Grant Weddell, who read the thesis in draft form, and gave useful advice and encouragement. Thanks are also due to my thesis committee (Larry Kerschberg, Paul Larson, Grant Weddell and Ross Willard), each of whom made worthy comments. My particular thanks to Ross, whose suggestions led to material improvements in some of the algorithms and proofs.

Funding for my studies came from several sources. Most important were an IBM Canada Fellowship and an Ontario Graduate Scholarship. Funding was also provided indirectly through research grants from the Natural Science and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario, and the University of Waterloo's Institute for Computer Research. I am grateful to each of these organizations for supporting my work.

Last but not least, I thank my family, and especially my wife Jane, who never chided me for taking far too long.

For Neil and Alana
My two incomparable maxima

It was on sleepless nights that you had thought it all out, in a state of great excitement, with palpitations of the heart and suppressed enthusiasm. And this suppressed, proud enthusiasm is a dangerous thing in young people. I ridiculed your article at the time, but let me tell you that, as an admirer of literature, I'm very partial to these first youthful and ardent literary efforts. Smoke, mist, and the sound of a dying chord in the mist. Your article is fantastic and absurd, but there is such a fresh breath of sincerity in it, there is such incorruptible youthful pride in it, there is the daring of despair in it. It is a sombre article, but that doesn't matter. I read your article and put it aside and—and as I put it aside I thought, This man will get himself into trouble one day!

Porfiry Petrovich in *Crime and Punishment*

# 1

# Introduction

A key theme in database work is the balance between sharing and independence. Sharing is a fundamental purpose of databases—indeed, sharing is implicit in the name of the subject. The sharing of data and operations improves resource usage and simplifies the maintenance of consistency. But sharing is not an unqualified virtue; it can result in unwanted coupling and efficiency bottlenecks. Thus, a good database design will always blend sharing and independence.

The balance between sharing and independence is also present in data modelling. For much of the early history of database science, it was implicitly accepted that there existed a *unifying data model* that could be shared by most, if not all, data, and that a key task of database scientists was to identify this model or *universal kernel* (Kerschberg *et al.* 76, Gilula 94). At some point between the entrenchment of the relational model and the advent of object-oriented databases, however, database scientists and database users stopped looking for a unifying model, and became interested in domain-specific modelling (Brodie *et al.* 84). The reasons for this change in course were many, but perhaps the key reason was disenchantment with the inflexibility and inefficiency of existing implementations. The apparent inapplicability of traditional database methods led to work on specialized databases for text, software, CAD, geographical information, management information, temporal data, and so on. Even the specializations were specialized: temporal database researchers claimed that the time domain was fundamentally different from other relational domains (Tansel *et al.* 93), while software version control researchers claimed that their *particular* time domain was fundamentally different from other temporal domains (Katz 90). The existence of so many subfields is an implicit challenge to the idea of a universal kernel.

This thesis returns to the idea of a unifying data model. The choice between unifying and domain-specific models is like any other choice between reuse and customization; the former aims to eliminate redundancy, while the latter aims to provide more expressive modelling for a specific problem. Experience shows that domain-specific models are not necessarily more expressive even though they cleave to the terminology of a domain. This is particularly true when it turns out that the domain is poorly defined, too narrow, or too transient; as an example, consider the brief spurt and then death of the field of 'office information systems' in the 1980's. Domain-specific models replicate existing procedures and activities of the domain, which makes it easier for domain experts to understand their operations. But if existing procedures are not well thought-out, then neither will be the domain-specific model. A unifying model, on the other hand, has the advantage of being tested in a variety of domains, and so may be more complete and consistent. In addition, operations which are not domain-specific may suggest new possibilities in a given domain.

Though we want a unifying model, we should not expect a universal model, or even claim that there should exist only one model. The advantages of reuse can be gained without requiring that every object is rooted in a single base class; it is only necessary that we recognize and control redundancy where it is troublesome. The plethora of domain-specific models is just such a troublesome redundancy.

This thesis investigates the use of *partial orders* as the basis for a unifying data model. Partial orders capture an important organizational requirement shared by many problems: the need for order. This requirement can be filled by a data model in which partial orders are the data schema, and partial order operations constitute the manipulation algebra. Partial orders are not domain-specific; thus, a general model based on order can be an important way to unify work in disparate domains.

## WHY ORDER?

Order is important because it is a fundamental property of information that is not captured explicitly in modern data models. The case for the importance of order in computing is well made by Parker (Parker 87, 89). Order underlies two main trends in data modelling: type hierarchies and logic. Both can be seen as instances of containment, which is a kind of ordering. A derived type contains its base type. Similarly, the truth value of a predicate contains the truth value of something implied by that predicate. Thus, any relationship that involves types or truth values can be seen as applications of containment, and thus of the ordering of subsets.

The relational model is based on unordered sets, and it is at its most elegant when representing unordered information. The need for order is so prevalent, however, that virtually every implementation of the relational model contains sorting and other operations that involve order. Since these operations are not part of the formal model, implementations are free to create their own semantics for them, resulting in inconsistent semantics across implementations. Even with ordering operations, problems that are heavily order-dependent, such as the classic *bill-of-materials* problem, continue to be a challenging issue, both theoretically and practically.

We focus on order, but do not deny the importance of unordered sets. The partial order model does not begin with a blank slate, but rather incorporates the relational model as a special case. Ordered sets are more general than unordered sets, and so the construction of a partial order model can be seen as a generalization of the relational model. Here we mean 'generalization' in the sense it is used in object-oriented inheritance: as a more abstract class that does not support all the operations of the specialized class. In the case of partial orders, the unsupported operations are difference and complement. Generalization can be contrasted with the more common *specialization* approach to extending the relational model, in which all the relational operations are preserved, and others are added. The specialization approach is common in domain-specific models, such as temporal database work (Tansel *et al.* 93).

## WHY PARTIAL ORDERS?

Given that order is important, why partial orders? A compelling argument for partial orders is the wide variety of applications that involve partial orders:

- temporal data can be mapped to interval orders, a subclass of partial orders (Fishburn 85, Lorentzos 93)
- concurrency scheduling and transaction control involve the management of events that are partially ordered (Pratt 86, Taylor and Coffin 94)
- shared access to computer resources can be represented by two-dimensional partial orders (Sandhu 88)
- object-oriented hierarchies with multiple inheritance are general partial orders (Chambers *et al.* 91)
- the analysis of client-server structures is partially ordered (Terribile 94)
- large software packages contain inclusion and dependency relationships that are naturally acyclic and transitive, and hence partially ordered (Borison 89, Waters 89)
- the notion of precision of information involves a partial order (Read *et al.* 92)
- the structure induced by functional dependencies in relational databases defines a partial order on instance relations (Lee 83)

Thus, partial orders seem to fulfill a basic requirement of a general-purpose data model: wide applicability.

Partial orders are also important because they span the closed interval between completely unordered sets and totally ordered sets. Total orders are not sufficient for many applications, because incomparability is a prominent phenomenon that must be captured explicitly. From incomparability one may infer opportunities for parallelism (for example, if two programs in a dependency graph are incomparable, they may be compiled in parallel), or one may

infer that certain facts are not derivable from each other (for example, if two sentences belong to incomparable substructures of a text, then they are unrelated).

The relational model succeeded partly because it was well grounded in set theory, and provided well-defined abstract operators. Partial orders, like relations, have a sound mathematical basis. The model described in this thesis is based on the mathematical notion of partial orders, and the bulk of the thesis describes the operators appropriate for combining partial orders, and the uses of those operators for specific applications.

Partial orders bring an important new property to database modelling. This property is that *partial orders describe structure without type relationships.* Virtually all existing modelling techniques rely on the notion of types: a schema is described by specifying a set of types and their relationships (which may also be typed), and the database itself is a collection of instances that satisfy the typing and relationship constraints. A relational schema, for example, is a set of types called 'relations' that organizes a set of types called 'attributes'. The entire structure of the relational model is expressed in terms of the relationships of type names; thus, the more types there are, the more kinds of structures that can be described. Conversely, the fewer types there are, the fewer structures can be described. In the limit, where only one type exists, a type-based model can express at most a single, cyclic relationship, as shown in Figure 1.

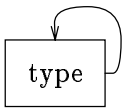
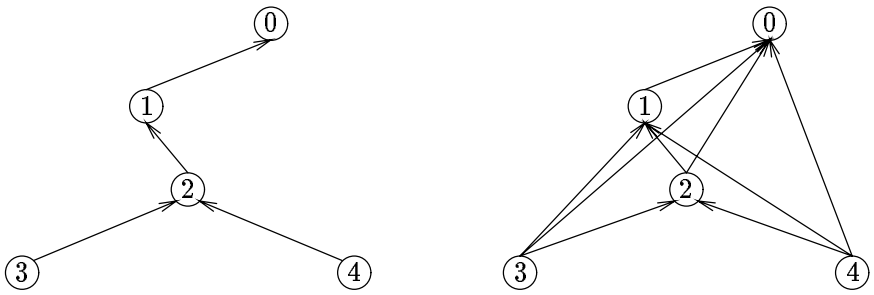



Figure 1: Relationship between two objects of the same type.

Partial orders, on the other hand, are specified as an ordering relation on an untyped base set. A partial order does not describe the relationships between types or classes of objects, but between instances of objects, whose types are immaterial. Thus, a partial order can capture the structure of an instance *within* the trivial type description of Figure 1.




(a) (b)

Figure 2: Transitive reduction vs. closure

The partial order model does not disallow types. It is possible to use the types of objects to describe an ordering of objects; for example, that source code files 'precede' their object code files. Typing is not required, as it is in most models, but it is possible. Note that, having induced an ordering relation with types, we can manipulate that relation without reference to the types, and can (if required) operate outside type constraints.

A second important property is that *partial orders naturally facilitate multiple organizations.* Type-based systems tend to encourage modellers to identify unique and fixed classifications for data, because the type of a datum is usually taken to be one of its permanent characteristics. In the relational model, this typing of data is expressed both by domains (each datum must belong to a single domain) and by the dependencies, that tend to restrict the relations to which attributes can belong. Indeed, the *universal relation* assumption, that all relations are projections of some universal relation, requires that only one type of relationship is possible between any set of attributes (Beeri *et al.* 78).

A partial order, on the other hand, explicitly separates the ordering relation from the base set. There is nothing to keep us from describing many partial orders on a single base set, each with a different ordering relation, and some of the relations perhaps contradictory to one another. This is useful because the ordering of information is often a property of the use to which it is put, and not an immutable characteristic of the data itself (Ginsburg and

Hull 83). Partial orders thus naturally have the flexibility known as *semantic relativism* (Brodie 84).

A basic requirement of any general-purpose data model is that it be easily understood by non-specialists. Formal logic is more general and elegant than the relational model, but as a general-purpose data model it has not prospered, because of the degree of sophistication needed to understand a collection of predicates. The relational model, on the other hand, can be intuitively grasped as tables that are manipulated with simple row and column operators. The simplicity of the partial order model remains to be seen. Since most people can grasp the notion of a directed acyclic graph, this is the most likely place in which to start explaining the model.

## WHY NOT DIRECTED ACYCLIC GRAPHS?

If DAGs are a good tool for explanation, why not simply base the model on DAGs? Since partial orders are often specified with Hasse diagrams—that is, with directed acyclic graphs—what is the advantage in studying partial orders, instead of a model based on DAGs, or even one based on more general graphs?

Partial orders, although they are often represented by a single graph, really denote classes of graphs. Consider the two graphs in Figure 2, both describing the same partial order. Graph (a) is the transitive reduction (a graph containing no edges derivable by transitive closure), and graph (b) is the transitive closure. It is common to keep the transitive reduction as the data instance, and to obtain the transitive closure by applying a closure operator. It would be equally valid, however, to use the transitive closure as the data instance, and provide the transitive reduction through a reduction operator.

The choice between the two alternatives is not a modelling choice, but one based on efficiency. If space is at a premium, we may prefer to store the reduction, since it is the minimum description. If the cost of querying is to be minimized, we may prefer to store the closure, since this permits us to answer pairwise comparisons in linear time.

The two alternatives also have different update properties. Re-

moving the edge between nodes 1 and 2 in the reduction graph also implicitly removes any relationship between nodes 1 and 3 and nodes 1 and 4. Removing the same edge in the closure graph still leaves node 1 connected to nodes 3 and 4. We can remove any edge in graph (a) and the result will still be a partial order; but if we remove the edge between 0 and 2 in (b), the result will not be a partial order, since transitivity is violated. Thus, each type of representation will facilitate a particular kind of update semantics, although either type can be used to support any semantics.

A partial order denotes a class of graphs consisting of the transitive reduction, the full transitive closure, and all intermediate closures. These differ not in their modelling power, but in their storage and update properties. The choice of which representation to use depends on the expected updates, and not on some aspect of modelling.

A second reason to prefer partial orders to graphs is to make the notion of incomparability an explicit one. Graph models may have nodes that are 'incomparable', in the sense that they do not succeed one another on some directed path through the graph. However, graph models usually do not attach any semantics to this kind of incomparability; indeed, they often provide navigational operators to get around it. In the partial order model, we exploit incomparability to represent problem semantics. We could also imagine distinguishing between different interpretations of incomparability; for example, the difference between a situation in which the order of two elements is unknown, and a situation in which two elements are known to be unrelated.

A third reason to prefer partial orders is to study the properties of an algebra for collections of ordering relations. Many graph-based models deal with a single graph on a given set of nodes (perhaps with several implicit subgraphs derivable from edge or node types). In the partial order model, we explicitly deal with multiple orders on a single base set, and we exploit this feature to partition the structure of a database. The existence of multiple orders immediately suggests the possibility of an algebra for combining orders. Models based on a single graph generally do not provide an algebra for combining

graphs. The possibility of multiple orders also immediately raises the fundamental question of database design—how can a database be decomposed into several components that can be recombined to produce the whole?

Existing graph-based models do not consider these issues, but a graph-based model that does address them could be constructed. One would expect that problems similar to the ones solved in this thesis would arise.

The final answer to the question 'why not DAGs?' is that the existence of DAGs is not an *a priori* reason to discard the partial order model, just as the existence of binary relations is not a reason to discard DAG models, and the existence of CODASYL networks is not a reason to discard relations. Equivalence of data models, like Turing completeness, addresses only the denotational power of a system, and not its elegance, efficiency, or comprehensibility.

## TEMPORAL DATABASES: AN EXAMPLE

We can show some of the utility of the partial order model by contrasting a partial order solution to the temporal database problem with the standard relational approaches.

Temporal database research begins with the observation that the relational model is inadequate to manage temporal data. A traditional relational database can be viewed as a snapshot of the data of an evolving enterprise; in other words, the data as it exists at one instant in time. Typically, the set of relations captures information that is valid 'now', but not at some time in the past. The updates performed on the relations keep the snapshot consistent with the current state of the enterprise, but in so doing, they lose information about the previous states of the system.[1] A *temporal* database, on the other hand, is one that provides access to both the current and previous states of the enterprise (Tansel *et al* 93).

A simple way to extend the relational model to handle previous states is to employ an additional attribute *time*, which specifies the

1 Previous states can be gleaned from transaction logs, but neither the relational algebra nor implementations of relational systems provide good methods for querying this data.

| NAME | SALARY | DEPT | TIME |
|---|---|---|---|
| John | 15k | toys | 11 |
| John | 15k | toys | 12 |
| John | 15k | toys | 13 |
| John | 15k | toys | 14 |
| John | 15k | toys | 15 |
| Inga | 25k | clothing | 71 |
| Inga | 25k | clothing | 72 |
| Inga | 25k | clothing | 73 |
| Inga | 25k | clothing | 74 |

Figure 3: Simple temporal relation.

time at which a tuple's attributes have particular values. Figure 3 shows a relation augmented with such a time attribute.

Such a relation is sometimes called an *unfolded* relation (folded relations will be discussed presently). Two things are apparent from this example of an unfolded relation. First, time must be an element of the key of such a relation, since it is often the only element that distinguishes tuples. Second, there is apparently a massive amount of redundancy in the relation. Any data that is constant over long periods of time would lead to many tuples being stored in the database. This redundancy has led temporal researchers to consider more complex and less redundant solutions.[2]

2 It is not clear that this concern about redundancy is well-founded. Any relation is only a model of the stored data, and not the stored data itself. The unfolded relation is clearly an easy model to understand, and hence is a strong candidate as the model the user should employ. If special constructs are needed in order to represent this model efficiently, then the appropriate place for these is in the implementation.

The redundancy of the unfolded relation is not identical to the redundancy of relations that are (for example) not in third normal form. In producing a relation in third normal form, we not only remove redundancies, but we eliminate certain types of update anomaly. By folding the unfolded relation, we do not eliminate update anomalies. Update anomalies are modelling concerns, not just implementation issues; thus, normalization does have some role to play in modelling, while controlling the redundancy of an unfolded relation

| NAME | SALARY | DEPT |
|---|---|---|
| [11,60] John | [11,49] 15k<br>[50,54] 20k<br>[55,60] 25k | [11,44] toys<br>[45,60] shoes |
| [0,20]∪[41,51] Tom | [0,20] 20k<br>[41.51] 30k | [0,20] hardware<br>[41,51] clothing |
| [71,NOW] Inga | [71,NOW] 25k | [71, NOW] clothing |
| [31, NOW] Leu | [31, NOW] 15k | [31, NOW] toys |
| [0,44] ∪[50, NOW] Mary | [0,44] ∪ [50, NOW] 25k | [0,44]∪[50, NOW] credit |

| DEPT | MANAGER |
|---|---|
| [11,49] toys | [11,44] John<br>[45,49] Leu |
| [41,47]∪[71,NOW] clothing | [41,47] Tom<br>[71, NOW] Inga |
| [45,60] shoes | [45,60] John |

Figure 4: Temporal relations.

The most obvious way to control the redundancy of the unfolded model is to produce a *folded* model, in which we store *interval* timestamps, rather than instantaneous timestamps. Depending on which temporal database approach one prefers, interval timestamps may be applied to attributes, tuples, or both. Attribute timestamping is shown in Figure 4. Each attribute in the temporal relation is time stamped with a set of intervals chosen from a totally ordered time domain, represented as pairs of integers. The special value NOW denotes the current time. A relational snapshot of this temporal database would include only tuples whose attributes all have timestamps including NOW.

While folding the relation removes the redundancy, it complicates the model. Folding results in two distinctly different types of data: conventional data, and time. Temporal data is non-atomic (it consists of sets of intervals, each of which is itself non-atomic), non-fixed

does not.

| DEPT | MANAGER |
|---|---|
| [45,49] toys | [45,49] Leu |
| [11,44] toys<br>[45,60] shoes | [11,60] John |
| [41,47] clothing | [41,47] Tom |
| [71, NOW] clothing | [71, NOW] Inga |

Figure 5: Weakly equivalent relation.

(the value NOW is actually a variable), and ordered. These characteristics of temporal data mean that we need additional operators for querying and update. Consequently, temporal database models introduce an additional `while` clause to the SQL `select` statement. The `while` clause permits us to add a temporal condition on the selection of tuples. The `where` clause is not sufficient because the ways in which we combine and query temporal data are different from conventional atomic data. We may want to ask whether certain things existed while other things existed, for example, or to ask for things that preceded a certain date, or to determine when some entity changed its state.

Folding introduces other problems as well. A timestamp must always be part of the key of a temporal relation (since it is always possible that the data does not change between instants in time), but folding means that the key may be spread among all the attributes of the tuple. At the very least this means more complex reasoning about the data. One indicator of this complexity is the notion of *weak equivalence*. Two temporal relations are said to be weakly equivalent if snapshots of the two relations are equivalent at any point in time, even though the relations themselves are not equivalent. Thus, the manager relation from Figure 4 is weakly equivalent to the manager relation in Figure 5.

Another disadvantage of folding is that it makes unions and differences more complex to perform, since one must now compute the intersection and union of sets of intervals.[3]

3 It is worth observing that the semantics of union and difference for folded

To summarize then, the attempt to use the relational model to handle temporal data is complicated by the desire to avoid redundancy. The interval approach to controlling redundancy introduces many new complications at the modelling level.

The partial order approach to temporal data begins, naturally enough, by adducing the order relationships in the data. The snapshot relation is an unordered set of unordered sets, but we gain little from using the partial order model on this structure. The temporal database, however, has at least two important ordering relations. The *precedence* order **P** tells us how events precede one another. The *containment* order **C** tell us which event lifespans are contained in one another. The precedence and containment partial orders, although related, are not reducible to one another.

**Theorem 1.1** *Given two temporal events $a$ and $b$, and the orders* **P** *and* **C**, *the following are true:*

1. *If $a$ contains $b$, then neither precedes the other.*

2. *If $a$ precedes $b$, then neither contains the other.*

We note that the converse of this theorem is not true; if $a$ does not precede $b$, then we cannot conclude that $a$ is contained in $b$ or $b$ is contained in $a$; similarly, if $a$ does not contain $b$, then we cannot conclude that $a$ is preceded by $b$ or $b$ is preceded by $a$. It is possible for two events to neither precede one another nor be contained in one another, as is the case when they have a non-trivial overlap.

The partial order approach to modelling the manager relationship must describe both the precedence order and the containment order, and show how they are related to the atomic data values. Figure 6 shows this model for the values given in Figure 5. There are three disjoint suborders in this one structure. The leftmost order shows how the non-temporal data values correspond to temporal intervals. The top righthand structure shows the precedence of temporal intervals. The bottom righthand structure shows the containment of

temporal relations are sometimes defined in terms of operations on the unfolded relation. This shows that the unfolded relation is the semantic bedrock of the model, and the folded relation a shakier representation built upon it.

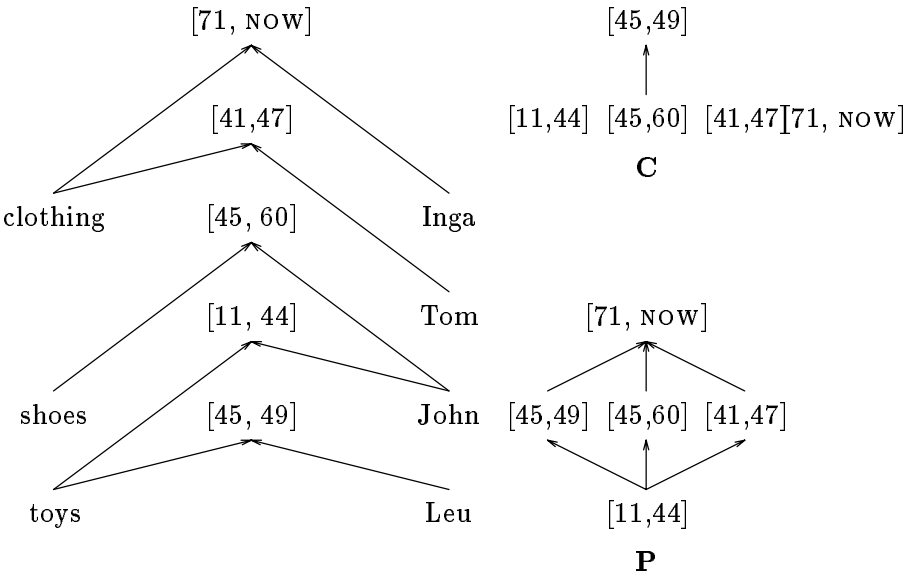



Figure 6: Partial order representation of managers.

temporal intervals. From these structures we can solve problems such as 'Who was manager of the toy department before John?' by applying operators (to be described in the next chapter) to these partial orders.

There are two noteworthy points about the partial order approach to temporal data. First, there is a natural redundancy control involved; each manager and department is mentioned only once, and 'reused' by any temporal interval that refers to that data. Second, we see that the concept of weak invariance is merely the observation that intervals can be clustered with one of several domains (in our example, with either managers or departments). In the partial order approach, clustering of this kind is part of the implementation, not part of the model.

The remainder of the thesis explores the partial order model and several major applications. In Chapter Two we present a more formal description of the partial order model, including definitions of the operators. The main contribution of this chapter is to augment the standard algebraic operations with additional operators that are necessary for database manipulation of partial orders.

In Chapter Three we present a novel implementation of the partial order operators, based on the use of realizers as a data structure. The main contributions of this chapter are algorithms for computing each operator. In addition, we present an algorithm for deriving a (non-minimal) realizer for partial orders of any dimension.

In Chapter Four we consider how the partial order model can be applied to software. The main contributions of this chapter are a new approach to makefiles, and an outline of how software construction and software version control can be unified.

In Chapter Five we consider how the partial order model can be applied to text databases. The main contributions of this chapter are to show how a variety of presentational and logical structures can be combined, and to present a new method for managing tables.

In Chapter Six we consider the relevance of the partial order model for describing basic database problems. The main contribution of this chapter is to outline how partial orders can be used as abstractions of the internals of a database, and not just as a model of the data it manages.

Finally, Chapter Seven considers methods for deploying the model and discusses some issues that should be considered in future research.

# 2

# The partial order model

The partial order model formalizes the manipulation of ordered sets of elements. A partial order is a set of elements that are structured according to an order relationship. An order relationship is both transitive and antisymmetric.

We often depict partial orders with Hasse diagrams, as we do in Figure 7. Hasse diagrams are graphs in which each node is an element of the base set and each edge is an element of the transitive reduction of the order relationship. We draw such diagrams so that the minimal element is at the bottom, wherever possible. We also draw directed edges $a \rightarrow b$ if $a < b$. Note that partial orders need not be represented by connected graphs.

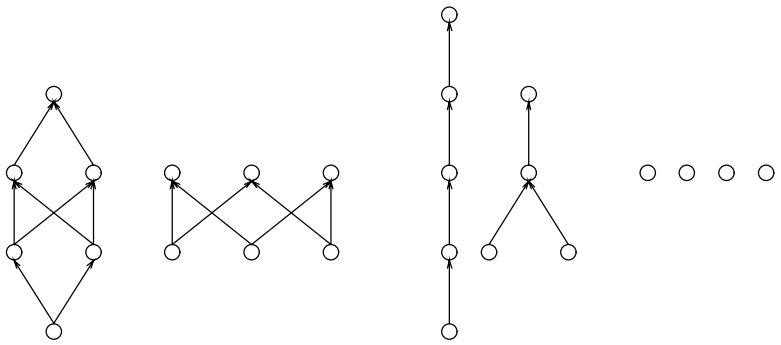
A partial order can structure elements of different types and different sizes. From the diagrams in Figure 7 we cannot tell whether the elements are numbers, strings, processes, or relational databases. Nor is this information formally necessary for us to be able to manipulate the objects. All that is needed is that the elements can be treated as atomic insofar as the order relationship is concerned.

A *partial order database* is simply a partial order. We permit the elements of a partial order to be composite; a partial order database will typically consist of a set whose elements are themselves partial orders.

We now formalize the model. First we introduce notation and definitions, next we discuss the operators of the model, and finally we present some of the algebraic properties of the model.

## PARTIAL ORDERS

A *partial order* is a pair consisting of a finite base set and an order relation. The mathematical theory of partial orders includes partial



(a) (b) (c) (d)

Figure 7: Examples of partial orders.

orders over infinite sets, but here we restrict ourselves to partial orders over finite sets, in the same way that the relational model restricts itself to finite relations (in particular, this means that the maxima and minima of our partial orders are always well-defined).

Let $A$ be a set consisting of the elements $a_1$, $a_2$, $a_3$, $\ldots, a_n$. A *binary relation on* $A$ is a set of ordered pairs $\{(a, b)\}$ where $a$ and $b$ are chosen from $A$. An *order relation on* $A$ is a binary relation $O$ on $A$ that is

1. reflexive: $\forall a,\ (a, a) \in O$
2. antisymmetric: if $(a, b) \in O$ and $(b, a) \in O$ then $a = b$
3. transitive: if $(a, b) \in O$ and $(b, c) \in O$, then $(a, c) \in O$

If $(a, b) \in O$, then we say that $a \leq b$. If $(a, b) \in O$ and $a$ and $b$ are distinct elements, then we say that $a < b$. If $a \leq b$ or $b \leq a$, then we say that $a$ and $b$ are *comparable*; otherwise, they are *incomparable*. Incomparable elements $a$ and $b$ are denoted by $a \sim b$. We denote partial orders with a bold letter and a subscript, as in

$$\mathbf{A}_{\mathbf{O}}$$

where $A$ is the base set of elements, boldface implies that the set is ordered, and the order relation is denoted by $\mathbf{O}$. Where it is clear that a specific order relation is implied on a given set, we will dispense with the subscript.

We define a special *empty order*, or $\emptyset$. This is the partial order consisting of no elements and an arbitrary ordering relation. The empty order serves as a zero element in the partial order algebra. We will define a unit element later.

We observe an *open world* assumption with respect to order; any elements of the world not belonging to the base set of a partial order $\mathbf{A}$ are assumed to have unknown relationships to the members of $A$. Note that an unknown relationship is not the same as known incomparability.

A *total order* is a partial order in which every pair of elements is comparable. A *linear extension* of a partial order $\mathbf{P}$ is a total order $\mathbf{L}$ such that $\mathbf{P} \subseteq \mathbf{L}$. A *realizer* $R$ of a partial order $\mathbf{P}$ is a set of linear extensions of $\mathbf{P}$ such that for every $a \sim b$ in $\mathbf{P}$, there exists at least one linear extension in $R$ in which $a < b$, and at least one linear extension in $R$ in which $a > b$ (Trotter 92). A realizer has only one linear extension if and only if the partial order is a total order.

## AN ALGEBRA FOR PARTIAL ORDERS

We want to manipulate partial orders as objects in an algebra. The kinds of operations we intuitively expect are those that modify the base set and order relation of a given partial order, those that extract suborders, and those that combine two or more partial orders to produce a new order.

The base set of a partial order can be manipulated with the standard set operators: union, intersection, difference, and product. These operators have been well studied in the relational model and its derivatives (Gilula 1994), so we do not investigate them further here. Instead, we concentrate on the operators that manipulate ordered sets.

The algebra should be *closed*: the result of applying any operator to any partial order should be a new partial order. Furthermore, we want the operators to be free of side-effects. The combination of closure and freedom from side-effects will permit substitution of arbitrary expressions in the algebra for any variable.

Closure is a strong requirement; it excludes many operators that

only sometimes produce a partial order as their result. In particular, difference and complement are undefined for partial orders. The difference of two partial order relations is a relation, but it need not be an order relation (as transitivity may be lost). Conversely, the complement of a partial order will almost always include contradictory information, since if the partial order contains $a \sim b$, the complement must include both $a < b$ and $b > a$.

### PREDICATES

Predicates are applied to partial orders to determine whether they satisfy some property. There are many possible predicates, but we require only the following:

- **total:** $\Updownarrow$

  $\Updownarrow \mathbf{A}$ if and only if $\mathbf{A}$ is totally ordered.

- **unordered:** $\Leftrightarrow$

  $\Leftrightarrow \mathbf{A}$ if and only if $\mathbf{A}$ is completely unordered.

- **containment:** $\subseteq$, $\subset$

  $\mathbf{B} \subset \mathbf{A}$ if and only if $B \subset A$ and $\forall x, y \in B$, $(x, y) \in \mathbf{B}$ if and only if $(x, y) \in \mathbf{A}$.

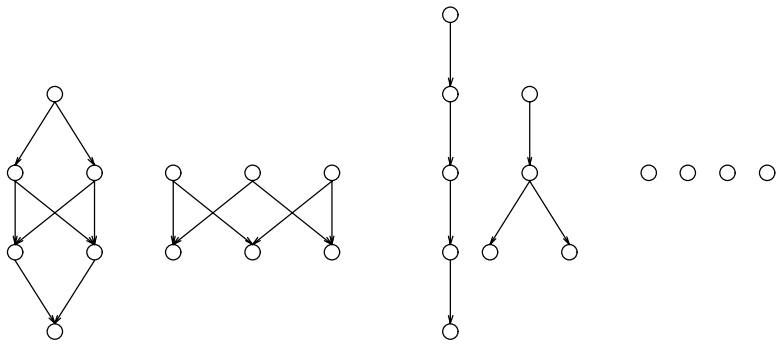
  $\mathbf{B} \subseteq \mathbf{A}$ if and only if $B \subseteq A$ and $\forall x, y \in B$, $(x, y) \in \mathbf{B}$ if and only if $(x, y) \in \mathbf{A}$.

  This definition of containment is stronger than what might be expected, namely, simple set containment of the order relation. This latter notion is sometimes called a *weak subposet*. What is meant here by containment is the notion of *induced subposet* (Stanley 1986).

- **equality:** $=$

  $\mathbf{A} = \mathbf{B}$ if and only if $A = B$ and $(x, y) \in \mathbf{A}$ if and only if $(x, y) \in \mathbf{B}$.

  Equality is a binary predicate. It may not be trivial to show equality if the data structure we use admits more than one form for a given partial order.



(a) (b) (c) (d)

Figure 8: Duals of the partial orders in Figure 7.

### OPERATORS OVER A SINGLE PARTIAL ORDER

The following operators manipulate a single partial order. Some of the operators require a predicate or element(s) of the base set as additional arguments.

- **base-set:** $\beta$

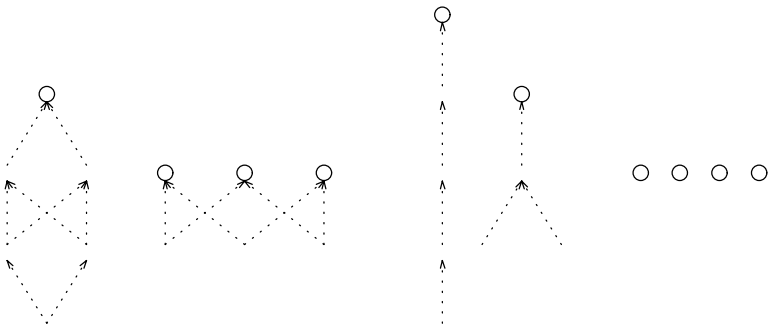
  $\beta\mathbf{A} = A$.

