ASPECTS OF PARTIAL INFORMATION IN DATABASES

Leonid Libkin

A DISSERTATION
in
COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.
1994

Peter Buneman
Supervisor of Dissertation

Mark Steedman
Graduate Group Chairperson

Averbakh–Kotov

Zurich, September 23, 1953

1.	d2–d4	Ng8–f6	27.	g4:f5	g6:f5
2.	c2–c4	d7–d6	28.	Rg1–g2	f5–f4
3.	Ng1–f3	Nb8–d7	29.	Be3–f2	Rf7–f6
4.	Nb1–c3	e7–e5	30.	Nc3–e2	Qd7:h3+!!
5.	e2–e4	Bf8–e7	31.	Kh2:h3	Rf6–h6+
6.	Bf1–e2	0–0	32.	Kh3–g4	Ng8–f6+
7.	0–0	c7–c6	33.	Kg4–f5	Nf6–d7
8.	Qd1–c2	Rf9–e8	34.	Rg2–g5	Rb8–f8+
9.	Rf1–d1	Be7–f8	35.	Kf5–g4	Nd7–f6+
10.	Ra1–b1	a7–a5	36.	Kg4–f5	Nf6–g6+
11.	d4–d5	Nd7–c5	37.	Kf5–g4	Ng8–f6+
12.	Bc1–e3	Qd8–c7	38.	Kg4–f5	Nf6:d5+
13.	h2–h3?	Bc8–d7	39.	Kf5–g4	Nd5–f6+
14.	Rb1–c1	g7–g6	40.	Kg4–f5	Nf6–g8+
15.	Nf3–d2	Ra8–b8	41.	Kf5–g4	Ng8–f6+
16.	Nd2–b3	Nc5:b3	42.	Kg4–f5	Nf6–g8+
17.	Qc2:b3	c6–c5	43.	Kf5–g4	Be7:g5
18.	Kg1–h2	Kg8–h8	44.	Kg4:g5	Rf8–f7
19.	Qb3–c2	Nf6–g8	45.	Bf2–h4	Rh6–g6+
20.	Be2–g4	Ng8–h6	46.	Kg5–h5	Rf7–g7
21.	Bg4:d7	Qc7:d7	47.	Bh4–g5	Rg6:g5+
22.	Qc2–d2	Nh6–g8	48.	Kh5–h4	Ng8–f6
23.	g2–g4?!	f7–f5	49.	Ne2–g3	Rg5:g3
24.	f2–f3	Bf8–e7	50.	Qd2:d6	Rg3–g6
25.	Rd1–g1	Re8–f8	51.	Qd6–b8+	Rg7–g8
26.	Rc1–f1	Rf8–f7		**0-1**	

Preface

In most applications, information stored in databases is not complete. There are various sources of partiality of information. First, some information may be missing. For example, in a database of employees some salaries may not be recorded. Second source of partiality is conflicts that occur when different databases are merged and they contradict each other. Another source of partiality is asking queries against several databases simultaneously. Even if all databases are complete, in most cases answers to such queries can only be approximated.

The field of partial information in databases has not received the attention that it deserves. Most work on partial information in databases asks which operations of standard languages, like relational algebra, can still be performed correctly in the presence of simple forms of partial information like missing values. We believe that the problem should be looked at from another point of view: the semantics of partiality must be clearly understood and it should give us new design principles for languages for databases with partial information.

The main goals of this thesis are to develop new analytical tools for studying partial information and its semantics, and to use the semantics of partiality as the basis for design of query languages.

This work should be distinguished from the body of work on partial information in artificial intelligence. In most artificial intelligence applications the main concern is the design of models for specific applications that could eventually lead to fast algorithms. In this thesis we are interested in representation and querying partial information in database systems. Consequently, we concentrate on general purpose solutions that are effectively implementable in the context of database query languages and provide a flexible basis for future modeling challenges.

We present a common semantic framework for various kinds of partial information which can be applied in a context more general than the flat relational model. This semantics is based on the idea of ordering objects in terms of being more informative. Such ordered semantics, which uses the ideas from the semantics of programming languages, cleanly intergrates all kinds of partial information and serves as a tool to establish connections between them. In addition, by analyzing mathematical properties of partial data, it is possible to find operations naturally associated with it that can be turned into programming language constructs. More precisely, having defined semantic domains for various kinds of collections of partial data, we can describe them as free algebras, and this gives us the desired sets of operations.

Various queries over partial databases can be formulated in terms of approximations. By analyzing different situations in which a precise answer can not be obtained for a query asked against several databases, we propose a classification of constructs that can be used to model approximations. Using the same approach as for collections, we define the semantics and the orderings of approximations and show their intimate connection with combination of disjunctive and conjunctive sets (so-called or-sets).

We discuss languages for databases with partial information. We follow the recently proposed approach to the design of query languages based on developing languages around operations naturally associated with the type constructors of their type systems. Such operations usually come from the universality properties of semantic domains of those types. A language for sets and or-sets is introduced and normalization theorem is proved. It allows to incorporate semantics into the language and to distinguish two levels of querying: structural and conceptual. We then use the semantic connection between sets, or-sets and approximations and show how to use this language to work with approximations. Languages for multisets are also discussed.

The language for sets and or-sets has been implemented on top of Standard ML. Its implementation is described and and two typical examples of queries are given. One deals with querying incomplete databases which often occur in computer aided design applications. The other example deals with querying independent databases.

Summing up, this thesis develops a new approach to dealing with partial information in databases. This approach is based on deep understanding of semantics of various kinds of partial information that may occur in many different contexts, and on designing languages naturally associated with partial information, rather than adapting existing languages for complete databases.

Acknowledgements

It has become a tradition to start the acknowledgement section of a dissertation by thanking the advisor. I shall certainly do so, but I want to keep expressing gratitude in the chronological order. It was five years ago that I finished a manuscript entitled *"Abstract Convexities in Lattices and Semilattices"* which was supposed to be my PhD thesis in mathematics. I never got a PhD in math; my thesis was disassembled and published in a number of papers, and shortly after finishing it I played my own Qd7:h3+ by switching to computer science. The way from lattices and convexity to partial information and query languages was not easy or fast: it lasted five years, went through five countries and, as for any critical move, there never will be a proof of correctness. This move is truly an example of partial information – we will never know for sure if it is correct, and this is why the game is taken as the epigraph for the thesis[1]. But there is still something that I can assert without a shadow of doubt: I would never be where I am today without having written that manuscript. And I would have never achieved that without help of a number of people.

Most of all I would like to thank my parents for their support that allowed me to do research when it was next to impossible. I also want to thank them for showing me that 5,000 miles is not an obstacle for their support and encouragement that I feel every day. I am immeasurably grateful to Ilya Muchnik, my advisor from 1985 till 1989, with whom I wrote thirteen papers, for being responsible for my *real* undergraduate education and for collaborating with me on so many projects. I want to thank János Demetrovics for inviting me to visit Budapest in 1988 and the Soviet authorities for unexpectedly allowing me to go; I recall that it was a pleasant surprise for me and perhaps a move like g2-g4 on their part. János and E.T. Schmidt from the Institute of Mathematics in Budapest convinced me that I should stop writing in Russian and helped me write my first English papers.

János gave an initial impulse to my transition to computer science. But it would not have been complete and I would not have ended up at Penn without help and good advice that came at the crucial moment. I would like to thank Georg Gottlob, Paolo Atzeni, Joachim Biskup and Victor Vianu for their help during my short but very important stay in Europe in 1989–1990. I also would like to thank Mila and Yuri Chekanovsky who helped complete this transition and

[1] The game is taken from Bronshtein [28].

supported me when I arrived to the US.

Having done with the past, let's move to the present. I am very grateful to my advisor Peter Buneman for numerous comments, ideas, suggestions and for being largely responsible for the development of the main principles upon which this thesis is based. I am also grateful to the members of the Penn database group (a.k.a. the "Tuesday Club"): Susan Davidson, Wenfei Fan, Anthony Kosky, Rona Machlin, Dan Suciu, Val Tannen and Limsoon Wong. Val has been extremely helpful since I came here. I can not think of a single piece of my database work in which in some way his influence is not present. I wrote seven papers with Limsoon and our collaboration was very pleasant and fruitful for me (and I hope for him as well). Dan's comments often helped improve those papers and consequently this thesis. Achim Jung from Darmstadt University gave me the first lessons in domain theory when we wrote our joint paper. He invited me to Darmstadt in October 92 where I learned about the problem of approximation in databases. I also would like to thank Hermann Puhlmann for his hospitality during my stay in Darmstadt. Carl Gunter was always very helpful, especially when I was trying to understand some fine points in the semantics of programming languages, before his excellent book appeared. Elsa Gunter has helped me a lot as my AT&T "mentor". She also influenced the implementation of OR-SML, the language that I built during my three-month stay at AT&T Bell Laboratories in 1993. I thank Paris Kanellakis for his comments on an earlier version of the thesis that have led to many improvements.

Many people read my papers upon which this thesis is based, and made useful suggestions. I was also very lucky to have presented the material of this thesis before very responsive audiences, and some penetrating questions asked during or after my talks influenced the contents of the thesis. It is impossible to mention all names, and I apologize for unintentionally omitting some people. For their comments, suggestions, questions and encouragement I thank Susan Davidson, János Demetrovics, Jean Gallier, Stephane Grumbach, Rick Hull, Tomasz Imielinski, Paris Kanellakis, Anthony Kosky, Alberto Mendelzon, Dale Miller, Inderpal Mumick, Shamim Naqvi, Teow-Hin Ngair, Atsushi Ohori, Jan Paredaens, Hermann Puhlmann, Jon Riecke, Anna Romanowska, Bill Rounds, Bernhard Thalheim, Kumar Vadaparty, Bennet Vance, Jan Van den Bussche, Dirk Van Gucht, Victor Vianu, Steve Vickers and Scott Weinstein. I thank Paul Taylor for his diagram macros, and Nan Biltz and Michael Felker for being a buffer between me and UPenn bureaucracy; all three saved me hours and perhaps even days that I could then use for research.

Finally, I gratefully acknowledge financial support provided by AT&T Doctoral Fellowship and NSF Grant IRI-90-04137.

Contents

Chapter 1

The Problem of Partial Information in Databases

In this chapter we give a brief introduction into the theory of partial information in databases. In the first section, we recall some major developments in the field and consider various types of null values which are used most often to introduce partial information into relational databases. We discuss representation systems and problems with query evaluation. These are the only two subfields in which significant progress has been made. We review extensions of partial information to the complex object (nested relational) data model.

Then we consider two different kinds of partial information that have not received the same amount of attention from the database community. One is disjunctive information represented via or-sets; the other is a number of constructions of similar structure called approximations. Having surveyed the results known for these two kinds of partiality, we summarize open problems that we solve or demonstrate new approaches towards solving, and outline the structure of the thesis.

1.1 Null values

1.1.1 Early work on null values in databases

Any practical database management system must deal with the concept of partial information. It was observed by Maier [113] that the fact that the structure of information may not fit the relational model is not its only major limitation. Equally important is the reason that even if information does fit the model, part of it may be missing for some reason. While there has been a flurry of activity lately in trying to go beyond the standard relational model, one can not say the same about partial information. The topic is still unexplored, there are few significant results and there is no clear understanding of what partiality really means.

Soon after Codd introduced his relational model, people realized that in real applications not all values may be present. For example, in a simple relation below that might be a part of a university or a corporation database, some values are missing and the symbol **ni** (no information) is used. Note that there could be several different reasons for using **ni**. For example, a person may not have a phone, or may have a phone but the number is unknown (for example, he may have forgotten it while filling out a form which was later entered in a database), or there could be no information whatsoever (if a clerk was entering the data and did not know anything about the phone in a particular office).