  $\beta$ extracts the base set of a partial order. The result can be viewed as a partially ordered set in which all elements are incomparable, or as a classical, unordered set. Thus, $\beta$ of a partial order can serve as a natural interface between set-based models and the partial order model.

- **dual:**

  $\overline{\mathbf{A}} = \{(x, y)$ such that $(y, x) \in \mathbf{A}\}$.

  The dual of a partial order inverts the order relation. The duals of the partial orders in Figure 7 are found in Figure 8 (note that we have only switched the orientation of the edges for this Figure, and have not reordered the diagram to put the minimal elements at the bottom).



(a) (b) (c) (d)

Figure 9: Maxima of the partial orders in Figure 7.

- **selection:** $\sigma$

  $\sigma(\mathbf{A}, p) = \mathbf{A}'$ where $\mathbf{A}' \subseteq \mathbf{A}$ and $A' = \{x \mid p(x)\}$.

  Selection extracts a contained partial order whose elements satisfy the predicate $p$. The predicate $p$ is applied to each element of $A$ independently, and $A'$ consists of the elements for which the predicate is true.

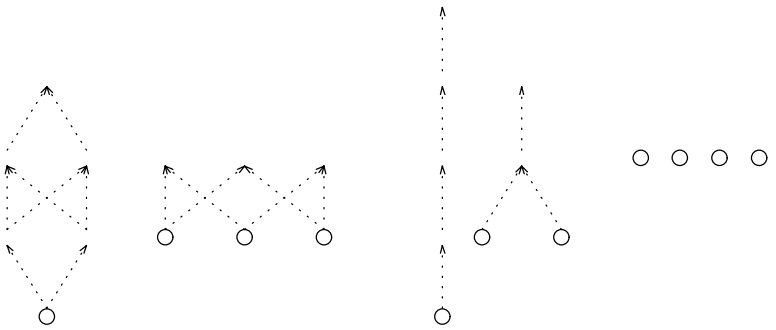
  The predicate $p$ may itself include a partial order expression, and is not necessarily limited to operators applicable to $\mathbf{A}$. $p$ may, for example, contain instances of $\sigma$. $p$ may also include operators that are specific to a given type of element, and not otherwise defined by the partial order model. Such operators will return false whenever they are applied to elements that are not of the intended type.

  When the predicate is a simple one (typically, testing for equivalence of elements to a fixed value), we exploit the notational convenience of subscripting the predicate. For example,

  $$\sigma_w \mathbf{A}$$

  selects all elements in $A$ that satisfy $w$.

  If $p$ is an expression whose value is not a truth value, it will be considered to be true if its value is a nonempty partial order, and false otherwise.



(a) (b) (c) (d)

Figure 10: Minima of the partial orders in Figure 7.

- **maxima:** $\top$

  $\top\mathbf{A} = \{x \text{ such that } (x, y) \in \mathbf{A} \text{ if and only if } x = y\}$.

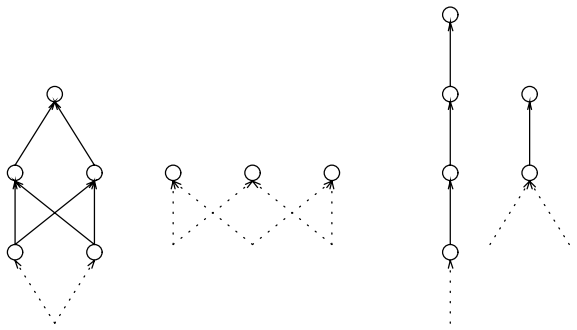
  $\top$ extracts the elements of the partial order that are not less than any other element. The result is an unordered set. The maxima of the partial orders in Figure 7 are shown in Figure 9.

- **minima:** $\perp$

  $\perp\mathbf{A} = \{x \text{ such that } (y, x) \in \mathbf{A} \text{ if and only if } x = y\}$.

  $\perp$ extracts the elements of the partial order that are not greater than any other element. The result is an unordered set. The minima of the partial orders in Figure 7 are shown in Figure 10.

  The symbol $\perp$ is not used consistently in the literature on partial orders. Some writers use it to indicate that two elements are comparable; thus, $a \perp b$ implies that $a \not\sim b$, in our notation. Other writers use $\perp$ to denote the minimum element of a partial order, if it has one (these are called *pointed* partial orders).



(a) (b) (c) (d)

Figure 11: $\pm$ of the partial orders in Figure 7.

- **up-tail:** $\pm$

  $\pm\mathbf{A} = \mathbf{B} \subset \mathbf{A}$ where $B = A - \perp\mathbf{A}$.

  $\pm$ extracts the induced suborder that results from removing the minima (or the lowest antichain) of the partial order. The up-tail of the partial orders in Figure 7 are shown in Figure 11.

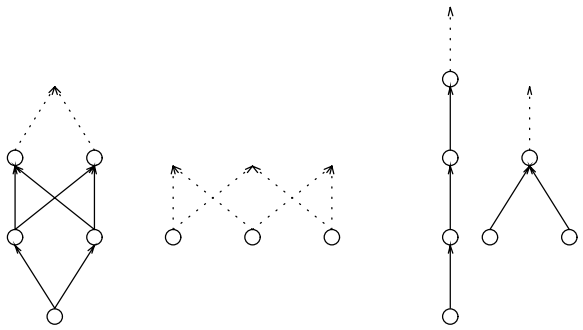
  The result of $\pm$ is well-defined, since there is always a unique set of minima for finite partial orders. Hence, the recursive application of up-tail is also well-defined. We sometimes use a subscripted form of the operator: $\pm_n\mathbf{A}$ removes the first $n$ antichains of $\mathbf{A}$.

- **down-tail:** $\mp$

  $\mp\mathbf{A} = \mathbf{B} \subset \mathbf{A}$ where $B = A - \top\mathbf{A}$.

  $\mp$ extracts the induced suborder that results from removing the maxima (or the highest antichain) of the partial order. The down-tail of the partial orders in Figure 7 are shown in Figure 12.

  The result of $\mp$ is well-defined, since there is always a unique set of maxima for finite partial orders. Hence, the recursive application of down-tail is also well-defined. We sometimes use a subscripted form of the operator: $\mp_n\mathbf{A}$ removes the first $n$ antichains of $\mathbf{A}$.



(a) (b) (c) (d)

Figure 12: $\mp$ of the partial orders in Figure 7.

For a partial order of height $h$, $h$ applications of $\mp$ will partition the partial order into $h$ antichains. Similarly, $h$ applications of $\pm$ will partition the partial order into $h$ antichains. The antichain partitions produced by $\mp$ and $\pm$ need not be the same.

- **up:** $\uparrow$

  $\uparrow(\mathbf{A}, e) = \{(a, b) \in \mathbf{A} \text{ where } (e, a) \in \mathbf{A}\}$.

  $\uparrow$ finds the induced suborder of $\mathbf{A}$ that is greater than or equal to the element $e$. It is similar to the mathematical notion of prime filter, but $\uparrow$ produces a partially-ordered set, whereas prime filter produces an unordered set[1](Davey and Priestley 90).

  Note that $\uparrow(\mathbf{A}, e) \subseteq \mathbf{A}$, and that $\bot \uparrow (\mathbf{A}, e) = e$.

  $\uparrow$ is defined analogously over a set of elements $\{e_1, e_2, \ldots e_n\}$.

1 More precisely, algebraists treat the filter as a set that can be considered either unordered or having an ordering induced by the original partial order. This is satisfactory if we are interested in the differences between structures only up to isomorphism. Here we give more emphasis to the idea that sets may have more than one order, and so we insist on greater clarity with regard to the concept of $\uparrow$.

- **down:** $\downarrow$

  $\downarrow(\mathbf{A}, e) = \{(b, a) \in \mathbf{A}$ where $(a, e) \in \mathbf{A}\}$.

  $\downarrow$ finds the induced suborder of $\mathbf{A}$ that is less than or equal to $e$. It is similar to the mathematical notion of prime ideal.

  Note that $\downarrow(\mathbf{A}, e) \subseteq \mathbf{A}$, and that $\top \downarrow(\mathbf{A}, e) = e$.

  $\downarrow$ is defined analogously over a set of elements $\{e_1, e_2, \ldots e_n\}$.

The next set of operators we discuss are applied to two or more partial orders. We can separate these operators into those that manipulate partial orders over a common base set of elements, and those that manipulate partial orders over a disjoint set of elements.

### OPERATORS OVER PARTIAL ORDERS WITH A COMMON BASE SET

A pair of partial orders over a common base set may contradict one another about the order of elements: it may be the case that $a < b$ in one partial order, but $b < a$ in the other. The operation of intersection combines the two partial orders and removes such contradictory orderings.

- **intersection:** $\cap$

  $\mathbf{A_1} \cap \mathbf{A_2} = \{(x, y)$ such that $(x, y) \in \mathbf{A_1}$ and $(x, y) \in \mathbf{A_2}\}$.

  Any partial order may be expressed as the intersection of some other partial orders; in particular, every partial order is the intersection of the linear extensions of any of its realizers.

- **contradiction:** $\sim$

  $\mathbf{A_1} \sim \mathbf{A_2} = \{a \mid \exists b \neq a$ such that $(a, b) \in \mathbf{A_1}$ and $(b, a) \in \mathbf{A_2}\}$.

  Contradiction returns the set of elements that participate in contradictory orderings in $\mathbf{A_1}$ and $\mathbf{A_2}$. The operator is not commutative; it returns the lesser element from the first order. This asymmetry allows us to obtain the other half of the incomparable elements simply by reversing the order of the operands.

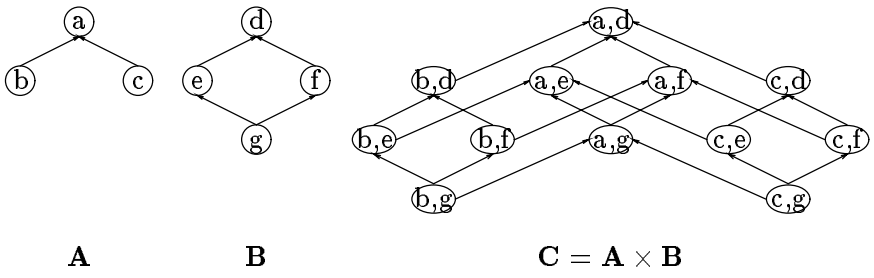



Figure 13: Cross product.

The output of contradiction is an unordered set. When closure demands, we may treat the unordered set as a set with an empty order relation.

### OPERATORS OVER PARTIAL ORDERS WITH DISJOINT BASE SETS

Partial orders over disjoint base sets cannot contain contradictory orderings; hence, the operation of union can be defined for such partial orders (difference could also be defined, but it would be trivial). Partial orders over disjoint base sets can be considered to be a special case of orders over a common base set, where it is guaranteed that the order relations are constrained to partitions of the base set. Thus, all the operations of the previous section can also be applied to partial orders over disjoint sets (contradiction and intersection will produce only trivial results for disjoint sets).

- **cross product:** $\times$

  $\mathbf{A_1} \times \mathbf{A_2} = \{((x_a, y_b), (x_c, y_d))$ where $(x_a, x_c) \in \mathbf{A_1}$
  and $(y_b, y_d) \in \mathbf{A_2}\}$.

  The operation of cross product is the principal method by which we combine partial orders that describe orthogonal relationships. An example is shown in Figure 13. Note that cross product produces a new base set and combines the two order relationships.

  The production of a new base set complicates the definition of an important concept, that of the unit element. We would like to have some unit partial order for which $\mathbf{A} \times \mathbf{1} = \mathbf{A}$. The

definition of cross product we have given seems to preclude this possibility, since the elements of the result are 2-tuples, while the elements of the arguments are singletons. Let $\mathbf{1}$ be the partial order over the base set $\{1\}$; we call it the *unit partial order*. We define it such that $\mathbf{A} \times \mathbf{1} \cong \mathbf{A} \cong \mathbf{1} \times \mathbf{A}$ and $\mathbf{1} \times \mathbf{1} \cong \mathbf{1}$.

- **lexicographical sum:** $\sum$

  $\sum_{\mathbf{A} \in \mathcal{F}}^{\mathbf{E}, f} \mathbf{A} = \{(x, y) \mid (x, y) \in \mathbf{A}, \mathbf{A} \in \mathcal{F}$ or

  $f(a) = \mathbf{A} \in \mathcal{F}$ and $f(b) = \mathbf{B} \in \mathcal{F}$and $(a, b) \in \mathbf{E}, a \neq b$, and $x \in A, y \in B\}$.

  Lexicographical sum is the principal operator by which we manage nested or hierarchical relationships. Intuitively, lexicographical sum is the expansion of a given partial order by replacing each of its elements with partial orders. An example is shown in Figure 14.

  Lexicographical sum requires an injective function $f : E \rightarrow \mathcal{F}$ that maps elements of $E$ to a family $\mathcal{F}$ of partial orders $\mathbf{A}$. The partial order whose elements are replaced (that is, $\mathbf{E}$) is known as the *expression order*. The partial orders that are substituted for the elements of the expression order (that is, the elements of $\mathcal{F}$) are known as the *factor orders*.

Most texts that discuss partial orders do not mention lexicographical sum,[2] but they typically do define the operations of *linear sum* and *disjoint union*. The linear sum of two partial orders is given by placing one 'above' the other, and connecting the minima of the top order to the maxima of the bottom order. The disjoint union of two partial orders is given by placing them next to each other. It is easy to see (although not generally mentioned) that linear sum and disjoint union are specializations of lexicographical sum for the two possible cases of two-element expression orders: the two-element chain and the two-element unordered pair. Linear sum and disjoint

2 An exception is Trotter (1992).

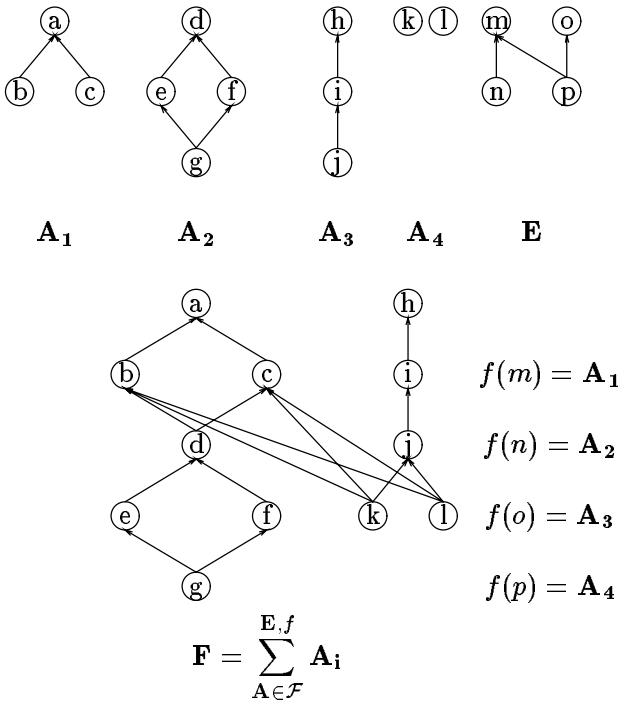



Figure 14: Lexicographical sum.

union are not primitive operators, but we define them for the sake of convenience.

- **linear sum:** $+$

  $\mathbf{A}+\mathbf{B} = \{(x,y) \mid x \in A \text{ and } y \in B \text{ or } (x,y) \in \mathbf{A} \text{ or } (x,y) \in \mathbf{B}\}$.

  Linear sum places the partial order $\mathbf{B}$ 'over' the partial order $\mathbf{A}$. It can be viewed as a restricted case of lexicographical sum in which the expression order is a total ordering of two elements (given implicitly by the order of the arguments to the linear sum).

- **disjoint union:** $\cup$

  $\mathbf{A} \cup \mathbf{B} = \{(x,y) \mid (x,y) \in \mathbf{A} \text{ or } (x,y) \in \mathbf{B}\}$.

  Union places two partial orders 'next to' each other. It can be viewed as a restricted case of lexicographical sum in which the expression order is an unordered doubleton.

We can easily define $n$-ary forms of linear sum and disjoint union: the linear sum of $n$ partial orders is the lexicographical sum over the n-element chain, and the disjoint union of $n$ partial orders is the lexicographical sum over the n-element antichain.

Another operation that is sometimes used in combinatorial applications of partial orders is *ordinal product* (Stanley 86). The ordinal product $\mathbf{P} \otimes \mathbf{Q}$ of two partial orders $\mathbf{P}$ and $\mathbf{Q}$ is equivalent to the lexicographical sum $\sum_{\mathbf{Q}}^{\mathbf{P},f} \mathbf{Q}$, where there are $|P|$ copies of $\mathbf{Q}$ (the definition of $f$ is immaterial so long as it is bijective). The main contribution of ordinal product (apart from notational shorthand) is that it introduces the question of what it means for copies of $\mathbf{Q}$ to be distinct. We can easily incorporate ordinal product into our definition of lexicographical sum if we understand each $\mathbf{Q}$ to be a 'clone' of some base partial order.

Cross product and lexicographical sum are operators for composing partial orders. It is natural to ask whether there are complementary operators for decomposing them. Consider two possible complementary operators: projection and reduction.

- **projection:** $\pi$

  $\pi\mathbf{A} = \{\mathbf{A_i} \mid \mathbf{A} \cong \mathbf{A_1} \times \mathbf{A_2} \times \cdots \times \mathbf{A_m}\}$.

  Projection is the factoring of a partial order into components whose cross product will reproduce the original partial order. Thus, cross product is analogous to multiplication of whole numbers, and projection is analogous to division.

- **reduction:** $\Re$

  $\Re\mathbf{P} = \mathcal{F}$, where $\mathbf{P} = \sum_{\mathbf{A}\in\mathcal{F}}^{\mathbf{E},f} \mathbf{A}$.

  Reduction produces a collection of partial orders that can be lexicographically summed to produce the original partial order.

Not every partial order can be projected into (non-trivial) components. In keeping with the terminology of number theory, we call a

partial order *prime* if its projection results in only two factors: the partial order itself, and the unit order. Every other partial order is called *composite*.

There is no unique prime decomposition of general partial orders (McKenzie 87). Connected partial orders do have a unique prime decomposition (Hashimoto 51).

Projection of partial orders is not the same as, yet resembles, projection of relations. Both operators produce $k$-tuples from $n$-tuples, where $k \leq n$, both involve the elimination of duplicates, and both are like multiplicative inverses. Relational projection, however, involves the specification of the substructure to be obtained, whereas partial order projection does not. Another difference between relational projection and partial order projection is that there is no notion of 'prime relations'; any $n$-ary relation can be non-trivially projected, whereas this is only true for composite partial orders. On the other hand, the product of relational projections is not necessarily the original relation, while the product of partial order projections is the original partial order.

Not every partial order can be reduced into (non-trivial) components. A *non-trivial lexicographical sum* is one in which the factor orders are not all single elements, and the expression order is not a single element. A partial order is *decomposable with respect to lexicographical sums* if it is isomorphic to a non-trivial lexicographical sum; if it is isomorphic to no non-trivial lexicographical sum, then it is said to be *indecomposable*.

A concept related to indecomposability is *t-irreducibility*. In order to define this concept, we need the notion of dimension. Any partial order can be expressed as the intersection of a set of total orders. Such a set is known as a *realizer* of a partial order, and its members are *linear extensions* of the partial order. The *size* of a realizer is the number of linear extensions it contains. For any given partial order, the size of the smallest realizer is the *dimension* of the partial order. A partial order $\mathbf{P}$ is $t$-irreducible if the dimension of $\mathbf{P}$ is $t$, and the dimension of any suborder of $\mathbf{P}$ is strictly less than $t$.

Given the above definitions, t-irreducibility and indecomposability are connected in the following result:

**Theorem 2.1 (Trotter 92)** *If a partial order* $\mathbf{P}$ *is t-irreducible for* $t \geq 2$*, then* $\mathbf{P}$ *is indecomposable with respect to lexicographical sums.*

Examples of indecomposable partial orders include the one-element partial order and the two-element antichain. Any partial order with a greatest or least element is decomposable, with the exception of the one-element partial order.

Reduction does not always produce a unique result. A chain of length $k$, for example, can be reduced in at least $k$ ways.

Since neither reduction nor projection are guaranteed to produce a unique result, we cannot define these operations as generally as intuition might prefer. We can imagine restricted versions of the operations. One possibility is 'attribute projection', which would mimic relational projection: a partial order of $n$-ary elements can be factored into $n$ components, one for each attribute. Each component would manifest only those aspects of the order relation that could be deduced from the attribute defining that component. Similarly, we can imagine a restriction of reduction which might be called 'linear sum reduction'; limiting the factors to those that can be combined with the linear sum operation. As we do not have need here for either reduction or projection, we leave the definition of restricted versions of these operations to some future study.

The operators that have been described in this section are derived from a variety of sources. $\uparrow$, $\downarrow$, $\times$, $+$, $\cap$, and $\cup$ are standard operators to be found in most books about partial orders. $\Sigma$ is a less well-known algebraic operator. $\sigma$ is derived from the relational model. $\perp$, $\top$, $\mp$, and $\pm$ are derived by analogy with Prolog and Lisp operators for dealing with lists. $\sim$ is an operator that was invented to give us something like difference, while still maintaining closure. $\beta$ was defined in order to permit conversions between partially ordered and unordered sets.

Some natural identities:

$$\downarrow(\mathbf{A},\ e) = \overline{\uparrow(\overline{\mathbf{A}},\ e)}$$

$$\mathbf{A} \cap \mathbf{A} = \mathbf{A}$$

$$\mathbf{A} \sim \mathbf{A} = \emptyset$$

$$\perp\mathbf{A} = \top\overline{\mathbf{A}}$$

$$\top\mathbf{A} = \perp\overline{\mathbf{A}}.$$

Note that

$$\mathbf{A} \cup \mathbf{A} \neq \mathbf{A}$$

$$\mathbf{A} + \mathbf{A} \neq \mathbf{A}.$$

If $\mathbf{A}$ is the unordered set, then

$$\top\mathbf{A} = \perp\mathbf{A} = A$$

$$\pm\mathbf{A} = \mp\mathbf{A} = \emptyset.$$

Some of the operators are reducible to combinations of the other operators, and hence are not primitive. Among these we have:

$$\beta\mathbf{A} = \mathbf{A} \cap \overline{\mathbf{A}}$$

$$\pm\mathbf{A} = \sigma(\mathbf{A}, A - \perp\mathbf{A})$$

$$\mp\mathbf{A} = \sigma(\mathbf{A}, A - \top\mathbf{A})$$

We next describe some of the algebraic properties of the primitive operators:

Associativity:
$$\mathbf{A} \cap (\mathbf{B} \cap \mathbf{C}) = (\mathbf{A} \cap \mathbf{B}) \cap \mathbf{C}$$
$$\mathbf{A} \cup (\mathbf{B} \cup \mathbf{C}) = (\mathbf{A} \cup \mathbf{B}) \cup \mathbf{C}$$
$$\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C}$$

Distributivity:
$$\mathbf{A} \times (\mathbf{B} + \mathbf{C}) = (\mathbf{A} \times \mathbf{B}) + (\mathbf{A} \times \mathbf{C})$$

Commutativity:
$$\mathbf{A} \cup \mathbf{B} = \mathbf{B} \cup \mathbf{A}$$
$$\mathbf{A} \cap \mathbf{B} = \mathbf{B} \cap \mathbf{A}$$
$$\mathbf{A} \times \mathbf{B} \cong \mathbf{B} \times \mathbf{A}$$

The $\emptyset$ order can be combined with other orders using the operators of the algebra. The following identities hold:

$$\top\emptyset = \bot\emptyset = \emptyset$$

$$\mathbf{A} \cup \emptyset = \mathbf{A}$$

$$\mathbf{A} \cap \emptyset = \emptyset$$

$$\mathbf{A} + \emptyset = \mathbf{A}$$

$$\overline{\emptyset} = \emptyset$$

$$\mathbf{A} \times 1 \cong 1 \times \mathbf{A} \cong \mathbf{A}$$

$$\sum_{\mathbf{A} \in \mathcal{F}}^{\emptyset, f} \mathbf{A} = \emptyset$$

$$\sum^{\mathbf{E}, f} \emptyset = \emptyset$$

$$\uparrow (\mathbf{A}, \top\mathbf{A}) = \top\mathbf{A}$$

$$\downarrow (\mathbf{A}, \bot\mathbf{A}) = \bot\mathbf{A}.$$

$$\uparrow (\mathbf{A}, \emptyset) = \downarrow (\mathbf{A}, \emptyset) = \emptyset$$

Next we consider an implementation for the operators of the partial order model.

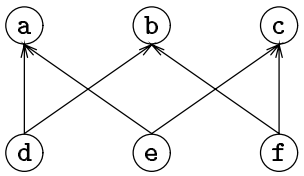
# 3

## Representing partial orders with realizers

A representation of the partial order model includes both a data structure to hold partial orders, and algorithms on that data structure that support the operators of the model. In this chapter we study *realizers* as a representation. We need not use realizers; any data structure that can manage a directed acyclic graph could be used. Most of these structures, such as adjacency matrices and adjacency lists, are already well-studied, and we do not need to investigate them here. We study realizers instead, because they are specific to partial orders, because they simplify some kinds of reasoning about the model, and because there are some natural relationships between redundancy control in realizers and the algorithms we develop for lexicographical sum. We will show at the end of this chapter that realizers are not substantially more consumptive of space than the reasonable alternative of storing the Hasse diagram.

Recall that a *linear extension* of a partial order is a total order that embeds the partial order, and that a *realizer* of a partial order is a set of linear extensions whose intersection is exactly the set of comparabilities and incomparabilities defined by the partial order. For distinct $a$ and $b$, we will say that $a < b \in R$ if $a$ precedes $b$ in every linear extension of $R$. We will also say that $a \sim b \in R$ if $a$ precedes $b$ in at least one linear extension of $R$, and $b$ precedes $a$ in at least one linear extension of $R$. Note that $a \leq a$ is always true for any $a$ in any linear extension. Figure 15 shows a partial order and one of its realizers.

Realizers are generally neither unique nor minimal. Even minimal realizers are not necessarily unique. Every total order, however, is its own realizer, and is both minimal and unique.




```
fecdba
deafbc
fdbeca
```

Figure 15: A partial order and its realizer.

Since we will typically deal with only two or three partial orders at one time, we will follow the convention of using $R$, $S$, and $T$ to denote specific realizers of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, respectively.

The *size* of a realizer is the number of linear extensions it contains. We denote the size of $R$ as $|R|$. Note that the size of a realizer $R$ for a partial order $\mathbf{A}$ is not the same as the *dimension* of $\mathbf{A}$; the dimension of $\mathbf{A}$ is the size of the smallest realizer for $\mathbf{A}$.

We sometimes need to refer to individual linear extensions of a given realizer. For this purpose, we assume some arbitrary order on the set of linear extensions, and denote each member of the set with array notation: $R[1]$, $R[2]$, $R[3]$, up to $R[|R|]$. Many algorithms iterate over a realizer's members, using a loop variable $k$. In all such algorithms, it will be acceptable to use $R[|R|]$ for any $k > |R|$. This avoids complicating the algorithms with checks that $k$ does not exceed $|R|$.

We will also need to refer to individual elements of a given linear extension of a given realizer. This is done with matrix notation: $R[i][1]$, $R[i][2]$, $R[i][3]$, up to $R[i][|A|]$, where $R$ is a realizer for the partial order $\mathbf{A}$. This notation permits one to imagine that a realizer is a matrix, with linear extensions as rows of the matrix, and elements of the extension as individual values in each row.

In the course of producing realizers, we will apply operations on the linear extensions of other realizers. We define the following two operations on linear extensions:

1. **catenation**

   Given linear extensions $X$ and $Y$, the catenation $X \cdot Y$ is the total order such that

   (a) $a < b \in X$ implies that $a < b \in X \cdot Y$

(b) $a < b \in Y$ implies that $a < b \in X \cdot Y$

(c) $a \in X, b \in Y$ implies that $a < b \in X \cdot Y$

Catenation represents the linear sum of linear extensions.

2. **augmentation**

   Given a linear extension $X$ and an element $k$, the augmentation $X(k)$ is the total ordering of ordered pairs $(X[i], k)$ such that $(X[i], k) < (X[j], k)$ if and only if $X[i] < X[j]$. It is thus a cross product with a singleton set.

   Augmentation converts the elements of a linear extension into ordered pairs, where the second member of each pair is a fixed element.

   We also make use of a dual operation called *pre-augmentation*. A pre-augmentation consists of a total ordering of ordered pairs $(k, X[i])$ such that $(k, X[i]) < (k, X[j])$ if and only if $X[i] < X[j]$.

The following lemmas will be used in proving the correctness of our algorithms.

**Lemma 3.1 (Realizer lemma)** *$R$ is a realizer of $\mathbf{A}$ if and only if it satisfies the following three conditions:*

1. *$a \in A$ if and only if $a \in R$*
2. *If $a < b \in \mathbf{A}$, then $a < b \in R$*
3. *If $a \sim b \in \mathbf{A}$, then $a \sim b \in R$*

PROOF If the three conditions hold, then $R \supseteq \mathbf{A}$, because $R$ contains each $a < b \in \mathbf{A}$ and each $a \sim b \in \mathbf{A}$. But it is also true that $\mathbf{A} \supseteq R$, because it is not possible for there to be any $a < b \in R$ such that $a < b \notin \mathbf{A}$. If such an $a < b \in R$ did exist, then it must be the case that either $a \sim b$ or $b < a$ is in $\mathbf{A}$. But by definition, $R$ would contain either $a \sim b$ or $b < a$, and thus it cannot contain $a < b$. By a similar argument, it is not possible for there to be any $a \sim b \in R$ such that $a \sim b \notin \mathbf{A}$. Thus $R$ is a realizer for $\mathbf{A}$. □

The realizer lemma is useful for demonstrating that a constructed set of linear extensions is actually a realizer.

The next lemma shows that, given certain restrictions, catenating linear extensions from a set of realizers to produce a new realizer will preserve the original partial orders.

**Lemma 3.2 (Catenation lemma)** *Let $\mathcal{S}$ be a set of realizers of disjoint partial orders, and $R$ be a realizer of a new partial order such that*

1. *Every linear extension of $R$ is a catenation of linear extensions in $\mathcal{S}$.*
2. *Every $R[j]$ contains exactly one linear extension from each element of $\mathcal{S}$.*
3. *Every linear extension in $\mathcal{S}$ appears in at least one $R[j]$.*

*Then $a \sim b \in \mathcal{S}$ implies that $a \sim b \in R$, and $a < b \in \mathcal{S}$ implies that $a < b \in R$.*

PROOF By conditions 1 and 2, $R$ is a realizer for a partial order over the same elements as $\mathcal{S}$.

Choose an arbitrary $a \sim b \in \mathcal{S}$. This implies that there is a linear extension in some element of $\mathcal{S}$ such that $a < b$, and another linear extension such that $a > b$. By condition 3, both of these linear extensions must exist in $R$, and so it must be the case that $a \sim b \in R$.

Similarly, choose an arbitrary $a < b \in \mathcal{S}$. This implies that $a < b$ in every linear extension in some element of $\mathcal{S}$. But since $R$ is composed solely of linear extensions from $\mathcal{S}$, then $a < b$ in every linear extension of $R$. Thus $a < b \in R$. □

The catenation lemma guarantees that any realizer produced solely by catenation of linear extensions preserves the relationships of the component partial orders; in effect, we can 'project' each component order out of the result.

Formally, a realizer is a set of linear extensions; it has no duplicates. Because testing for duplicates can be expensive, we permit our realizers to contain duplicate linear extensions. Duplicate elimination can sometimes be efficiently done based on knowledge about the type of partial order that is being represented; for example, if we know the partial order is unordered, then only two linear extensions are needed in its realizer.

Finally, note well that we do not depend on having a minimal realizer for any partial order. Some algorithms would be simplified if we could guarantee a minimal realizer, and some bounds would be improved. But determining the minimal realizer for any partial order of dimension greater than 2 is known to be NP-complete (Yannakakis 82); thus, we do not require minimality in our realizers.

We now consider algorithms for each operator and predicate.

TOTAL

The predicate $\Updownarrow$ is implemented by testing that all linear extensions are identical.

**Algorithm 3.1** *Assume that* **A** *has realizer* $R$. *Return* `true` *if* **A** *is totally ordered, and* `false` *otherwise.*

1. If $|R| = 1$, return `true`.
2. If $|R| > 1$, then if $R[i][j] \neq R[k][j]$, for some $i, j, k$, then return `false`; otherwise, return `true`.

**Theorem 3.1** *Algorithm 3.1 implements* $\Updownarrow$ **A**.

PROOF Trivially true. □

This predicate can be tested in time $O(|R| \times |A|)$. If the algorithm returns true, then **A** can be represented by a minimal realizer consisting of any single linear extension in $|R|$.

UNORDERED

The predicate $\Leftrightarrow$ can be implemented by showing that the set of maximal elements is the entire set (or alternatively, by showing that

the set of minimal elements is the entire set).