Name	Salary	Room	Telephone
John	15K	075	**ni**
Ann	17K	**ni**	**ni**
Mary	**ni**	351	x-1595

In 1975 Codd [39] perhaps did not consider it as a serious problem and suggested a simple solution: a fact about a tuple is either true (1) or false (0) or neither ($\frac{1}{2}$) which is the case when we do not have a complete information. However, a few years later, Grant [63] showed that Codd's solution leads to wrong results if we are to select certain tuples from the database. He proposed an alternative solution which was, in fact, introduction of the Skolem constants for nulls, formally studied by Biskup [23] a few years later.

The example given by Grant [63] and Codd [40] is essentially the following. Suppose we have a person whose name is in a database but salary is unknown, as for Mary in the above example. Suppose that we want to partition the table into two: T_1 containing employees with salaries less than 15K and T_2 of employees with salaries at least 15K. Of course, we believe that $T_1 \cup T_2$ should produce the original table back. But as a matter of fact, according to Codd's query evaluation algorithm in the presence of null values [39], Mary will not be included in T_1 nor in T_2.

Still, one very important observation was made in Codd [40]. Since every null value can be replaced by a non-null value, each relation with nulls is represented by a *set* of relations without partial information. Moreover, this set could be considered as the *semantics* of the given incomplete relation. Thus, the most important lesson that we learn from the early work on partial information is that there is a need in better mathematical models for partial information and in better understanding of its semantics.

In the late 70s and early 80s there were three major developments in the theory of partial information. First, the idea to use orderings as a means to express partiality emerged. Second, a rather rudimentary approach to disjunctive information was developed and an attempt was made towards a design of a query language specifically for partial information. Third, the distinction between various assumptions on partiality was made and it was shown how those assumptions lead to different semantics and query evaluation algorithms. Let us consider all three.

Orderings and partial information

We believe that the idea of expressing partiality of information by means of orderings is due to Vassiliou [172]. Two years after his initial work, this idea was further developed by Biskup [23].

As a simple example, consider values that may occur in a database. Then **ni** is more partial, or less informative, than any nonpartial value v such as 15K or 'Mary'. Therefore, we impose an order according to which $\mathbf{ni} \leq v$ for any nonpartial value v.

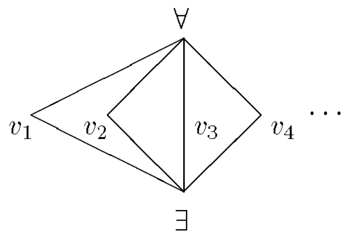
Since databases are obtained by applying record and set constructors, we need to extend the orderings respectively. For records the most natural way to do it is componentwise. For records with fields labeled by $l_1, \ldots, l_n$, we define

$$[l_1 : v_1, \ldots, l_n : v_n] \leq [l_1 : v_1', \ldots, l_n' : v_n'] \quad \text{iff} \quad \forall i = 1, \ldots, n : \ v_i \leq v_i'$$

For sets there are various ways to extend a partial order, and typically the following one, perceived as a generalized subset ordering, was considered:

$$X \sqsubseteq Y \text{ iff } \forall x \in X \ \exists y \in Y : \ x \leq y$$

Let us briefly consider two of the early works dealing with ordering on objects. Biskup [23] considered two null values. One is $\exists$ and its meaning is the same as **ni** in the example above: there is no information about the value of an attribute and there exists a complete value that can be substituted for it. The other is a somewhat less natural value $\forall$ meaning that *any* value is a right substitution for it. For instance, a record $[l_1 : v_1, l_2 : \forall]$ is just a short notation for a set of records $[l_1 : v_1, l_2 : v]$ for all possible values v. That is, $\forall$ is not really a null value. This is further confirmed by the ordering imposed on values:



Biskups's paper made two major contributions to the field. First, he showed that truth of certain logical formulae about databases with added $\exists$ and $\forall$ values is intimately connected with the ordering. Second, he showed how to evaluate some of the standard database operations in the presence of those values.

Another approach to incorporating orderings as a means to express partiality into the relational model was proposed by Zaniolo [181]. He considered one kind of null, **ni**, and defined the ordering on tuples and sets in a way similar to the one given above; the only difference is that he allowed to compare tuples over different sets of attributes by inserting nulls in the missing columns. For example, a tuple [Name $\Rightarrow$ 'Joe', Age $\Rightarrow$ 25] is less informative than [Name $\Rightarrow$ 'Joe', Age $\Rightarrow$ 25, Salary $\Rightarrow$ 15K] because the former is extended to [Name $\Rightarrow$ 'Joe', Age $\Rightarrow$ 25, Salary $\Rightarrow$ **ni**], which is less informative than the latter under the componentwise ordering.

The notion of being more informative is extended from tuples to relations by the ordering given above, that is, $R_1 \sqsubseteq R_2$ iff $\forall t_1 \in R_1 \exists t_2 \in R_2 : \ t_1 \leq t_2$. This is a preorder, and it might be the case that both $R_1 \sqsubseteq R_2$ and $R_2 \sqsubseteq R_1$ hold; in this case R_1 and R_2 are information-wise equivalent and we write $R_1 \cong R_2$. By an *x-relation* Zaniolo means an equivalence class of relations with respect to $\cong$; an equivalence class of a relation R is denoted by $\hat{R}$. It is easy to express the generalized notion of a tuple t belonging to an x-relation $\hat{R}$ using the following fact: $t \in R'$ for some $R' \in \hat{R}$ iff $t \leq t'$ for some $t' \in R$. We use the notation $t \hat{\in} \hat{R}$ for this notion of being an element. Then one can redefine the union, intersection and difference on equivalence classes $\hat{R}_1$ and $\hat{R}_2$ as equivalence classes given by the following relations: $\{t \mid t \hat{\in} \hat{R}_1 \text{ or } t \hat{\in} \hat{R}_2\}$, $\{t \mid t \hat{\in} \hat{R}_1 \text{ and } t \hat{\in} \hat{R}_2\}$, $\{t \mid t \hat{\in} \hat{R}_1 \text{ and } \neg(t \hat{\in} \hat{R}_2)\}$ respectively.

Defining join is slightly trickier. First we say that two tuples t_1 and t_2 are *joinable* if, for any common attribute A, either in one of the two the A-value is **ni** or in both the A-values coincide. Since any two tuples can be viewed as tuples over the same set of attributes, we define the *join* of t_1 and t_2 of two joinable tuples by taking its A-value to be **ni** if both A-values in t_1 and t_2 are **ni**, or v is either A-value of t_1 is v or A-value of t_2 is v and $v \neq$ **ni**. For example,

$$[\text{Name} \Rightarrow \text{John}, \text{Age} \Rightarrow 25] \vee [\text{Name} \Rightarrow \text{John}, \text{Room} \Rightarrow 76] = [\text{Name} \Rightarrow \text{John}, \text{Age} \Rightarrow 25, \text{Room} \Rightarrow 76]$$

The reason is that both tuples are first extended by adding **ni** to the missing fields; then they are found to be joinable and then the join is taken. Now, given a set X of attributes, a *join* of two equivalence classes of relations $\hat{R}_1$ and $\hat{R}_2$ on X is defined by $\hat{R}_1 \bowtie_X \hat{R}_2 = \hat{R}$ where

$$R = \{t_1 \vee t_2 \mid t_1 \hat{\in} R_1, \quad t_2 \hat{\in} R_2, \quad t_1 \text{ and } t_2 \text{ are total on } X\}$$

Zaniolo [181] showed that the algebra thus defined can be used to query databases with partial information. In particular, he showed how to represent universal quantification and negation in queries.

Disjunctive information and query languages

In his classical papers, Lipski [109, 110] introduced two very important concepts that have influenced the theory of partial information ever since.

First, he proposed a special data model for partial information. This data model[1] is based not on null values but rather on assigning sets to objects and attributes. The idea is that for a given object x and a given attribute a, the value that x may have on a is taken from this assigned set X_x^a. This data model is the first instance of the use of *disjunctive information* in the database literature dealing with partial information. Disjunctive information is of special importance in this thesis and we shall discuss it later in details.

The second idea is based on the assumption that, in the presence of partial information, it is often impossible to evaluate queries precisely. Therefore, one should look for a reasonable approximation. We believe that Lipski [109] was the first to explicitly state the requirements that two bounds for a query Q constitute the answer for partial databases:

1. *The lower approximation* to the answer to Q, that is, those objects for we which one can conclude with certainty that they belong to the answer to Q.

2. *The upper approximation* to the answer to Q, that is, those objects for we which one can conclude that they may belong to the answer to Q.

However, it was not until ten years later that it was observed by Buneman, Davidson and Watters [31, 32] that those pairs of approximations may not only be regarded as results of query evaluation but may also be used as a representation mechanism for certain kinds partial data. Studying such approximation constructs is central to this thesis and we shall present a thorough study of them later.

However, another idea from Lipski's papers [109, 110] was overlooked by many. Unlike most other researchers, Lipski did not try to tie his data model to the standard relational data model and consequently he did not use languages like the relational algebra. Instead, he designed a special language, that arose quite naturally from the structures he was considering. Thus, it was the first instance (and unfortunately one of very few) when, instead of adapting existing languages to work with partial information, a new language was designed specifically for the purpose of working with partial information. This is the approach we advocate throughout this thesis and we shall see its many features later.

[1] Called in [109, 110] information systems, which is in direct conflict with the informations systems used in programming semantics [67].

Open worlds and closed worlds

It was observed by Reiter [142] that certain assumptions on the nature of partiality are to be made if we want to provide a notion of correctness of query evaluation algorithm. To explain those assumptions, consider the following relation:

Name	Salary	Room
ni	**ni**	076
Mary	17K	**ni**

Once all or some information about missing values (**ni**'s) is known, we have a relation that represents better knowledge than the one above. However, there is a question *what* values are allowed in the new relation?

One possible interpretation, called the *closed world assumption* or *CWA*, states that we can only improve our knowledge about records that are already stored but can not invent new ones. For example, it is legal to add any record $\boxed{v_1 \mid v_2 \mid 076}$ which improves upon the first record in the relation. It is also possible to add a record $\boxed{\text{Mary} \mid \text{17K} \mid 561}$ which is better knowledge than that represented by the second record in a database. However, it is *not* possible to add a record $\boxed{\text{Ann} \mid \mathbf{ni} \mid 561}$ as it does not improve any of the records already in the database. Indeed, it can not be seen as an improvement of the knowledge represented by the first record (since the office number is 561 and not 076), nor the second one (as the name is Ann, not Mary). That is, the database is *closed* for adding new records.

Contrary to that, the *open world assumption* or *OWA* allows adding records to database as well as improving already existing records. Under the open world assumption, adding any record considered above to the database is perfectly legal. That is, the database is *open* for adding new records.

There is another interpretation of the CWA and the OWA. Facts stored in a database are presumed to be positive facts. Then, under the CWA, we assume that if a fact is not represented in the database, then it is not true, i.e. we have a perfect picture of the world and nothing can be added to it. Under the OWA, this is not the case and not having a fact stored in a database does not tell us whether it should or should not be there.

To summarize, Figure 1.1 shows how to replace missing values according to both assumptions.

Reiter [142] defined the concept of a CWA answer to a query. He proved that minimal CWA answers contain precisely one tuple, that CWA query evaluation distributes over intersection and union, and that for a database that is consistent with the family of negations of facts stored in it, the CWA evaluation algorithm gives exactly the same result as the OWA evaluation algorithm.

Name	Salary	Room
ni	**ni**	076
Mary	17K	**ni**

CWA

Name	Salary	Room
John	15K	076
Ann	**ni**	076
Mary	17K	561

OWA

Name	Salary	Room
John	15K	076
Ann	13K	325
Mary	17K	561

Figure 1.1: Illustration to CWA and OWA

Computational complexity of problems related to CWA or OWA was studied by Vardi [171]. He assumed a very simple model of partiality, namely values of a subset U of a set of attributes V are missing. Then a V-relation R' OWA-represents a U-relation R if $R \subseteq \pi_U(R')$, and it CWA-represents R if $R = \pi_U(R')$. Vardi considered certain problems related to dependency satisfaction and inference for both representations. He obtained a number of results of the following flavor: if a problem for OWA representations lies in a complexity class $\mathcal{C}$, then the same problem for CWA representations lies in the corresponding nondeterministic complexity class $\mathcal{NC}$. However, the situation is reverse for evaluation of boolean queries in all representations satisfying a given set of dependencies. Then for CWA the problem is PSPACE-complete whereas it is co-r.e.-complete for OWA representations.