**Algorithm 3.2** *Assume that* $\mathbf{A}$ *has realizer* $R$. *Return* `true` *if* $\mathbf{A}$ *is unordered, and* `false` *otherwise.*

1. If $\top\mathbf{A} = A$, return `true`; otherwise, return `false`.

**Theorem 3.2** *Algorithm 3.2 implements* $\Leftrightarrow \mathbf{A}$.

> PROOF If the set of maximal elements is the original set, then there can be no order relationships among any of the elements. Thus $\mathbf{A}$ must be unordered. □

This predicate can be tested in $O(|A| + M)$ time, where $M$ is the time it takes to determine the maximal elements. If the predicate returns `true`, then $\mathbf{A}$ can be represented by a minimal realizer consisting of $R[1]$ and $\overline{R[1]}$.

EQUALITY

Equivalent partial orders can be represented by different realizers; thus the test for equality does not reduce to testing for identical realizers. We can test for equality by showing that every element has the same prime ideal in both partial orders.

**Algorithm 3.3** *Assume* $\mathbf{A}$ *has realizer* $R$ *and* $\mathbf{B}$ *has realizer* $S$. *Return* `true` *if* $\mathbf{A} = \mathbf{B}$, *and return* `false` *otherwise.*

1. If $A \neq B$, return `false`.
2. $\forall a \in A$, if $\beta\downarrow(\mathbf{A}, a) = \beta\downarrow(\mathbf{B}, a)$ then return `true`; otherwise return `false`.

**Theorem 3.3** *Algorithm 3.3 implements* $=$.

> PROOF First we show that if corresponding prime ideals are identical, then so are $\mathbf{A}$ and $\mathbf{B}$. Assume that $\forall a \in A$, $\beta\downarrow(\mathbf{A}, a) = \beta\downarrow(\mathbf{B}, a)$, but $\mathbf{A} \neq \mathbf{B}$. Then either
>
> 1. $\exists(x, y) \in \mathbf{A}$ where $(x, y) \notin \mathbf{B}$. But then it must be the case that $x \in\downarrow(\mathbf{A}, y)$ and $x \notin\downarrow(\mathbf{B}, y)$.

2. $\exists (x, y) \in \mathbf{B}$ where $(x, y) \notin \mathbf{A}$. But then it must be the case that $x \in \downarrow (\mathbf{B}, y)$ and $x \notin \downarrow (\mathbf{A}, y)$.

Both of these conditions lead to a contradiction, so if corresponding prime ideals are identical, then so are the partial orders.

Next we show that if any corresponding prime ideals are different, then so are $\mathbf{A}$ and $\mathbf{B}$. Assume that $\exists a \in A$ for which $\beta \downarrow (\mathbf{A}, a) \neq \beta \downarrow (\mathbf{B}, a)$. Then either

1. $\exists y \in \beta \downarrow (\mathbf{A}, a)$ and $y \notin \beta \downarrow (\mathbf{B}, a)$. But then $(y, a) \in \mathbf{A}$ and $(y, a) \notin \mathbf{B}$, and so $\mathbf{A} \neq \mathbf{B}$.
2. $\exists y \in \beta \downarrow (\mathbf{B}, a)$ and $y \notin \beta \downarrow (\mathbf{A}, a)$. But then $(y, a) \in \mathbf{B}$ and $(y, a) \notin \mathbf{A}$, and so $\mathbf{A} \neq \mathbf{B}$. $\square$

Two points should be noted about the algorithm for equality testing. First, it is not specifically designed to exploit (or require) realizers as a data structure; it is purely algebraic. To work with realizers, this algorithm requires that $\downarrow$ be implemented for realizers. The second point is that the basic underlying effect of the algorithm is to perform transitive closure; if this is done naively (as for instance, if we first compute $\downarrow$ for $a$ and then for elements $b$ where $a < b$), then much redundant work will be performed. Thus, the order in which the prime ideals are computed is important. The run time for equality testing is $O(|A| \times I)$, where $I$ is the time to compute $\downarrow (\mathbf{A}, a)$.

CONTAINMENT

Testing for containment can be reduced to testing for set containment and then doing an equality test on the induced suborder.

**Algorithm 3.4** *Assume that* $\mathbf{A}$ *has realizer* $R$ *and* $\mathbf{B}$ *has realizer* $S$*, and test for* $\mathbf{A} \subseteq \mathbf{B}$*.*

1. *If* $A \not\subseteq B$*, then return* `false`*.*
2. *From each linear extension of* $S$*, remove any element that is not in* $A$*. The new realizer* $T$ *denotes a partial order* $\mathbf{C}$*.*

3. *If* $\mathbf{C} = \mathbf{A}$, *return* `false`*; otherwise return* `true`.

**Theorem 3.4** *Algorithm 3.4 implements containment.*

PROOF Clearly true, because the definition of containment requires that there be only one induced partial order for any subset of elements. □

The run time for containment is $O(|B| \times |R| + E)$, where $E$ is the time needed to test for equality between $\mathbf{C}$ and $\mathbf{A}$.

DUAL

The dual of $\mathbf{A}$ is constructed by simply reversing all the linear extensions in any realizer of $\mathbf{A}$.

**Algorithm 3.5** *Assume that* $\mathbf{A}$ *has realizer* $R$. *Construct* $S$, *a realizer of* $\overline{\mathbf{A}}$, *from* $R$.

1. For $i$ from 1 to $|R|$, construct

$$S[i] = R[i][|A|] \cdot R[i][|A| - 1] \cdot R[i][|A| - 2] \cdots R[i][1]$$

**Theorem 3.5** *Algorithm 3.5 constructs* $S$, *a realizer of* $\overline{\mathbf{A}}$.

PROOF For every $a < b$ in $\mathbf{A}$, it must be the case that $a < b$ in all $R[i]$ and no $R[j]$ contains $b < a$. But then all $S[i]$ contain $b < a$ and no $S[j]$ contains $a < b$, so for every $a < b$ in $\mathbf{A}$, $b < a$ in $S$.

For every $a \sim b$ in $\mathbf{A}$, there exists an $R[i]$ that contains $a < b$ and an $R[j]$ that contains $b < a$. But then $S[i]$ contains $b < a$ and $S[j]$ contains $a < b$. Thus for every $a \sim b$ in $\mathbf{A}$, $a \sim b$ in $S$.

$S$ contains all elements of $\overline{\mathbf{A}}$, every $a < b \in \overline{\mathbf{A}}$, and every $a \sim b \in \overline{\mathbf{A}}$. Thus by the realizer lemma, $S$ is a realizer of $\overline{\mathbf{A}}$. □

Dual can be computed in $O(|A| \times |R|)$ time. It can be performed in place by swapping the $k$th and $n-k+1$th items in each realizer, for $k$ from 1 to $\lfloor n/2 \rfloor$.

It is also possible to implement dual very inexpensively by simply keeping a single bit that determines the direction in which to read the linear extensions in a realizer.

The size of the realizer of a dual is the same as the size of the original realizer.

UP

Up of a chosen element in a partial order is constructed by finding all elements that succeed it in every linear extension of a given realizer. This means removing all elements that are below the chosen one in some linear extension.

**Algorithm 3.6** *Assume that* $\mathbf{A}$ *has a realizer* $R$*, and that* $e \in A$*. Construct* $S$*, a realizer of* $\uparrow(\mathbf{A}, e)$*, from* $R$*.*

Let the index of an element $k$ in $R[i]$ be denoted by $k_i$. In other words, $k = R[i][k_i]\ \forall i, k$. Then the algorithm is as follows:

1. $S = R$.
2. For all $S[i][j] = a$ such that $j < e_i$, remove $a$ from every linear extension of $S$.

**Theorem 3.6** *Algorithm 3.6 constructs* $S$*, a realizer of* $\uparrow(\mathbf{A}, e)$*.*

PROOF

1. $S$ contains no $a$ such that $a \sim e$. For every $a \sim e$ in $\mathbf{A}$, there is some $R[i]$ that contains $a < e$ and some $R[j]$ that contains $e < a$. But in this case, $a$ satisfies step 1 of the algorithm, and so $a$ must have been removed from every linear extension in $R$. Thus $S$ contains no $a$ such that $a \sim e$.
2. $S$ contains no $a$ such that $a < e$. If $a < e$ in $\mathbf{A}$, then $a < e$ in all $R[i]$. But in this case $a_i < e_i$, and

so any such $a$ would have been removed by step 1 of the algorithm.

3. $S$ contains all $a \sim b$, where $e < a$ and $e < b$. For all such $a \sim b$, there is some $R[i]$ that contains $e < a < b$ and some $R[j]$ that contains $e < b < a$, and there is no $R[k]$ where $a < e$ or $b < e$. But in this case, $a > e$ and $b > e$ in each linear extension, and so they are not removed by step 1 of the algorithm. Thus, $S[i]$ contains $e < a < b$ and $S[j]$ contains $e < b < a$, and so $S$ contains $a \sim b$.
4. By analogous argument to 3, $S$ contains all $a < b$, where $e < a$ and $e < b$.
5. $S$ contains all $a$ such that $e < a$. The algorithm removes $a$ from $R[i]$ only if there is an $R[j]$ such that $R[j][a_j] < R[j][e_j]$; but if that is the case, then $e \not< a$.

By 1, 2, and 5, $S$ contains the elements in $\uparrow(\mathbf{A}, e)$. By 3 and 4, $S$ contains all relationships in $\uparrow(\mathbf{A}, e)$. By the realizer lemma, then, $S$ is a realizer of $\uparrow(\mathbf{A}, e)$. □

We can find $e$ in each linear extension in linear time. With one pass through the realizer, we compute a bit map for each element in $A$, setting a bit for any element $a < e$ in linear time. Finally, we construct $S$ consisting only of elements $a > e$ in all $R[i]$. Thus, up can be constructed in $O(|A| \times |R|)$ time.

The size of the realizer for up is the same as the size of the original realizer. However, some of the new linear extensions may be superfluous.

DOWN

Down is constructed analogously to up.

**Algorithm 3.7** *Algorithm 3.7 is identical to Algorithm 3.6, except that we reverse all the inequalities.*

**Theorem 3.7** *Algorithm 3.7 constructs $S$, a realizer of $\downarrow(A, e)$.*

PROOF A proof for Algorithm 3.7 can be constructed from the proof for Algorithm 3.6, simply by reversing all inequalities. □

The cost of constructing down is the same as the cost of constructing up, by duality.

BASE SET

The base set of **A** is constructed by simply returning any one of the linear extensions of any realizer of **A**, treating it as an unordered set.

**Algorithm 3.8** *Assume $S$ is a realizer of* **A**. *Return $A$, the base set of* **A**.

1. Return S[1] as an unordered set.

**Theorem 3.8** *Algorithm 3.8 constructs $A$, the base set of* **A**.

PROOF Trivially true. □

Base set can be computed in $O(|A|)$, the time needed to copy any linear extension. If we wish the base set operator to return a partial order instead of a set, then $S[1]$ and $\overline{S[1]}$ will suffice. The time is still $O(|A|)$.

MAXIMA

Maxima is implemented by finding all elements whose prime filters are empty.

**Algorithm 3.9** *Assume that* **A** *has realizer $R$. Construct $V$, a set representing $\top\mathbf{A}$, from $R$.*

1. If $\uparrow(\mathbf{A}, a) = a$, then $a \in V$.

**Theorem 3.9** *Algorithm 3.9 constructs $V = \top\mathbf{A}$.*

PROOF Trivially true.□

There are strategies that can improve the performance of $\top$ for some partial orders. First, we know that it is not necessary to find the complete prime filter of every element, since as soon as it is known that the prime filter is non-empty, its element cannot be maximal. This is a useful strategy when many elements have large prime filters. Second, we know that the elements at the rightmost end of any linear extension must be maximal, and that maximal elements are more likely to be on the right side of linear extensions; thus, it would be reasonable to start looking for maximal elements here. This is a useful strategy when the set of maximal elements is relatively small. Neither strategy is useful when the partial order is bipartite and has small height. In the worst case, we will scan the entire realizer for each element, and thus maxima will need $O(|A|^2 \times |R|)$ time.

If we wish maxima to return a partial order instead of a set, then any ordering $\mathbf{V}$ along with $\overline{\mathbf{V}}$ will suffice.

MINIMA

Minima can be implemented as the dual of maxima.

**Algorithm 3.10** *Assume that* $\mathbf{A}$ *has realizer* $R$. *Construct* $V$, *a set representing* $\perp\mathbf{A}$, *from* $R$.

**Theorem 3.10** *Algorithm 3.10 constructs* $V = \perp\mathbf{A}$.

1. If $\downarrow(\mathbf{A}, a) = a$, then $a \in V$.

   PROOF A proof for Algorithm 3.10 can be constructed from the proof for Algorithm 3.9 by substituting $\downarrow$ for $\uparrow$. □

Minima has the same cost as maxima. If we wish minima to return a partial order instead of a set, then any ordering $\mathbf{V}$ along with $\overline{\mathbf{V}}$ will suffice.

DOWN-TAIL

Down-tail can be implemented by computing the maximal elements and then removing them from each linear extension.

**Algorithm 3.11** *Assume $R$ is a realizer of $\mathbf{A}$. Construct $S$, a realizer of $\mp\mathbf{A}$.*

1. $S = R$.
2. Construct $Z = \top\mathbf{A}$.
3. Remove $Z$ from all linear extensions in $S$.

**Theorem 3.11** *Algorithm 3.11 constructs $S$, a realizer of $\mp\mathbf{A}$.*

PROOF Trivially true. □

Down-tail can be performed in $O(|A|^2 \times |R|)$ time. The dominant step is finding $\top$.

UP-TAIL

Up-tail can be implemented by computing the minimal elements and then removing them from each linear extension.

**Algorithm 3.12** *Assume $R$ is a realizer of $\mathbf{A}$. Construct $S$, a realizer of $\pm\mathbf{A}$.*

1. $S = R$.
2. Construct $Z = \bot\mathbf{A}$.
3. Remove $Z$ from all linear extensions in $S$.

**Theorem 3.12** *Algorithm 3.12 constructs $S$, a realizer of $\pm\mathbf{A}$.*

PROOF Trivially true. □

Up-tail can be performed in $O(|A|^2 \times |R|)$ time. The dominant step is finding $\bot$.

Contradiction can be implemented by computing $L \cap M$ for each $a$, where $L$ is the set of elements that are greater than $a$ in one partial order, and $M$ is the set of elements less than $a$ in the other partial order. We refer to $L \cap M$ as contradictory elements.

**Algorithm 3.13** *Assume that* $\mathbf{A}$ *has realizer* $R$ *and* $\mathbf{B}$ *has realizer* $S$*, and that* $A = B$*. Construct* $Z$*, a set of elements that contains all* $a$ *such that* $a < b$ *in* $\mathbf{A}$ *and* $b < a$ *in* $\mathbf{B}$*.*

1. For each $a$ in $A$, construct $L_a$, a set of $b$ such that $b \in L_a$ if and only if $a < b \in \mathbf{A}_1$.
2. For each $a$ in $A$, construct $M_a$, a set of $b$ such that $b \in M_a$ if and only if $b < a \in \mathbf{A}_2$.
3. For each $a$ in $A$ do

$$Z = Z \cup a \quad \text{if} \quad L_a \cap M_a \neq \emptyset$$

**Theorem 3.13** *Algorithm 3.13 constructs* $Z$*, a set of contradictory elements for* $\mathbf{A}_1$ *and* $\mathbf{A}_2$*.*

PROOF $Z$ clearly contains only contradictory elements, so we need only show that $Z$ contains all contradictory elements. Assume that there is a contradictory element $x$ that is not in $Z$. For each $a$ such that $a < x \in \mathbf{A}_1$ it must be the case that $a < x \in \mathbf{A}_2$, otherwise it would be in $Z$. But then $x$ is not contradictory. By a similar argument, we can show this for $x < a \in \mathbf{A}_1$. Finally, if $x \sim a \in \mathbf{A}_1$ or $\mathbf{A}_2$, then $x$ cannot be contradictory. Thus there is no contradictory $x$ that is not in $Z$. □

Contradiction can be computed in $O(|A|^2 \times (|R| + |S|))$ time.

One method for implementing the sets used in contradiction is with bitmaps. This method requires $|A|^2$ bits of space. Each element is associated with a bitmap of size $|A|$ that will represent the set of elements less than the given one. We process a single linear extension

starting from the right. We construct a mask as we go, setting bits for each element that we encounter in the linear extension, and computing the logical and of the mask with the bitmap for the corresponding element. We process all linear extensions in both realizers but never visit any element more than once; thus, $O(|A|^2 \times (|R| + |S|))$ time is needed.

Another approach to contradiction is to reduce it to previously defined operators. $\mathbf{A} \sim \mathbf{B}$ is $\perp(\mathbf{A} \cap \overline{\mathbf{B}}) - top(\mathbf{A} \cap \overline{\mathbf{B}})$

SELECTION

Selection can be implemented by applying the selection predicate to each of the members of one linear extension, and then removing all unsatisfying elements from each linear extension.

**Algorithm 3.14** *Assume $R$ is a realizer of $\mathbf{A}$, and that $p$ is a selection predicate. Construct $S$, a realizer of $\sigma(\mathbf{A}, p)$.*

1. $S = R$.
2. $Z = \{a \mid p(a)\}$
3. Remove $A \backslash Z$ from all linear extensions in $S$.

**Theorem 3.14** *Algorithm 3.14 constructs $S$, a realizer of $\sigma(\mathbf{A}, p)$.*

PROOF Trivially true. □

The cost of selection is $O(|A| \times (|R| + q))$, where $q$ is the cost of executing $p$ on any element of $A$.[1] Removing $Z$ from $S$ can be done as selection is evaluated.

INTERSECTION

The intersection of $\mathbf{A}$ and $\mathbf{B}$ is constructed by taking the union of the two realizers.

1 This also assumes that the difference in the cost of executing $p(a)$ and $p(b)$ is $O(1)$, for any $a, b$.

**Algorithm 3.15** *Assume that* $\mathbf{A}$ *has realizer* $R$*, and* $\mathbf{B}$ *has realizer* $S$*. Construct* $T$*, a realizer of* $\mathbf{A} \cap \mathbf{B}$*, from* $R$ *and* $S$*.*

1. For $i$ from 1 to $|R|$, construct

$$T[i] = R[i]$$

2. For $i$ from $|R| + 1$ to $|S| + |R|$, construct

$$T[i] = S[i - |R|]$$

**Theorem 3.15** *Algorithm 3.15 constructs* $T$*, a realizer of* $\mathbf{A} \cap \mathbf{B}$*.*

PROOF Since the algorithm does not change existing linear extensions, it can only remove comparabilities. Thus if $T$ is not a realizer of $\mathbf{A} \cap \mathbf{B}$, it must be because $\exists a < b \in \mathbf{A} \cap \mathbf{B}$ where $a < b \notin T$ or $\exists a < b \in T$ where $a < b \notin \mathbf{A} \cap \mathbf{B}$. But if $a < b \in \mathbf{A} \cap \mathbf{B}$, then $a$ must be to the left of $b$ in every linear extension of $\mathbf{A}$ and every linear extension of $\mathbf{B}$. But if this is the case, then $a$ is to the left of $b$ in every linear extension of $T$, and so $a < b \in T$. □

Intersection can be computed in $O(|A| \times (|R| + |S|))$ time.

With this algorithm, the size of the realizer for intersection is the sum of the sizes of the realizers of the arguments. The constructed realizer is likely to contain a non-minimal number of linear extensions, and it may also contain duplicate linear extensions.

SUM

Sum is constructed by catenating linear extensions of $\mathbf{A}$ and $\mathbf{B}$ such that the $\mathbf{A}$ component always precedes the $\mathbf{B}$ component.

**Algorithm 3.16** *Assume that* $\mathbf{A}$ *and* $\mathbf{B}$ *are disjoint partial orders with realizers* $R$ *and* $S$ *respectively. Construct* $T$*, a realizer of* $\mathbf{A}+\mathbf{B}$*, from* $R$ *and* $S$*.*

1. For $k$ from 1 to $\max(|R|, |S|)$, construct

$$T[k] = R[k] \cdot S[k]$$

**Theorem 3.16** *Algorithm 3.16 constructs $T$, a realizer of $\mathbf{A} + \mathbf{B}$.*

PROOF $T$ is constructed by catenation of linear extensions, and contains all linear extensions in $R$ and $S$. Thus, by the catenation lemma we know that $T$ preserves $R$ and $S$.

Since each $T[i]$ is a catenation of a linear extension from $A$ with a linear extension from $B$, $T$ establishes that $a < b$, $\forall a \in A, b \in B$.

Since each $T[i]$ contains $A \cup B$, and since $T$ contains all $a < b$ and $a \sim b$ in $\mathbf{A} \cup \mathbf{B}$, then by the realizer lemma, $T$ is a realizer of $\mathbf{A} + \mathbf{B}$. □

The size of the realizer of a sum is the maximum of the sizes of the factor realizers.
Sum can be done in $O(max(|R|, |S|) \times (|A| + |B|))$ time.

DISJOINT UNION

Disjoint union of **A** and **B**, like sum, is constructed by catenating linear extensions of **A** with those from **B**. We arrange that there will be one linear extension in the result in which the **A** component precedes the **B** component, and at least one linear extension in the result in which the **B** component precedes the **A** component. This constraint ensures that every element of $A$ is pairwise incomparable with every element of $B$.

**Algorithm 3.17** *Assume that* **A** *and* **B** *are disjoint partial orders with realizers $R$ and $S$ respectively. Construct $T$, a realizer of* $\mathbf{A} \cup \mathbf{B}$*, from $R$ and $S$.*

1. If $|R| = |S| = 1$, construct $T$ as follows:

$$T[1] = R[1] \cdot S[1]$$

$$T[2] = S[1] \cdot R[1]$$

2. If $|R| > 1$ or $|S| > 1$, then construct

$$T[1] = R[1] \cdot S[1]$$

and for $k$ from 2 to $\max(|R|, |S|)$, construct

$$T[k] = S[k] \cdot R[k]$$

**Theorem 3.17** *Algorithm 3.17 constructs $T$, a realizer of $\mathbf{A} \cup \mathbf{B}$.*

PROOF $T$ is constructed by catenation of linear extensions, and contains all linear extensions in $R$ and $S$. Thus, by the catenation lemma we know that $T$ preserves $R$ and $S$.

$T[1]$ and $T[2]$ establish that $a \sim b$, $\forall a, b$ where $a \in A$ and $b \in B$.

Since each $T[i]$ contains $A \cup B$, and $T$ contains all $a < b$ and $a \sim b$ in $\mathbf{A} \cup \mathbf{B}$, then by the realizer lemma, $T$ is a realizer of $\mathbf{A} \cup \mathbf{B}$. □

Note that disjoint union of non-empty partial orders always results in a partial order of dimension two or greater, since there is always at least one incomparable pair of elements. The size of a disjoint union is the maximum of the sizes of the factor realizers (except for the case of disjoint union of total orders).

Disjoint union can be done in $O(max(|R|, |S|) \times (|A| + |B|))$ time.

### LEXICOGRAPHICAL SUM

Lexicographical sum is the expansion of an *expression* partial order $\mathbf{E}$ by substituting *factor* partial orders $\mathbf{A}$ for each of the elements of $\mathbf{E}$, as shown in Figure 16. The basic idea of the algorithm for producing a realizer is to simulate this process on the realizer of the expression order; that is, we expand each element of the realizer for $\mathbf{E}$ by substituting linear extensions from each $\mathbf{A} \in \mathcal{F}$. The main constraint is that each linear extension of each $\mathbf{A}$ must appear at least once in the result.

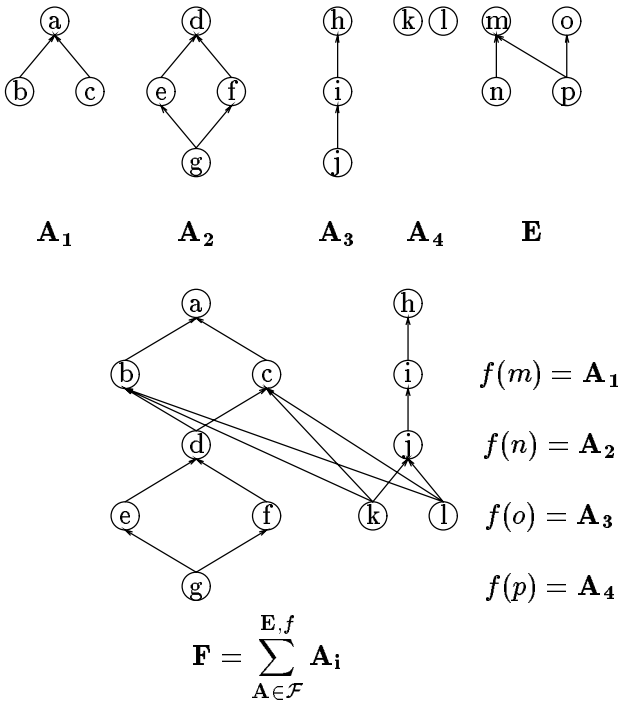



Figure 16: Example of lexicographical sum.

**Algorithm 3.18** *Assume that $\mathcal{F}$ is a family of disjoint partial orders $\mathbf{A_1}, \mathbf{A_2}, \cdots \mathbf{A_n}$, and that each $\mathbf{A_i}$ has a realizer $R_i$. Let $\mathbf{E}$ be a partial order with $n$ elements, and let $S$ be a realizer of $\mathbf{E}$. Let $f$ be a one-to-one mapping $f : E \rightarrow \mathcal{F}$. Construct $T$, a realizer of $\sum_{\mathbf{A}\in\mathcal{F}}^{\mathbf{E},f} \mathbf{A_i}$, from $S$ and the set of $R_i$.*

We will let $m = |S|$, $d = max(|R_i|)$, and $q = max(d,\ m)$. Since $f$ is a one-to-one map from elements of $E$ to $\mathbf{A_i}$, it is also a one-to-one map from elements of linear extensions in $\mathbf{E}$'s realizer to the set of realizers of $\mathbf{A_i}$. That is,

$$f : S[j][k] \rightarrow R_i$$

We will use the notation

$$f(S[j][k])[r]$$

to denote the $r$th linear extension of $R_i$.

Given the preceding, the algorithm is as follows:

$$
\begin{aligned}
R_1 &= bca,\ cba \\
R_2 &= gefd,\ gfed \\
R_3 &= jih \\
R_4 &= kl,\ lk \\
S &= nmpo,\ pnom \\
f(S) &= R_1R_2R_3R_4 \\
&\quad\ R_3R_1R_4R_2 \\
T &= bcagefdjihkl \\
&\quad\ jihcbalkgfed
\end{aligned}
$$

Figure 17: Steps in the production of the realizer for the lexicographical sum in Figure 16.

1. For $1 \leq j \leq q$, construct

$$T[j] = f(S[j][1])[j] \cdot f(S[j][2])[j] \cdot f(S[j][3])[j] \cdots f(S[j][n])[j]$$

The use of Algorithm 3.18 to produce the realizer for Figure 16 is shown in Figure 17.

**Theorem 3.18** *Algorithm 3.18 constructs $T$, a realizer of* $\sum_{\mathbf{A}\in\mathcal{F}}^{\mathbf{E},f} \mathbf{A}$.

PROOF

1. There are $q$ linear extensions in $T$. The algorithm constructs each $T[j]$ by using the $j$th linear extension from each $R_i$. Since no $R[i]$ has more than $q$ linear extensions, each linear extension in each $R_i$ appears in some $T[j]$.
2. Each $T[j]$ consists of at most one linear extension from each $R_i$, because $f$ is one-to-one, and so each element of a given $T[j]$ is mapped to a different element of $R_i$.

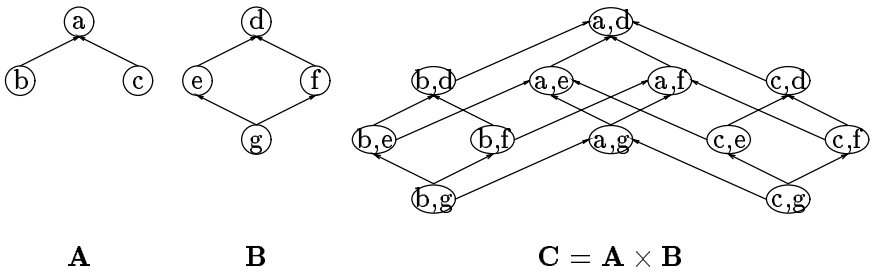



Figure 18: Example of cross product

3. $T$ contains all $a < b$ in $\sum_{\mathbf{A}\in\mathcal{F}}^{\mathbf{E},f} \mathbf{A}$, where $a \in A_i$ and $b \in A_j, i \neq j$. Assume without loss of generality that $x < y \in \mathbf{E}$, and $f(x) = A_i, f(y) = A_j$. If $x < y$ in $\mathbf{E}$, then there is no linear extension in $S$ where $x > y$. Thus there is no linear extension in $T$ where $A_i > A_j$. Thus, $a < b, \forall a \in A_i, b \in A_j$.

4. By a similar argument to 3, $T$ contains all $a \sim b$ in $\sum_{\mathbf{A}\in\mathcal{F}}^{\mathbf{E},f} \mathbf{A}$ where $a \in A_i$ and $b \in A_j$.

By 1 and 2, $T$ satisfies the catenation lemma, and hence preserves all $\mathbf{A_i}$. By 3 and 4, $T$ preserves $a < b$ and $a \sim b$ from $\sum_{\mathbf{A}\in\mathcal{F}}^{\mathbf{E},f} \mathbf{A_i}$. Thus, by the realizer lemma, $T$ is a realizer of $\sum_{\mathbf{A}\in\mathcal{F}}^{\mathbf{E},f} \mathbf{A_i}$. □

The size of the realizer of a lexicographical sum is the maximum of the sizes of the factor and expression realizers.

The algorithms for producing disjoint union and linear sum are special cases of the general algorithm for producing lexicographical sum. Disjoint union is a lexicographical sum with an expression order whose realizer is $\{ab, ba\}$; linear sum is a lexicographical sum with the expression order whose realizer is $\{ab\}$.

Lexicographical sum can be done in $O(q \times \cup_i |A_i|)$ time.

CROSS PRODUCT

Cross product involves the generation of a new base set $C = A \times B$. These new elements must then be ordered so that if we restrict elements of $\mathbf{C}$ to their $A$ component (eliminating duplicates), we will

$$
\begin{aligned}
R &= bca,\ cba \\
S &= gefd,\ gfed
\end{aligned}
$$

$$
\begin{aligned}
X[1] &= bg \cdot be \cdot bf \cdot bd \\
X[2] &= cg \cdot ce \cdot cf \cdot cd \\
X[3] &= ag \cdot ae \cdot af \cdot ad \\
X[4] &= cg \cdot ce \cdot cf \cdot cd \\
X[5] &= bg \cdot be \cdot bf \cdot bd \\
X[6] &= ag \cdot ae \cdot af \cdot ad
\end{aligned}
$$

$$
\begin{aligned}
Y[1] &= bg \cdot cg \cdot ag \\
Y[2] &= be \cdot ce \cdot ae \\
Y[3] &= bf \cdot cf \cdot af \\
Y[4] &= bd \cdot cd \cdot ad \\
Y[5] &= bg \cdot cg \cdot ag \\
Y[6] &= bf \cdot cf \cdot af \\
Y[7] &= be \cdot ce \cdot ae \\
Y[8] &= bd \cdot cd \cdot ad
\end{aligned}
$$

$$
\begin{aligned}
T = {} & bg \cdot be \cdot bf \cdot bd \cdot cg \cdot ce \cdot cf \cdot cd \cdot ag \cdot ae \cdot af \cdot ad \\
& cg \cdot ce \cdot cf \cdot cd \cdot bg \cdot be \cdot bf \cdot bd \cdot ag \cdot ae \cdot af \cdot ad \\
& bg \cdot cg \cdot ag \cdot be \cdot ce \cdot ae \cdot bf \cdot cf \cdot af \cdot bd \cdot cd \cdot ad \\
& bg \cdot cg \cdot ag \cdot bf \cdot cf \cdot af \cdot be \cdot ce \cdot ae \cdot bd \cdot cd \cdot ad
\end{aligned}
$$

Figure 19: Realizer for cross product in Figure 18

produce the partial order $\mathbf{A}$, and if we restrict elements of $\mathbf{C}$ to their $B$ component (eliminating duplicates), we will produce the partial order $\mathbf{B}$. This suggests that we can construct partial linear extensions for the cross product $\mathbf{C}$ by substituting the elements of $C$ in the linear extensions of the realizers for $\mathbf{A}$ and $\mathbf{B}$. This substitution can be performed by augmentation (recall that augmentation is like cross product with a singleton set). In order to construct complete linear extensions, we then catenate these partial extensions.

**Algorithm 3.19** *Assume that* $\mathbf{A}$ *and* $\mathbf{B}$ *are disjoint partial orders with realizers* $R$ *and* $S$ *respectively. Construct* $T$*, a realizer of* $\mathbf{A} \times \mathbf{B}$*, from* $R$ *and* $S$*.*

We assume without loss of generality that $|R| = r$, $|S| = s$, $|A| = a$, and $|B| = b$. We first construct intermediate orders, using augmentation and pre-augmentation. Then we catenate the intermediate orders to form the linear extensions of $T$.

1. For $1 \leq i \leq r$ and $1 \leq j \leq a$, construct:

$$X[j + (i-1) * a] = (R[i][j])S[1]$$

2. For $1 \leq i \leq s$ and $1 \leq j \leq b$, construct:

$$Y[j + (i-1) * b] = R[1](S[i][j])$$

3. For $0 \leq i \leq s - 1$, construct:

$$T[i+1] = X[ia+1] \cdot X[ia+2] \cdots X[ia+a]$$

4. For $0 \leq i \leq r - 1$, construct:

$$T[s+i+1] = Y[ib+1] \cdot Y[ib+2] \cdots Y[ib+b]$$

**Theorem 3.19** *Algorithm 3.19 constructs $T$, a realizer of* $\mathbf{A} \times \mathbf{B}$.

PROOF

1. $T \subseteq \mathbf{A} \times \mathbf{B}$
   Assume the contrary. Then $\exists z = ((x_1, y_1), (x_2, y_2)) \in T$, $z \notin \mathbf{A} \times \mathbf{B}$. Either there is a linear extension in $R$ such that $x_2 < x_1$, or a linear extension in $S$ such that $y_2 < y_1$. But if there is a linear extension in $R$ such that $x_2 < x_1$, then by the construction of $T$ in step 3, there exists some $T[i]$ in which $(x_2, y_j) < (x_1, y_k)$ is true, $\forall (y_j, y_k) \in \mathbf{B}$. Similarly, if there is a linear extension in $S$ such that $y_2 < y_1$, then there exists some $T[i]$ in which $((x_j, y_2), (x_k, y_1))$ is true, $\forall (x_j, x_k) \in \mathbf{A}$. In either case there is a contradiction with $z \in T$.

2. $\mathbf{A} \times \mathbf{B} \subseteq T$
   Assume $\exists z = ((x_1, y_1), (x_2, y_2)) \in \mathbf{A} \times \mathbf{B}$ , $z \notin T$. Then there must be at least one linear extension in $T$ for which $(x_2, y_2) < (x_1, y_1)$. But this linear extension cannot be a catenation of pre-augmentations of $S[1]$, because the elements are either chosen from a single $X[i]$ (in which case $x_1 = x_2$ and $y_1 < y_2$) or else the elements are chosen from distinct $X[i]$ and $X[j]$ (in which case $x_1 < x_2$). Similarly, this linear extension cannot be a catenation of augmentations of $R[1]$, because the elements are either chosen from a single $Y[i]$ (in which case $y_1 = y2$ and $x_1 < x_2$) or else they are chosen from distinct $Y[i]$ and $Y[j]$ (in which case $y1 < y_2$). Hence no such $z$ exists. □

Note that the size of the realizer of a cross product is the sum of the sizes of the realizers of its two factors. Cross product can be performed in $O((|R| + |S|) \times |A| \times |B|)$.

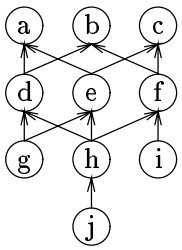



Figure 20: Example partial order.

## PRODUCING A REALIZER

Our algorithms accept realizers and produce new ones, and so closure is maintained. We must, however, have some way of starting: we need to be able to produce a realizer for a given partial order that is described in some other way, perhaps with an adjacency matrix or a covering relation.

We would like to produce a small realizer, but producing a minimal realizer for a partial order of dimension greater than 2 is an NP-complete problem. This fact has led many researchers to restrict their attention to classes of partial orders whose dimensions are known to be less than 3. Interval orders and series-parallel orders are two classes of partial orders with dimension less than 3.

For our purposes minimal realizers are not necessary. As long as we have some realizer for a partial order, we can manipulate it with the algorithms previously presented. As long as our realizers are not too big, we can process partial orders with reasonable efficiency. In any case, we cannot restrict ourselves to partial orders of dimension 2 or less if we want to use all the operators in the model. Applying cross product to partial orders that contain incomparable elements will generally result in a partial order with dimension greater than 2. Thus, if we intend to implement cross product, we must be prepared to handle higher-dimension partial orders.

We therefore consider a polynomial-time algorithm for generating a non-minimal realizer $R$ for a given partial order of arbitrary dimension. Before presenting a formal description of the algorithm, we describe it intuitively. Consider the example of Figure 20. Any linear extension of a partial order contains all of its comparabilities,

Our goal is to construct a set of linear extensions that contains all of its incomparabilities as well. We will start with a pair of linear extensions that contain some known subset of incomparabilities, and refine it, adding new linear extensions until all incomparabilities are accounted for.

We will call the initial pair of linear extensions *antichain extensions*. We so denote them because they contain all the incomparabilities that arise from some partition of the partial order into antichains. By Dilworth's chain covering theorem (Dilworth 50), we can always partition a partial order into $n$ antichains, where $n$ is the height of the partial order. For the example in Figure 20, we may generate the following set of antichains:

$$\{\ \mathtt{j}\ \},\ \{\ \mathtt{g},\ \mathtt{h},\ \mathtt{i}\ \},\ \{\ \mathtt{d},\ \mathtt{e},\ \mathtt{f}\ \},\ \{\ \mathtt{a},\ \mathtt{b},\ \mathtt{c}\ \}$$

Antichains have the nice property that they can always be represented by a partial order of dimension 2, consisting of one chain and its dual. The algorithm produces one antichain extension by catenating antichains, and a second by catenating their duals.[2] For the antichains in our example, two possible antichain orders are:

```
jghidefabc
jihgfedcba
```

The antichain extensions are a complete description of the partial order in which there is no incomparability except within the chosen antichains. Most partial orders, of course, have some additional incomparabilities; in Figure 20, for example, we have $\mathtt{c} \sim \mathtt{d}$, $\mathtt{b} \sim \mathtt{e}$, $\mathtt{a} \sim \mathtt{f}$, $\mathtt{f} \sim \mathtt{g}$, $\mathtt{i} \sim \mathtt{d}$, $\mathtt{i} \sim \mathtt{e}$, $\mathtt{i} \sim \mathtt{a}$, $\mathtt{j} \sim \mathtt{i}$, and $\mathtt{j} \sim \mathtt{g}$. Consequently, we need additional linear extensions to contain the contradictory orderings needed to express these additional incomparabilities.

The additional linear extensions we need can be constructed by modifying the antichain orders to produce new linear extensions. We call the algorithm for doing this the *bubbling* algorithm. Consider for example the incomparability $\mathtt{i} \sim \mathtt{a}$. Both antichain extensions contain $\mathtt{i} < \mathtt{a}$, so we must construct a new linear extension in which

2 The order in which antichains are catenated to produce an antichain extension is fixed by the partial order. The order within any given antichain is arbitrary, so long as the dual order is present in the complementary antichain.

`a` $<$ `i`. We do this by moving `i` towards `a` within a linear extension, bubbling the elements between them as necessary. We will move `i` to the right, pushing elements that are incomparable to `i` to the left, and preserving the relative position of any elements that are known to be comparable to them. For example, beginning with

`jihgfedcba`

we can produce

`jhgifedacb`

simply by rearraging the antichains to which `a` and `i` belong. In this step we have bubbled incomparable elements out of the region between `a` and `i`. In the next step, we produce

`jhgedifacb`

`i` and `f` are comparable and adjacent in the linear exension and so they constitute a subchain. We can move this subchain to the right of `ed` because `i` is incomparable to both `e` and `d`. Note that it is not necessary to check whether `e` or `d` are comparable to `f`; if they were, then by transitivity they would be comparable to `i`. Also note that we need not compute `a` and `d` to each other, since we are preserving their relative order in the linear extension. In the last step, we move `a` to the other side of the subchain, because it is incomparable to `i` (again, it is not necessary to check `a`'s comparability to `f`):

`jhgedaifcb`

The result of this sequence of swaps is a linear extension of the partial order in which `a` $<$ `i`; thus, in conjunction with the antichain orders, we now have that `i` $\sim$ `a`. Note that along the way we also managed to represent `i` $\sim$ `d` and `i` $\sim$ `e`. We can represent all other incomparabilities in a similar fashion.

In general, the bubbling algorithm for producing $y < x$ (given that we have a linear extension in which $x < y$ and it is known that $x \sim y$) requires that we move a subchain whose minimum is $x$ to the right. The subchain starting at $x$ consists of elements known to be greater than $x$. We construct this subchain as we go, appending

elements that satisfy the chain condition, and bubbling elements that do not satisfy the chain condition to the left of $x$. In effect, we are recognizing a linear extension of $\uparrow x$, and moving any elements that are not part of $\uparrow x$ to the left of $x$.

We now formalize the bubbling algorithm.

**Algorithm 3.20** *Given a partial order* $\mathbf{P}$*, an incomparability* $x \sim y \in \mathbf{P}$*, and a linear extension* $R$ *in which* $x < y$*, construct a linear extension of* $\mathbf{P}$ *in which* $y < x$*.*

> Let $i$ and $j$ be integers such that $R[i] = x$ and $R[j] = y$.
> Let $chain = 1$.
>
> For $i + 1 \leq k \leq j$ do if $x \sim R[k]$ then move the element at position $k$ to position $k - chain$, shifting all elements from position $k - chain$ to $k - 1$ to the right.
>
> Otherwise, $chain = chain + 1$.

We now show that bubbling is correct.

**Theorem 3.20** *Bubbling always constructs a valid linear extension.*

> PROOF We show that each iteration of the bubbling loop preserves the validity of the linear extension. Assume without loss of generality that we have a subchain of elements for which $x$ is the minimum element, that we wish to bubble the subchain to the right, and that $z$ is the leftmost element to the right of $x$ and not in the subchain. $z$ is either incomparable to $x$ or greater than $x$ (because it is to the right of $x$ in a valid linear extension). If $z$ is greater than $x$, the the linear extension remains valid if we append $z$ to the subchain. If $z$ is incomparable to $x$, then $z$ cannot be comparable to any element in the subchain, because of transitivity. Moving $z$ to the left of $x$ preserves all orderings except betwen $x$ and members of the subchain, and thus results in a valid linear extension.
>
> Because the loop terminates when $y$ is moved to the left of $x$ and each linear extension produced is valid, the algorithm halts with a valid linear extension. □

Given a partial order $\mathbf{P}$ described by the adjacency matrix corresponding to the Hasse diagram, construct a realizer $R$ of $\mathbf{P}$.

1. Construct a set $L$ of incomparabilities in $P$.
2. Choose a set of antichains that partitions $P$, and construct antichain extensions $X$ and $Y$. Remove the antichain incomparabilities from $L$. Set $R = \{X, Y\}$.
3. While $L$ is not empty, choose an incomparability $l_i$ from $L$ and generate a new linear extension $Z_i$, using the bubbling algorithm to ensure that $l_i$ is represented in $X \cap Y \cap Z_1 \cap Z_2 \cdots Z_i$. Remove $l_i$ from $L$, as well as any $l_j$ that may have been constructed during bubbling. Add $Z_i$ to $R$.

**Theorem 3.21** *$R$ is a realizer of $\mathbf{P}$.*

PROOF $R$ contains only valid linear extensions of $\mathbf{P}$, and so it contains every $(a, b) \in \mathbf{P}$. By the definition of the algorithm, $R$ also contains every $a \sim b \in \mathbf{P}$. Since every linear extension in $R$ contains exactly the elements of $P$, then by the realizer lemma, $R$ is a realizer of $\mathbf{P}$. □

We note that the bubbling algorithm is polynomial in the number of incomparabilities, since each incomparability requires at most moving every element in a single linear extension.

The bubbling algorithm for producing linear extensions can simplify some of the operator implementations. With suitably constructed antichain orders, we know that either the maxima or minima will be found as a contiguous sequence at the extreme end of the antichain orders; thus, at least one of maxima or minima is easy to determine.

### EFFICIENTLY STORING REALIZERS

A realizer $R$ appears to be a highly redundant structure, since it contains $|R|$ copies of every element in the partial order. Many

$$\begin{aligned}\mathbf{P} &= xabcdefzghijkly \\ & \quad xghijklabcdefyz \\ & \quad xabcdefyjklzghi\end{aligned}$$

(a) redundant storage of realizer

$$\begin{aligned}\mathbf{P} &= xAzDy \\ & \quad xDAyz \\ & \quad xAyCzB\end{aligned}$$

$$\begin{aligned}A &= abcdef \\ B &= ghi \\ C &= jkl \\ D &= BC\end{aligned}$$

(b) realizer with subsequence removal

Figure 21: Example of subsequence removal.

order relationships are likely to be repeated in a realizer, simply because it is a requirement that each linear extension contain all elements of the partial order. For example, a partial order of $n$ elements, only two of which are incomparable, will require a realizer with at least two linear extensions, and thus $2n$ in storage, even though it is possible to represent the same information in size $n+2$ (the total order plus one antichain of 2 elements). It is natural to wonder whether we can eliminate the redundancy in realizers.

One obvious approach is to extract common subsequences. If two or more linear extensions contain a common subsequence, we

can substitute a reference, and store the subsequence only once. A simple abstract representation of references employs distinguished symbols, in much the same way that nonterminals are distinguished symbols in context-free productions. Figure 21 is an example in which four subsequences can be identified; instead of storing 45 symbols, we store 21 'terminals' and 9 'nonterminals', for a total of 30 symbols.

The nonredundant form of the realizer closely resembles a set of productions for a grammar. Such a grammar for a realizer has a finite language, which is the set of linear extensions of the realizer. We can also view the productions of the grammar as denoting a containment structure of subsequences—that is, a partial order of subsequences.

If a grammar is our storage structure, we would like to find the smallest grammar for a given realizer. Let us refer to the size of the smallest grammar for a realizer as the *footprint* of the realizer. The footprint is not necessarily the minimal size of a storage structure for the partial order, as there may be another realizer with a smaller footprint. We expect that smaller realizers will always have a smaller footprint than larger ones:

**Conjecture 3.1** *Footprint is a monotonically increasing function of realizer size.*

If the conjecture is correct, then to reduce the space used to store a partial order, it is best to begin with a realizer that is as small as possible. Unfortunately we cannot in general start with a minimal realizer, since finding a minimal realizer for partial orders of dimension greater than 2 is an NP-complete problem, and so finding a minimal footprint is also likely to be NP-hard.

Some of the operator implementations that have been described so far are well-matched to the use of a grammar as a storage structure. Operations that use catenation of existing linear extensions are, in effect, generating productions of the grammar. These operators include lexicographical sum, disjoint union, and linear sum.

On the other hand, the implementations of other operators generate redundant linear extensions. Up-tail, down-tail, up, down, and selection may all produce redundant extensions, because in re-

moving some elements they may remove the differences between linear extensions. Redundant linear extensions can be removed by lexicographically sorting the realizer and then doing pairwise comparisons on linear extensions. In general we expect this to take $O((\log_2 |R| + |A|)|R|)$, where $|A|$ is the number of elements in the partial order, and $|R|$ is the size of the realizer.

## COMPARING REALIZERS TO TRANSITIVE REDUCTIONS

The transitive reduction (or Hasse diagram) is the most common way of presenting a partial order. Would this also be a good storage technique? We will see that the realizer is a competitive storage technique; sometimes better, sometimes worse, but usually within the same order of polynomial.

Let us assume that the edges in the transitive reduction are stored as ordered pairs; then we compute the cost of a transitive reduction as 2 times the number of edges in the reduction. The cost of a realizer will be computed as the number of elements in the partial order times the number of linear extensions in the realizer. Then consider the cost of storing the following cases:

**chain** Storing a totally ordered set of $p$ elements takes $p$ if done by realizer, and $2p$ if done by transitive reduction.

**antichain** Storing an unordered set of elements takes $2p$ storage in either case, if done in the most straightforward manner (the realizer has two linear extensions, and the transitive reduction needs one edge for each element, the second member of the edge being some null element). It can also be done in $p$ storage in either case (if we permit a listing of nodes for the transitive reduction, and if we use a single bit to indicate the complement of one linear extension in the realizer).

**complete bipartite** Consider the partial order shown in Figure 22, which is a complete bipartite partial order. It contains 16 edges, and hence costs 32 to store as a transitive reduction. The realizer for this partial order is of dimension 2, and so its cost is only 16. For all complete bipartite partial orders on a

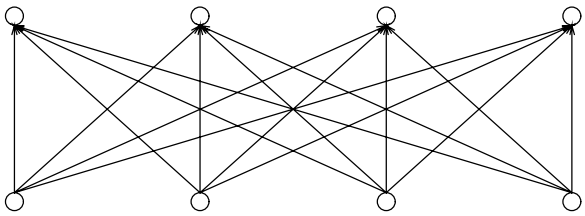


Figure 22: A complete bipartite partial order.

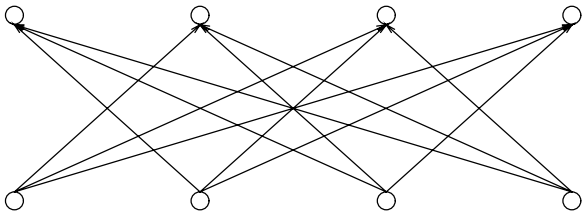


Figure 23: A standard example of a 4-dimensional partial order.

base set of size $p$, the dimension is 2, and so the cost of the realizer is $4p$, while the cost of the transitive reduction is $2p^2$.

**standard example** One common technique for constructing partial orders of arbitrary dimension is with the so-called *standard example of an $n$-dimensional partial order*. Standard examples have the property that their dimension is known to be $n$. A standard example for $n = 4$ is shown in Figure 23 (note that it is bipartite but not complete). The standard example always has $2n$ elements; since it has dimension $n$, it can be stored as a realizer with $n$ linear extensions, for a total size of $2n^2$. The transitive reduction, on the other hand, requires $n \times (n-1) \times 2$ space, or $2n^2 - 2n$.

Every partial order containing a standard example for $n$ must have a dimension of at least $n$; thus we would expect all such partial orders to require the same order of magnitude of space with either technique. However, it is not the case that every partial order of dimension $n$ contains a standard example, so the foregoing is not a proof that realizers are always roughly as efficient as the transitive reduction.

**dimension 2** Every partial order of dimension 2 is stored in less

space with a realizer if the average out-degree of the elements in the transitive reduction graph is greater than 1. The size of a transitive reduction for such a partial order must be $2np$, where $n$ is the average out-degree and $p$ is the number of elements in the partial order. Since the realizer for a 2-dimensional partial order requires only $2p$ space, it will be more efficient whenever $n > 1$.

Clearly, this observation can be extended to partial orders of dimension $k$: realizers require as little as $kp$, while the transitive reduction will require $2np$ for average out-degree $n$. Thus, the transitive reduction becomes more efficient when $2n < k$.

These examples suggest that realizers are comparable to transitive reductions in terms of the storage cost. The comparison may be somewhat optimistic, because we computed the cost of the minimal realizer, and the determination of such realizers is known to be NP-complete for dimensions greater than 2. On the other hand, we did not make use of any techniques for removing redundancy in realizers, so our estimates for the cost of realizer storage are not minimal.

# 4

# Software

We now consider two of the basic problems of software engineering: building an executable program from a collection of components, and managing multiple versions and variants of a program. These two problems are commonly handled with separate systems that do not interact very elegantly. We shall see that the problems are closely related at the abstract level, and that the partial order model is a useful framework for managing this relationship. A system that incorporates the partial order model is able to leverage the close relationship between these problems and thus gain many advantages.

## BUILDING PROGRAMS

Most programs consist of many interrelated parts. When a part is updated, the program should be rebuilt. Parts are typically stored in separate files, and are built by compiling and linking the files to generate an executable module. The key problem in building programs is this: when we make some changes to a program, how do we determine the minimal set of parts to compile?

This problem is usually solved by comparing two sets of data about the parts. The first set of data tells us what the relationship between the parts should be, and the second tells us their current relationship. We will call the former the *expected structure*, and the latter the *existing structure*. Whenever there is a difference between the expected and existing structure, we need to make an update.

The expected structure of the parts is typically given by a *build dependency* graph. The build dependency graph tells us, for each part, which other parts it depends on. Build dependency is a partial ordering of the program's parts. Build dependency must be acyclic, since we cannot build a part whose construction presupposes itself.

The build dependency is also transitive; a part cannot be built until all of its components are built, and this requirement is recursive.

The existing structure of the parts is typically inferred from the timestamps on the parts: that is, by their temporal ordering. This ordering is almost always total, since timestamps are usually assumed to be unique and comparable. If more than one timestamper is used, however (as when the program's parts are managed on different computers, each with its own timestamper), then the timestamp order may be partial, because some parts may have identical timestamps.

Given a build dependency and a temporal ordering, the straightforward building procedure is: build all the parts that are older than the parts on which they depend. The logic of this rule is simple; if a part changes, then it follows that the dependent parts may also need to change. Of course, rebuilding one part makes it newer than any parts that depend on it, and so rebuilding tends to be a recursive process.

We cannot always automatically produce the appropriate updates every time we change a part. Manually installed dependencies must often be manually maintained. For example, changing a variable of type `int` to one of type `float` necessitates looking at all assignments of that variable to other variables of type `int`, to see if the implied truncation is really what is wanted.

If we manually change the type or purpose of an external variable then we probably need to manually change code everywhere the variable is referenced. Some parts, however, are always constructed automatically, by compilers, linkers, preprocessors, archivers, and other tools. The problem of building software is usually restricted to only these operations, and aims at satisfying the following *build condition*: A program is properly built if all of its automatically-generated parts are newer than the parts on which they depend.

Consider the example in Figure 24, in which the parts of a system are individual files. $\mathbf{S_F}$ is a partial order describing the build dependencies. A file in this partial order depends on all files that can be reached by a upward path. For the purposes of the example, we assume that any file $f_i$ can be automatically generated from

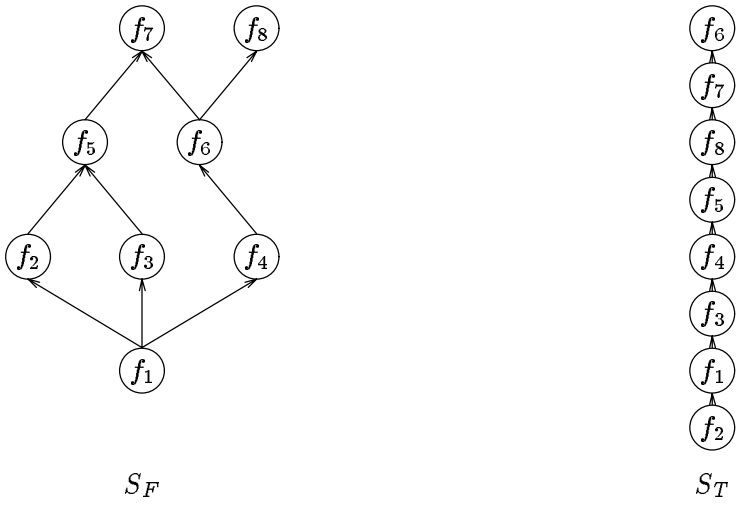



Figure 24: Program construction example.

its ancestors in the build dependency structure.[1] The partial order $\mathbf{S_T}$ describes the temporal order of the files; those files with earlier timestamps are closer to the minimal element of the partial order.

If the build condition is satisfied, then no action need be taken. In our example, we determine whether all the parts in $\mathbf{S_F}$ are newer than the parts they depend on; if they are not, we must rebuild the program. It is straightforward to prove the following:

**Theorem 4.1** *The following conditions are equivalent:*

1. $\mathbf{S_F}$ *and* $\mathbf{S_T}$ *satisfy the build condition.*
2. $\mathbf{S_T}$ *is a linearization of* $\mathbf{S_F}$.
3. $\mathbf{S_T} \cap \mathbf{S_F} = \mathbf{S_F}$.

If the build dependency order and the temporal order satisfy the build condition, then no computation is required to build the program. When the two orders are not consistent, we must update

1 Our notion of automatic generation is flexible. $f_1$, for instance, might be the include file `stream.h`, $f_2$, $f_3$, and $f_4$ might be C++ source code files, $f_5$ and $f_6$ might be libraries, and $f_7$ and $f_8$ might be programs that use the classes defined by $f_5$ and $f_6$. 'Generating' $f_2$ from $f_1$ means running the preprocessor to include $f_1$ bodily in $f_2$, while 'generating' $f_7$ from its ancestors means first generating the ancestors, compiling $f_7$, and then linking $f_7$ to the class library.

some of the parts. We wish to update the minimal number of parts, and we wish to produce a complete program. As we noted earlier, every time we update a part, we make it newer than the ones that depend on it, and so they must also be updated. This propagation of the update introduces the following property:

**Theorem 4.2** *The update structure is minimal only if it is a suborder of the build dependency structure.*

> First, note that an update is in general a partial order; it is a set of parts that must be updated in some order, possibly a partial order (that is, the order for some parts may be immaterial).
>
> Assume that we have a minimal update order that is not a suborder of the build dependency structure. Then there is some target file $f_i$ that will be updated before all of its ancestors are updated. But the transitivity of dependencies implies that $f_i$ must be updated after any of its ancestors are updated, and so $f_i$ must be updated more than once. Since in a minimal sequence of updates each file must be updated only once, any update sequence that is not a suborder of the build dependency structure cannot be minimal. □

How can we apply the partial order operators to determine which parts to update? First, we can determine the parts that do not satisfy the build condition by applying the contradiction operator:

$$\mathbf{S_T} \sim \mathbf{S_F}$$

$\mathbf{S_T} \sim \mathbf{S_F}$ finds the contradictions between the build dependency and the temporal order; for every pair $(a, b)$ in $\mathbf{S_T}$ for which $(b, a)$ is in $\mathbf{S_F}$, $\mathbf{S_T} \sim \mathbf{S_F}$ returns $a$. Assuming that we update a part by rebuilding it (and thus making it newer than any other part), then we must recognize that each such update will create new contradictions that lead to more rebuilding. This transitive effect, and the construction of the program itself, is captured by the following query:

$$\mathbf{R} = \uparrow (\mathbf{S_F}, (\mathbf{S_T} \sim \mathbf{S_F}))$$

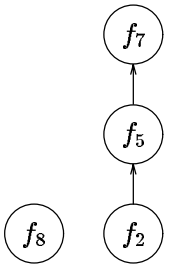



Figure 25: Files to be updated.

The result of this query on the example in Figure 24 is shown in Figure 25. Note that the result is also a partial order; $f_5$ cannot be built before $f_2$ is built, but the order in which $f_2$ and $f_8$ are built is immaterial. Incomparability is useful information if parallel or distributed building is possible. The sequence of files to be built can be determined by applying the following recursive query:

$$\mathbf{Q_i} = \perp\mathbf{R_i}$$

$$\mathbf{R_{i+1}} = \pm\mathbf{R_i}$$

The query is repeated until $R_i$ is empty. In our example, the first application of this query yields $f_2$ and $f_8$, the second application yields $f_5$, and the third yields $f_7$.

The problem of determining a valid building order for software has been reduced to applications of the operators of the partial order model. This has many advantages over current approaches, which we briefly review.

`make`

The standard program for building programs is `make` (Feldman 79). `make` operates according to a textual description called a *makefile*. A makefile is a list of rules for constructing individual targets from their components. Rules in a makefile take the following form:

```
target_1: part_1 part_2 . . .part_n
                instruction_1
                instruction_2
```

```
                .
                .
            instruction_m

target_2: part_1 part_2 . . .part_p
            instruction_1
                .
                .
```

The rules describe the set of parts that make up each target, and they provide a list of instructions for generating a target from its parts. Programmers are free to specify all targets explicitly, or they can take advantage of the built-in rules of `make`, designed to simplify the construction of programs that use standard file suffixes and generators.

There is a long list of complaints about `make`, and many programs have been created to extend or supplement it.[2] Here we examine two particular categories of problem.

The first category contains complaints about `make`'s implementation. `make`'s syntax is an example: most novices quickly run afoul of `make`'s distinction between tabs and space characters. Syntax is also an issue for `make` rules, which can be written only for files that are identifiable by their suffix. A third type of syntax problem is that different `make`s accept different syntaxes; a valid makefile for one system may be invalid for another.

Implementation problems also result from `make`'s use of operating system utilities to update its targets. This feature means that `make` is susceptible to operating system differences; for example, a `cd` in a DOS `make` remains in effect for all succeeding instructions, while in UNIX it is in effect only for the single instruction in which it appears. Problems also arise because a single-tasking operating system cannot spawn subshells for each instruction, and so makefiles that require a multitasking environment are not portable to single-tasking systems.

2 Somogyi suggests that the number of complaints is, paradoxically, strong evidence of the popularity and pervasiveness of `make` (Somogyi 87).

Program inconsistencies can arise because `make` is lax about representing dependencies. The dependency of a file on its `#included` files is typically not represented in a makefile. Programs such as `makedepend` and `mkmf` can be run to add such dependencies to a makefile, or to create a makefile that contains them. `make` also does not take into account the fact that the system is dependent on the makefile itself; if the makefile changes, the system will not be reconstructed, even though the system definition may have changed.

On the other hand, some dependencies are too strongly represented: programmers will often hardwire specific linkers, compilers, include file directories, and a host of other system-specific information into their makefiles. Current versions of `make` are initialized by configuration files that establish system-specific defaults, but programmers can and often do override these defaults, thus reducing the value of such parameterization. `imake` is a preprocessor that generates makefiles from a list of system characteristics and a high-level description of the program's structure. While `imake` helps reduce the tendency to embed system-specific dependencies, it is somewhat arcane, and consequently there is little enthusiasm for it.

Another type of inconsistency is that user-defined targets are transitively evaluated, while suffix-based dependencies are not. Consider the command `make file.a` with the following makefile:

```
.SUFFIXES .c .b .a

.c.b: ; @echo "making a .b file"
.b.a: ; @echo "making a .a file"
```

If file `file.b` does not exist, `make` will report that it does not know how to construct `file.a`, even though the rules specify how to construct `file.b` from `file.c`, and so `make` should be able to infer how to construct `file.b`, and then to construct `file.a`. Though `make`'s suffix rules form a directed acyclic graph, `make` evaluates each rule on the basis of the files present before it begins construction. Since some transitive manipulations are extremely common (for example, compiling and then linking, or generating intermediate parser code and then compiling), `make` has special-purpose rules for these situations. The ability to handle general transitive construction of

targets identified by suffixes is one of the features of `cake`, a variant of `make` (Somogyi 87).

`make` is sometimes criticized for being unable to handle parallel building; it serializes the construction of a program. `dmake` and GNU `make` are improved `make`s that support parallel builds. `dmake` also supports distributed builds.

A second category of problems results from `make`'s design and philosophy. One such problem is `make`'s reliance on files. `make` has no way to store information other than through the files it controls, and hence it cannot keep track of the evolution of a system or of its components except at the file level.[3] The reliance on files also encourages needless recompilation, since many spurious dependencies occur when files have access to more components than they actually use (Borison 89). Another problem with relying on the file system is that most file systems do not provide transaction support beyond that associated with reading and writing a single file; this is a problem in parallel builds if programs generate files with a fixed name (`lex` and `yacc`, for example, generate files with the fixed names `lex.yy.c`, `y.tab.c`, and `y.tab.h`).

Another design problem is the use of timestamps as an indication of existing structure. Using timestamps means that any update causes wholesale reconstruction of the entire system 'downstream' of the updated file, even if the change to the original file was as simple as adding a comment or even just reading it and writing it out. Timestamps are only one type of decision predicate that may be used to evaluate a file's consistency. Bitwise comparison may be a better decision predicate: if a fresh build of a file is bitwise identical to a previous version, then there need be no transitive propagation of the update (Borison 89). With more sophisticated notions of equivalence, the consistency of the temporal order with the build order is not a necessary condition for building. Consistency with the temporal order turns out not to be a sufficient condition, either; if we retrieve an earlier version of a part file (along with its original timestamp), it will predate parts that are dependent on it, but these

3 This was initially thought to be an advantage, since it meant that `make` could be interrupted and restarted (Feldman 79).

parts almost certainly need to be rebuilt (Fowler 90).

In general, it is not possible to determine the minimal update operations without more information about the operations we are allowed to perform on the parts, and hence on the temporal order relation. If we are allowed, for example, to simply change the timestamps of offending parts, then contradictions can be repaired without any transitive effect (although in this case the resulting program may not do what we expect). In the usual situation, the only operation we have is to update the file, and hence make it more recent than any other file; in effect, to move the file to the top of the temporal order. But it is possible to imagine other operations (such as retrieving intermediate versions of a file) that would move the file to the middle of the temporal order, and perhaps result in little or no recompilation.

### Discussion

The partial order model does not solve every problem of `make`; indeed, it does not even address the implementation problems. A poor implementation of partial orders might easily have more syntactic problems than `make`. What the partial order model does do is create a clear separation between the logical modelling of a program, and the details of the systems that implement that model. This is the basic advantage of any kind of data model; Waters achieves similar advantages by separating the task and structural information in makefiles (Waters 89).

The literature on `make` and its imitators describes these programs by enumerating their features. Instead of presenting a model of program design, these programs are explained by means of examples of makefiles and details about their syntax, features, and peculiarities. Similarly, makefiles themselves are programs in which 'the architecture of the system being described is typically obscured by a blizzard of detailed, task-specific information' (Waters 89). If program building were described in terms of partial orders (or of any data model), it would permit a new kind of dialogue. Instead of describing an implementation, documentation would describe a model, and present the implementation as one way to achieve the model. Instead of

constructing a highly implementation-specific description the architecture would be expressed in a logical form that could be mapped onto a variety of implementations. Just as it is not appropriate to explain relational databases by starting with the syntax and features of particular implementations of SQL, it is probably not appropriate to describe software by the program used to construct it, nor is it appropriate to describe software tools through exegesis of their implementations.

With the exception of `BUILD` (Waters 89) and some proprietary systems, `make` and its imitators treat the program building process as an end in itself; one constructs a makefile solely in order to build an executable file. By showing that program construction can be achieved algebraically, the partial order model introduces the opportunity of treating the program structure as a database that can be queried, updated, and used in more complicated forms of processing. The larger and more modular a software system, the more necessary it becomes to treat the structure of the components as (dynamic) data about the system.