1.1.2 Types of nulls

So far we have considered only one null value, **ni**, following Zaniolo [181]. There are other kinds of nulls in the literature.

Existing unknown values. In all examples above, we have not said anything about existence of a value that can be substituted for a null. For example, in the CWA completion of the database in figure 1.1, Ann has no salary. There could be several reasons for that. First, we may simply lack information about Ann's salary for some reason. For example, she was hired but is not on the payroll yet. Secondly, it could be the case she does not have a salary. For example, she might be working voluntarily, without getting paid.

In order to represent the first case, when a value does exist but is unknown at present time, existing unknown null values have been introduced. These have been studied most, see Codd [39, 40], Biskup [23], Maier [113], Grahne [62] etc. We shall often use **un** to denote such nulls.

Nonexisting nulls. As we have just mentioned, one of the reasons for a value to be missing is that it does not exist. Such values are denoted by **ne**; they were studied by Lerat and Lipski [94]. The main reson that such values appear in a database is that some attributes are not always applicable. For example, not every employee may have a telephone; the "children" attribute is certainly not applicable to all people, nor are "maiden name" and even more so "spouse's business phone number".

There is some confusion about considering **ne** as a null. Indeed, **ne** represents perfect knowledge in exactly the same way as any usual value. Knowing that Ann's maiden name is Smyth is as good as knowing she is not married and does not have one, if our concern is partiality of information. We shall see shortly that the intuition that **ne** "is not really a null value" will be confirmed when we consider ordering on those values in more detail.

No information nulls. These are nulls **ni** we have considered in the previous section. Having **ni** in a database simply means that there is no knowledge whatsoever about the situation.

Having introduced these three kinds of nulls, let us reexamine the first example of a relation with incomplete information given in this thesis. If we use nulls as follows:

Name	Salary	Room	Telephone
John	15K	075	**ne**
Ann	17K	**un**	**ni**
Mary	**un**	351	x-1595

We certainly have better knowledge than we had using only the **ni** null value. First, we know that John does not have a phone; moreover, we also obtained the knowledge that Mary and Ann do have some salary but at this time it is unknown what their salaries are. Hence, information-wise, **ni** is the worst situation possible, while having either a value or **ne** gives us complete knowledge about the situation. **un** is an intermediate situation: it is better than **ni** but certainly worse than any value, and it is incomparable with **ne**.

Now, applying the idea of representing partiality by means of an order on values, we obtain the ordering for the three kinds of nulls we studied in this section, see figure 1.2. Perfect knowledge, i.e. knowledge that can not be improved, is represented by elements which are not dominated by any other elements in this poset. In particular, **ne** is such.

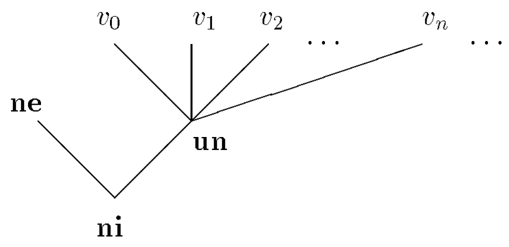


Figure 1.2: Order on null values

Open null values. Another kind of null values was introduced by Gottlob and Zicari [59] in the context of closed world databases. Assume we have a database with two kinds of null values, **ne** and **un**, and further assume the closed world assumption. Now, assume that we would like to relax this closed world assumption for a given attribute, but retain it for the others. The idea of Gottlob and Zicari was to introduce a new null, called **open**, which then will mean that the corresponding attribute is "open", i.e. it may have arbitrary values and not only those consistent with the information already stored in a database. We shall return to open null values later when we study semantics of partiality.

Generic nulls. In many cases we are not concerned with the meaning of null values and simply want to distinguish nulls from non-nulls. Then we use *generic nulls*, which will be denoted by $\perp$. Generic nulls are often used in the literature if general properties of partial information are investigated, see Buneman et al. [33], Levene and Loizou [98].

1.1.3 Semantics and query evaluation

Assume we are given a relational database with nulls and a query written in the relational algebra. How does one *evaluate* that query on an incomplete relation?

This is the question that has been studied most in the theory of partial information. A number of approaches resulted in two landmark papers: Imielinski and Lipski [78] and Abiteboul, Kanellakis and Grahne [8] which are, in my opinion, the most profound contributions into the theory of partial information in relational databases[2].

An incomplete database can represent many complete ones, which are often called *possible worlds*. Let R be a relation, and let $[\![R]\!]$ be the semantics of R, that is, the set of all possible worlds that R can denote. We explain later how $[\![\cdot]\!]$ can be defined. For now it is only important to understand that $[\![R]\!]$ is a *family* of relations. Let Q be a relational algebra query. We can

[2]Some of the results from [8] can also be found in the book Grahne [62].

define Q on $[\![R]\!]$ by

$$Q([\![R]\!]) = \{Q(R') \mid R' \in [\![R]\!]\}$$

The question arises: how can we define the action of Q on the incomplete relation R? The most natural requirement for this action of Q on R, which will be denoted by $Q^*(R)$, is to represent precisely $Q([\![R]\!])$. That is, $[\![Q^*(R)]\!] = Q([\![R]\!])$. Using terminology of Grahne [62], we call a pair $\langle [\![\cdot]\!], \Theta\rangle$ a *strong representation system* if $[\![Q^*(R)]\!] = Q([\![R]\!])$ holds for any query Q which is written in a sublanguage of the relational algebra that uses only operations from Θ.

As it was noted in Imielinski and Lipski [78], the structure of $[\![R]\!]$ is too irregular to allow $\langle [\![\cdot]\!], \Theta\rangle$ be a strong representation system for most Θ. Therefore, they suggested that one has to settle for something less. Their idea was to look at the set of *certain* answers to Q which is defined as

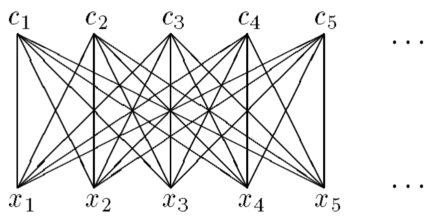
$$Q_c(R) = \bigcap Q([\![R]\!]) = \bigcap \{Q(R') \mid R' \in [\![R]\!]\}$$

Now we say that $\langle [\![\cdot]\!], \Theta\rangle$ is a *weak representation system* if for any query Q it is possible to find a query Q^* which represents the certain answer to Q, that is,

$$\bigcap [\![Q^*(R)]\!] = Q_c(R)$$

It was observed in Grahne [62] that the concepts of strong and weak representation systems coincide when Θ includes all operations of the relational algebra.

The next step is to define some classes of relations with null values and the semantic function $[\![\cdot]\!]$ for them. *Codd tables* are defined as relations in which *variables* can occur as well as constants and every variable occurs at most once. Variables represent null values, and each variable can be substituted by any value. That is, in terms of orderings, the basic domain of values that can occur in Codd tables is shown below. It is a complete bipartite graph between variables x_i's and constants c_j's. In other words, every variable x_i is less informative than every constant c_j and consequently can be replaced by it.



An *inequality table* is obtained from a Codd table by adding a finite number of inequalities between variables and between variables and constants. *Equality tables* are obtained from Codd's tables by declaring some variables equal. That is, the condition that every variable may occur at most once is removed. A combination of equality and inequality tables, that is, an equality table with a set of inequalities attached to it, is called a *global table*. Finally, a *conditioned table* is a global table with local conditions attached to each record. Those local conditions are conjunctions of equalities and inequalities. Below we give an example of each kind of tables.

0	x
y	1
v	z

Codd table

0	x
y	1
x	y

Equality table

$x \neq 1$	$v \neq z$
0	x
y	1
v	z

Inequality table

$x \neq 1$	$v \neq 2$
0	x
x	1
v	v

Global table

$x \neq 1$	$v \neq 2$	
0	x	$z = z$
x	1	$v = 0$
v	v	$v \neq x$

Conditioned table

To define the semantic function $[\![\cdot]\!]$, we fist define valuations as partial maps from variables to constants. Given a valuation ν, it can be extended to tables in a natural way (that is, by requiring that all conditions hold under the valuation ν). If a valuation does not satisfy the condition associated with a given record, it is not defined on that record. Similarly, if the global condition is not satisfied, then the valuation is not defined on a table with that global condition. That is, valuations extended to relations remain partial functions.

For a given table R, let $\mathrm{VAR}(R)$ be the set of all variables that occur in R. For a given valuation ν, let $dom(\nu)$ be the set of variables on which ν is defined. Now we can define $[\![\cdot]\!]$ under both closed and open world assumptions:

$$[\![R]\!]_{\mathrm{CWA}} = \{R' \mid \exists \nu : \ \mathrm{VAR}(R) \subseteq dom(\nu) \ \& \ \nu(R) = R'\}$$

$$[\![R]\!]_{\mathrm{OWA}} = \{R' \mid \exists \nu \exists R'' : \ \mathrm{VAR}(R) \subseteq dom(\nu) \ \& \ \nu(R) = R'' \ \& \ R'' \subseteq R'\}$$

That is, the main difference between CWA and OWA interpretations is that the latter allows adding any number of records that do not contain variables.

The main results of Imielinski and Lispki [78] are the following:

1. If Θ contains all operations of the relational algebra, then $\langle [\![\cdot]\!]_{\mathrm{OWA}}, \Theta \rangle$ is a strong representation system for tables without the global condition.

2. If Θ contains all operations of the relational algebra except difference and selection with negations present in the conditions, then $\langle [\![\cdot]\!]_{\mathrm{OWA}}, \Theta \rangle$ is a weak representation system for equality tables.

3. If Θ consists of projection and selection only, then $\langle [\![\cdot]\!]_{\mathrm{OWA}}, \Theta \rangle$ is not a weak representation system for equality tables.

4. If Θ consists of projection and selection only, then $\langle [\![\cdot]\!]_{\mathrm{OWA}}, \Theta \rangle$ is a weak representation system for Codd tables.

5. If Θ consists of projection, selection and union, then $\langle [\![\cdot]\!]_{\mathrm{OWA}}, \Theta\rangle$ is not a weak representation system for Codd tables.

6. If Θ consists of projection and join, then $\langle [\![\cdot]\!]_{\mathrm{OWA}}, \Theta\rangle$ is not a weak representation system for Codd tables.

7. If Θ does not contain difference, then $\langle [\![\cdot]\!]_{\mathrm{CWA}}, \Theta\rangle$ is a weak representation system iff so is $\langle [\![\cdot]\!]_{\mathrm{OWA}}, \Theta\rangle$.

Abiteboul, Kanellakis and Grahne [8] studied complexity of certain problems related to the CWA semantics of the tables. Two most important problems they studied are *membership* and *containment*.

The membership problem has a parameter Q which is a query that can be evaluated in polynomial time. It has two inputs: a relation R' without incomplete information and a conditioned table R. The question is whether $R' \in [\![R]\!]_{\mathrm{CWA}}$.

The containment problem two parameters, Q_1 and Q_2, which are queries that can be evaluated in polynomial time. It has two conditioned tables R and R' as an input. The question is whether $Q_1([\![R]\!]_{\mathrm{CWA}}) \subseteq Q_2([\![R']\!]_{\mathrm{CWA}})$.

It was shown that the general containment problem lies in Π_2^{p} and the general membership problem lies in $\mathcal{NP}$. In the case when the parameter of the membership problem is the identity query id, the membership problem becomes polynomial for Codd tables but is $\mathcal{NP}$-complete for equation and inequation tables. When both parameters of the containment problem are id, the problem is in $\mathcal{NP}$ for global tables and equality tables, and even in PTIME when one input is a global table and the other is a Codd table. However, it is Π_2^{p}-complete if one input is an equality table and the other is a conditioned table, or if one input is an inequality table and the other is a Codd table. More results of this flavor can be found in [8].

Query evaluation algorithms for databases with null values have also been studied by Reiter [144]. He used his earlier framework of representing databases as first-order theories [143] and showed how to incorporate existing but unknown nulls into it. In that setting, he demonstrated a sound query evaluation algorithm which is also complete under certain restrictions.

1.1.4 Extension to complex objects

All examples considered so far use the standard flat relational model. In the past few years many attempts have been made to go beyond that model. Most of them focus on *nested relations* or *complex objects*. We give a brief description of those and then discuss the problem of adding partial information into the complex object data model. The reader interested in development of the theory of nested relations per se should consult Schek and Scholl [156], Thomas and Fischer [167], Paredaens et al. [131] and the collection of articles [4].

The basic idea is that attributes may be relation-valued themselves. For example, in the following simple database the attribute Sections is relation-valued as any course may have a number of sections with different teaching assistants. Attributes Course and Instructor are single-valued: their values are like CS1 or Brown.

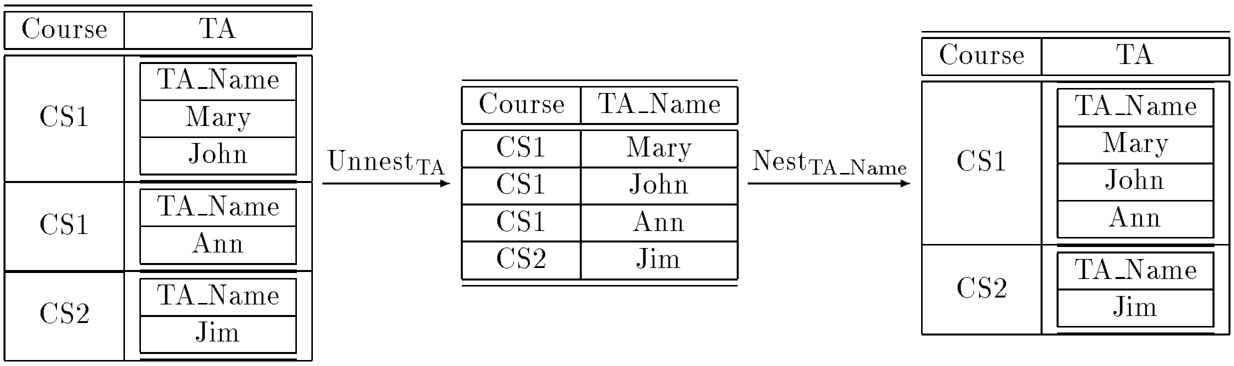
<table>
<tr><th>Course</th><th>Instructor</th><th>Sections</th></tr>
<tr><td>CS1</td><td>Smith</td><td><table><tr><th>Section#</th><th>TA</th></tr><tr><td>001</td><td>Ann</td></tr><tr><td>002</td><td>John</td></tr></table></td></tr>
<tr><td>CS2</td><td>Brown</td><td><table><tr><th>Section#</th><th>TA</th></tr><tr><td>003</td><td>Michael</td></tr><tr><td>004</td><td>Jim</td></tr></table></td></tr>
</table>

All operations of the relational algebra can be used with nested relations as well. However, for nested relations we need more than just relational algebra as it does not allow us to go deep inside the relations. Two operations for doing so have been proposed – *nest* and *unnest*. The *unnest* operation unnests values of some attributes. For example, unnesting the Sections attribute in the example above produces the usual (flat) relation

Course	Instructor	Section#	TA
CS1	Smith	001	Ann
CS1	Smith	002	John
CS2	Brown	003	Michael
CS2	Brown	004	Jim

Nesting over a group of attributes collects tuples with equal projections onto those attributes into new relations, thus creating an additional level of nesting. For example, nesting over Section# and TA in the above flat relation will give us the original nested relation.

The operations of nesting and unnesting are not mutually inverse, and doing *unnest* followed by *nest* we may lose some information. In the following example, we start with a nested relation and unnest the TA attribute and then nest over that attribute. The result, however, is different from the original relation:



There are several algebras for nested relations based on adding *nest* and *unnest* to the flat relational algebra, see Schek and Scholl [156] and Thomas and Fischer [167]. Colby [41] proposed an algebra in which operations can be defined recursively to go arbitrarily deep into nested relations, and showed that such an algebra is equivalent to the standard algebras of Schek and Scholl and Thomas and Fischer. Therefore, we can speak of *the nested relational algebra*, meaning any of these.

There are two main problems with existing algebras for nested relations. One is using *nest* and *unnest* in majority of queries. Every *nest* or *unnest* requires restructuring of data, which makes those algebras ineffective. Second problem is very cumbersome syntax of the nested relational algebras. Indeed, to ask a query about atomic values in a complex object of nesting depth two, two *unnest* operations must be performed, then a relational algebra query must be asked which may or may not be followed by some *nest* operations.

Therefore, if we aim at the design of a language capable of working with nested relations, we should find a better language to start with. Fortunately, new languages for complex objects have been invented recently which do not have many deficiencies of the standard languages, see Buneman et al. [34], Buneman, Tannen and Wong [26] and Libkin and Wong [105]. We present these languages in chapter 3.

In many applications nested relations are restricted to those in *partitioned normal form*, see Abiteboul and Bidoit [3], Roth, Korth and Silberschatz [151] and Van Gucht and Fischer [169]. In such relations, the single-valued attributes form a key, and each nested subrelation is in the partitioned normal form itself. The relation shown in the beginning of this subsection is such. An example of a non-partitioned normal form relation is given below. It can not be in the partitioned normal form because it does not have any single-valued attributes.

Employee		Salary History	
Name	Department	Year	Amount
Ann	CS	1992	12K
Mary	Math	1993	14K
Name	Department	Year	Amount
Jim	Physics	1992	10K
John	CS	1993	11K

Null values were introduced into partitioned normal form complex objects in Roth, Korth and Silberschatz [151]. They considered three kinds of null values: **ni**, **un** and **ne**. They defined an algebra on such complex objects with null values and claimed that the algebra was a precise generalization of the nested relational algebra restricted to partitioned normal form complex objects. By "precise" they meant that queries commute with *unnest* in the following sense: if a query Q sends a nested relation R into R', then unnesting R over zero-order attributes and then performing Q on the result is the same as unnesting R' over zero-order attributes. However, it was shown by Levene and Loizou [96] that projection in the algebra of Roth, Korth and Silberschatz is *not* a precise generalization of the standard projection.

To remedy this, Levene and Loizou introduced the notion of information-wise equivalent nested relations in [98]. This notion is based on the idea of ordering. They started with one generic null and the ordering $\perp \leq v$ for any value v and extended it component-wise to tuples. To extend it to sets, they used the ordering $\sqsubseteq$ shown in the section on orderings and null values. Then, if $R_1 \sqsubseteq R_2$ and $R_2 \sqsubseteq R_1$, they said that R_1 and R_2 are information-wise equivalent. Under this notion of equivalence, it is possible to generalize the nested relational algebra in the *precise* way, that is, in the way that agrees with respective operations on complete relations up to the information-wise equivalence.

One major problem with the approach of Levene and Loizou [98] is that they used the standard nested relational algebra and inherited all of its problems and drawbacks. In particular, the description of their notion of null-extended join operator is almost one-page long, and many other operations are rather hard to grasp.

Finishing this section, let us mention briefly some other directions of research on null values that we do not address in the thesis. Updates in relational databases with null values have been studied in Abiteboul and Grahne [5] and Grahne [62]. Functional dependencies in relational databases with existing unknown nulls were studied in Vassiliou [173] and Atzeni and Morfuni [17]. Dependencies in incomplete databases specified by Horn clauses are one of the main subjects of Grahne [62]. Dependencies in relations with existing unknown nulls are also studied in the context of the weak instance model, see Honeyman [74]. A generalization of the weak instance model that incorporates nonexisting nulls was given by Atzeni and De Bernardis [18]. More information on dependencies in incomplete relational databases can be found in Thalheim [166].

Dependencies in nested relations with generic nulls are the main topic of Levene and Loizou [97].

1.2 Disjunctive information and or-sets

1.2.1 Definition and examples of or-sets

As we mentioned before, the idea of using disjunctive information as a means to express partiality was already present in Lipski [109, 110]. It was not until almost ten years later that the first attempt was made to introduce disjunctions explicitly into the standard relational model.

Consider the following example. Suppose we have two databases, D_1 and D_2 shown below:

D_1 :

Name	SS#	Age
John	123456789	24
Mary	987654321	32

D_2 :

Name	SS#	Age
John	123456789	27
Ann	564738291	25

Assume that we merge D_1 and D_2. It is clear that records $\boxed{\text{Mary} \mid 987654321 \mid 32}$ and $\boxed{\text{Ann} \mid 564738291 \mid 25}$ should be in the resulting database. But what is the value of the Age field for John? Since SS# identifies people uniquely[3], we have *conflicting* information coming from two databases, and this conflict must be recorded in the newly created database until one finds out if John is 24 or 27 years of age.

Therefore, both ages – 24 and 27 – are stored in the new database. However, the semantics of the Age attribute (which is now set-valued) is different from the usual interpretation of sets in databases. Rather than suggesting that John is both 24 and 27 years old, it says that John is 24 *or* 27.

Since such disjunctive sets, also called *or-sets*, have semantics that differs from the ordinary sets, we shall use a special notation $\langle\rangle$ for them. That is, the result of merging D_1 and D_2 is

D :

Name	SS#	Age
John	123456789	$\langle 24, 27\rangle$
Mary	987654321	32
Ann	564738291	25

[3] Or at least is supposed to.

Again, we emphasize that the or-set $\langle 24, 27\rangle$ denotes *one of its elements*. So semantically it is either 24 *or* 27.

It is interesting to note that one practical implementation of or-sets was done in early 80s in Hungary, as I was told by János Demetrovics [46]. Their primary motivation was police database, and their observations showed that different witnesses of the same event often contradicted each other; hence the need for or-sets. For example, one witness could say that a car used by robbers was green, another saw a red car and the third witness could have seen a car that was both red and green. A data model for such a database should allow all three statements to be stored in an appropriate way. Therefore, using only ordinary sets was no longer sufficient, and a rudimentary model of disjunctive information was used in that project.

In early papers dealing with objects that may include or-sets (Imielinski and Vadaparty [82], Liu and Sinderraman [111], Ola [128]) a very limited model was considered. In fact, in those papers or-sets could only appear as entries in the usual relations, as it is in the example above. In Liu and Sinderraman [111] and Ola [128] extensions of the traditional relational algebra were studied. As we mentioned before, this is not the approach we advocate here. Rather, we prefer Lipski's approach [110] that new languages should be designed for new kinds of partiality. That we should follow Lipski's approach is further confirmed by many difficulties encountered in the above mentioned papers. For example, to obtain the correctness result, in Ola [128] some rather ad-hoc types of tuples are introduced and representation systems are defined via those types. Contrary to [111, 128] which used extensions of the relational algebra, in Imielinski and Vadaparty [82] a logical language was used. Another logical language for or-sets was proposed in Sakai [153] but it was not feasible for many applications as it had an *a priori* upper bound on the number of elements in or-sets.

In subsequent papers, such as Imielinski, Naqvi and Vadaparty [80, 81], Rounds [152] and Libkin and Wong [104] more general data models were considered. In particular, it was possible to freely combine sets, records and or-sets.

As we have said above, an or-set $\langle 1, 2, 3\rangle$ denotes a single integer, of which we only know that it is either 1 or 2 or 3 but do not know which one. That is, or-sets are used to represent a special case of *partial* information. A singleton or-set corresponds to precise information; that is, $\langle 1\rangle$ denotes the integer 1. An empty or-set can be interpreted as inconsistency as its meaning is "choose one out of nothing".