The abstraction of the build process is an important step in generalizing the consistency rule. The partial order model permits the use of any decision predicate in place of comparison of build to temporal dependency. Consider how the partial order model could support the use of bitwise equivalence to reduce the number of transitively-induced updates. Recall that the query for rebuilding a system was:

$$\uparrow (\mathbf{S_F},\ (\mathbf{S_T} \sim \mathbf{S_F}))$$

This query was appropriate under the conditions that (i) only $\mathbf{S_T}$ could be updated and (ii) each update made the affected files the most recent; that is, for each $a$ in $\mathbf{S_T} \sim \mathbf{S_F}$, the update changed the timestamp for $a$ such that $\forall b \in \mathbf{S_T},\ a > b$. Under bitwise comparison, we relax (i) and permit changes to $\mathbf{S_F}$, as well as $\mathbf{S_T}$. Suppose that $a \in \mathbf{S_T} \sim \mathbf{S_F}$, when rebuilt, is $a'$, and that $a' = a$; that is, they are bitwise indistinguishable. Then we need not consider transitive updates based on $a$, since its recompilation returned the identical file. We can register this fact by removing all edges $(b, a)$ from $\mathbf{S_F}$: because we have shown that the recompilation of $a$ is not different from $a$, we have in effect shown that (at least for this build)

$a$ does not depend on its ancestors $b$.[4] We can now recompute $\uparrow (\mathbf{S_F}, (\mathbf{S_T} \sim \mathbf{S_F}))$ and proceed with the next contradiction (of course, there are iterative solutions to this problem that would not involve recomputation of the original query). Conventional `makes` cannot be easily extended in this way, because the use of build and temporal dependencies is hardwired into their architecture. Moreover, *only* these dependencies are present in conventional `makes`; the partial order model permits different dependencies, and also more than two sets of dependencies, to be combined in a consistency condition.

The partial order model is the natural generalization of the structures used to represent software. If we accept that neither the temporal order nor the parts dependency can be cyclic, and that both are transitive, then a partial order is the most restricted model that satisfies both of these conditions. The partial order can easily accomodate partially ordered temporal structures, and in fact does so with the existing query.

Partial orders by their very nature include the information required to do parallel and distributed builds; by definition, all incomparable files can be generated in parallel. The result of the build query is also a partial order, one that specifies the maximum level of parallelism that is possible while still achieving consistency. If it is useful to build some components in parallel, then the rules for building those components can themselves be described with a partial order.

Before leaving the problem of constructing programs from their parts, it is worthwhile to reconsider the reason for decomposing a program into parts. Modularization is done for many reasons:

- to encourage short, readable routines
- to facilitate information hiding and code reuse
- to express interfaces or other high-level abstractions
- to control access
- to allow partial compilation

4 What we have actually shown is that the ancestors $b$ do not cause any *change* in $a$, and so the formal dependency of $a$ on $b$ is, in this instance, moot.

Modularization has many benefits. However, one of its costs is the potential for program inconsistency. `make` and its imitators were developed to address this problem. The problem of consistency in updates has also been investigated in the relational database model, resulting in the theory of dependencies and normal forms. It is interesting to consider the implications of this theory for software.

Consider modelling the parts of a program as values in a relational database. Each file might be represented by an atomic value of an attribute. There could be several domains, one for each type of file; for example, a `.c` domain, a `.h` domain, a `.o` domain, and so on. The build dependency structure represents the instance, and from that instance we can distill functional dependencies; for example, every `.c` file determines exactly one `.o` file, giving the dependency `.c` $\rightarrow$ `.o`[5] In normalizing such a system we would attempt to create schemas that contain only those attributes that are functionally dependent on the schema's key, so as to be able to guarantee the consistency of updates.

The preceding example is not introduced to suggest that program construction reduces directly to a normalization problem. Instead, we simply illustrate the fact that in program construction, atomic updates are performed on individual files, without taking into account the dependencies of other parts of the program on those files. In effect, the software is stored in relations consisting of a single attribute, and dependencies are not observed at update time. `make` and its offshoots are fundamentally a kind of transaction system that attempts to provide consistency after the fact, by means of automatic tools for updating data, based on the dependencies expressed in the makefile. The partial order model is a natural way to express and manage those dependencies.

5 Typically, there is a bijection between `.c` and `.o` files, but this need not be the case. It need not be true that one `.c` file alone determines a `.o` file, since several `.c` files may be compiled into a single `.o` file. But it is always the case that, for any given program, each `.c` file can participate in the production of only one `.o` file.

Program building is the problem of ensuring the consistency of a set of parts. Version control is the problem of managing the evolution of a collection of parts. Programmers who work on large, long-term projects often want to represent the process of deriving one part from another, and they also want to represent the process of constructing parts that come in various incompatible alternatives, such as an *optimized* part and a *debugging* part. We will refer to the steps in the evolution of a part as its *versions*, and alternatives that exist at any one time as its *variants*.

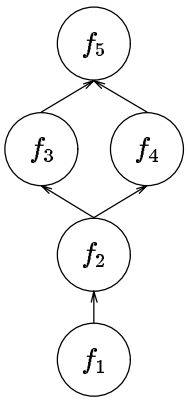
Given a collection of versions and variants, how can we manage them? Partial order concepts can be applied to the problem of version control, because versions and variants involve ordering relationships. Versions are always comparable to one another, while variants are always independent of one another; in partial order terms, versions correspond to chains, while variants correspond to antichains.[6] Let us consider the use of partial orders for version control by studying a small example.

The simplest software system requiring version control is a single file that is changed over time. Figure 26 shows a partial order **H** that describes a *history* for a hypothetical file. $f_1$ is the original file and $f_2$ is a later version. $f_3$ and $f_4$ are versions of $f_2$, and variants of each other. Finally, $f_5$ is a version of both $f_3$ and $f_4$.

File histories are partially ordered. They are transitive: $f_3$ is derived from $f_1$, because $f_3$ is derived from $f_2$ and $f_2$ is derived from $f_1$. They are also acyclic: it is not possible for a file to be derived from itself.[7]

6 This choice of definition is not the one used in the software engineering community; indeed, the community appears to have given up on achieving consensus on any useful distinction. The choice between 'version' and 'variant' is, for the software engineering community, much the same as the choice between 'attribute' and 'relationship' for the database community (Kent 79). We venture to use the terms more precisely here simply because we can define them in terms of a significant mathematical property, rather than by appeal to intuition.

7 It is possible for a derived file to have identical content to an earlier version, but the two files are distinguishable by their derivation history.




$\dot{H}$

Figure 26: File history.

Even for the simple example of a single file, there are useful partial order queries:

1. *What are the most recent version(s) of the file*?

   $$\top\mathbf{H}$$

   Note that the result can be more than one file.

2. *Describe the changes in the file since version* $b$.

   $$\uparrow(\mathbf{H}, b)$$

3. *When was the variable* `size` *first used in this file?*

   $$\bot\sigma(\mathbf{H}, variable('\texttt{size}'))$$

   This query assumes that there is a predicate that tests for the presence of variables of a given name. This may be approximated by a string searching program, or there may be some other database that can be consulted.

Most interesting software systems consist of more than one file. A simple system with three independent files, for example, can be described by the independent file histories $\mathbf{H_1}$, $\mathbf{H_2}$, and $\mathbf{H_3}$, as

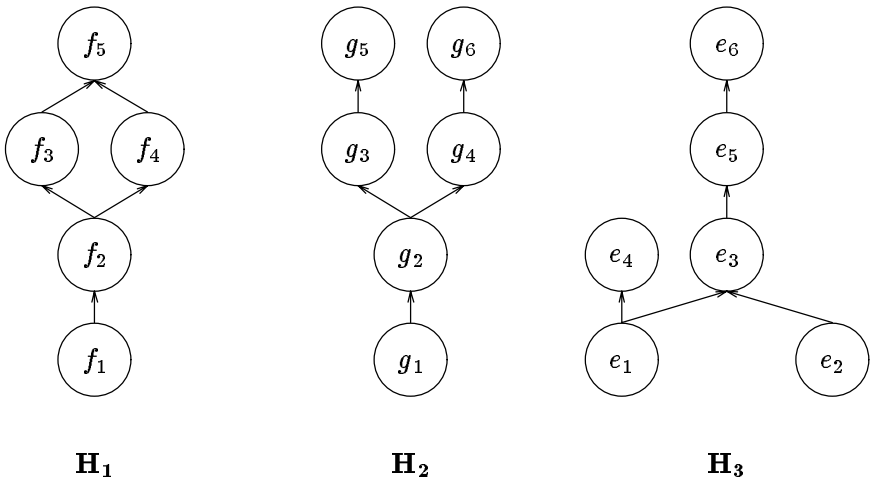



Figure 27: Independent histories.

shown in Figure 27. If the files are completely independent, then the system is simply the disjoint union of the three file histories. Disjoint union preserves the independence of the files betweeen histories; for example, $g_5$ is incomparable to $e_6$ (and both files are in $\top(\mathbf{H_1} \cup \mathbf{H_2} \cup \mathbf{H_3})$).

In most cases files are not completely independent, and systems are composed of collections of files structured in various ways. One such ordering is the build dependency described in the previous section on program building. Adding a build dependency to our example might give the organization shown in Figure 28. Here there is a simple ordering of file histories such that $\mathbf{H_1}$ and $\mathbf{H_3}$ are incomparable, and both are greater than $\mathbf{H_2}$. This ordering of three file histories is expressed by the partial order $\mathbf{D}$. We can find the relationships between any two files in any of the file histories if we compute the lexicographical sum of the file histories, using $\mathbf{D}$ as the expression order:

$$\mathbf{V} = \sum_{\mathbf{H_i} \in \mathcal{H}}^{\mathbf{D}, f} \mathbf{H_i}$$

Under union, $g_5$ and $e_6$ were incomparable; under this lexicographical sum, they are comparable.[8]

8 It is worthwhile recalling that disjoint union is a special case of lexicographical sum, with an empty partial order as the expression order.

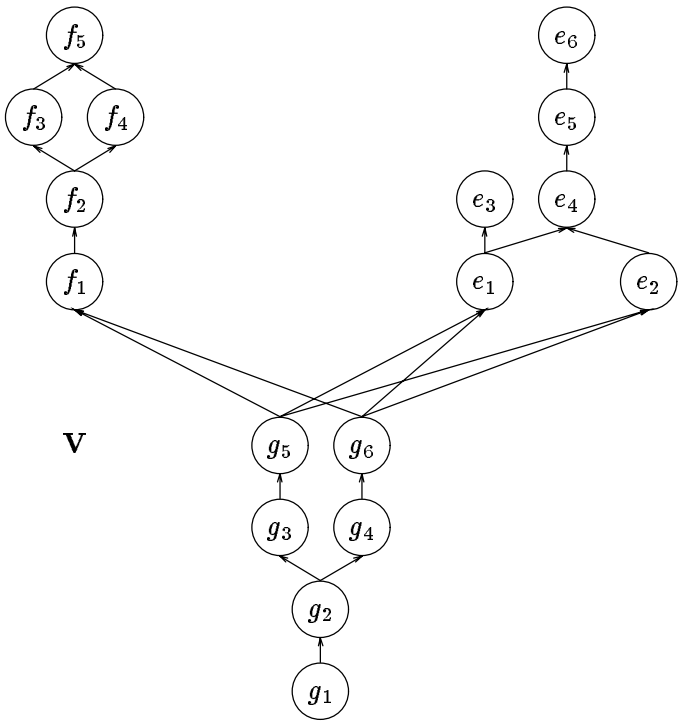



Figure 28: Ordered file histories.

The lexicographical sum of file histories organizes complete file histories. This is appropriate only if we are indifferent about which file is chosen from a history. Such a system is conceivable if each of the files presents exactly the same interface to the whole system, and if we have no other reason to want to select particular versions or variants in the history of any given file. In general, however, versions and variants are constructed to be used in a synchronized way: we get a fast system, for example, by using the fast variants of the parts (if any; otherwise we use the standard parts) that comprise that system. Thus, we need not one, but several build dependencies, one for each version of the system that we intend to produce. We will refer to these as *version orders*, and denote them by $\mathbf{V_i}$. The elements of each $\mathbf{V_i}$ are suborders of the file histories. In Figure 29, the individual file histories have been presented to show the correlation of files in each file history with versions of the system. The version orders are not shown explicitly, but they are the 'planes'

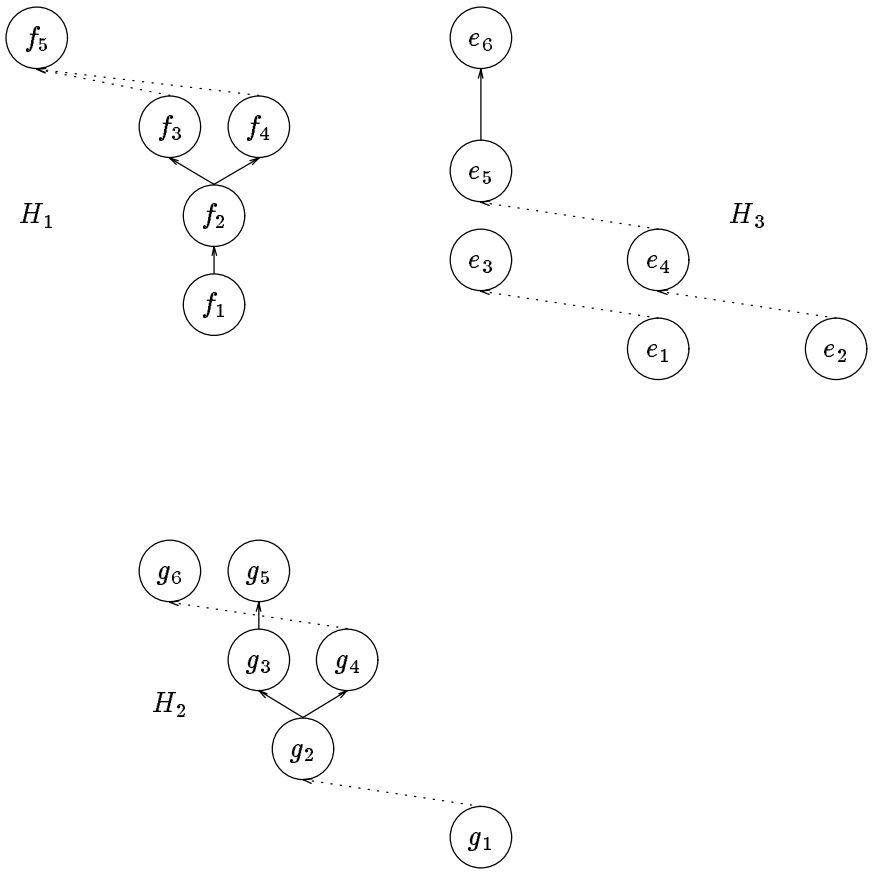



Figure 29: File histories organized by version orders.

that would intersect the file histories in this figure, if it is viewed as an isometric projection. In each case, we have the same build dependency $\mathbf{D}$. Thus, we have the following three version orders:

$$\begin{aligned}
\mathbf{V_1} &= \sum^{\mathbf{D}} \sigma(\mathbf{H_1}, \{f_1, f_2, f_3, f_4\}), \sigma(\mathbf{H_3}, \{e_1, e_2\}), \sigma(\mathbf{H_2}, \{g_1\}) \\
\mathbf{V_2} &= \sum^{\mathbf{D}} \emptyset, \sigma(\mathbf{H_3}, \{e_3, e_4\}), \sigma(\mathbf{H_2}, \{g_2, g_3, g_4, g_5\}) \\
\mathbf{V_3} &= \sum^{\mathbf{D}} \sigma(\mathbf{H_1}, \{f_5\}), \sigma(\mathbf{H_3}, \{e_5, e_6\}), \sigma(\mathbf{H_2}, \{g_6\})
\end{aligned}$$

Each $\mathbf{V_i}$ is a lexicographical sum, where $\mathbf{D}$ is the expression order, and the factors are suborders of the file histories (we use the selection operator $\sigma$ to select suborders that contain the designated subsets

of each of $\mathbf{H_1}$, $\mathbf{H_1}$, and $\mathbf{H_3}$.)

An instance of a system is not immediately given by a version order. Where a suborder of a file history (that is, a factor in the lexicographical sum) consists of more than one file, we need to choose exactly one to participate in the system. In most cases we want the most recent file; if a factor has a maximum element, then this is the most recent, and hence the one we choose. If a factor has several maximal elements, then we must choose one of the variants according to some other criterion. We will assume that such rules have been specified, and that for each $\mathbf{V}$ we can identify exactly one file in each participating $\mathbf{H_k}$. Thus, a system consistent with $\mathbf{V_1}$ comprises either $f_3$ or $f_4$, either $e_1$ or $e_2$, and $g_1$.

A version of the system is not required to include a file from each of the file histories; $\mathbf{V_2}$ in the above example uses no file from $\mathbf{H_1}$. This reflects the fact that as a system develops, new capabilities will be added, requiring files that were not previously used, and old capabilities will be dropped or folded into existing capabilities, and thus the system may not require files that were previously used. Individual $\mathbf{V_i}$ are thus not necessarily isomorphic to $\mathbf{D}$, though they usually have a significant overlap.

The collection of $\mathbf{V_i}$ can itself be ordered. We want an ordering that preserves our notion of system evolution, because we want to be able to answer questions such as, 'which is the most recent version of the system?' Consider two version orders $\mathbf{V_i}$ and $\mathbf{V_j}$, where we want $\mathbf{V_i} < \mathbf{V_j}$; that is, we think of $\mathbf{V_i}$ as being an 'earlier' version of the system. The elements of $\mathbf{V_i}$ and $\mathbf{V_j}$ are suborders of complete file histories $\mathbf{H_k}$. In the general case, $\mathbf{V_i}$ and $\mathbf{V_j}$ will have a non-trivial overlap—that is, they will have some files in common—but some files will be unique to $\mathbf{V_i}$ or $\mathbf{V_j}$. There is little restriction on this overlap; $\mathbf{V_j}$ might have a completely different set of files from $\mathbf{V_i}$, or it it might differ in only one file. A natural condition to place on the overlap arises from the histories $\mathbf{H_k}$. This is the condition of monotonicity:

$$\mathbf{V_i} \leq \mathbf{V_j} \text{ iff } \forall x, y \in H_k,\ (x \in H_k \cap \mathbf{V_i}) \ \leq \ (y \in H_k \cap \mathbf{V_j})$$

This condition says that if two versions of a system use files from the same file history, then it cannot be the case that the later

version of the system uses an earlier version of the file. Versioning is thus a monotonically increasing function on file histories. Note that the restriction permits a version to use the same files as a previous version. Versioning is monotonically increasing, but not necessarily strictly increasing.

A variety of queries are possible with a monotonic description of the versions of a system:

1. *What versions of the system use file $f_i$?*

$$\sigma(\mathbf{V},\ \sigma(\mathbf{V_k}, f_i))$$

   File $f_i$ may be a specific file, or it may be a predicate on files; for example, we can select versions of the system that use the 'fast' files, or the 'debugged' files, or 'files owned by Charlie'. The nested select gives the correct answer in this case, because if there is no $f_i$ in $\mathbf{V_k}$, then the result of the select is an empty order.

2. *Find the first version of the system that uses no components that are in version $\mathbf{V_i}$.*

$$\perp\sigma(\mathbf{V},\ \beta - \beta\mathbf{V_i} = \emptyset)$$

   Note that the first $\beta$ takes as argument each element of $\mathbf{V}$ (that is, $\beta$ is instantiated with individual $\mathbf{V_k}$).

3. *How does version $\mathbf{V_j}$ differ from version $\mathbf{V_i}$?* Users may have different queries in mind for this problem. One possible query simply shows which files belong to $\mathbf{V_j}$ and not $\mathbf{V_i}$:

$$\beta\mathbf{V_j} - \beta\mathbf{V_i}$$

   and of course, the converse is also interesting. These results do not show any changes in structure, however. A more interesting query shows the parts of a given file history structure that have changed between $\mathbf{V_i}$ and $\mathbf{V_j}$:

$$\sigma(\mathbf{H_k},\ \beta\mathbf{H_k} \cap ((\beta\mathbf{V_j} - \beta\mathbf{V_i}) \cup (\beta\mathbf{V_i} - \beta\mathbf{V_j})))$$

The above query fragment finds the suborder of $\mathbf{H_k}$ in which $\mathbf{V_j}$ differs from $\mathbf{V_i}$. We can use this query to determine the changes in a given file history, and if we have a partial order that contains all file histories (that is, a partial order $\mathbf{Z}$ that is the set of file histories), then

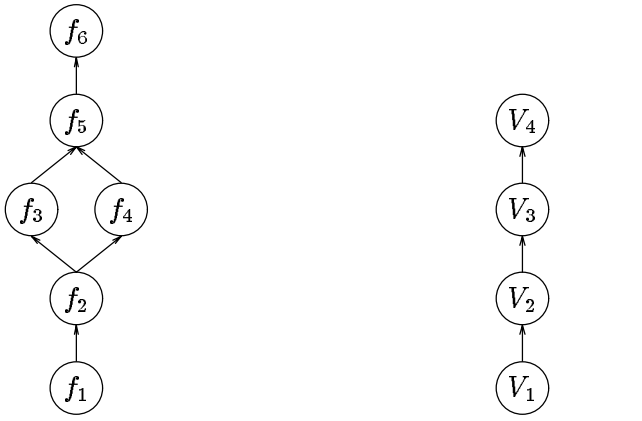
$$\sigma(\mathbf{Z},\ (\beta\mathbf{V_j} - \beta\mathbf{V_i}) \cup (\beta\mathbf{V_i} - \beta\mathbf{V_j}))$$

will return those file histories, reduced to the parts that have changed between versions $\mathbf{V_j}$ and $\mathbf{V_i}$.

Monotonicity may seem too stringent a condition. Is it never the case that one would want to use $f_3$ in $\mathbf{V_4}$, for example, rather than the most current version $f_5$ used in $\mathbf{V_3}$? What if, for example, $f_5$ contains an untested feature that appears to be buggy, while $f_3$ is known to be robust? There are two approaches to handling this situation while still preserving monotonicity, presented in Figure 30. In approach (a), one creates file $f_6$ and gives it the same content as file $f_3$, adding it to the file history such that $f_5 < f_6$. This will obtain the semantics of the $f_3$ element, while still preserving the monotonicity of $\mathbf{V_3} < \mathbf{V_4}$. While $f_3$ has the same content as $f_6$, they differ in their evolution history; $f_6$ was obtained by considering and (temporarily) rejecting $f_5$, while $f_3$ was created without knowledge of $f_5$. The rationale for this approach is that it preserves the information about how $f_6$ was derived.

The second approach to preserving monotonicity is to create a variant of some older version order in such a way that we can preserve monotonicity. So instead of generating $\mathbf{V_4}$ as a version of $\mathbf{V_3}$, we create $\mathbf{V_4}$ as a version of $\mathbf{V_1}$, where $\mathbf{V_4}$ is the same as $\mathbf{V_3}$ except that it contains $f_3$. That is, instead of creating a situation in which montonicity will require that we reject the attempt to reuse $f_3$, we create $\mathbf{V_4}$ as a version of $\mathbf{V_1}$, and continue on from there. Then $\mathbf{V_4}$ will be incomparable to $\mathbf{V_2}$ and $\mathbf{V_3}$. The rationale for this approach is that by returning to a previous version of some file, one is really starting a new chain of development, and terminating the current chain.

What are the advantages of monotonicity?




$f_3 = f_6$
(a)


$f_3 \in V_4$, $V_4$ starts a new chain
(b)

Figure 30: Two methods for preserving monotonicity.

- Given a set of points in some file histories, we can always determine the earliest and latest version containing these points (in general, a version may consist of a set of variants).

- It is possible to order an unordered set of versions possessing only knowledge of the file histories. Thus, ordering information need not be stored for versions.

- It is possible to derive intermediate versions; that is, one can specify the 'slack' in the file histories that permits the creation of version $\mathbf{V_j}$, such that $\mathbf{V_i} \leq \mathbf{V_j} \leq \mathbf{V_k}$, where previously only $\mathbf{V_i}$ and $\mathbf{V_k}$ existed. This type of query can be useful if one wants to include some of the changes since a previous version, but not all of them. Determining the 'slack' will let one know if it is possible to synthesize an intermediate version, and what its contents would be.[9]

These advantages are lost if there is no necessary relationship between file histories and versions.

To this point, our example of version control has been explained in terms of program files. The partial order model does not require files or software, however; we can contemplate managing the histories of any object, including documentation, specifications, connection channels, processes, or other resources. Such object histories can be structured in version and system orders analogously to how we have suggested managing file versions. Moreover, since version and system orders are themselves objects, they can also be subject to history control, version control, and system control. Thus the partial order model for version control can be applied to itself.

We recap the essential elements of a partial order approach to version control:

**An object history H** is an ordering of a set of versions and variants of a single object. The order is given by the object's evolution.

9 As mentioned in the section on building programs, using an intermediate version of a file is a kind of update to the system that enables us to avoid needless recompilation.

**A version order V** is an ordering of a set of object histories. The order is a dependency structure of specific elements in the object histories, and is expressed as a lexicographical sum.

**A system order S** is an ordering of a set of version orders. The order is given by the principle of monotonicity, applied to the underlying object histories.

**A system O** is a version order **V** restricted to a single object from each factor in the lexicographical sum. The rule for restriction may combine partial order operators (for example, 'choose the maximum element') and domain-specific operators (for example, 'choose only from elements that have been checked by two or more programmers').

We next briefly consider conventional approaches to managing versions and variants of programs.

### RCS and other systems

The most widely used program for version control is Revision Control System, or RCS (Tichy 82) (Tichy 85). RCS is a tool for managing file histories. An RCS-controlled file can store both versions and variants, which can be checked out to a user's file space, edited, and then checked in again. RCS maintains the latest version of the file and keeps older versions with backwards file differences. A file can be checked out by multiple programmers, but only the initial checkout can be checked in again without coordination between the programmers. RCS can assist in coordination by doing simple merges of multiple edits to a file.

RCS does not manage modules or projects. Whatever can be represented in a file can be controlled by RCS, however, and so many programming teams use RCS to control directories or text-file descriptions of modules. Like `make`, RCS is popular because it is a simple tool that does a useful job, and because it can be applied to many problems.

From a database standpoint, RCS is a tool for providing data storage, transaction control, and some normalization (because it stores differences between files). It has no model; if a model is

desired, it must be imposed by the programming team. Thus, RCS is essentially a shared data structure.

The contemporary trend is towards version control systems that handle many more domain-specific tasks. Generally these are large, monolithic systems best suited to large programming teams. IBM's CMVC is one such environment. CMVC provides RCS's capabilities, and adds more features:

- more sophisticated control over users, including control over which hosts can access CMVC
- explicit control of larger units of software: a project is known as a *family*, and is broken down into a hierarchy of *components*; a *release* is a variant; a *level* is a collection of files that crosses releases (essentially a version)
- defect management, including predefined schemas for solving defects
- graphical tools for displaying file histories and project structure

CMVC and its ilk add project management structure to RCS, but do not essentially change its status as a shared data structure.

A novel form of version control was used in the Thoth and Waterloo Port operating systems (Cargill 79). In most version control systems, variants and versions of files are stored with special software, and are extracted into a conventional file system for editing and compilation. In Cargill's approach, variants and versions are stored in a conventional file system, and special software is used to extract compilable programs. Part of the Thoth file system is shown in Figure 31. One novel aspect of the Thoth file system is that it allows any file to be both a directory and a file; that is, to have both content and substructure. This feature is exploited in Thoth to express versions of a file as subfiles.

The file hierarchy divides the operating system SYS into a collection of nested modules; the figure shows the file system module FSYS and the KERNEL. Each of these modules contains several functions; for example, `.Find_son` and `.Mark`. If there are differing

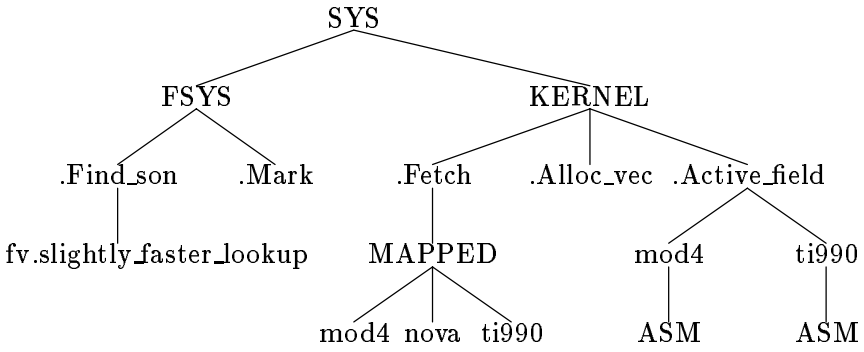



Figure 31: Part of the Thoth filesystem.

versions of these files, they are stored as subfiles; for example, the file `.Find_son` has the version `fv.slightly_faster_lookup`. For variants that are architecture-dependent, one stores the function in a file with the name of the architecture; in the example, we see `mod4`, `nova`, and `ti990`. Assembler variants of functions are kept in subsidiary files named `ASM`. Finally, variants that differ in functionality are stored in subdirectories that have names indicating the difference in functionality; a mapped kernel, for example, uses the functions in the subdirectory MAPPED.

Cargill's approach has two interesting characteristics. First, versions and variants can be examined and manipulated with standard file system utilities; a special-purpose program is not needed. This is because no special database or difference file is used to store the code. Consequently, there is no checking-in, checking-out, or merging of files (but on the other hand, no transaction control in the presence of multiple updaters, either). The disadvantage is that a special program must be used to collect all relevant modules for compilation, and that there is more redundancy than would be the case if edit differences were employed.

The Thoth structure can also be used to support a type of object-oriented inheritance. The software hierarchy defines 'versions' of modules that can be overridden by lower-level versions, just as a derived object may override functions that it inherits from a base object. These overridings are all compile-time bindings, however; Thoth does not support run-time polymorphism.

> Common data structure descriptions and constants are entered just once and are shared by relevant programs (Feldman 79).
>
> We need to economize by building system families whose members share common parts (Tichy 82).
>
> Programming in an object-oriented language largely revolves around specifying sharing relationships (Ungar 91).

Cargill's approach demonstrates the close relationship between version control and object-oriented design. This relationship is not accidental: beneath both of these activities lies the fundamental activity of sharing.

Versions and variants of a program are related by common purpose or essence, and often share many parts. The purpose of a version control system is to manage both the shared and the unique parts, and combine them to make a working system. The shared parts of a system are not only recognized, they are often explicitly stored: RCS and other systems store only the differences between the current version of a file and any previous versions.

Object-oriented systems are essentially techniques for controlling redundancy. One of the key differences between object-oriented and standard approaches is the emphasis on redundancy control, and the use of inheritance to express sharing relationships.

The partial order model is uniquely suited for both version control and object-oriented systems, because there is a deep connection between sharing and partial orders (Pratt 76). Sharing is one interpretation of incomparability in an ordering relationship. When we say that $c$ is shared by $a$ and $b$, it means a three-way relationship in which $a$ and $b$ are unique, but have common content; in other words,

a containment relationship that involves some incomparability.

How are version control and program building related? We noted previously that one problem with `make` was its reliance on file systems that provide, at best, primitive support for transactions. A version control system is one way to provide better transaction support. This has led some researchers to suggest that rather than add version support to `make`, we should simply use a versioning file system (Fowler 85). This solution has the undoubted advantage of simultaneously addressing the need of many other file-based utilities for version control (Garfinkel *et al.* 94).

Simply providing a versioning file system, however, fails to exploit the common structure underlying both program building and version control. Just as sharing is basic to both version control and object-oriented structures, sharing is an essential aspect of program building. `make` is a program for expressing the dependencies between code modules, dependencies that often involve the sharing of one module by several others. The core activity of `make` is to ensure that all automatically-derived modules employ the same version of a module that they share. If code modules were independent of each other, there would be no need for a program builder.

`make` and RCS are both designed to derive objects from shared components. `make` invokes compilers, linkers, and other programs to derive new objects. It may not be as readily apparent, but RCS is also a derivation tool: RCS invokes programs to derive older versions of a given file, because it does not store each version in its complete form, but rather stores backward differences for each version. In effect, RCS stores shared components, and uses its backward differences to derive complete files. RCS thus has a limited ability to generate derived objects based on shared components. In addition, while `make` is expressly designed to combine objects, RCS cannot in general derive a single object from several others, because there is no general algorithm for merging a collection of updates that were separately applied to an object.[10] On the other hand, RCS retains and can access a set of derivations that constitute a history, while

10 RCS provides a program called `rcsmerge`, but this is not guaranteed to produce the correct results; it is only a starting point for further manual editing.

`make` cannot. This is similar to the distinction between a relational database, which maintains only a current snapshot of the data, and a temporal database, which retains a history of the data—in effect, a sequence of snapshots.

The partial order model makes it possible to unify the two processes of program building and version control. Recall that building a system was reduced to testing for contradictions between the dependency structure and the temporal order. This same query can be used even if a partial order consisting of all versions and file histories is employed, because this is still a test of the contradictions between a (much larger) dependency partial order and a (much larger) temporal order. The make could be restricted to only one version, or alternatively, multiple versions of the system can be built in parallel, each with only the necessary components being updated.

The need to derive and manage consistent components in software goes beyond the problems of `make` and RCS. Dependency structures are also present in such problems as installing and deinstalling software. To install a large package, one must add all the files that it depends on, and that are not already present. Assume that we have a package $\mathbf{P}$ and an existing system $\mathbf{F}$. The structure we need to install is:

$$\sigma(\mathbf{P}, P - F \cap P)$$

Note that the result is not a set, but a partially ordered object, and that this is necessary, since one must install directories before one installs the files that they contain. The same query can be used to determine the files to deinstall a package (one might take the dual of the resulting partial order, so that files would be removed before their directories are removed).