In [80], Imiellinski, Naqvi and Vadaparty designed a data model and a logical query language for or-objects, following the approach of Abiteboul and Kanellakis [7]. Consequently, the semantics and query language are rather involved. They also obtained some complexity results for their logical language. In particular, they were able to demonstrate *co-*$\mathcal{NP}$-completeness result, and they were successful in identifying certain restricted tractable fragments that are useful in real-life applications.

A similar notion of disjunctive deductive databases was also studied in Minker [115]. However, it is important to make a clear distinction between or-sets and disjunctive deductive databases (cf. [80]). In the latter arbitrary disjunctions are allowed. In contrast to that, we regard or-sets as a *type constructor.* Hence, or-sets can appear only in certain places specified by a database schema. Furthermore, in the field of deductive databases, a database is considered as a theory, whereas representation of objects involving or-sets is purely structural. Finally, or-sets are distinguished from other forms of disjunctive information by having two distinct interpretation, which are described in the following subsection.

1.2.2 Structural and conceptual queries

As we have just said, or-sets are distinguished from other kinds of disjunctive information by having two distinct interpretation. An or-set can either be treated at the *structural* level or at the *conceptual* level. The structural level concerns the precise way in which an or-set is constructed. The conceptual level concerns the meaning of or-sets. It sees an or-set as representing an object which is equal to a member of the or-set. For example, the or-set $\langle 1, 2, 3\rangle$ is structurally a collection of numbers; however it is conceptually a number that is either 1, 2, or 3.

If an or-set is sitting inside another structure, such as a relation, it is not immediately clear what the whole object is conceptually. Consider our example of the database D that was obtained by merging D_1 and D_2. Its representation that has been shown is on the structural level. To see what its meaning is, observe that John's age is (conceptually) either 24 or 27. Therefore, the whole D is conceptually either

Name	SS#	Age
John	123456789	24
Mary	987654321	32
Ann	564738291	25

or

Name	SS#	Age
John	123456789	27
Mary	987654321	32
Ann	564738291	25

The two views of or-sets are complementary. Consider a design template used by an engineer. The template may indicate that component A can be built by either module B or module C. Such a template, as explained in [80], is structurally a complex object whose component A is the or-set containing B and C. Moreover, A, B and C can in turn have the similar structure. A designer employing such a template should be allowed to query the structure of the template, for example, by asking what are the choices for component A. On the other hand, the designer should also be allowed to query about possible completed designs, for example, by asking if there is a cheap complete design, or if all completed designs have do not exceed certain cost is some of the choices have been made. In the latter case, as the designer is still in the process of creating a design, the "complete design" is purely conceptual. Both views of or-sets are important and should be supported.

The structural interpretation of or-sets is quite clear. However, the conceptual interpretation requires further exposition. For example, to go to the conceptual level from the structural level, we need operators prescribing the interaction of or-sets, records and ordinary sets. Several of them can be considered. For example, taking or-set brackets outside of records or sets by listing explicitly all possible choices, as we just did with the database D. Such operators provide an idea of what to include in a *structural* query language. But what kind of operators should be provided in a *conceptual* query language? Should there be an operator for testing whether two objects are conceptually equivalent? Should there be an operator for testing whether one object is amongst the objects that a second object can conceptually be? Fortunately, it is not necessary to make such chaotic "enhancements." We will show later that the operators informally described above are sufficient to construct a *normal form* (or, conceptual representation) of every object unambiguously.

1.3 Approximations

In this we section consider another kind of partiality which often arises when one tries to query independent databases that do not necessarily agree with each other. As it was observed by many, an answer to a query against a number of independent databases can at best be approximated. That is, it is unrealistic to expect a precise answer.

In this section we start with an example that illustrates the problems arising in querying independent databases. We then proceed to introduce a number of models that are used for approximated answers. There is a tradition to give food names to those. It started when Buneman, Davidson and Watters [31, 32] introduced *sandwiches,* which consist of lower and upper approximations and denote precisely what is in between, hence the name. Other constructions were called *mixes* (Gunter [66]), *snacks* (Ngair [121], Puhlmann [141] although they were studied much earlier by pure mathematicians: Płonka [135, 136], Balbes [19]), *scones*[4] and *salads* (Libkin [103]). The generic name for these constructions is *edible powerdomains* (Libkin [103]). It is probably not a very good naming convention, as names do not reflect the structure of specific approximations. However, we follow the tradition and later introduce a new systematic notation for all the constructs.

1.3.1 Example: Querying independent databases

The general problem of querying independent databases is the following: given a set of databases $D_1, \ldots, D_n$ and a query q that can not be answered by using information from one of D_i's, approximate the answer to q by using information from all $D_1, \ldots, D_n$. These problems have been

[4]This is not a good choice of name suggested by Jung and then used by Puhlmann [141] as it is in conflict with the notion of a scone used in category theory and recently in the categorical models of polymorphic languages, see Mitchell and Scedrov [117].

investigated theoretically, and they gave rise to a number of constructions called *approximations*. Intuitively, given a query q, the databases are divided into two groups, one giving the upper approximation to the answer to q and the other giving the lower approximations.

Consider the following problem. Suppose the university database has two relations, Employees and CS1 (for teaching the course CS1):

Employees

Name	Salary
John	15K
Ann	17K
Mary	12K
Michael	14K

CS1

Name	Room
John	076
Michael	320

Assume that our query asks to compute the set TA of teaching assistants. We further assume that only TAs can teach CS1 and every TA is a university employee. Also, for simplicity, we make an assumption that the Name field is a key. Of course this may not be the case, and solutions we consider in this thesis work if no assumptions about keys were made. This assumption, however, makes the examples easier to understand.

To be able to reason about entries in different tables at the same time, we assume that all tables have the same attributes by putting nulls in the missing columns:

Employees

Name	Salary	Room
John	15K	$\perp$
Ann	17K	$\perp$
Mary	12K	$\perp$
Michael	14K	$\perp$

CS1

Name	Salary	Room
John	$\perp$	076
Michael	$\perp$	320

Let us briefly outline how the TA query can be answered. We know that every person in CS1 is a TA; therefore, CS1 gives us the certain part of the answer. Moreover, every TA is an employee, hence finding people in the Employees relation who are not represented in the CS1 relation gives us the possible part of the answer to the TA query. Notice that it is possible to find possible TAs because Name is a key. If it were not, we would have to use or-sets.

Of course, in the real life applications, the situation is not always that close to ideal. Let us just briefly list the problems one should have in mind while querying independent databases:

- Databases could be inconsistent. Then anomalies must be removed before a query could be evaluated. There are, however, a number of subproblems:

1. Which database to believe? Each one can be updated.
2. If in the example above we believe Employees and the Name field is *not* a key, assume we have one John in Employees and two Johns in CS1. Then one of the Johns in CS1 must be deleted. But which one?

- Even if databases are consistent, but the Name field is not a key, there is no way to evaluate the TA query unambiguously. For example, there could be two Johns with different salaries in Employees, but only one in CS1. Assume a query "give the list of sure TAs" was asked. Then what is John's salary?

Notice that these problems have not been addressed in Buneman, Davidson and Watters [31, 32]. In the thesis we shall show how to solve these problems using various tools for programming with approximations and or-sets.

1.3.2 Simple approximations

A pair of relations CS1 and Employees is called a *sandwich* (for TA). The Employees relation is *an upper bound:* every TA is an Employee. The CS1 relation is *a lower bound:* every entry in CS1 represents a TA. Notice that in our example records in CS1 and Employees are *consistent:* for every record CS1, there is a record in Employees consistent with it. That is, they are joinable and their join can be defined. For example,

$$\begin{array}{|c|c|c|}\hline \text{John} & \text{15K} & \perp \\ \hline \end{array} \vee \begin{array}{|c|c|c|}\hline \text{John} & \perp & 076 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline \text{John} & \text{15K} & 076 \\ \hline \end{array}$$

Hence, a sandwich (for a query Q) is a pair of relations R_1 and R_2 such that R_1 is an upper bound or an upper approximation to Q, R_2 is a lower bound or a lower approximation to Q, and R_1 and R_2 are consistent.

Assume a pair of consistent relations R_1 and R_2 is given. What is the semantics of the sandwich (R_1, R_2)? To emphasize that R_1 is an upper approximation, we denote it by U from now on. Similarly, we denote the lower approximation R_2 by L.

To answer the question about semantics of (U, L) – at this stage, only informally – we appeal to the idea of representing partial objects as elements of ordered sets. In a graphical representation, ordered sets will be shown as triangles standing on one of their vertices. That vertex represents the minimal, or bottom element[5]. The side opposite to that vertex represents maximal elements. In our interpretation of the order as "being less partial", maximal elements correspond to complete descriptions, i.e. those that do not have any partial information at all.

[5] We almost always consider ordered sets with minimal elements.

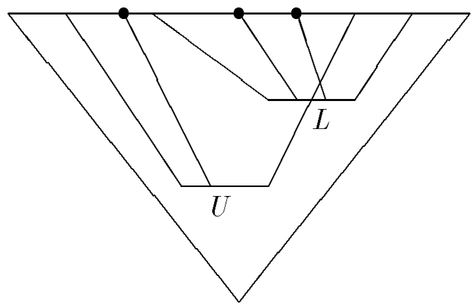


Figure 1.3: Sandwich (U, L) and its semantics

The graphical representation of a sandwich (U, L) is shown in figure 1.3. Trapezoids standing on U and L represent graphically elements of the whole space which are bigger than an element of U or L respectively. The semantics of a sandwich is a family of sets such as the one denoted by three bullets in the picture. There are two properties of such sets X that include them into the semantic space of a sandwich:

1. For every element $l \in L$, there is an element $x \in X$ such that $l \leq x$.

2. All X lies in the trapezoid standing on U. That is, for every $x \in X$, there exists $u \in U$ such that $u \leq x$.

Observe that in our particular example depicted in the picture, L is assumed to have two elements. Since both of them are under elements of the three-bullet set, which in turn are all above some elements of U, (U, L) satisfies the consistency condition, i.e. it is a sandwich.

Now, assume that the Name field is a key. Then we can replace certain nulls in relations CS1 and Employees by corresponding values, taken from the other relation. The reason is that certain tuples are joinable, and corresponding joins can be taken to infer missing values. One such join was shown in the beginning of this section. Since Name is a key, we know that there is only one John and we assume that the same John is represented by both databases. Hence we infer that he is in the office 076 and his salary is 15K. Similarly for Michael we infer that he is in the office 320 and his salary is 14K. Thus, we can replace Employees and CS1 by Employees$'$ and CS1$'$ as shown below:

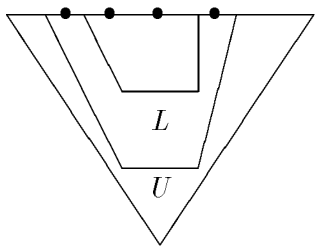


Figure 1.4: Mix (U, L) and its semantics

Employees′

Name	Salary	Room
John	15K	076
Ann	17K	$\perp$
Mary	12K	$\perp$
Michael	14K	320

CS1′

Name	Salary	Room
John	15K	076
Michael	14K	320

We can regard CS1′ and Employees′ as another approximation for TA. But this one satisfies a much stronger consistency condition than sandwiches: every record in CS1′ is also found in Employees′. We call a pair satisfying this consistency condition a *mix*. An example of a mix is shown in figure 1.4.

Mixes were introduced by Gunter [66] as an alternative approximation construct, whose properties are generally easier to study than properties of sandwiches because of its consistency condition in which no joins are involved. We shall discuss this phenomenon in details later.

Semantics of mixes is defined in exactly the same way as semantics of sandwiches: we look at sets that represent all elements of the lower approximation and whose elements are representable by the upper approximation. In Figure 1.4, a set shown by four bullets is such.

1.3.3 Approximating by many relations

Let us consider a more complicated situation. Assume now that CS1 has two sections: $CS1_1$ and $CS1_2$, and each section requires a teaching assistant. Assume that we have a pool of prospective TAs for each section that includes those graduate students who volunteered to be TAs for that section. Now suppose that the selection of TAs has been made, and those who have been selected were entered in the database of employees, while the database of prospective TAs remained unchanged. This situation can be represented by an example below:

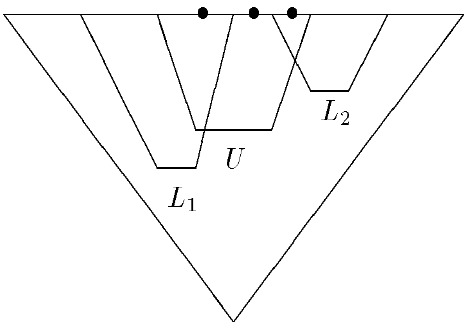


Figure 1.5: Scone $(U, \{L_1, L_2\})$ and its semantics

Employees

Name	Salary	Room
John	15K	$\perp$
Ann	17K	$\perp$
Mary	12K	$\perp$
Michael	14K	$\perp$

$CS1_1$

Name	Salary	Room
John	$\perp$	076
Jim	$\perp$	$\perp$

$CS1_2$

Name	Salary	Room
Michael	$\perp$	320
Helen	$\perp$	451

Since all the selections have been made, at least one of prospective TAs for each section is now a TA and therefore there is a record in Employees for him or her. That is, in each of the subrelations of CS1, at least one entry is consistent with the Employees relation.

Let us summarize the main difference between this construction and sandwiches or mixes considered in the previous section.

1. The lower approximation is no longer a single relation but a *family of relations*.
2. The consistency condition does not postulate that all elements in the lower approximation are consistent with the upper approximation, but rather that there *exists* and element in each of the subrelations of the lower approximation that is consistent with the upper.

Such approximations are called *scones*. We shall denote the lower approximation by $\mathcal{L}$ and its components by L_1, L_2 etc. The graphical representation of a scone with the two-element $\mathcal{L}$ is shown in Figure 1.5.

The semantics of a scone is a family of sets X that satisfy the following two properties:

1. For every set $L \in \mathcal{L}$, there exist $l \in L$ and $x \in X$ such that $l \leq x$.
2. All X lies in the trapezoid standing on U. That is, for every $x \in X$, there exists $u \in U$ such that $u \leq x$.

For example, in Figure 1.5 the set denoted by three bullets is such. Observe that the second property is exactly the same for scones as it is for sandwiches and mixes, but the first one is different and it reflects the differences in the structure of scones and sandwiches.

Now let us look at the data represented by $\mathrm{CS1}_1$ and $\mathrm{CS1}_2$. Assume again that the Name field is a key. Observe that some preprocessing can be done before any queries are asked. In particular, there is no entry for Jim in the Employees relation. Hence, Jim could not have been chosen as a possible TA for a section of CS1. Similarly, Helen can be removed from $\mathrm{CS1}_2$. Having removed Jim and Helen from $\mathrm{CS1}_1$ and $\mathrm{CS1}_2$, we can now infer some of the null fields as we did before in order to obtain mixes from scones. Doing so in our example yields:

Employees

Name	Salary	Room
John	15K	076
Ann	17K	$\perp$
Mary	12K	$\perp$
Michael	14K	320

Name	Salary	Room
John	15K	076

$\mathrm{CS1}_1$

Name	Salary	Room
Michael	14K	320

$\mathrm{CS1}_2$

We now see that the condition expressing consistency of this approximation is much stronger than the condition we used for scones. In fact, all elements in $\mathrm{CS1}_1$ and $\mathrm{CS1}_2$ are elements of Employees. In other words, taking into account that some entries can be nulls, we see that the new consistency condition says that every element of every set in the lower approximation is bigger than some element of the upper approximation. Such constructions are called *snacks*, see Ngair [121], Puhlmann [141]. The graphical representation of a snack with two-element $\mathcal{L}$ is given in Figure 1.6.

The semantics of snacks is defined precisely in the same way as the semantics of scones. For example, in Figure 1.6 the four-element set denoted by the bullets is in the semantics of $(U, \{L_1, L_2\})$. Thus, it is only the consistency condition that makes scones different from snacks. The importance of this condition will be studied later in the thesis.

Finally, what if we have arbitrary data coming from two independent databases that may not be consistent (as was discussed in the beginning of this section). For instance, we saw that there may be anomalies in the data that ruin various consistency conditions. Then we need a model that would not require any consistency condition at all. Such a model was introduced and studied by Libkin [103]. Since it is in essence "all others put together", it is called *salad.*

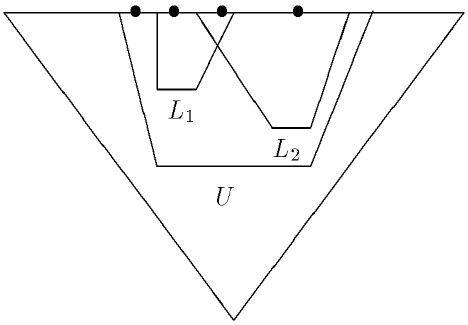


Figure 1.6: Snack $(U, \{L_1, L_2\})$ and its semantics

At this point we give a historical remark. Snacks were introduced long before scones. In fact, snacks were studied by Ngair in 1991, see [121]. A few initial results on scones were obtained by Jung and Puhlmann only in late 1992. The reason it happened (despite the fact that scones may appear more natural as a model of approximation, as we have seen) is that the development of models for approximating partial data has been done in a rather ad-hoc manner: new consistency conditions were introduced and studied. Since snacks appear simpler than scones, they were invented first.

Later in the thesis we shall present a systematic approach that lists all possible consistency conditions in conjunction with various data structures, thus giving us all possible approximation constructs. We shall characterize each of them mathematically, and then develop a unifying approach that encompasses all of them.

1.4 Toward a general theory of partial information

As we have seen, there are a number of models for partial information in the database literature. Some of them are quite ad-hoc, based on specific needs arising in particular applications. We have covered three main sources of partiality: null values, disjunctive information and approximations. There are no solid theoretical foundations for any of these, nor are there any results that show how they are connected. Moreover, most models of partiality are developed only for the flat relational model, and virtually nothing is known for more complicated database models. This situation in the field of partial information was summarized by Kanellakis in his recent survey [89]:

> "... for the representation and querying of incomplete information databases, there are many partial solutions but no satisfactory full answer. It seems that the further away we move from the relational data model, the fewer analytical and algebraic

tools are available."

Thus, to address the problem of partial information in databases and to move closer to satisfactory solutions that work for a large class of data models, one has to come up with new analytical tools and show their applicability not only in the study of the extended data models but also in the development of new query languages for databases with partial information. Making progress in achieving these goals is the major motivation for this work. The main contribution of the thesis is the following:

> *In this thesis we make a step toward a general theory of partial information in databases. We do it by developing a new approach to partial information that integrates all kinds of partiality within the same semantic framework. In addition to giving us necessary analytical and algebraic tools to study various kinds of partial information, this framework also naturally suggests operations that should be included into the language that works with partial information. Techniques that are developed for analyzing the structure of partial information can be applied to the study of the languages that deal with it.*

This general statement can be decomposed into the two key ideas upon which our approach to partial information is based.

I. Partiality of data is represented via orderings on values.

As we saw, this idea in its rudimentary form was already present in Biskup [23], Zanilo [181] and several other papers. However, they all had two major limitations. First, they dealt with the relational model or very limited complex object model (such as Roth, Korthand Silberschatz [151]). Second, the class of null values was always given *a priori*, hence all the models lacked generality that we seek.

A few recent results provided a basis for overcoming these limitations. Buneman, Jung and Ohori [33] started developing a general framework for representing partial objects as elements of certain ordered sets that have been used extensively in the semantics of programming languages [67]. Their results were further extended by Ohori [124, 125], Levene and Loizou [95], Libkin [99] and Jung, Libkin and Puhlmann [88], which resulted in the theoretical foundation for the studying of the problem of partial information.

It was shown in Buneman, Jung and Ohori [33] that the existing models of null values and many data models fit very nicely with their approach. They were successful in defining some notions of the relational database theory such as scheme and functional dependencies. However, they encountered certain difficulties. For example, there is no "universal" way to order subsets of ordered sets. They suggested using the Smyth ordering [157] known in the semantics of concurrency, mainly because it gives the natural join for free. But using the Smyth ordering often leads to counterintuitive results. Other papers mentioned above used the Hoare ordering [68], but also without any justification. Thus, the problem of choosing the way of extending the orderings to various collections is a central one and should be addressed.

Once the problem of ordering various database objects has been resolved, one should look at the fundamental properties of the semantic domains they give rise to. The reason why this is so important is the second central idea of this thesis.

II. Semantics suggests programming constructs.

How does one choose primitive operations upon which database query languages are constructed? An approach that has been increasingly popular in the past few years is to look at the operations that are *naturally* defined by the data types involved. This is an example of *data-oriented* programming of Cardelli [35].

To answer the question about operations naturally associated with a data type, one has to understand first what the values of that data type are *semantically* and what is the structure of the semantic domain of the data type. In particular, one often tries to find *universality properties* of those semantic domains which guarantee existence and uniqueness of operations on data that can be turned into programming language syntax. This approach has been applied to a number of data types and has proved extremely useful, see Buneman et al. [26, 34], Libkin and Wong [104, 105, 106], Suciu [161].

Therefore, in this thesis we shall be looking for the universality properties of various kinds of partial data as a main tool for the language design. In other words, the mathematical properties of semantics of partial data will naturally suggest the programming primitives to be included in the languages. Thus, the purpose of developing the semantics of partiality is twofold. First, we use it to *integrate* all kinds of partial information. Second, we use it to *design languages* for incomplete databases.

Let us summarize the main contributions of the thesis.

1. We define a general model of representation of database objects in certain partially ordered sets to capture the notion of being less partial.
2. We define the semantics for sets and or-sets and use it to show how they must be ordered;
3. We propose the "update" semantics, which explains being less partial as obtained via a sequence of elementary updates that add information, and show that it leads to the same orderings.

4. We analyze semantics and orderings on approximations and show how they can be encoded with sets and or-sets.

5. We exhibit universality properties for semantic domains of all kinds of partial information.

6. We study the interaction of sets and or-sets and demonstrate a computable isomorphism between iterated semantic domains of those.

7. We use the universality properties together with the above computable isomorphism to design a language for sets and or-sets.

8. We show how the meaning of or-objects can be incorporated into the language by means of a process called normalization, and investigate structural and computational aspects of that process.

9. We demonstrate that the universality properties for approximations do not lead to a reasonable programming language because of the complexity of the operations involved, and show how to use the language for sets and or-sets to program with approximations.

10. We describe implementation of the language for sets and or-sets and show how it can be used to query incomplete and independent databases.

The structure of the thesis

In chapter 2 we present the mathematical background which is necessary to understand this thesis. In chapter 3 we lay the foundation for our study of semantics of partiality and languages to work with partial information. The notion of partial information in databases is re-examined and connected with certain partially ordered spaces of descriptions used in the semantics of programming languages. Several main concepts of the relational database theory are redefined in such a setting. Also in chapter 3 we explain the new approach to the design of query languages whose gist is turning universality properties of collections into syntax.

In chapter 4 we study the semantics of partial data. Orderings on collections and approximations are defined via the semantics and updates and are shown to be the same. These results explain how sets an or-sets of partial descriptions arise as free constructions. They also demonstrate a natural way of combining sets and or-sets to encode approximations. We show that all approximations arise as free constructions as well. We also construct an isomorphism between iterations of semantic domains for sets and or-sets.

In chapter 5 we apply the approach that makes programming syntax out of universality properties to study languages for databases with partial information. We add order on objects as a primitive and study the resulting languages. We then introduce the language for sets and or-sets and show how to *normalize* objects and explain why normalization provides us with passage from the structural to the conceptual level. Finally, we discuss two approaches to programming with approximations. One is the structural recursion and the other is encoding approximations with sets and or-sets.