Another example of the need for dependency maintenance occurs in mainframe environments. Large applications often require that many jobs be done in a certain sequence, possibly involving some concurrency. Mainframe systems thus include facilities for scheduling a set of tasks according to a defined precedence structure. This problem is similar to the program building problem: instead of building an executable, however, task scheduling builds a result in the state of some system through the execution of tasks. These

task structures need to be tracked, maintained, queried, logged, and otherwise managed in much the same way as do the components of a software system. The partial order model is a natural way to model such problems.

# 5

# Text

Text is more common than any other type of data, but we still lack a good data model for it (Tompa 89) (de Rose *et al.* 90) (Tompa and Raymond 91) (Raymond *et al.* 96). Text gains its expressive power from two fundamental inventions: the alphabet, and the use of positional notation (Goodman 76) (Harris 86). The alphabet is a small set of atomic characters that can be composed into a very large number of structures, through the use of positional notation. Positional notation is an ordering of characters; from positional notation we get words, sentences, chapters, headings, footnotes, marginalia, parallel translations, tables, page furniture, and most of the other structures in text. Computerized text adds the possibility of dynamic orders; that is, the ability for the user to change the order of the components (Raymond and Tompa 88) (Raymond 92) (Raymond 93). Whether dynamic or static, any system that relies on positional notation clearly has order as an integral aspect. Thus, it seems clear that order should take a primary role in any model for text.

## ORDERS IN TEXT

There are many different ordering relationships in text. We shall try to build up a model for text by exploring each of these relationships in turn.

The most important class of ordering relationships in text is *containment*. Containment expresses the fact that some parts of a document are components of other parts. The most commonly recognized example of containment is the so-called 'logical' structure of documents; for example, a research article is a collection of sections, each section is a collection of paragraphs, each paragraph is

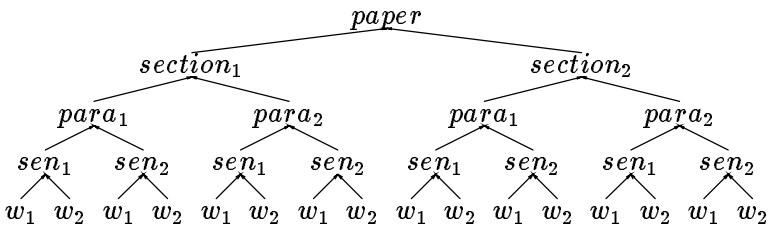



Figure 32: **L**

a collection of sentences, and each sentence is a collection of words, as in Figure 32. The 'logical' structure is usually assumed to be the most important structure of a document (sometimes declared as 'the' structure of a document). We will denote this structure by **L**. **L** is hierarchical, and thus partially ordered.

The elements in Figure 32 are numbered because documents generally consist of ordered parts: thus, a sentence is not just a collection of words, it is a sequence of words. For the moment, we will ignore these sequences, and treat all collections as sets.

The relationships presented in Figure 32 are only one of several forms of containment. Figure 32 is not itself the document; rather, it is one representation of the document, a highly redundant one. In a document of reasonable size, words and phrases are certainly repeated many times, and if the document has 'boilerplate' text, then even sentences or sections may be repeated. Normally we think of these redundant values as equivalent.[1] This equivalence, or alternatively, this redundancy in data, can and should be expressed in the data model, since that is the only way that it can be abstractly managed.

Figure 33 shows **L** along with a new partial order, **R**, which captures the equivalence of words. The new structure distinguishes

1 Equivalence is a semantic, and therefore subjective, notion. The claim of equivalence made here is based on the fact that updates to the text are usually applied consistently to all replicated values. When we change the spelling of 'color' to 'colour', for example, we expect it to be done consistently for every instance of the string 'color' (with the exception of documents that, for example, discuss the use of both British and American spellings of colour!). This consistency requirement is a statement of the equivalence of the instances of the string.

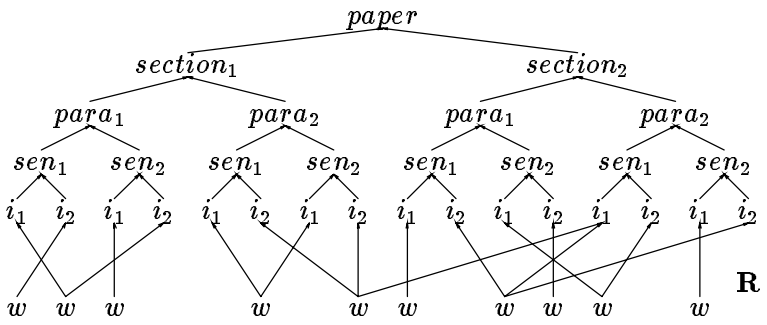



Figure 33: **C**

between words and uses of words: we call the latter *instances*. Words are values, and instances are occurrences of those values in some ordered structure. **L** organizes instances, and **R** organizes words. If $a < b$ in **R**, this implies that $a$ contains $b$; thus, the word $w$ contains its instances $i_j$. **L** and **R** can be viewed as two partial orders that meet at their 'fringes'. We refer to the combination of **L** and **R** as **C**.[2]

**R** is clearly a partial order. It is antisymmetric, because otherwise we may have an instance that contains a word, which is not allowed. Since it is only a two-level structure, transitivity is trivially true. **C** is also a partial order; since neither **L** nor **R** orders elements within the fringe (recall that we are ignoring the sequence order for the moment), and since the two structures have only the fringe elements in common, then it cannot be the case that the interaction of the two orders results in some non-order relationship.

There are several points to note about **C**. Whereas the **L** structure has a maximum, **R** has many maxima. The maxima of **R** are the set of instances in the document, while the maximum of **L** is the root of the logical structure of the document, and $\perp$**C** returns the set of unique words. **L** could have a set of maxima, if the partial order described a collection of documents, rather than only one.

Because it separates the notion of 'word' from that of 'instance',

2 The 'combination' of these two partial orders is not a union, since the orders are not defined on disjoint sets. We have no operator that produces this structure; it is simply a structure consisting of two partial orders that share some elements.

**R** is capable of describing situations in which words have a variety of non-identical instances. This permits us to capture surface differences such as the following:

- capitalization ('Army', 'army')
- abbreviation ('Dr.', 'M.D.')
- spelling variation ('draft', 'draught')
- ligatures (hemorrhage, hæmorrhage)
- hyphenation (boiler room, boiler-room)
- line breaks (hyster-esis)
- special symbols ('&' for 'and', '$' for 'dollars')
- digits and numbers ('2' and 'two')

We need not have the same rules about variations through the whole text; if some instances of *Dr.* are not instances of *M.D.*, that fact can be modelled. This should be contrasted with text searching programs which permit the user to specify variations that must be applied globally (or as is more typical, text searching programs which simply enforce their own idea of what constitutes a variation, including truncation and stop word elimination).

The choice of what constitutes a surface variation and what constitutes an actual difference in word is, of course, a choice to be made by the modeller, and not the model. The modeller should be careful not to extend the **R** structure so far that it actually captures synonyms. It is probably wise to put synonym information in a different structure, and use **R** to describe only those clusters in which forms are interchangeable. The semantic synonym structure can be built on top of the variations described in **R** (if 'physician' is considered synonymous with 'Doctor', then it will also be synonymous with 'M.D.' and 'Dr.').

Figure 33 shows a structure in which only shared words are represented, but it is also possible to express the sharing of individual characters, of sentences, or of other fragments. To represent

the sharing of individual characters, we would extend the structure so that $\top\mathbf{R}$ would consist of the set of atomic characters used in the text (perhaps the letters a-z, A-Z, the numbers 0-9, and some punctuation characters). The next level down in the redundancy structure (that is, $\top \mp \mathbf{R}$) could be words, space-delimited tokens, syllables, or n-grams. The next level down (that is, $\top \mp_2 \mathbf{R}$) could be phrases, sentences, sets of words, or other distinct fragments. This process can be continued through the **R** structure. The choice of what to model in **R** depends partly on the types of queries we want to support, and partly on the constraints we need to enforce during update. This generalization of **R** will result in an organization of overlapping sets, while **L** is an organization of nested multisets.[3]

The various examples of **R** have an interesting property: they are abstractions of *indexes*. **R** captures redundancy, but in so doing it also expresses the essential property of an inverted list—the unification of copies of data. Both **R** and inverted lists support querying based on data values. **R** goes beyond the basic inverted list, however, since it organizes the values into a partial order, expressing the transitivity between 'inverted' values. Thus, we might think of the various **R** structure as 'transitive value' structures.

While **R** captures the abstract notion of an index, it does not mandate a specific data structure for that index. **R** supports querying of ordered data values, and states the relationships of sets or sequences of values to one another, but does not specify how these relationships are to be stored or updated. It is possible to develop several data structures that can support **R**, but they belong to the implementation layer of the system, rather than the modelling layer.

3 Here we follow the normal convention of assuming that the logical structure of the document is strictly hierarchical. One could also consider other models, in which **L** might organized overlapping sets.

Some sample queries that can be posed on this structure:

1. *Find all sentences.*

$$\sigma(\mathbf{L}, \text{sen})$$

2. *Find all paragraphs containing the word 'automata'.*

$$\sigma_{para} \uparrow (\mathbf{C}, '\texttt{automata}')$$

3. *Find all paragraphs not containing the word 'compress'.*

$$\beta\sigma_{para}\mathbf{L} - \beta\sigma_{para} \uparrow (\mathbf{C}, '\texttt{compress}')$$

The above queries return the set of elements that satisfy the query conditions. We can apply $\downarrow$ or $\uparrow$ as desired to return the partial order rooted at those elements. We use the convention that quote-delimited names refer to literal values (that is, values stored in the **R** structure), and non-delimited names refer to elements in the **L** structure. This permits us to disambiguate the word 'section' from the concept *section*, should the two appear in the same document. Other conventions are possible.

Next, let us consider the sequential structure of the data. Typically, each element of **L** is totally ordered within its parent; thus, a sentence is a total ordering of words, a paragraph is a total ordering of sentences, and a section is a total ordering of paragraphs. A totally ordered structure is not mandatory; hypertexts, for example, may have a variety of possible orderings for paragraphs or sections, and sentences may have words that can be reordered or contain incomparable alternatives (such as the ubiquitous 'and/or'). We will capture the partial order of the components by **S**. **S** for our example is shown by the dashed edges in Figure 34. Since **S** represents the sequence structure explicitly, we no longer require subscript numbers.

**S** is a union of disjoint sequences, so **S** is trivially partially ordered. The combination of **S** with **C** is also a partial order. **S** orders only incomparable elements within **C**, and therefore does not conflict with either **R** or **L**. We will refer to the combination of **S** and **C** as **T**.

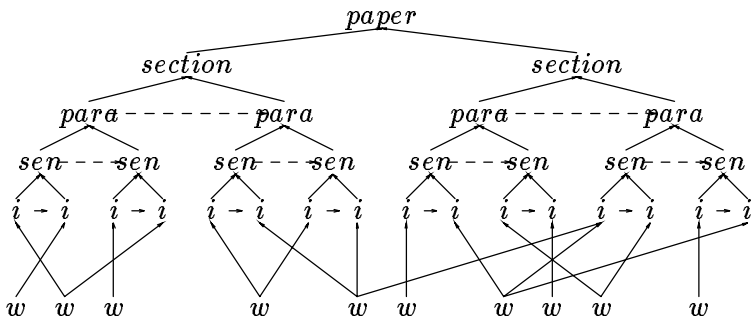



Figure 34: **T**

We next show some sample queries that we can pose on this structure. In order to simplify the reading of some of these queries, we have decomposed them into subqueries, and we explain each subquery in bottom-up fashion. The use of variables to name subqueries should not be taken to imply that general variables can be used in a partial order query; this is merely a shorthand to avoid the complexity of a purely functional notation.

1. *Find the first paragraph in every section.*

   $$\sigma_{para}\perp\mathbf{S}$$

   $\perp\mathbf{S}$ selects the first elements in each sequence and $\sigma_{para}$ restricts the choice to paragraphs. Note that the query works even if there is more than one 'first' paragraph (that is, there are incomparable minima in $\mathbf{S}$).

   This query assumes that paragraphs belong only to sections. If paragraphs occur in other elements of the document (such as figures), then it is necessary to limit the results to only those paragraphs within sections:

   $$\sigma_{para}\perp\mathbf{S} \cap \downarrow(\mathbf{L},\ section)$$

2. *Find all sections whose first paragraph contains the word 'compress'.*

   The query can be solved as follows:

   (a) $W = \sigma_{para} \uparrow (\mathbf{C},\ \texttt{'compress'})$ finds all paragraphs containing an instance of 'compress'.

(b) $X = \sigma_{para} \perp \mathbf{S}$ finds the first paragraphs.

(c) $Y = W \cap X$ finds the first paragraphs containing an instance of 'compress'.

(d) $Z = \uparrow (\mathbf{L}, \beta Y)$ finds the parts of $\mathbf{L}$ containing (first) paragraphs that contain an instance of 'compress'.

(e) $\sigma_{section}(\mathbf{L}, Z)$ finds the sections containing first paragraphs that contain 'compress'.

3. *Find all sections between section $m$ and section $n$ that contain the words 'compact' or 'compress', but not 'automata'. Assume that there are $k$ sections in total, and that $k > n$.*

   The query can be solved as follows:

   (a) $X = \pm_m \sigma_{section} \mathbf{S}$ removes the first $m$ sections.

   (b) $Y = \mp_{k-n} \sigma_{section} \mathbf{S}$ removes the sections from $n$ to $k$.

   (c) $Z = \beta\sigma_{section}(\uparrow (\mathbf{C}, \textit{'compress'}) \cup \uparrow (\mathbf{C}, \textit{'compact'})) - \beta\sigma_{section} \uparrow (\mathbf{C}, \textit{'automata'})$ finds sections containing 'compress' and 'compact', but not 'automata'. Note that we require a set difference operation.

   (d) Finally, $X \cap Y \cap Z$ finds all sections in the desired region that contain the appropriate content.

4. *Find all sentences in which the word 'fast' precedes the word 'compression', possibly with some intervening words.*

   The query can be solved as follows:

   (a) $X = \uparrow (\sigma_i \mathbf{S}, \uparrow (\mathbf{C}, \textit{'fast'}))$ selects strings of word instances that start with 'fast'.

   (b) $Y = \downarrow (\sigma_i \mathbf{S}, \uparrow (\mathbf{C}, \textit{'compression'}))$ selects strings of word instances that end with 'compression'.

   (c) $Z = X \cap Y$ selects strings of words where 'fast' precedes 'compression'.

   (d) $\sigma_{sen} \uparrow (\mathbf{L}, Z)$ finds all sentences that contain the desired strings of word instances.

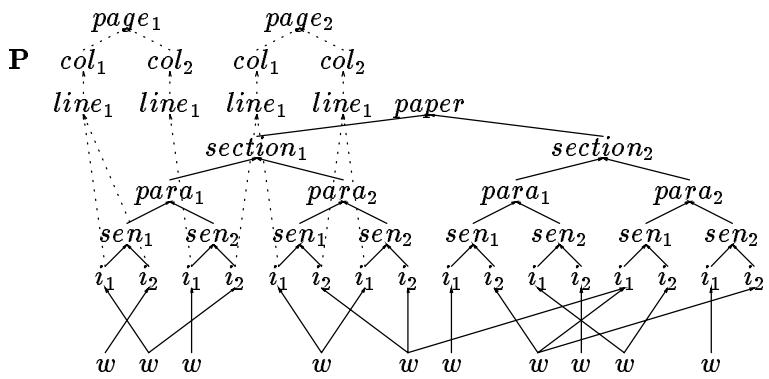



Figure 35: **D**.

The constructs that make up a document consist of ordered structures of characters. Some of these constructs are sets, some are total orders, and some have incomparability (and so, are non-degenerate partial orders). It is the combination of these constructs that makes it difficult to employ purely set-based models to manage text. We may want to do boolean querying of the document, or we may want to do sequence-based searching of the document. Consequently, we need to support both **S**-type structures and **R**-type structures.

We have not yet exhausted the containment structures in text. An important containment structure is the one induced by the physical medium—the layout order **P**. For most documents, this structure includes such elements as lines, columns, and pages; some documents have much more complex presentation or medium-related artifacts, such as sheets, footnotes, marginalia, and parallel translations. **P** for our current example is shown in Figure 35.

**P** is a partial order, because it is based on containment; lines are contained within columns, and columns are contained within pages. **P** may be a non-hierarchical partial order, if some elements belong to more than one ancestor; for example, a figure that spans two columns may be considered to be 'contained' within both of them.

**P** and **T** describe the same collection of characters, but the boundaries of the two structures do not coincide; figures can cross page boundaries, lines can be split at column boundaries, and words can be split at line boundaries (via hyphenation).

**T** and **P** are each partial orders, and so is the combination of

the two. As with the **L** and **R** components of **T**, we observe that **P** and **T** meet at a fringe,[4] and induce no relationships among that fringe. As was the case with **L** and **R**, the combination of **P** and **T** is neither a cross product nor a lexicographical sum. We will refer to this combination as the *document order*, or **D**. **D** supports queries on the combined layout and logical properties of the text.

Here are some sample queries that can be solved with **D**:

1. *Find instances of the word 'transition' on odd-numbered pages.*

$$\uparrow(\mathbf{D},\ 'transition') \cap\ \downarrow(\mathbf{P},\ odd)$$

   We assume that the user has supplied a predicate *odd* that returns `true` when applied to odd-numbered pages, and `false` when applied to any other element of a partial order.

2. *Find all sentences that are contained within exactly one line.*

   This query can be solved by finding $\sigma(\mathbf{L},\ \sigma_{sen}\ \&\ Z)$ where

   (a) $X = \downarrow \mathbf{L}$ finds the sub order rooted at a given element (which will only be *sen*s in this case).

   (b) $Y = \sigma_{line} \uparrow (\mathbf{P}, X)$ extracts the lines containing $X$

   (c) $Z = |\beta Y| = 1$ returns true if the number of lines is exactly 1.

   If we know that

   (a) only *sen* are immediately above $w$ in **D**

   (b) *sen* are only immediately above $w$ in **D**

   (c) only *line* are immediately above $w$ in **L**

   (d) *line* are only immediately above $w$ in **L**

   then we can pose the query in another form, using $\top$, $\bot$, $\pm$, and $\mp$:

$$\sigma(\mathbf{L},\ \sigma_{sen}\ \&\ |\beta\bot\pm \uparrow (\mathbf{P},\ \top\mp \downarrow \mathbf{L})| = 1)$$

4 In fact, **P** and **T** may construct a new fringe, since some atomic elements may be split—as for example when word instances are hyphenated.

The advantage of the second form of the query is that it lends itself to certain kinds of optimization; we can limit our search for matching *sen* and *line* to a restricted part of the order. The advantage of the first form of the query is that it is correct in situations in which *sen* and *line* may not be the immediate ancestors of $w$ in their respective orders.

We have introduced a few simple extensions to the algebra here, for the sake of greater query expressiveness. First, we made use of a COUNT-like function to determine the cardinality of a set. Second, we use logical connectives and compound predicates. Third, because the definition of selection is that the selection predicate is applied to one element at a time, we made use of a shorthand for some subexpressions: in particular, $\downarrow \mathbf{L}$ instead of $\downarrow (\mathbf{L}, x)$. The latter form requires us to use a dummy variable, which we would like to avoid.

We have to this point explored containment structures. Texts also often exhibit semantic relationships between non-contained sections of the document. The most common of these is *cross-references*. Cross-references are used for at least three purposes:

1. An aid to navigation; the cross-reference is an embedded pointer to some non-contiguous part of the document that is of importance.

2. As a statement of a semantic relationship that is not expressible by proximity or nesting.

3. As a technique for reducing redundancy; rather than repeat some content, the author indicates that it is given elsewhere.

Cross-references need not be either transitive or anti-symmetric, and hence they need not be partially ordered. Cross-references are transitive and antisymmetric, however, if they are used for redundancy reduction, because this represents containment, which, as we have seen, is a partial order structure. A camouflaged example of asymmetry occurs when a cycle of cross-references is used to express the fact that several fragments are best read together (as when two

fragments have 'see also' references to each other). Such a structure can be viewed as representing the fact that cyclic fragments share a common (unnamed) essence. This is in effect a statement of containment, or alternatively, a kind of redundancy reduction (the redundant alternative being to include the content of the sink fragment with the source fragment).

The most likely case of symmetry is the unstructured use of links in hypertext, where cycles are common (often occurring when data pages are linked by a 'return' link to the starting or root pages). Even in these cases, there is a distinction between 'return' links and links that denote the subdivison of the network, and we may wish to separate the two kinds of links into two different partial orders.[5]

A more abstract structure, not frequently recognized in text management systems, is the *class* structure of text objects. The class structure captures the common aspects of text objects, just as the class structure of an object-oriented system captures the inheritance relationships among the classes. Class structure is not related to containment boundaries; for example, we may have a class of 'inflections', including such things as emphasized phrases, 'nota bene' fragments, and other methods for indicating inflection through typography. More generally, all of the forms of text structures that we have investigated—logical, redundancy, and layout structures—are themselves part of a kind of class structure of text.

In its most general form, a class structure will be partially ordered, because it is an indication of shared and non-shared aspects of the object descriptions. Document class structure, then, is not fundamentally different from any other multiple inheritance hierarchy one might produce in an object-oriented approach.

We recap the essential elements of a partial order approach to text as outlined so far:

**A logical structure L** is an ordering of subdivisions of the text. The order is given by the containment of the subdivisions

5 The intent here is not to convince the reader that every structure in text *must* be a partial order. There is no reason why this should necessarily be the case; if symmetric or non-transitive structures occur, then an order relation is simply the wrong way to represent them. However, we do challenge the view that any possibility of cycles renders partial orders inapplicable.

within one another.

**A redundancy structure R** is an ordering of values in the text. The order is given by treating a value as containing its instances.

**A layout structure P** expresses the placement of textual glyphs on a display medium. The order is given by the containment of glyphs within structures of the medium.

**A cross reference structure** (if it is acyclic and transitive) is an ordering of elements according to links. The order is given by the sequence that the links denote.

**A class structure** is an ordering of types of structures. The order is given by the common characteristics or content of the types.

The running example we have examined has mostly hierarchical components. However, the partial order model (and the example queries that have been shown) will apply even if logical, redundancy, or layout structures are non-hierarchical, or if they contain non-hierarchical elements.

## TABLES

Tables are perhaps the most common non-hierarchical text structure. Software for making tables has been available since at least 1979 (Lesk 79), but the problem does not appear to be a solved one, as research into tables has continued (Beach 85) (Vanoirbeek 92) (Wang and Wood 93a, 93b) (Shin *et al.* 94). The main issue that keeps researchers interested in tables is separating the purely presentational aspects of tables from their logical structure. Partial orders do not address all the issues of table layout, but they do provide interesting insights into the problem of separating presentation from abstraction. Finding the correct abstraction for tables is an important part of constructing a data model for texts.

Consider the following simple two-dimensional[6] table:

6 In this section, when we use the word 'dimension', we mean in the sense of

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |

It is relatively easy to find a partial order in such tables, but somewhat more difficult to choose the most appropriate order. We might choose, for example, to represent the table by the partial order **T** shown in Figure 36. This partial order appears to combine a row-

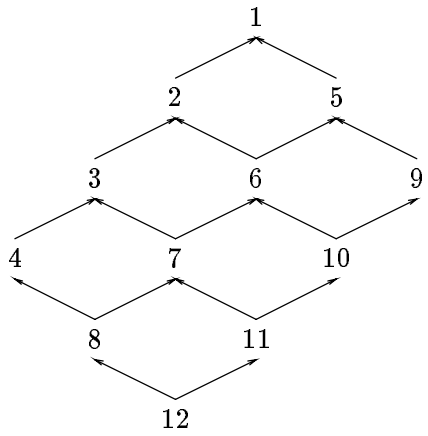



Figure 36: Possible partial order **T** for table.

major traversal of the table with a column-major traversal. **T** has the following properties:

- **T** is a planar lattice; every pair of elements has a greatest lower bound and a least upper bound.
- For every pair of elements $(a, b)$ in **T** such that $a < b$, define the *contained rectangle* as the set of elements in the table that belong to the rectangle defined with $b$ as its upper left hand corner and $a$ as its lower right hand corner. Then the contained rectangle corresponds to the partial order $\uparrow(\mathbf{T}, a) \cap \downarrow(\mathbf{T}, b)$. Note that the contained rectangle may be a row or a column. Conversely, every sublattice of the partial order corresponds to a contained rectangle of the table.[7]

Cartesian space, and not in the sense of the minimal realizer for a partial order.

7 This is similar to the range selection operation in spreadsheets.

- All incomparable elements $a$ and $b$ in $\mathbf{T}$ are elements in the table such that $a$ is below and to the left of $b$, or $b$ is below and to the left of $a$.

- Consider the intersection of $\mathbf{R}$, the total order given by a row-major traversal of the elements, and $\mathbf{C}$, the total order given by a column-major traversal of the elements. $\mathbf{R} \cap \mathbf{C} = \mathbf{T}$; hence, $\mathbf{T}$ has dimension at most 2. But since $\mathbf{T}$ contains incomparable elements, it has dimension at least 2. Hence, $\mathbf{T}$ has dimension 2.

- $\mathbf{T}$ describes both the table and its transpose.

$\mathbf{T}$ is a complete topological description of the layout of the table; indeed, it is simply the graph derived by making each entry in the table a node, each entry bottom an edge, and each entry right-hand side an edge.

$\mathbf{T}$ has some nice properties; its insensitivity to transposition means that $\mathbf{T}$ denotes at least two layouts. Transposition can be useful for fitting a rectangular table into a constrained space. $\mathbf{T}$ is not, however, insensitive to arbitrary row and column interchanges, which may result from sorting the data (among other things). Interchanging rows and columns in $\mathbf{T}$ could be done if we had an operator to 'swap chains'. Each row and column in the table corresponds to a chain in $\mathbf{T}$; chains that are produced only by leftward moves correspond to rows, while chains that are produced only by rightward moves correspond to columns. Thus, interchanging rows would correspond to swapping the appropriate leftward chains.

'Swapping chains' is not an operation in the partial order algebra, but it could be done in an external application. Even if swapping chains was a partial order operation, $\mathbf{T}$ appears not to be the ideal abstract description of a table. What we want is a description that permits us to restrict the row and column interchanges.

Consider a description of the data values themselves that is independent of their layout. Our example has two dimensions, $X$ and $Y$. The dimensions contain individual values $x_1, x_2, x_3, x_4$, and $y_1, y_2, y_3$. Every data point in any valid table layout is denoted by one value from each of these dimensions; hence, the total data space

is the expression $X \times Y$; that is, the binary relation over all data values.[8] If we were not displaying the values in a table, this cross product would be a complete and sufficient description of the data. A table, however, is something more than the data values: it is an ordered presentation of the data values. In the partial order model for tables, we let the data space be defined by the cross product of a collection of data dimensions, and we specify the layout by ordering those dimensions. By ordering the individual $x_i$ and $y_j$ within $X$ and $Y$, and by using a partial order cross product instead of a set cross product, we derive a layout of the data corresponding to a table. The order $x_2 < x_4 < x_3 < x_1$ and $y_2 < y_3 < y_1$, for example, will lead to the following table:

| | $x_2$ | $x_4$ | $x_3$ | $x_1$ |
|---|---|---|---|---|
| $y_2$ | | | | |
| $y_3$ | | | | |
| $y_1$ | | | | |

A different table can be constructed simply by changing the order of one or more of the dimensions. Changing the order of a dimension corresponds to interchanging rows or columns of the table.

In the partial order model, then, a *data space* is the cross product of two or more data dimensions:

$$M = A \times B \times C \times \cdots$$

A *table* is a layout of a data space that is specified by ordering the values of the dimensions:

$$\mathbf{M} = \mathbf{A} \times \mathbf{B} \times \mathbf{C} \times \cdots$$

A data space is produced by a set-theoretic cross product operator; a table is produced by a partial-order cross product operator. The partial order model thus clearly separates the data from its layout; the former is unordered and manipulated with set-based operations, while the latter is partially ordered and manipulated with partial-order operations.

8 Strictly speaking, the table consists of the mapping $X \times Y \rightarrow V$, but since $V$ is functionally determined by every element in the cross product, we can think of the table as being restricted to the elements in the cross product.

A simple cross product is the right model if it is the case that the data dimensions are independent and orthogonal. Often the dimensions are not independent, and a given dimension makes sense only as a subdivision of some other dimension. A table that shows the sales of products of various companies, for example, describes products that are specific to each individual company; company $C_1$ has products $p$, $q$, and $r$, but company $C_2$ only has product $s$. This structure can be expressed by a lexicographical sum of partial orders, where the expression order is the partial order of companies, and the factor orders are the orderings of each company's products. Lexicographical sum gives us the ability to handle nested tables.

Often a display that looks like a nested table is really a case of not having enough dimensions in the output medium. If we have $n$ data dimensions but only $m$ dimensions in the output medium, then in order to render a table, we must compress the data dimensions into the output dimensions. Typically, $m = 2$, so we refer to this as the problem of *planarizing* the data. To planarize $n$ data domains, we must produce $m$ groups of data domains, each group being linearly ordered; each group will be mapped to an output dimension, with the linear order giving the order of 'nesting' of the data domains mapped to that dimension. We will call such a structure a *planarization order*. Two planarization orders for the data space $A \times B \times C \times D$ are shown in Figure 37; the planar table of Figure 37(a) is specified with a planarization order whose realizer is

$$A\ B\ C\ D$$
$$D\ A\ B\ C$$

Similarly, the planar table of Figure 37(b) is specified by the planarization order whose realizer is

$$A\ C\ D\ B$$
$$D\ B\ A\ C$$

We express planarization orders as orderings of the underlying data domains, and not of the ordered domains, because planarization determines only the nesting of data domains, and is independent of the ordering of the individual data values. Note that the planarization order on $m$ dimensions actually permits $m$ layouts, because we allow

| | | | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|---|---|
| $A_1$ | $B_1$ | $C_1$ | | | |
| | | $C_2$ | | | |
| | | $C_3$ | | | |
| | $B_2$ | $C_1$ | | | |
| | | $C_2$ | | | |
| | | $C_3$ | | | |
| $A_2$ | $B_1$ | $C_1$ | | | |
| | | $C_2$ | | | |
| | | $C_3$ | | | |
| | $B_2$ | $C_1$ | | | |
| | | $C_2$ | | | |
| | | $C_3$ | | | |

(a)

| | | $D_1$ | | $D_2$ | | $D_3$ | |
|---|---|---|---|---|---|---|---|
| | | $B_1$ | $B_2$ | $B_1$ | $B_2$ | $B_1$ | $B_2$ |
| $A_1$ | $C_1$ | | | | | | |
| | $C_2$ | | | | | | |
| | $C_3$ | | | | | | |
| $A_2$ | $C_1$ | | | | | | |
| | $C_2$ | | | | | | |
| | $C_3$ | | | | | | |

(b)

Figure 37: Varying table layout

any order of mapping groups of domains to output dimensions (in effect, we permit all transpositions). We can have more restricted planarizations if there is some canonical (total) order of dimensions, and a total ordering of data domains.

It is important to distinguish true nesting of data domains (which is captured by lexicographical sum) from the planarization of data domains. Nesting is an expression of abstract structure, while planarization is a compromise made in order to fit data into a less-than-sufficient number of output dimensions. Planarization always produces a table that suggests relationships which are not actually

present in the data. This is an unavoidable consequence of using fewer dimensions than the data requires. A casual glance at Figure 37(a) might well lead one to infer that $B_i$ is a substructure of $A_j$, rather than an independent data domain.[9]

| | | $D_1$ | | $D_2$ | | $D_3$ | |
|---|---|---|---|---|---|---|---|
| | | $B_1$ | $B_2$ | $B_1$ | $B_2$ | $B_1$ | $B_2$ |
| | $C_1$ | | | | | | |
| $A_1$ | $C_2$ | | | | | | |
| | $C_3$ | | | | | | |
| | $C_1$ | | | | | | |
| $A_2$ | $C_2$ | | | | | | |
| | $C_3$ | | | | | | |

$$\mathbf{G} = A \times C \times \mathbf{B} \times \mathbf{D}$$

| | | $D_1$ | | $D_2$ | | $D_3$ | |
|---|---|---|---|---|---|---|---|
| | | $B_1$ | $B_2$ | $B_1$ | $B_2$ | $B_1$ | $B_2$ |
| | $C_1$ | | | | | | |
| $A_1$ | $C_2$ | | | | | | |
| | $C_3$ | | | | | | |
| | $C_1$ | | | | | | |
| $A_2$ | $C_2$ | | | | | | |
| | $C_3$ | | | | | | |

$$\mathbf{G} = \mathbf{A} \times C \times \mathbf{D} \times B$$

Figure 38: Varying gridline specification

$M$ is a data space, **M** is an ordering that defines a table, and **P** is a planarization order that puts that table into a smaller output space. We can in addition use partial orders to specify some of the decorative aspects of tables. Many tables include gridlines or *rules* that separate groups of elements.[10] These groups are really

9 The careful observer will note that every value of $B_i$ appears under every value of $A_j$, and thus could infer that the domains might indeed be independent.

10 Hart and other typographers recommend minimizing the use of rules (Hart

elements of a partition of the data values, and the gridlines are physical manifestations of the partition. If these groups can be ordered in a fashion consistent with their underlying elements, then we have an additional partial order $\mathbf{G}$ that specifies the gridlines. Figure 38 shows some examples. In Figure 37 (b), for example, $\mathbf{G} = \mathbf{A} \times \mathbf{C} \times \mathbf{D} \times \mathbf{B}$; that is, $\mathbf{G}$ is identical to $\mathbf{M}$, and thus denotes the gridlines seen in the figure. This is the maximum gridding possible for this table. We can remove individual gridlines by selectively removing ordering information from $\mathbf{G}$. If all data dimensions are unordered, then we have no gridlines. If dimension $A$ is totally ordered, then horizontal gridlines will be drawn that separate the elements of $A$. If dimension $A$ is partially ordered, then we will have gridlines only between comparable elements. Not every partial order can be accommodated by this model; we are restricted to a specific type of reduction of the total order implied by $\mathbf{M}$: $\mathbf{G}$ must be a weak suborder of $\mathbf{M}$. If this is the case, then $M \subseteq \mathbf{G} \subseteq \mathbf{M}$, and the gridline specification will be consistent with the ordering given by $\mathbf{M}$.

Another problem that the partial order model is suited to handle is the presentation of extremely big tables that require more than one page (or screen) for display. The logical model of such a table remains $\mathbf{M}$; however, we now divide $M$ into subsets $m_1$, $m_2$, $m_3$, ..., $m_k$ that will each fit on one page, and we display $\sigma(\mathbf{M}, x \in m_i)$ on page $i$. If the page sequence is ordered (or partially ordered), then the subsets can also be ordered, thus specifying their placement on the page sequence. Note that we do not insist that the subsets be disjoint; in some large tables, it may be useful to repeat part of the table content in each of the subtables, perhaps for comparison of a few designated values with all other values in the set.

The partial order model for tables can also be extended to structures that are not usually managed by table software. A list, for

83) (Bringhurst 92). In part, this appears to be a holdover from the properties of mechanical photocomposition devices (Beach 86). In any case, the present discussion is not meant to contradict this advice; here we discuss only mechanism, not policy. The mechanism by which gridlines can be specified is equally applicable to specifying white space or some other delimiter that is preferred by a particular designer.

$A_1$

$B_1$

$C_1$

$C_2$

$C_3$

$B_2$

$C_1$

$C_2$

$C_3$

$A_2$

$B_1$

$C_1$

$C_2$

$C_3$

$B_2$

$C_1$

$C_2$

$C_3$

Figure 39: Linearized table

example, can be treated as a table that is linearized. Thus, the data space used in Figure 37 and 38 could be linearized as in Figure 39 (we leave out dimension $D$ for the sake of space). This table/list is described by the expression $\mathbf{A} \times \mathbf{B} \times \mathbf{C}$. Similarly, page layout might be handled by the tabular model; consider the tabular nature of multiple columns, marginalia, callouts, and page furniture (Beach 86).

The set of partial orders that constitutes a table can be inserted in the document's partial order structure just as if they were words or characters; a document **D** can, without loss of generality, include elements which are complete tables. All the operators of the partial order model will be applicable to such a structure.

### Discussion

The lack of separation between layout and abstraction in traditional table software is not just a theoretical problem; it is an enormous complication in the construction and use of the software. Table descriptions in *tbl* and LaTeX are a throwback to fixed-format data entry such as was common in punch card days. Table layout instructions in such languages use cryptic forms for the sake of efficiency in data entry, but the result is an unforgiving interface (for example, missing delimiters cause the system to abort). Perhaps more annoying is that data is positionally dependent, and so attempting to rearrange a table is often very tedious. In both *tbl* and LaTeX, interchanging rows is easy, but interchanging columns requires editing every row in the table. Transposing rows and columns is complicated when there is a non-trivial gridline structure, since the description of the gridlines is spread over several places in the table.

The partial order model permits us to separate the layout of tables from their abstraction. It is highly generalizable, being applicable to tables of dimension $n$ laid out in a space of dimension $m$, for all $0 < m \leq n$. It is more flexible than a total order specification; total orders describe exactly one table, whereas partial orders define classes of equivalent tables, thus leaving open the possibility of choosing different table layouts to satisfy some other typesetting constraints,[11] or to take advantage of media that can display incomparable data in an unordered fashion. The model also extends naturally to permit additional partial orders for describing gridlines, or perhaps to manage other semantic constraints.[12]

The model presented here does not address every important problem in table layout. One of the more difficult problems is choosing appropriate column sizes, especially when columns contain paragraphs of text. Elegant solutions to this sort of problem require

11 For example, one might choose the ordering that results in the best layout of contained text.

12 Consider for example using **M** as a constraint system for updating the table; in order to update a value, it is necessary to update all values 'above' it in the order. This is one way to ensure that tables that summarize their own data are kept consistent under update. Spreadsheets implement this type of principle, although not with a mathematical model as a foundation.

interaction between the paragraph layout module and table layout module, an interaction not common in batch typesetters.

## SUMMARY

Text modelling is complicated by the problem that definitions of 'text', like definitions of 'art', tend to invite counterexamples that overturn the definition. Standard Generalized Markup Language (SGML) for example, structures computerized text in a static, hierarchical repository containing embedded markup. Yet computerization itself encourages rapid and frequent change to text, as well as multiple (and possibly external) structures on texts. The conclusion we should draw from this experience is not that one model is better or worse; rather that any model is more likely to change the domain than capture it. The domain-specific approach to modelling is not helpful when the domain is too fluid. We should not try to found a text database system on an abstract definition of text, any more than we attempted to found traditional database systems on abstract definitions of business.

The partial order model does not attempt to define text, nor is it especially designed for text. Instead, it follows the general philosophy of the relational model: find a well-chosen mathematical abstraction, and by its nature it will be applicable to many problems. We have seen that many structures in texts can be described as partial orders. The most important contribution of the partial order model is its extraction of order as an abstract concept, whose manipulation and update is fundamental to many problems in text management.

Markup-based approaches tend to encourage a hierarchical view of text, not just because it is easier to handle hierarchical markup, but because it is easier if most structures in the text are derived from the underlying total order of the standard file representation. This assumption is powerful for managing hierarchical and proximity-based relationships. It becomes a liability when trying to manage non-hierarchical relationships, such as tables or overlapping sections. It also leads to errors when the proximity that is in the representation is not part of the model; thus, the last word of one sentence is

proximate to the first word of the next sentence, but the two words are not related, as they belong to different structures.

Partial orders are not limited to hierarchies, and so they can express tables and overlapping relationships in a natural manner. Containment is the most important relationship in text, and hierarchies do express containment, but not all containment is hierarchical. The most general form of containment is the partial order.

SGML addresses the variety of structures in text with a variety of mechanisms.

- the 'logical' structure can be represented by the element tags in the document, and the element declarations in the document type definition
- the 'redundancy' structure can be represented by the use of entities and entity references
- the cross-reference structure can be represented by the use of ID/IDREF attributes
- the layout structure can be represented by CONCUR
- class structures may be approximated by use of extension mechanisms

These mechanisms are not equally powerful or well-developed; CONCUR, for example, is not widely implemented or used (Barnard 88). Similarly, extension mechanisms for class structures are not part of SGML, and so they are typically expressed as guidelines (for example, the Text Encoding Initiative's *Guidelines* (TEI 94)) rather than enforceable constraints. SGML's mechanisms are also not mutually orthogonal; it is possible, for example, to represent a hierarchy with ID/IDREF values, or even with entities and entity references (indeed, it is possible to represent directed acyclic graphs with entity references). The lack of orthogonality in these mechanisms complicates the document modeller's problem, as do their differing implementation-specific characteristics.

The basic strength of the partial order model is that it describes the various structures of text with the single abstraction of order,

rather than through a variety of mechanisms. This unification of text structures makes it easier to combine and manipulate the components of the resulting structures, because closure is more easily obtained for a single abstraction and its few operators, than for a collection of distinct mechanisms. It can also be argued that it is easier to understand a text as a variety of instances of one class of structure, than as a single instance of a variety of classes of structures.

The contemporary approach to text splits querying into two distinct parts: a specification of structural properties (such as 'paragraph' or 'section') and a value specification (typically specified by string searching). The problem with naive string searching is that it directly involves the user with the representation. The user must worry about such issues as case, embedded tags, accents, variant spellings, and punctuation (or rather, the typical user does not even think of these things, and so the search fails to be complete). The partial order model permits the data modeller to unify value-based searching with structure-based searching; instead of strings being a different kind of data value, they are partial orders of atomic values. Explicitly representing sharing exposes the difference between a data value (of which there is only one copy) and the position of that value in some order (of which there may be as many instances as there are orders). Including strings in the logical model also has important update implications, which are handled informally in other systems.

The ability to have more than one ordering relation on a single base set is useful for capturing the many orders (some conflicting) that can be imposed on a single text. Markup-based systems can represent multiple orderings, but simply representing them is insufficient; one also needs operators to manipulate, combine, and update these orders. The partial order model provides such operators.

We have seen in the chapter on software that partial orders are useful for representing versions and variants of software. Documents also come in versions and variants, and it is often important to organize and retain this structure. Knowing the structure of the set of versions of a text is an essential aspect of bibliography (Williams

and Abbott 89). The ability to use the same model for structuring both a single version of a document and its family of versions clearly offers an important advantage in the reuse of common concepts and implementations.

# 6

# Partial orders and databases

We have formally defined the partial order model, shown one representation, and argued for its utility by exploring several applications. Many data models have been similarly justified, and it appears that the partial order model would be just one more model among many. In this chapter we show that the partial order model has an additional significance, because it appears to be useful for describing data *about* the database, and modelling systems *within* the database. Such areas as dependency theory, indexing, transactions, and inheritance graphs are usefully modelled with partial orders. Putting the partial order model at the foundation of database work can more closely unify the study of data and meta-data.

## DEPENDENCY THEORY

> Almost every concept in the existing relational database theory has a counterpart in lattice theory. This suggests that further study of relations should be carried out within the framework of lattice theory (Lee 83).

Data dependencies in a relational database are an expression of the containment relationships of subsets of attributes, as inferred from the data they represent. Containment relationships, as we have seen, are expressible as partial orders. This connection between partial orders and database dependency theory was first observed by T.T. Lee (Lee 83). We briefly restate Lee's results here.

Consider a relation $R$ with attributes $\{A_1,\ A_2, \cdots A_n\}$. A subset of the attributes partitions the set of tuples in an instance, where

| Name | Size | Distance | Moon |
|---|---|---|---|
| Mercury | small | near | no |
| Venus | small | near | no |
| Earth | small | near | yes |
| Mars | small | near | yes |
| Jupiter | large | far | yes |
| Saturn | large | far | yes |
| Uranus | medium | far | yes |
| Neptune | medium | far | yes |
| Pluto | small | far | yes |

Figure 40: Solar system data.

two tuples belong to the same equivalence class if they have identical values on the given attributes. Now consider all such partitionings of a given instance, and the subsets of attributes that are denoted by them (in general, more than one subset will correspond to a given equivalence class). These subsets can be arranged in a containment lattice, where the minimum element corresponds to a partition that places each tuple in a separate equivalence class, and the maximum element corresponds to a partition with a single element containing all tuples. Lee refers to this containment lattice as the *relation lattice*.[1]

Consider the example in Figure 40 that contains data about the solar system. We can generate one partition of the tuples for each subset of the attributes, where the elements of the partition have equivalent values for the chosen subset of attributes. Given a subset of attributes $A$, $\theta(A)$ is the induced partition of the tuples. (In the following, $M_e$ stands for Mercury and $M_a$ for Mars, and each attribute is represented by the first letter of its label). We then have

1 Although Lee does not explicitly say so, not every instance suffices to generate the relation lattice; one wants an instance that will generate the maximum number of valid partitionings. In general, Lee's paper does not clearly distinguish between a relation schema and a relation instance.

the following distinct partitions:

$$\begin{aligned}
\theta(S) &= \{M_e,\ V,\ E,\ M_a,\ P\},\{U,\ N\},\{J,\ S\} \\
\theta(D) &= \{M_e,\ V,\ E,\ M_a\},\{J,\ S,\ U,\ N,P\} \\
\theta(M) &= \{M_e,\ V\},\{E,\ M_a,\ J,\ S,\ U,\ N,\ P\} \\
\theta(SD) &= \{M_e,\ V,\ E,\ M_a\},\{J,\ S\},\{U,\ N\},\{P\} \\
\theta(SM) &= \{M_e,V\},\{E,\ M_a\},\{J,\ S,\ U,\ N\},\{P\} \\
\theta(DM) &= \{M_e,V\},\{E,\ M_a\},\{J,\ S,\ U,\ N,\ P\} \\
\theta(SDM) &= \{M_e,V\},\{E,\ M_a\},\{J,\ S\},\{U,\ N\},\{P\} \\
\theta(NSDM) &= \theta(N) = \theta(NS) = \theta(ND) = \theta(NM) \\
&= \theta(NSD) = \theta(NSM) = \theta(NDM) \\
&= \{M_e\},\{V\},\{E\},\{M_a\},\{J\},\{S\},\{U\},\{N\},\{P\}
\end{aligned}$$

We define $\theta(\emptyset)$ as the partition consisting of a single element representing all tuples. Note also that because the data is a relation, any subset including the key (that is, the planet's name) will always result in the maximal partitioning of the tuples. These partitions can be used to generate the relation lattice $\mathbf{L}$ for the data, given in Figure 41.

The relation lattice embodies a great deal of information about the constraints, which we state here without the proofs (they can be found in Lee's paper):

1. The partition that maps to the minimum element of the relation lattice corresponds to the set of superkeys of the relation schema.

   For the solar system example, the superkeys are all combinations of attributes that include $N$. In the partial order model, the superkeys are identified by $\perp\mathbf{L}$.

2. The ordering relationship of the relation lattice is the set of functional dependencies for the relation schema.

   For the solar system example, the functional dependencies are all trivial, with the exception of $N \rightarrow SDM$. The regular structure of the lattice suggests this fact.

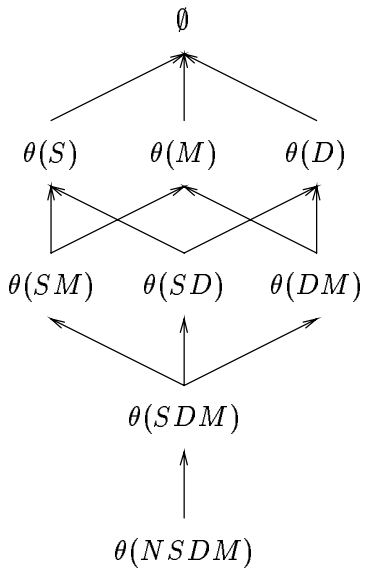



Figure 41: Relation lattice **L** for solar system data.

3. Boyce-Codd normal form corresponds to a relation lattice in which each (prime) atomic filter is isomorphic to a Boolean lattice. An *atomic element* is one that covers a minimal element; in this example, the only atomic element is $\theta(SDM)$. An *atomic filter* is the prime filter of an atomic element, which in this example is given by $\uparrow \theta(SDM)$. Since $\uparrow \theta(SDM)$ is isomorphic to a Boolean lattice, the example is in Boyce-Codd normal form.

   We can give an intuitive explanation of this result. Since all superkeys are found in the minimum element of the lattice, then the atomic elements must correspond to the largest partitions derived solely from non-key attributes. If all atomic filters correspond to a Boolean lattice, then we know that every possible combination of non-key attributes can be distinguished by at least one tuple, and so no non-key attributes can be determined by other non-key attributes. If some atomic filter does not correspond to a Boolean lattice, however, then there must exist some dependencies between non-key elements, and consequently the relation cannot be in Boyce-Codd normal form.

| Name | Size | | | Distance | | Moon | |
|---|---|---|---|---|---|---|---|
| | small | medium | large | near | far | yes | no |
| Mercury | × | | | × | | | × |
| Venus | × | | | × | | | × |
| Earth | × | | | × | | × | |
| Mars | × | | | × | | × | |
| Jupiter | | | × | | × | × | |
| Saturn | | | × | | × | × | |
| Uranus | | × | | | × | × | |
| Neptune | | × | | | × | × | |
| Pluto | × | | | | × | × | |

Figure 42: Solar system data in binary form.

### Concept analysis

The algebraic theory of relational databases is closely related to Wille's theory of concept analysis (Wille 82). Concept analysis is based on a table that resembles a relation, but whose entries are limited to binary values. The solar system data given in the previous section is actually a modification of a concept analysis example given by Wille in his original paper; Wille used the binary form shown in Table 42.

The goal of concept analysis is to find clusters of related data. Given a binary table as in Figure 42, we identify as concepts any subset of objects that have identical values on a subset of attributes. More formally, a *concept* is a pair $(O, A)$, where $O$ is the set of tuples that possess all attributes in $A$, and $A$ is the set of attributes common to all objects in $O$.

Given two concepts $(O_1, A_1)$ and $(O_2, A_2)$, we say that $(O_1, A_1) \leq (O_2, A_2)$ if $O_1 \subseteq O_2$ (which is equivalent to stating that $A_1 \supseteq A_2$). This ordering relation over the total set of concepts produces a *concept lattice*. The concept lattice for the solar system example is shown in Figure 43.

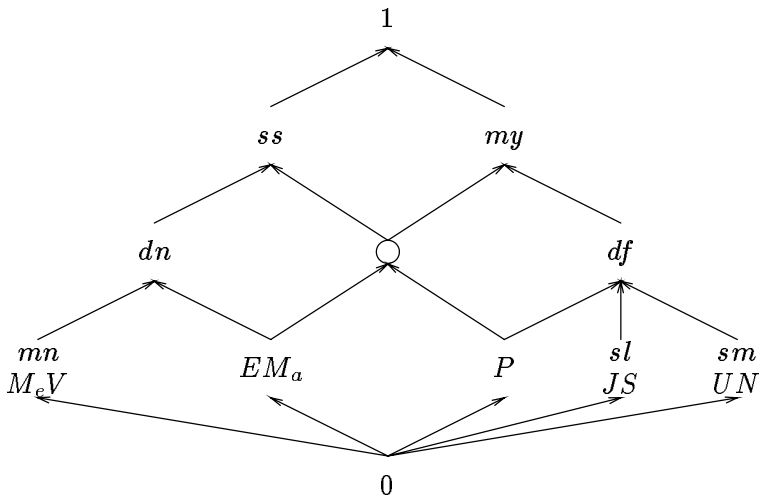



Figure 43: Concept lattice.

### Discussion

Concept lattices and relation lattices both manage equivalence classes defined by tuples that share the same values on some subset of attributes. The key difference is that the concept lattice is primarily a lattice of subsets of attributes, while the relation lattice is a lattice of subsets of tuples. Figure 44 relabels the concept lattice for the solar system example to indicate the subsets of tuples it organizes. A relation lattice expresses all functional dependencies for a schema, while the concept lattice expresses only trivial functional dependencies that are given by containment of the attribute sets.

These partial orders on relational data are interesting for several reasons. One reason is expressiveness. Relational theory assumes that data is best managed as unordered sets. The consistency of relational data, however, requires dependency theory, whose meta-data is not set-oriented data. Thus, we have the irony that some of the meta-data of the relational model falls outside the bounds of what that model is capable of describing naturally. By using partial orders, we permit this kind of meta-data to be managed.

Partial orders of relational data can also lead to new group operators. Furtado and Kerschberg's quotient algebras involve group operators that manipulate relations partitioned in exactly the man-

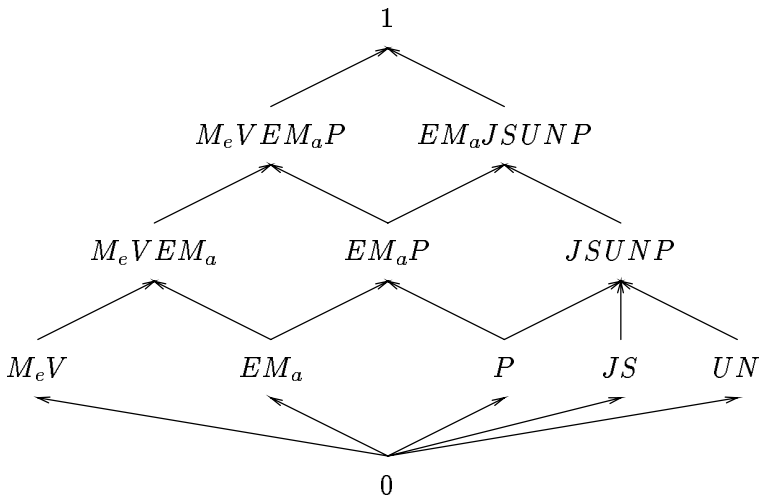



Figure 44: Concept lattice, showing subsets.

ner described by Lee (Furtado and Kerschberg 77). Furtado and Kerschberg's quotient relation $Q_R$ is tantalizingly close to the relation lattice. $Q_R$ is, as they suggest, isomorphic to $\mathbf{2}^R$, under the condition that there are no functional dependencies.

Lee demonstrated that Boyce-Codd normal form is a structural property of the relation lattice, and that therefore one can both check for BCNF and generate BCNF by manipulating the relation lattice. It is interesting to consider whether there are deeper issues at hand. Atoms, for example, are not merely a construct that is only useful in determining BCNF; it is known that every Boolean lattice is uniquely determined by its atoms, as is implicit in the following theorem (Davey and Priestley 92):

**Theorem 6.1** *The following are equivalent:*

1. *$B$ is a Boolean lattice*
2. *$B \cong \mathcal{P}(\mathcal{A}(\mathcal{B}))$, where $\mathcal{A}$ denotes the set of atoms of $B$, and $\mathcal{P}(\mathcal{A}(\mathcal{B}))$ denotes the power set of the set of atoms of $B$*
3. *$B$ is isomorphic to $\mathbf{2}^n$, for some $n \geq 0$*

The atoms of a Boolean lattice are a complete description of the lattice. Every element of a Boolean lattice can be considered to be

the join of the set of atoms that are less than it. Also, every atomic filter of a Boolean lattice is itself Boolean. Thus, the atoms of a Boolean lattice are its fundamental structures.

The atoms of a relation lattice correspond to sets of non-key attributes that are maximal, in the sense that they define the partitions of the tuples that are most highly refined (keys, of course, will always produce the trivial partition of singleton sets).

## OBJECT-ORIENTED MODELLING

Much of data modelling today is done with object-oriented models. The database community has long known that it would be interesting to structure type information (Smith and Smith 77), but more recently type structures have come to be considered an essential tool for modelling non-traditional database applications. We shall see that some of the main features of the object-oriented approach result from the structural properties of the partial orders that lie beneath it.

Object-oriented approaches claim three basic virtues:

1. ease of data modelling
2. encapsulation of functionality
3. software reuse

The ease of data modelling depends on the extent to which the user's concepts can be directly mapped to objects. This appears to be the most oversold and least quantifiable aspect of object-oriented approaches. Encapsulation and software reuse, on the other hand can be quantified. Encapsulation is a set of techniques for controlling access to objects; the goal of encapsulation, like type checking, is to introduce constraints in order to catch semantic errors in software. Software reuse is the removal of redundancy in software; as with database normal forms, the goals of software reuse are to correctly propagate updates, to achieve consistency, to reduce resource consumption, and to decompose systems into their natural parts.

Each of these three virtues gains at least part of its power from underlying order relationships. Object-oriented modelling, for ex-

ample, is based on aggregation and generalization structures (Smith and Smith 77). An aggregate is the sum of its components; since aggregates can share components and can themselves be aggregated, an aggregation structure is partially ordered. Generalizations, on the other hand, represent the common features of their instances; the attributes of a generalization are the intersection of the attributes of the instances. Generalization is the dual of containment, and so it is also a partial order relationship. Object-oriented modelling is done by identifying aggregates and generalizations, and organizing them in a structure that is usually called the *inheritance graph*.

Software reuse is a dependency structure in which one fragment of code provides functionality by pointing to another fragment that actually implements the desired functionality. This is a partial ordering, since cyclic dependencies are not permitted; a function cannot be implemented by pointing to some other fragment whose implementation requires the original function.[2] Software reuse is a kind of containment, since one can think of base classes as parts of derived classes. In C++, reuse is often implemented as containment, since the memory structure of a derived class includes all its base classes (Terribile 94).

Encapsulation is an equivalence relation that partitions the set of classes into those that can access a given object and those that cannot. Like any other form of permission structure, encapsulations tend to occur as a containment structure; an object can access all the resources that are permitted to objects that it can access. There are exceptions to this, however; C++'s `friend` structure, for example, is never transitive, since friends of friends cannot access what the friends can access.

Modelling, reuse, and encapsulation can all result from the same underlying partial order. A standard C++ inheritance graph can serve all three purposes: the classes are the model of the system, the inherited members are the software reuse, and the encapsulation rule is that base class members can only be accessed by the base classes or classes derived from them. Thus, applying the $\uparrow$ operator to the un-

2 Note that recursive definitions of functions are not cyclic; here a total function is defined in terms of a partial function.

derlying partial order simultaneously tells us about modelling, reuse, and encapsulation. In most practical programs, however, modelling, reuse, and encapsulation involve different structures.

Software reuse is separated from modelling if we share a model without sharing implementation. In C++ this is done with abstract base classes, which have no instances. Abstract base classes are used to share abstract interfaces with a set of (derived) concrete classes, while still enabling them to employ different implementations. The object-oriented community refers to this as *inheritance of interface* without *inheritance of implementation.* Another way to separate reuse from modelling is to reuse software completely outside the inheritance graph. The best example of this in C++ is templates, which reuse both implementation and interface, but are not related by inheritance (that is, not related by data type). Yet a third possibility is to reuse only implementation, and not interface. Contrast the inheritance-based

```
class DerivedX : public X
{
}
```

and the containment-based

```
class ContainsX
{
   X my_X;
}
```

A `DerivedX` is a kind of `X`, but `ContainsX` is not. Updates to `X` (that is, members that are added or deleted) are automatically propagated to `DerivedX`; one can immediately invoke the newly-added functions from a `DerivedX` object. This is not the case for a `ContainsX` object; newly-added `X` functions cannot be invoked as `ContainsX` operations, and functions that are deleted from `X` may even cause `ContainsX` to become inoperative. `ContainsX`'s interface is not the same as that of `X` (except insofar as the programmer tries to make these interfaces the same). `ContainsX` does, however, reuse the implementation of `X`, because it need not implement any

`X` functionality; it only needs to convert invocations on itself into invocations on `X`.

Reuse and modelling need not employ the same relationships, and neither do encapsulation and modelling necessarily employ the same relationships. C++'s notions of `friend` and `private` can be used to violate the default permissions of the inheritance graph in an arbitrary manner. As previously noted, `friend` is not a transitive relationship; thus, a structure that expresses the `friend` relationship will have many disjoint components. This seems unsatisfactory, since one might expect that all `friend` elements should be in one structure. On the other hand, the only relationship between each of these `friend` elements is that they are of the same type; there are no inferences that can be drawn about two classes that may each have a `friend` relationship with one other class.

The three order structures support the basic queries one poses on a given object: *What data members can it access? What is its implementation?* and *What kind of thing is it?* Each of these questions can be answered by use of the $\uparrow$ or $\downarrow$ operators on the appropriate partial order. Naturally, all object-oriented systems implement these queries in some manner, but the implementation is often a hidden part of the language. One of the difficulties in learning C++, for example, is recognizing which issues are resolved at compile-time (templates and non-virtual functions) and which are resolved at run-time (virtual functions). As in other software environments, object-oriented systems provide not an algebra, but a collection of special rules and customs that implement the chosen set of constraints.

$\uparrow$ and $\downarrow$ are two of the simpler operations in the partial order algebra. Other operators suggest ways in which we could support much more powerful class structures than is currently done. One simple observation is that the partial order algebra provides a more powerful way of assembling class structures from existing substructures. Generally, class hierarchies are put together manually, one class at a time. The partial order operations of lexicographical sum and cross product, make it possible to combine multiple class structures in one operation. Similarly, the operations of selection and

projection could be used to decompose a given class structure into several orthogonal pieces.

A very interesting parallel to the partial order model is found in some *classless* object-oriented systems, such as SELF (Chambers *et al.* 91) (Ungar and Smith 91) (Ungar *et al.* 91). In a classless object-oriented system, objects have no type, and so they do not obey type constraints. Modelling, reuse, and encapsulation are done on an instance-by-instance basis, rather than on a class-by-class basis.

Classless systems have the advantage of flexibility and simplicity. Classless systems can be exceedingly simple. The SELF system, for example, has no classes, no types, no variables, and few control primitives. It has objects, which can point to one another, and it has messages which are sent to objects (often, to one's self, hence the name of the system). Objects have named slots; these are like the fields in a record. A message is a slot name; sending a message to an object causes it to access the slot. If the slot contains a value, the value is returned; if the slot contains a function, the function is executed. If a message arrives for which an object has no slot, then the message is propagated to the object's parents (that is, the objects it points to) to find a slot with the given name.

It is interesting to compare SELF to the partial order model. In broad outline, a SELF program is a partial order of object instances. SELF uses object relationships to propagate messages; the partial order model uses element relationships to propagate information.[3] SELF is a classless system; the partial order model is a typeless system. The main differences between the two approaches are that SELF does not apply an algebra to its object structures, and the partial order model does not have a notion of computing via propagating messages through the partial order. It would be interesting to consider applying algebraic operations to SELF structures, and also to consider extending the partial order model with SELF-like triggers or similar notions from active databases.

One of the advantages of a classless system is that objects may change their parents at run time. This corresponds to changing the

3 The message traffic in SELF is not partially ordered; messages can be sent between arbitrary objects. The intent of the structure of a SELF program, however, is to capture as much of the expected message traffic as possible.

class of an object in a class-based system. Most class-based systems do not permit changing the class of an object, partly because that kind of change would preclude compile-time binding (and thus lead to greater inefficiency), and partly because in class-based systems, the class of an object is seen as an immutable part of its character; this is also the common assumption in database modelling. The ability to change the parents of an object at run-time complicates one of the more difficult problems in multiple inheritance: how do we resolve cases of ambiguous inheritance? Consider a situation in which the object `Derived` inherits from both `Base1` and `Base2`, both of which contain a member function `DoIt`. If `Derived` is sent the message `DoIt`, should it use `Base1::DoIt` or `Base2::DoIt`? In C++, such ambiguity is a compile-time error, and the programmer must specify exactly one choice. In some other systems, the choice is made by using a fixed, total ordering of all classes, and choosing the one with higher ranking in this total order. Both of these approaches work best if one assumes that the parents of an object are fixed. Since the parents of an object are dynamic in SELF, it needs a more powerful method for resolving ambiguity. SELF's developers resolve some cases of ambiguity with the *sender path tiebreaker rule* (Chambers *et al.* 91). This rule chooses the implementation that is on a path between the sending and receiving object, as shown in Figure 45. Here the sending object and the receiving object belong to a chain containing `Base2`, so it is `Base2`'s implementation of `DoIt` that is chosen. The sender path tiebreaker rule can be easily implemented in the partial order algebra by checking the result of $\uparrow(\mathbf{S}, \mathit{sender}) \cap \downarrow(\mathbf{S}, \mathit{receiver})$.[4]

That the partial order model can emulate the sender path tiebreaker rule is not as important as the fact that having an algebra at one's disposal allows for many other rules to be implemented, rather than hardwiring a single rule in the language, as is done in SELF. The partial order algebra also permits us to write additional rules to cover situations in which the sender path tiebreaker rule is ineffective; for example, if the sender and receiver are incomparable, the

4 Strictly speaking, we must also check $\downarrow(\mathbf{S}, \mathit{sender}) \cap \uparrow(\mathbf{S}, \mathit{receiver})$, since we do not know that the receiver is necessarily less than the sender.

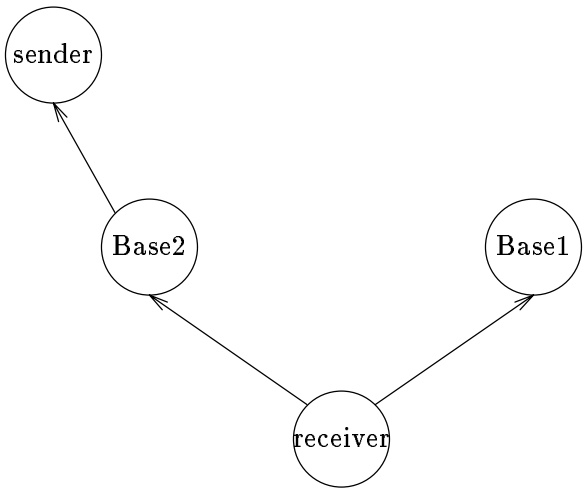



Figure 45: Sender path tiebreaker rule

sender path tiebreaker rule cannot apply.

Object-oriented systems make use of partial orders without providing an algebra that would confer greater flexibility. Using a partial order to manage object-oriented systems also suggests close connections between areas such as version control and redundancy control. One can think of derived objects as versions of their base classes, or base classes as an attempt to normalize the redundancy in derived classes (Raymond and Tompa 92). The partial order model can serve as a framework that unifies all three of these domains.

### MANAGING REDUNDANT DATA

Databases often contain multiple instances or forms of data. These forms are designed to improve querying. Three of the more common forms are:

**sorted data** Sorted data facilitates access on the imposed ordering. An index is an ordering of a subset of the data to support fast access. Typically indexes are constructed for keys, and the index may also contain some non-key data, if it is frequently accessed when keys are accessed.