In chapter 6 we describe an implementation of the language for sets and or-sets on top of Standard ML (hence called OR-SML). We give examples of queries which require disjunctive information and demonstrate how to use the language to answer those queries. The language is extended in a way that allows dealing with bags and aggregate functions. It is also extensible by user-defined base types. The language has been implemented as a library of modules in Standard ML. This allows the user to build just the database language as an independent system, or to interface it to other systems built in Standard ML. Since the system is running on top of Standard ML and all database objects are values in the latter, the system benefits from combining a sophisticated query language with the full power of a programming language.

Finally, in chapter 7 we summarize the main contributions of this thesis and outline prospects for further research.

Relationship with work of others

Most of the results in the thesis are my own. However, on several occasions I did include some of the results that are due to my colleagues or that have been obtained jointly.

In the first part of chapter 3 I mix my own results from Libkin [99] and Jung, Libkin and Puhlmann [88] with the results from Buneman, Jung and Ohori [33]. In the second part of that chapter I present the approach which was originally developed by Buneman, Breazu-Tannen, Naqvi and Wong [25, 26]. Many properties of the languages it gives rise to have been studied in my joint papers with Wong [105, 106, 107, 108] and I include some of the results that have been proven jointly by us.

Or-sets are the main topic of my paper with Wong [104]. The normalization theorem for set semantics was proved by us independently; the proof in the thesis as well as other variations of the normalization theorem are my own. The losslessness theorem that I prove is mine, although there is a related losslessness result by Wong that appeared in [104].

E. Gunter influenced the implementation of OR-SML in many ways and some of the examples in our paper [69] that I use here are due to her.

Chapter 2

Mathematical Background

In this chapter we give mathematical background which is necessary to understand the results of this thesis. We present basic definitions and some results about ordered sets, universal algebras (paying particular attention to ordered algebras freely generated by posets), categories, adjoint functors and associated monads, abstract rewrite systems and term rewrite systems. Covering basic domain theory, we give a somewhat unusual presentation of powerdomains. In view of this, we sketch a few proofs.

2.1 Ordered sets and domains

A *preorder* on a set A is a reflexive transitive relation. A preorder is called a *(partial) order* if it is antisymmetric. A set with a partial order on it is called a *poset*. We shall use symbols $\leq, \precsim$ and the likes to denote orders.

Let $\langle A, \leq \rangle$ be a poset, and $x, y \in A$. We say that x and y are *consistent* (denoted by $x \uparrow y$) if there exists $z \in A$ such that $x, y \leq z$. A subset $X \subseteq A$ is *downward closed*, or an *order-ideal*, if $x \in X$ and $x' \leq x$ imply $x' \in X$. The order ideal generated by a set X is denoted by $\downarrow X$; $\downarrow X = \{y \leq x \mid x \in X\}$. If $X = \{x\}$, then $\downarrow X$, also denoted by $\downarrow x$, is called *principal*. The concept of an *upward closed* set or a *filter* is defined dually; for a filter generated by X we use the notation $\uparrow X$.

A subset $X \subseteq A$ is called *directed* if a common upper bound exists for any two elements of X, that is, given $x_1, x_2 \in X$, there exists $x \in X$ such that $x \geq x_1, x_2$. A poset is called complete (abbreviated – *cpo*) if every directed subset $X \subseteq A$ has a least upper bound $\sqcup X$. An element of a cpo is called *compact* if it can not be below a least upper bound of a directed set X without being below an element of X. That is, x is compact if $x \leq \sqcup X$ for a directed X implies $x \leq x'$

for some $x' \in X$. A cpo is called algebraic if every element is the least upper bound of compact elements below it, see C. Gunter [67][1].

A *domain* is an algebraic cpo with bottom. Given a domain D, $\leq$ denotes its order and $\mathbf{K}D$ is the set of its compact elements.

A cpo D is *bounded complete* if supremum of $X \subseteq D$, denoted by $\sqcup X$, exists whenever X is bounded above in D, i.e. there is $a \in D$ such that $a \geq x$ for all $x \in X$. We shall use a more convenient notation $a_1 \vee \ldots \vee a_n$ instead of $\sqcup\{a_1, \ldots, a_n\}$. An element x of a bounded complete cpo D is compact if, whenever $\sqcup X$ exists and $x \leq \sqcup X$, $x \leq \sqcup X'$ where $X' \subseteq X$ is finite.

In a bounded complete cpo the set of compact elements below any element is always directed; therefore, a bounded complete cpo is algebraic if any element is the supremum of all compact elements below it. Algebraic bounded complete cpos are also called *Scott-domains.* Equivalently, a Scott-domain is a domain which happens to be a complete meet-semilattice.

A Scott-domain is called *distributive* if every principal ideal $\downarrow x$ is a distributive lattice. It is called *qualitative* if all $\downarrow x$ are Boolean lattices, cf. Girard [57].

Given $X, Y \subseteq D$, the *lower*, the *upper* and the *convex powerdomain orderings* are given by

$$X \sqsubseteq^{\flat} B \Leftrightarrow \forall x \in X \exists y \in Y \;:\; x \leq y$$

$$X \sqsubseteq^{\sharp} Y \Leftrightarrow \forall y \in Y \exists x \in Y \;:\; x \leq y$$

$$X \sqsubseteq^{\natural} Y \Leftrightarrow X \sqsubseteq^{\flat} Y \text{ and } X \sqsubseteq^{\sharp} Y$$

Sometimes they are called *the Hoare*, *the Smyth* and *the Plotkin* orderings respectively.

A subset of an ordered set is called a chain if every two elements in it are comparable, and an *antichain* if no two elements in it are comparable. If $\langle X, \leq \rangle$ is an ordered set and $Y \subseteq X$, then $\max_{\leq} Y$ and $\min_{\leq} Y$ are sets of maximal and minimal elements of Y. We will use just $\max Y$ and $\min Y$ if the ordering is understood. $\mathbb{A}_{\text{fin}}(X)$ stands for the set of all finite antichains of X.

For an arbitrary poset A, we denote $\langle \mathbb{A}_{\text{fin}}(A), \sqsubseteq^{\flat} \rangle$ by $\mathcal{P}^{\flat}(A)$ and $\langle \mathbb{A}_{\text{fin}}(A), \sqsubseteq^{\sharp} \rangle$ by $\mathcal{P}^{\sharp}(A)$. Note that we can canonically embed A into both $\mathcal{P}^{\flat}(A)$ and $\mathcal{P}^{\sharp}(A)$:

$$\forall a \in A: \quad \eta(a) = \{a\} \in \mathbb{A}_{\text{fin}}(A)$$

At this point, let us make a number of observations about the two constructs we have just introduced. First of all, $\mathcal{P}^{\flat}(A)$ is always a join-semilattice with bottom element and $\mathcal{P}^{\sharp}(A)$ is always a meet-semilattice with top element. Indeed, the join and meet operations are given by

$$X \sqcup^{\flat} Y = \max(X \cup Y)$$

[1] The name "algebraic" comes from lattice theory where it was motivated by the fact that algebraic lattices are exactly the lattices of subalgebras/congruences of algebras. Analog of the first result for certain cpos was given in Libkin [102].

$$X \sqcap^\sharp Y = \min(X \cup Y)$$

and empty set is the bottom (top) element with respect to $\sqsubseteq^\flat$ ($\sqsubseteq^\sharp$).

Furthermore, if A is a meet-semilattice, then the meet operation with respect to $\sqsubseteq^\flat$ exists:

$$X \sqcap^\flat Y = \max\{x \wedge y \mid x \in X, y \in Y\}$$

and, if A is bounded complete, then the join with respect to $\sqsubseteq^\sharp$ exists:

$$X \sqcup^\sharp Y = \min\{x \vee y \mid x \in X, y \in Y, x \vee y \text{ exists}\}$$

Another observation is almost obvious but it will be used numerous times in this thesis:

Lemma 2.1 *a)* $X \sqsubseteq^\flat Y$ *iff* $\max X \sqsubseteq^\flat \max Y$*;*
b) $X \sqsubseteq^\sharp Y$ *iff* $\min X \sqsubseteq^\sharp \min Y$*;*
c) $X \sqsubseteq^\natural Y$ *iff* $\max X \sqsubseteq^\flat \max Y$ *and* $\min X \sqsubseteq^\sharp \min Y$. □

Another observation that will be used later as a very important tool for the language design, is the following simple fact stating the universality properties of $\mathcal{P}^\flat(\cdot)$ and $\mathcal{P}^\sharp(\cdot)$.

Lemma 2.2 *a) Let A be a poset. Then for every join-semilattice with bottom element $\langle S, \vee, \bot\rangle$ and every monotone map $f : A \to S$, there exists a unique semilattice homomorphism $f^+ : \mathcal{P}^\flat(A) \to S$ that makes the following diagram commute:*

$$\begin{tikzcd} A \arrow[r, "\eta"] \arrow[dr, "f"'] & \mathcal{P}^\flat(A) \arrow[d, dashed, "\exists! f^+"] \\ & S \end{tikzcd}$$

b) Let A be a poset. Then for every meet-semilattice with top element $\langle S, \wedge, \top\rangle$ and every monotone map $f : A \to S$, there exists a unique semilattice homomorphism $f^+ : \mathcal{P}^\sharp(A) \to S$ that makes the following diagram commute:

$$\begin{tikzcd} A \arrow[r, "\eta"] \arrow[dr, "f"'] & \mathcal{P}^\sharp(A) \arrow[d, dashed, "\exists! f^+"] \\ & S \end{tikzcd}$$

Proof. We prove a) only. Consider a finite antichain $X = \{x_1, \ldots, x_n\}$ in A and define

$$f^+(X) = \begin{cases} \perp & \text{if } n = 0 \\ f(x_1) \vee \ldots \vee f(x_n) & \text{otherwise} \end{cases}$$

That f^+ is a homomorphism follows from monotonicity of f and $X = \eta(x_1) \sqcup^\flat \ldots \sqcup^\flat \eta(x_n)$ and its uniqueness follows from the definition. $\square$

It is well-known that both $\mathcal{P}^\flat(\cdot)$ and $\mathcal{P}^\sharp(\cdot)$ preserve bounded-completeness, see Gunter [67].

We now turn our attention to the Plotkin construction, for which we also give a somewhat unusual description. Define *conv*(A) as a subset of $\mathbb{A}_{\mathrm{fin}}(A) \times \mathbb{A}_{\mathrm{fin}}(A)$ that consists of pairs (X, Y) with $X \sqsubseteq^\natural Y$. These pairs are ordered by[2] $(X_1, Y_1) \sqsubseteq^\natural (X_2, Y_2)$ iff $X_1 \sqsubseteq^\sharp X_2$ and $Y_1 \sqsubseteq^\flat Y_2$. Notice that (X, Y) is in *conv*(A) iff there exists a finite set $Z_{(X,Y)} \subseteq A$ such that $X = \min Z_{(X,Y)}$ and $Y = \max Z_{(X,Y)}$; moreover, in this case $(X_1, Y_1) \sqsubseteq^\natural (X_2, Y_2)$ iff $Z_{(X_1,Y_1)} \sqsubseteq^\natural Z_{(X_1,Y_1)}$.

We define $\mathcal{P}^\natural(A)$ as $\langle \mathit{conv}(A) - \{(\emptyset, \emptyset)\}, \sqsubseteq^\natural \rangle$. The universality property for this construction is given by the following lemma which uses $\sqsubseteq$*-ordered semilattices*, i.e. semilattices $\langle S, *, \sqsubseteq \rangle$ in which $\cdot$ is monotone with respect to the partial order $\sqsubseteq$. That $\mathcal{P}^\natural(A)$ is a $\sqsubseteq^\natural$-ordered semilattice follows from the observation that $(X_1, Y_1) *^\natural (X_2, Y_2) = (X_1 \sqcap^\sharp X_2, Y_1 \sqcup^\flat Y_2)$ is a semilattice operation monotone with respect to $\sqsubseteq^\natural$.