**cached data** Cached data is usually a derived result that is maintained in the expectation that it will be needed in the future. Materalized views and other temporary tables are one kind of cached data. Query plans that are maintained by a query optimizer are a cache of meta-data.

**replicated data** Replication means keeping multiple copies of data, sometimes at more than one site. Replication is done either to reduce contention for some widely-used data set, or to avoid latencies in a communications network (replication may also be done to improve the robustness of the database, but we are not primarily concerned with this aspect here.)

These different forms of data can improve querying, but they also introduce problems. The first problem is in maintaining consistency under update. When an update is pending, we need to know which forms are dependent on data that will be updated, so that all forms can be changed (or at least checked) at the same time. The second problem is in query planning; if there are many forms of data, then there are potentially many ways to obtain a desired result. Even though the reason for adding forms of data is to improve queries, the additional forms increase the space of possibilities and thus complicate the problem of finding an efficient query plan.

We will consider one possible approach to using partial orders to manage different forms of data. We want to capture the fact that some collections of these forms contain more information than other collections. We will do this with a partial order $\mathbf{C}$, in which every element is a set of tables, and $a < b$ if there is some query on the set of tables $b$ that produces the set of tables in $a$, and no query which can do the reverse. $\mathbf{C}$ is a partial order, because the relationship is antisymmetric (by definition) and transitive (if query $q_1$ produces $b$ from $a$, and query $q_2$ produces $c$ from $b$, then the composition $q_1 q_2$ produces $c$ from $a$). The maximal element of $\mathbf{C}$ is the universal relation (from which all other relations can be generated) and the minimal elements consist of the individual tables that actually occur in the database.

The partial order $\mathbf{C}$ has the property that if some element $c$ satisfies a query, then so does every element in $\uparrow(\mathbf{C}, c)$. This is

because $\mathbf{C}$ preserves monotonicity of information content. Thus, given a valid query strategy that employs a certain set of tables $c$, we can use $\mathbf{C}$ to find alternative sets of tables, and thus to suggest other query strategies. If we know that $c$ is the minimum table set, in the sense that it is the lowest possible element of $\mathbf{C}$ that satisfies the query, then we can find all table sets that can lead to a solution, since they must all belong to $\uparrow(\mathbf{C}, c)$.[5]

$\mathbf{C}$ expresses one type of information containment for derived data. The replications found in a distributed database, for example, can be viewed as derived data, and so they will appear in the partial order $\mathbf{C}$. But replicated data is also structured according to the location at which it is found. Assume that the database is spread over $i$ sites, where we use the term 'site' loosely–it may mean separate machines, separate locations, or just separate logical processes. Then for each site $i$ of the database, we can construct a partial order $\mathbf{S}_i$ that models the 'cost' of collecting at site $i$ a group of tables from any or all of the sites. Each element of $\mathbf{S}_i$ is a set of tables; $a < b$ in $\mathbf{S}_i$ if the cost of collecting the set $a$ is less than the cost of collecting the set $b$ (we may compute cost by considering the speed of the communications link, the size of the table, or other factors). The minimum element of each $\mathbf{S}_i$ will be the set of local tables, with cost 0. The sets of tables that can be collected with lowest non-zero cost will be found at $\perp \pm \mathbf{S}_i$; and so on. The partial orders $\mathbf{S}_i$ describe the preferred sets of tables to use to solve queries at a given site $i$.

Using $\mathbf{S}_i$ and $\mathbf{C}$, we can construct algebraic approaches to solving various problems in query planning. If we want the minimal cost sets of tables that can solve a query known to be solvable with set $c$, for example, then we can use the following query:

$$\perp\sigma(\mathbf{S}_i,\ \beta \uparrow (\mathbf{C}, c))$$

This finds all sets of tables that can solve the query, selects the suborder of $\mathbf{S}_i$ that includes only those sets, and finds the minimal elements of that suborder.

5 It may be the case that there is a set of minimal elements $c_i$, each of which can satisfy the query. In that case the collection of possible query sets is found in $\uparrow(\mathbf{C}, c_i)$.

Storing this information in a meta-database has advantages. It provides a structured mechanism in which to handle the update of the meta-information—new derived tables can be added and deleted, but the same queries can be executed to find the minimal cost sets. It also permits us to dynamically modify some parameters of the system; for example, we can modify the definition of 'cost' to avoid hot spots in data access. If we increase the cost of accessing a hot spot from any site, then it tends to encourage the use of other copies of the data. The addition of these copies, and the increasing of the cost of the hot spot, can be done automatically.

Indexes can also be structured with partial orders.[6] If we have several indexes, we can construct a partial ordering based on the containment of index order. As an example, a warehouse database may be indexed by the serial number of the parts and the date that the parts entered the warehouse. If it is the case that serial numbers and date of entry into the warehouse are both increasing over time, then there is an *order dependency* between the two values (Ginsburg and Hull 83). Given the order dependency between serial number and date of entry, an index on serial numbers can be used to narrow searches for date of entry.

Order dependencies, in effect, state that indexes **A** and **B** are isomorphic. We can manage non-isomorphic indexes based on containment. It may be the case that some of the indexes in a system are refinements of other indexes; that is, index **A** is a total ordering of the data, and index **B** is a partial order such that $\mathbf{B} \subset \mathbf{A}$. Then there is a containment relationship between the two indexes.

## DISCUSSION

Data modelling is based on the principle that abstraction promotes system evolution, flexibility, and rigour. Database practitioners have long advocated data modelling for the enterprise's data, but they have not been as interested in imposing abstractions on the *internals* of database systems. The internals of database systems such as DB/2 are a panoply of tables, views, indexes, query optimiz-

6 Recall that the chapter on text databases showed the similarity between an inverted index and the redundancy structure **R**.

ers, storage managers, transaction control systems, locking mechanisms, gateways, and all the other manifold appurtenances of modern database software.

It would be useful for abstraction to be applied to database internals, and not just to the data they manage. Indeed, there appears to be a significant trend towards *less* abstraction for the managed data, and more for database internals. The burgeoning presence of the Internet and the variety of data resources it provides are rapidly developing a situation in which most data will not be under the control of centralized database systems, but will be found in heterogeneous, distributed systems of all kinds, with no centralized control, no database administrator, and questionable levels of consistency. This kind of environment necessarily makes the interfaces of systems the most prominent feature of database study. This trend can be seen in work on multidatabases, federated databases, and standards for componentized databases like Microsoft's OLE. Interfaces, of course, embody an abstraction of the system they encapsulate.

The concept of using a data model to describe meta-data is not new. Relational systems store their catalogs in tables that can be accessed with SQL. This technique is at best underutilized, however, partly because much of the interesting meta-data in a database involves order or dependencies. Unlike set-based models, the partial order model can describe at least some of the features of database internals, such as transactions, consistency control, and the management of redundant forms of data. The partial order model is a natural way to express multiple sets of such dependencies and orderings, and to manipulate them algebraically.

# 7

# Future work and conclusions

Let us recap the contributions of the thesis.

The main contribution is the recognition that partial orders have value in modelling data. Partial orders possess three essential properties that are either non-existent or underused in most database models:

- they make order explicit
- they can express typeless structure
- they can express multiple structures of data

These properties are essential for modelling ordered data.

A second contribution is the definition of the model, including its set of operators. The operators were derived from a blend of sources, including algebra, existing database practice, and experience with attempting to apply the model to practical problems. For several operators we have shown reductions to more fundamental operators.

A third contribution is the elaboration of a representation based on realizers. This work includes tractable algorithms for each operator, and additionally an algorithm for producing a realizer from an arbitrary partial order. Some informal arguments were presented to show that realizers are a reasonably efficient representation.

A fourth contribution is the application of the model to three important database problems—software, text, and meta-data. In each case we proposed a decomposition of the problem into partial orders, and showed some examples of the application of the partial order algebra to solve queries on the data. In each case, the partial order model

- solves interesting and useful problems

- suggests new functionality
- exhibits generality
- supports integration into a more general framework

The partial order model was particularly useful in describing table manipulation and a new approach to the problem of building software.

Next we consider possibilities for future work.

## DEPLOYING THE MODEL

In previous chapters, it was implicitly assumed that the partial order model might be implemented and used without reference to other models and systems. While this may be an appropriate stance for explaining a new model, it is not a practical stance for deploying it. Existing databases and 'legacy systems' will continue to survive and even to thrive; the partial order model will no more replace existing relational installations than did relational installations supplant IMS installations. A data model remains only a curiosity if it cannot gracefully adapt to existing problems, replacing older models where it can and cooperating with them otherwise.

How can the partial order model be introduced into the database community? One approach to deploying the model is to simply build a complete partial order database for some specific practical application that is not well served by traditional databases. The text and software communities have applications of just this sort, which were extensively investigated in previous chapters. By providing a complete implementation, we can demonstrate the potential of the model. By concentrating on non-traditional data, we support functionality that is not otherwise available, and we are not immediately competing with efficient, mature systems. Another advantage of a full implementation is that we have complete control over the system, and so we can more rigorously impose the model's restrictions and more fully exploit its features. Such a system is also self-reliant; it does not depend on any other systems.

The disadvantage of full implementation is that we must recreate

all the standard elements of database functionality—security, logging, transactions, query optimization, storage management—and we are unlikely to do these things as efficiently as they are done by more mature systems.[1] Object-oriented database companies have had to face this problem.

A second approach to deploying the model is to provide it as a layer on top of an existing database system. In this approach, users interact with the model, but its machinery is implemented as an application of some other database. In effect, we are building a wrapper, as it is still possible for the existing database system to handle transactions sent directly to it. The advantage of this technique is that we do not have to construct the functionality provided by the underlying system, and can take advantage of mature systems for transaction control and secondary storage management. If a clean interface exists between the partial order model and the underlying system, it may even be that the partial order system is transportable between different implementations of the underlying system. The disadvantage of this approach is impedance mismatch; the partial order layer is likely to introduce or rely on constructs that are not efficiently supported by the underlying model, or perhaps to exploit features in which implementations have semantic differences.

If layering is the chosen strategy, then we need to select an underlying system. The possibilities include relational, object-oriented, network, or hierarchical systems; each has certain strengths and weaknesses. Relational systems are attractive for their simplicity and wide availability. A partial order can easily be expressed as a binary relation, or perhaps as a series of totally ordered unary relations (each one holding a linear extension for a realizer of the partial order). The simplicity of the data representation is countered by the expected high cost of frequent joining to obtain the transitive closure.

Network and hierarchical databases are appealing because they support certain partial orders more or less directly (that is, they

1 On the other hand, because the partial order model appears able to describe at least some aspects of these database internals, we may be able to handle the problems more generally, if not more efficiently.

support those partial orders that are network or hierarchical structures). Such systems may also have other advantages; IMS, for example allows for secondary indexes (hence, multiple structures), and it can also provide excellent performance. The main disadvantage of these systems as an underlying layer is that they are only available in a limited number of environments, and are no longer being actively developed. Another disadvantage is the shrinking population of programmers who know these systems.

Object-oriented systems like SELF are appealing because they closely resemble a partial order on object instances. SELF is also a relatively efficient programming language; its developers spent considerable effort to introduce runtime efficiencies. One disadvantage of object-oriented systems is that they are too new to be significantly more mature than a completely new implementation of the partial order model. Another disadvantage is that there will be some temptation to 'reach' through the partial order layer to implement semantics directly in the objects, thus defeating the purpose of separating the system into layers.

A third approach to deploying the partial order model is to extend an existing data model with partial order operations, making a hybrid model. The user can then interact with either the partial order model or the existing model, and perhaps with a combination of the two. This approach in effect creates a federated database. For example, we could extend the relational model with partial order operations for ordered data sets. Many domain-specific models have followed the extension route; that is why the proposed SQL 3 standard includes extensions for multimedia and other non-traditional forms of data. One proposal for text is typical: a new non-atomic domain is defined ('text') and various domain-specific operators are employed to manipulate the internal structure of the domain (Blake *et al.* 95). Externally, the values of the domain can be treated as atomic data and manipulated with standard relational operators. The advantage of the federated approach is that we do not impact existing applications, and we leverage the system at the user's level; the user still employs many of the same tools that were used with the legacy system. The disadvantage of the approach is that the

hybrid is more complex than either model on its own, and the system as a whole becomes harder to understand and to support. The size of the proposed SQL 3 standard document is one hint about the limits of the extension-based approach.

A fourth approach to deploying the model is to step back from the mathematical definition and implement a simple, intuitive version of the model. In the case of partial orders, one might construct a model that has two concepts: unordered sets, and totally ordered sequences. This restricts the domain of partial orders to what are known as series-parallel orders. Such a restriction has the great virtue of simplicity. Simplicity means that the underlying system may be more easily understood by users and more quickly implemented, and may be able to take advantage of certain tricks to gain efficiency (series-parallel orders, for example, are known to be of dimension 2). The main disadvantage is that the system will probably be treated more as a kind of data structure than as a model, and so the peculiarities of a given implementation are likely to sneak into the modelling of information.

A final approach to deploying the model is to ignore the possibility of a practical implementation, and simply use the model as a theoretical stance for analyzing actual implementations that provide some or all of the features of the partial order model. This, the pedagogical approach, is the easiest and the least satisfying of all.

## DIRECTIONS FOR FUTURE INVESTIGATION

There is much that needs to be done to demonstrate the effectiveness of the partial order model. The first and best demonstration would be a working prototype: nothing succeeds like a successful implementation. In addition to an implementation, three areas ripe for work are storage structures, languages, and theory.

**storage structures** More practical experience is needed with storage structures before we can claim to understand how to implement partial orders in a reasonable manner. A key problem is deciding what types of partial orders are most common, how they are accessed, and what updates are likely.

Methods for using realizers to represent partial orders were outlined in some detail, but this analysis showed only that realizers are a reasonable option. We need lower bounds for most of the operators, and for the more expensive ones we need algorithms that can take advantage of useful special cases. Various techniques for compressing realizers (such as the proposed use of replacement strings) should also be investigated in more detail.

All of the algorithms for realizers implicitly assumed that they were in primary storage. It would be worthwhile to look at secondary storage costs and lower bounds. For example, Ullman and Yannakakis have shown that $O(n^3)$ I/O operations are needed to perform transitive closure (Ullman and Yannakakis 90).

The analysis of realizers did not include any work on the cost of updating partial orders, approaches to shared buffers, added redundancy for performance reasons, or the use of indexes to speed querying. Each of these areas could profit from more research effort.

**languages** Most advances in databases are accompanied by (some would say led by) new ideas in programming languages, with the three obvious examples being data typing, logic programming, and object-oriented programming. While it is not the case that the partial order model was inspired by Stott Parker's work in partial order programming, it is a nice coincidence that this model comes at roughly the same time and along similar lines to Parker's very interesting work in programming languages (Parker 92).

Parker's concept is of a programming language for managing constraints over a partially ordered domain. A *partial order programming problem* has the form

| | |
|---|---|
| minimize | $u$ |
| where | $v_1 \leq u_1, v_2 \leq u_2, \ldots$ |

In effect, it is a linear programming problem defined over a partially ordered domain. Among other contributions, Parker has shown that if the partial order programming problem is both continuous and monotone, then it well-defined and computable, and equivalent to the domains of functional and logic programming. Thus, partial order programming is Turing complete.

Parker's language is clearly a calculus, since one merely states the conditions that should obtain without specifying how the result is computed. Possessing both a calculus and an algebra for partial orders leads to the the natural question: what is the relationship between these two languages? One would like to prove (or disprove) some kind of completeness result for the two languages, as was done for relational algebra and relational calculus. If completeness is shown, one would like to construct algorithms for conversion of algebraic problems into calculus problems and vice versa. How to compare the two languages is not immediately obvious, since Parker's language deals with only one partially ordered domain.

It would also be interesting to consider the relationship between the partial order algebra, calculus, and a SELF-like (that is, object-oriented) view of partial orders. As with Parker's calculus, it is not immediately clear how to compare SELF to partial orders, since a SELF program is apparently a single partial order. If it were possible to show constructive transformations of programs from one language to the other, we would have a variety of additional options for implementation, and a variety of methods of explaining the functionality of partial orders.

**theory** Much more work can be done on theoretical aspects of the partial order algebra. We adduce only a few obvious issues here.

One question is whether the set of operators is sufficient for most queries. If not, we need to determine the necessary additional operators. Aggregation operations such as min, max,

and average appear to be natural candidates, although it is not clear if these are part of any basis set of operators, or whether they are reducible to some operations on some partial order that we have not yet identified.

A second area of investigation is studying restricted classes of partial orders—examples include lattices, series-parallel orders, and interval orders. If we have efficient techniques for recognizing and manipulating instances of these classes, we can exploit this information for more efficient processing of data. Many systems already 'recognize' and 'manipulate' such classes with reduced operators. Some text searching systems, for example, manipulate the class of totally ordered data, although they generally do not do so in an algebraic fashion.

A third area of investigation is decomposability. Database design is concerned with the ways in which data can be decomposed into components that can be recombined to produce the original. There are several advantages to decomposition:

- a decomposition may have desirable update properties, as for example when components are produced by a normalization process.
- a decomposition may have better access characteristics, if the components are distributed, or if their smaller size requires less I/O.
- a decomposition can assist users in understanding a domain, because the components can be separately perceived.

How to decompose partial order databases properly is an important area for further work. The examples in previous chapters have shown some decompositions that seem plausible, but they were derived intuitively, which is both a time-consuming and error-prone process. A more rigorous decomposition technique awaits. One approach is to start by checking for t-irreducibility and primality; these are formal limitations on decomposition. Given that the candidate database is composite or reducible, then we need a program that can determine

the possible decompositions, so that we can choose between them based on other criteria, such as efficiency or comprehensibility.

A fourth area for investigation is formalizing the connections between versioning, time, redundancy, transactions, update, dependency, sharing, and inheritance. These concepts are closely related, as can be seen by their (occasionally camouflaged) appearance in so many of our examples. A version system for files, for example, can also be viewed as a temporal database for those files. If we store all the versions of the file, then we will have redundancy. If we recognize the dependencies between the versions—the parts that do not change between one version and another—then we can eliminate redundancy by storing only the differences between each version. This form of redundancy control, however, can itself be viewed as a log of transactions that could have been used to produce the versions. If the file has multiple variants (recall that variants are incomparable derivatives), then the variants can be viewed as sharing their common ancestors. Finally, in some cases versions of a file may be thought of as specializations, and the version order can be treated as an inheritance structure. This simple example shows that there are important, and as yet unformalized relationships between the concepts of dependency, sharing, update, and redundancy.

## CONCLUSIONS

The essence of data modelling is using abstraction to better understand some part of the world. Good data models are simple enough to be understood by nonspecialists, they are general enough to have wide applicability, and they are formal enough to support reasoned analysis. No data model succeeds solely on the strength of its formalism or its intuitive qualities. Petri nets are as formal as one could wish but they have made little impact. A decade's experience in over-optimistic object-oriented modelling has shown that trying to track the language of a given domain does not solve all modelling

problems.

The work in this thesis began with the following assumptions:

- order is a fundamental property of information
- disorder should be explicitly recognized
- typeless structure is as important as typed structure
- data has more than one organization
- algebra is the natural means to manipulate ordered sets

The partial order model is a natural outcome of these assumptions. The model captures both order and disorder, and does so in a mathematically sound manner. The model is by definition typeless, although types can be used. The model supports multiple structures on a data set, and it provides algebraic mechanisms to combine and compare these structures. Through several examples we have seen that the partial order algebra is capable of concisely expressing solutions to domain-specific problems. Finally, we have seen that the partial order model is capable of managing some of the meta-data of a database, as well as its data.

Perhaps the most significant contribution of a data model is to show that its structures are not simply artifacts of a peculiar kind of data organization, but that instead they are deep structural truths about many kinds of information. Normalization in the relational model, for example, was not simply a clever method for decomposing tables, it was a formal approach to redundancy control through management of typed data dependencies. The partial order model shows promise for making a comparable contribution: showing that both typed and untyped dependencies can be captured in the single mechanism of order. Consequently, the partial order model suggests that it is possible to unify a very large class of phenomena.

# 8

# References


**Abbott** *et al.* **89** C. Abbott, D.M. Levy, M. Dixon, M. Weiser, 'Document Interchange and Reconfigurable Editors' *WOODMAN '89 Workshop on Object-Oriented Document Manipulation* p. 205-214, Rennes, France (May 29-31, 1989).

**Barnard** *et al.* **88** D. Barnard, R. Hayter, M. Karababa, G. Logan, J. McFadden, 'SGML-Based Markup for Literary Texts: Two Problems and Some Solutions' *Computers and the Humanities* **22** p. 265-276 (1988).

**Beach 85** R.J. Beach, 'Setting Tables and Illustrations with Style', CS-85-45, Department of Computer Science, University of Waterloo, Waterloo, Ontario (May 1985).

**Beach 86** R.J. Beach, 'Tabular Typography' *Proceedings of the International Conference on Text Processing and Document Manipulation*, p. 18-33, University of Nottingham (April 14-16, 1986).

**Beeri** *et al.* **78** C. Beeri, P.A. Bernstein, and N. Goodman, 'A Sophisticate's Introduction to Database Normalization Theory', *4th VLDB*, p. 113-124, West Berlin, Germany (1978).

**Borison 89** E.A. Borison, 'Program Changes and the Cost of Selective Recompilation', CMU-CS-89-205, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania (July 1989).

**Breitbart** *et al.* **92** Y. Breitbart, H. Garcia-Molina, A. Silberschatz, 'Overview of Multidatabase Transaction Management' *Proceedings of 1992 CAS Conference*, IBM Canada Laboratory, p. 23-56, Toronto, Ontario (November 9-12, 1992).

**Bringhurst 92** R. Bringhurst, *The Elements of Typographic Style* Hartley & Marks, Vancouver, British Columbia (1992).

**Brodie 84** M.L. Brodie, 'On the Development of Data Models' in *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, p. 19-47, Springer-Verlag, New York, N.Y. (1984).

**Cargill 79** T. A. Cargill, 'A View of Source Text for Diversely Configurable Software' CS-79-28, Department of Computer Science, University of Waterloo, Waterloo, Ontario (1979).

**Chambers** *et al.* **91** C. Chambers, D. Ungar, B. Chang, U. Holzle, 'Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF', *Lisp and Symbolic Computation* **4** (3) (1991).

**Clifford** *et al.* **93** J. Clifford, A. Croker, A. Tuzhilin, 'On the Completeness of Query Languages for Grouped and Ungrouped Historical Data Models', *Temporal Databases: Theory, Design, and Implementation*, p. 496-533, Benjamin/Cummings Publishing Company, Redwood City, California (1993).

**Codd 70** E.F. Codd, 'A Relational Model of Data for Large Shared Data Banks' *Communications of the ACM* **13** (6) p. 377-387 (June 1970).

**Codd 90** E.F. Codd, *The Relational Model for Database Management: Version 2* Addison-Wesley, Reading, Massachusetts (1990).

**Davey and Priestley 90** B.A. Davey, H.A. Priestley, *Introduction to Lattices and Order* Cambridge University Press, Cambridge, England (1990).

**de Rose** *et al.* **90** S.J. DeRose, D.J. Durand, E. Mylonas, A.H. Renear, 'What is Text, Really?' *Journal of Computing in Higher Education* **1** (2) p. 3-26 (Winter 1990).

**Dilworth 50** R.P. Dilworth, 'A Decomposition Theoreom for Partially Ordered Sets', *Ann. Math.* **51** p. 161-165 (1950).

**Feldman 79** S.I. Feldman, 'Make—A Program for Maintaining Computer Programs' *Software—Practice and Experience* **9** p. 255-265 (1979).

**Fishburn 85** P.C. Fishburn, *Interval Orders and Interval Graphs*, John Wiley & Sons, New York, N.Y. (1985).

**Fowler 90** G. Fowler, 'A Case for `make`' *Software—Practice and Experience* **20** (S1) p. S1/35-S1/46 (June 1990).

**Fowler 85** G.S. Fowler, 'A Fourth Generation Make' *Proceedings of the USENIX Summer Conference*, p. 159-174, Portland, Oregon (June 1985).

**Furtado and Kerschberg 77** A.L. Furtado, L. Kerschberg, 'An Algebra of Quotient Relations' *SIGMOD '77* p. 1-8, Toronto, Ontario (August 3-5, 1977).

**Garey and Johnson 79** M. R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., San Francisco, California (1979).

**Garfinkel** *et al.* **94** *The Unix Haters' Handbook*, edited by S. Garfinkel, D. Weise, and S. Strassman, IDG Books Worldwide, Inc. Indianapolis, Indiana, (May 1994).

**Gilula 94** M. Gilula, *The Set Model for Database and Information Systems*, Addison-Wesley, Reading Massachusetts (1994).

**Ginsburg and Hull 83** S. Ginsburg, R. Hull, 'Order Dependency in the Relational Model *Theoretical Computer Science* **26** p. 149-195 (September 1983).

**Gischer 84** J.L. Gischer, 'Partial Orders and the Axiomatic Theory of Shuffle' STAN-CS-84-1033, Department of Computer Science, Stanford University, Stanford, California (December 1984).

**Goodman 76** N. Goodman, *Languages of Art*, Hackett Publishing Co. (1976).

**Harris 86** R. Harris, *The Origin of Writing*, Gerald Duckword & Co., London, England (1986).

**Hart 83** H. Hart, *Hart's Rules for Compositors and Readers at the University Press Oxford, 39th edition*, Oxford University Press, Oxford, England (1983).

**Katz 90** R. H. Katz, 'Toward a Unified Framework for Version Modeling in Engineering Databases' *Computing Surveys* **22** (4) p. 375-408 (December 1990).

**Kerschberg** *et al.* **76** L. Kerschberg, A. Klug, D. Tsichritzis, 'A Taxonomy of Data Models' *2nd VLDB*, Brussels, Belgium p. 43-64 (September 8-10, 1976).

**Lee 83** T.T. Lee, 'An Algebraic Theory of Relational Databases' *The Bell System Technical Journal* **62**(10) p. 3159-3204 (December 1983).

**Lesk 79** M.E. Lesk, 'Tbl—A Program to Format Tables' Bell Laboratories, Murray Hill, New Jersey (January 16, 1979).

**Lorentzos 93** N.A. Lorentzos, 'The Interval-Extended Relational Model and its Application to Valid-Time Databases *Temporal Databases: Theory, Design, and Implementation*, p. 67-91, Benjamin/Cummings Publishing Company, Redwood City, California, (1993).

**Parker 87** D.S. Parker, 'Partial Order Programming', Technical Report, Computer Science Department, University of California, Los Angeles, California (1987).

**Parker 89** D.S. Parker, 'Partial Order Programming' *Proceedings of the 16th ACM Conference on Principles of Programming Languages*, p. 260-266, Austin, Texas (1989).

**Plaice and Wadge 93** J. Plaice, W. W. Wadge, 'A New Approach to Version Control' *IEEE Transactions on Software Engineering* **19** (3) p. 268-276 (March 1993).

**Pratt 86** V. Pratt, 'Modelling Concurrency with Partial Orders' STAN-CS-86-1113, Department of Computer Science, Stanford University, Stanford, California (June 1986).

**Raymond and Tompa 88** D.R. Raymond, F.W. Tompa, 'Hypertext and the Oxford English Dictionary' *Communications of the ACM* **31** (7) p. 871-879 (July 1988).

**Raymond 92** D.R. Raymond, 'Flexible Text Display with Lector' *IEEE Computer* **25** (8) p. 49-60 (August 1992).

**Raymond and Tompa 92** D.R. Raymond, F.Wm. Tompa 'Applying Database Dependency Theory to Software Engineering', CS-92-56, Department of Computer Science, University of Waterloo, Waterloo, Ontario (December 31, 1992).

**Raymond** *et al.* **92** D.R. Raymond, F.W. Tompa, D. Wood, 'Markup Reconsidered' *First International Workshop on Principles of Document Processing* Washington, D.C. (October 21-23, 1992).

**Raymond 93** D.R. Raymond, 'Visualizing Texts' *9th Annual Conference of the UW Centre for the New Oxford English Dictionary* p. 20-33, Oxford, England (September 27-28, 1993).

**Raymond** *et al.* **96** D.R. Raymond, F.W. Tompa, D. Wood, 'From Data Representation to Data Model: Meta-Semantic Issues in the Evolution of SGML' *Computer Standards and Interfaces* **18** p. 25-36 (1996).

**Read** *et al.* **92** R.L. Read, D.S. Fussell, A. Silberschatz, 'A Multi-Resolution Relational Data Model' *18th International VLDB*, p. 139-150, Vancouver, British Columbia (August 1992).

**Salminen and Tompa 92** A. Salminen, F.W. Tompa, 'Data Modelling with Grammars' unpublished manuscript, Department of Computer Science, University of Waterloo, Waterloo, Ontario (February 1992).

**Sandhu 88** R.S. Sandhu, 'The NTree: A Two Dimension Partial Order for Protection Groups' *ACM Transctions on Computer Systems* **6** (2) p. 197-222 (May 1988).

**Shin** *et al.* **94** K.H. Shin, K. Kobayashi, A. Suzuki, 'Tafel Musik: Formatting Algorithm for Tables' *Principles of Document Processing '94*, Seeheim, Germany (April 11-12, 1994).

**Smith and Smith 77** J.M. Smith and D.C.P. Smith, 'Database Abstractions: Aggregation and Generalization' *ACM Transactions on Database Systems* **2**(2) p. 105-133 (June 1977).

**Somogyi 87** Z. Somogyi, 'Cake: A Fifth Generation Version of Make' *Australian Unix System User Group Newsletter* **7** (6) p. 22-31 (April 1987).

**Stanley 86** R.P. Stanley, *Enumerative Combinatorics*, Wadsworth and Brooks/Cole (1986).

**Summers 91** J.A. Summers, *Precedence-Preserving Abstractions for Distributed Debugging*, M.Math Thesis, University of Waterloo, Waterloo, Canada, 1991.

**Tansel** *et al.* **93** *edited by* A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass, *Temporal Databases: Theory, Design, and Implementation* Benjamin/Cummings Publishing Company, Redwood City, California (1993).

**Taylor and Coffin 94** D.J. Taylor, M.H. Coffin, 'Integrating Real-Time and Partial-Order Information in Event-Driven Displays' CASCON '94, IBM Canada Centre for Advanced Studies, Toronto, Ontario, p. 157-165 (October 31-November 3, 1994)

**Terribile 94** M.A. Terribile, *Practical C++* McGraw-Hill, New York, N.Y. (1994).

**Tichy 82** W.F. Tichy, 'A Data Model for Programming Support Environments and its Application', *Automated Tools for Information System Design* p. 31-48, North-Holland, Amsterdam, The Netherlands (1982).

**Tichy 85** W.F. Tichy, 'RCS—A System for Version Control' *Software—Practice and Experience* **15** (7) p. 637-654 (July 1985).

**Tompa 89** F.W. Tompa, 'What is (tagged) text?' *Proceedings of the Fifth Annual Conference of the UW Centre for the New Oxford English Dictionary*, p. 81-93, UW Centre for the New OED, University of Waterloo, Waterloo, Ontario (September 18-19, 1989).

**Tompa and Raymond 91** F.W. Tompa, D.R. Raymond, 'Database Design for a Dynamic Dictionary' *Research in Humanities Computing I, Papers from the 1989 ACH-ALLC Conference*, p. 257-272, Oxford University Press, Oxford, England (September 1991).

**Trotter 92** W.T. Trotter, *Combinatorics and Partially Ordered Sets: Dimension Theory* Johns Hopkins University Press, Baltimore, Maryland (1992).

**Ungar and Smith 91** D. Ungar, R.B. Smith, 'SELF: The Power of Simplicity' *Lisp and Symbolic Computation* **4**(3) (1991).

**Ullman and Yannakakis 90** J.D. Ullman, M. Yannakakis, 'The Input/Output Complexity of Transitive Closure' *SIGMOD '90*, Atlantic City, New Jersey, p. 44-53 (May 23-25, 1990).

**Ungar** *et al.* **91** D. Ungar, C. Chambers, B.W. Chang, U. Hölzle, 'Organizing Programs Without Classes', *Lisp and Symbolic Computation* **4**(3) (1991).

**Vanoirbeek 92** C. Vanoirbeek, 'Formatting Structured Tables', *Proceedings of EP '92* p. 291-309 Cambridge, England (1992).

**Walden 84** K. Walden, 'Automatic Generation of Make Dependencies', *Software—Practice and Experience* **14** (6) p. 575-585 (June 1984).

**Wang and Wood 93a** X. Wang, D. Wood, 'Tabular Abstraction for Tabular Editing and Formatting', *Proceedings of 3rd International Conference for Young Computer Scientists* Tsinghua University Press, Beijing, China (1993).

**Wang and Wood 93b** X. Wang, D. Wood, 'An Abstract Model for Tables', *Proceedings of the 1993 TEXUser's Group Meeting* **14**(3) p. 231-237, Birmingham, England (July 26-30, 1993).

**Waters 89** R.C. Waters, 'Automated Software Management Based on Structural Models' *Software—Practice and Experience* **19** (10), p. 931-955 (October 1989).

**Wille 82** R. Wille, 'Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts', in *Ordered Sets*, ed. by Ivan Rival, NATO ASI Series **83**, Reidel, Dordrect, p. 445-470 (1982).

**Williams and Abbott 89** W.P. Williams, C.S. Abbott, *An Introduction to Bibliographical and Textual Studies, 2nd. Ed.* Modern Language Association of America, New York, N.Y. (1989).

**Yannakakis 82** M. Yannakakis, 'On the Complexity of the Partial Order Dimension Problem', *SIAM J. Alg. Discr. Meth.* **3**, p. 351-358 (1982).