Lemma 2.3 *Let A be a poset. Then for $\sqsubseteq$-monotone semilattice $\langle S, *, \sqsubseteq \rangle$ and every monotone map $f : A \to S$, there exists a unique $\sqsubseteq$-monotone semilattice homomorphism $f^+ : \mathcal{P}^\natural(A) \to S$ that makes the following diagram commute (where $\eta(a) = (\{a\}, \{a\})$):*

$$\begin{tikzcd} A \arrow[r, "\eta"] \arrow[dr, "f"'] & \mathcal{P}^\natural(A) \arrow[d, dashed, "\exists! f^+"] \\ & S \end{tikzcd}$$

Proof. Define f^+ by

$$f^+(\{x_1, \ldots, x_m\}, \{y_1, \ldots, y_n\}) = f(x_1) * \ldots * f(x_m) * f(y_1) * \ldots * f(y_n)$$

[2] Note abuse of notation, but it will not lead to ambiguities.

It is easy to see that f^+ is monotone with repsect to the additional order. It can also be seen that the above representation does not change if non-minimal (non-maximal) elements are added to the first (second) component. Hence, $f^+((X_1, Y_1) *^\natural (X_2, Y_2)) = (*_{x \in X_1 \cup X_2} f(x)) * (*_{y \in Y_1 \cup Y_2} f(y)) = (*_{x \in \min(X_1 \cup X_2)} f(x)) * (*_{y \in \max(Y_1 \cup Y_2)} f(y)) = f^+((X_1, Y_1)) * f^+((X_2, Y_2))$. That the diagram commutes follows from the definition of f^+. Its uniqueness follows from $(\{x_1, \ldots, x_m\}, \{y_1, \ldots, y_n\}) = \eta(x_1) *^\natural \ldots *^\natural \eta(x_m) *^\natural \eta(y_1) *^\natural \ldots *^\natural \eta(y_n)$. □

In the semantics of programming languages, usually the *ideal completion* is applied to $\mathcal{P}^\flat(A)$, $\mathcal{P}^\sharp(A)$ and $\mathcal{P}^\natural(A)$, where A is taken to be $\mathbf{K}D$ for some domain D. It is easy to see that in this case we obtain the standard constructions of the Hoare powerdomain [67], the Smyth powerdomain [157] and the Plotkin powerdomain [137]. However, for the purpose of this thesis we shall not need use the ideal completion.

For more information on domain theory, the reader is referred to Gunter [67], Gunter and Scott [68] and Abramsky and Jung [12].

2.2 Algebras

In this section we recall a few definitions from universal algebra. A *signature* is just a collection Ω of symbols, or operation names, with associated arities. An *algebra* is a pair $\langle A, \Omega \rangle$ where A is a set, called *carrier*, and each operation ω in Ω of arity n is interpreted as a function from A^n to A. We refer the reader to standard textbooks (Grätzer [64], Wechler [177]) for definitions of the concepts of homomorphism, subalgebra etc. If it does not give rise to ambiguity, we occasionally confuse an algebra with its carrier.

Let $\langle A, \Omega \rangle$ be an algebra and $X \subseteq A$. Then $[X]$ denotes the subalgebra of $\langle A, \Omega \rangle$ generated by X. Let $\mathcal{K}$ be a class of algebras of the same signature. We say that $\langle A, \Omega \rangle$ is *freely generated* by X in $\mathcal{K}$ if two conditions hold:

(i) $\langle A, \Omega \rangle$ is generated by X in , that is, $[X] = A$, and

(ii) for any algebra $\langle B, \Omega \rangle \in \mathcal{K}$ and any map $f : X \to B$ there exists a unique homomorphism $f^+ : \langle A, \Omega \rangle \to \langle B, \Omega \rangle$ such that the following diagram commutes, where η is the embedding of X into A:

$$\begin{tikzcd} X \arrow[r, hook, "\eta"] \arrow[dr, "f"'] & \langle A, \Omega \rangle \arrow[d, dashed, "\exists! f^{+}"] \\ & \langle B, \Omega \rangle \end{tikzcd}$$

Freely generated algebras need not exist for an arbitrary $\mathcal{K}$ and generally it is a hard result to prove their existence, see Grätzer [64]. One important case in which a positive result is well known is when $\mathcal{K}$ is a variety, or an equational class.

In this thesis we shall mostly work with ordered algebras. In mathematical literature freely generated ordered algebras are typically considered with respect to embeddings that disregard order, see Grätzer [64] and Bloom [24]. This no longer satisfies our needs in the denotational semantics which will be used throughout this thesis. The need for the theory of freely generated ordered algebras was recognized, for example, by Stoughton in his work on full abstraction [160]. Although there are still no general results about existence of ordered algebras freely generated by posets, most classes of algebras we shall consider do possess them, and we shall not be concerned with the lack of underlying mathematical theory, at least for the purpose of this work.

An *ordered algebra* $\langle A, \Omega \rangle$ has a predicate $\leq$ as one of the symbols in the signature; its interpretation is a partial order on the carrier. A monotone homomorphism $f : \langle A, \Omega \rangle \to \langle B, \Omega \rangle$ is a homomrphism which is monotone with respect to $\leq$. We say that $\langle A, \Omega \rangle$ is *freely generated* by $X \subseteq A$ in a class $\mathcal{K}$ if the condition (i) above holds, and for every other $\langle B, \Omega \rangle \in \mathcal{K}$ and a *monotone* $f : X \to B$ there exists a unique *monotone* f^{+} that makes the diagram above commute.

Occasionally, we shall also be slightly imprecise saying that an algebra $\langle A, \Omega \rangle$ is freely generated by a set X which is not a subset of A if the emdedding

$$X \xhookrightarrow{\eta} A$$

is understood. Of course by that we mean that $\langle A, \Omega \rangle$ is freely generated by $\eta(X)$.

Thus, we can reformulate lemma 2.2 as follows: For any poset A, $\mathcal{P}^{\flat}(A)$ is the free join-semilattice with bottom generated by A, and $\mathcal{P}^{\sharp}(A)$ is the free meet-semilattice with top generated by A.

2.3 Adjunctions and monads

The reader may skip this section and still be able to understand the rest of the thesis. However, certain concepts defined here are very useful for understanding the mathematical structure underlying the main principles of the language design. We refer the reader to Barr and Wells [21] or MacLane [112] for the definition of categories, functors and natural transformations.

First of all, let us define a number of categories that will be useful later.

Set, the category of sets;

FSet, the category of finite sets;

Poset, the category of posets and monotone maps;

FSL, the category of finite semilattices and semilattice homomorphisms. Two important subcategories are $\mathbf{FSL}_0$ and $\mathbf{FSL}_1$ that contain join (or meet) semilattices with bottom (top); the morphisms are required to preserve the special elements. If semilattices of arbitrary cardinality are considered, the corresponding categories are denoted by $\mathbf{SL}_0$ and $\mathbf{SL}_1$.

Ω-Alg, the category of $\Omega-$algebras and homomorphism between them.

Let $\mathbf{A}$ and $\mathbf{B}$ be categories and $\mathbb{F} : \mathbf{A} \to \mathbf{B}$ and $\mathbb{G} : \mathbf{B} \to \mathbf{A}$ be two functors. Then $\mathbb{F}$ is *left adjoint* to $\mathbb{G}$ and $\mathbb{G}$ is *right adjoint* to $\mathbb{F}$, written $\mathbb{F} \dashv \mathbb{G}$, if the following two conditions hold:

(i) There exists a natural transformation $\eta : \mathsf{id} \to \mathbb{G}\mathbb{F}$, and

(ii) For any object A of $\mathbf{A}$, any object B of $\mathbf{B}$ and any arrow $A \xrightarrow{f} \mathbb{G}(B)$ in $\mathbf{A}$ there exists a unique $\mathbb{F}(A) \xrightarrow{g} B$ in $\mathbf{B}$ such that the following diagram (in $\mathbf{A}$) commutes (where η_A is the A-component of η):

$$\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, "f"'] & \mathbb{G}\mathbb{F}(A) \arrow[d, dashed, "\mathbb{G}(g)"] \\
& \mathbb{G}(B)
\end{tikzcd}$$

The property expressed by the diagram is called the *universal mapping property* or just *universality property.* It is closely related to the freeness conditions considered in the previous sections, as a few examples below show. In all of them, the right adjoint is the forgetful functor $\mathbb{U}$ that "forgets" the additional structure.

1. Powerset can be considered as a functor $\mathbb{P} : \mathbf{Set} \to \mathbf{SL}_0$ that takes a set and returns its powerset considered as a semilattice under the inclusion ordering. Its action on morphisms

is defined by "mapping" a function f from a set X to a set Y over subsets of X, i.e. $\mathbb{P}(f)(A) = \{f(a) \mid a \in A\}$. Then $\mathbb{P}$ is left adjoint to $\mathbb{U} : \mathbf{SL}_0 \to \mathbf{Set}$. In other words, $\mathbb{P}(X)$ is the free join-semilattice with bottom generated by X.

Restricting to finite sets, we obtain an adjunction $\mathbb{P}_{\mathrm{fin}} \dashv \mathbb{U}$.

2. $\mathcal{P}^\flat(\cdot)$ can be considered as a functor from **Poset** to $\mathbf{SL}_0$. Its action on a monotone map $f : X \to Y$ is given by $\mathcal{P}^\flat(f) : \mathcal{P}^\flat(X) \to \mathcal{P}^\flat(Y)$ where $\mathcal{P}^\flat(f)(A) = \max\{f(a) \mid a \in A\}$. According ot the lemmas proved above, $\mathcal{P}^\flat(\cdot)$ is left adjoint to $\mathbb{U} : \mathbf{SL}_0 \to \mathbf{Poset}$. Similarly for $\mathcal{P}^\sharp(\cdot) : \mathbf{Poset} \to \mathbf{SL}_1$, we have $\mathcal{P}^\sharp(\cdot) \dashv \mathbb{U}$. Note that the action of $\mathcal{P}^\sharp(\cdot)$ on morphisms is given by $\mathcal{P}^\sharp(f)(A) = \min\{f(a) \mid a \in A\}$.

3. More generally, let $\mathcal{K}$ be a full subcategory of Ω-**Alg**. Assume that for each set X, a free algebra $F_{\mathcal{K}}(X)$ generated by X in $\mathcal{K}$ exists. Then $F_{\mathcal{K}}$ can be considered as a functor from **Set** to Ω-**Alg** whose action on morphisms is given by the universality property. Then $F_{\mathcal{K}} \dashv \mathbb{U}$.

Associated with every adjunction $\mathbb{F} \dashv \mathbb{G}$ there is another natural transformation, $\epsilon : \mathbb{F}\mathbb{G} \to \mathsf{id}_{\mathbf{B}}$. The details of its construction can be found in Barr and Wells [21] and MacLane [112].

The next construct to be introduced is closely associated with adjunctions. Given a category $\mathbf{A}$, a *monad* on it is a triple $\langle \mathbb{T}, \eta, \mu \rangle$ where $\mathbb{T}$ is an endofunctor (i.e. a functor $\mathbb{T} : \mathbf{A} \to \mathbf{A}$) and $\eta : \mathsf{id} \to \mathbb{T}$ and $\mu : \mathbb{T}^2 \to \mathbb{T}$ are natural transformations such that the following diagrams commute:

$$
\begin{tikzcd}
\mathbb{T} \arrow[r, "\eta\mathbb{T}"] \arrow[dr, "="'] & \mathbb{T}^2 \arrow[d, "\mu"] & \mathbb{T} \arrow[l, "\mathbb{T}\eta"'] \arrow[dl, "="] \\
& \mathbb{T} &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbb{T}^3 \arrow[r, "\mathbb{T}\mu"] \arrow[d, "\mu\mathbb{T}"'] & \mathbb{T}^2 \arrow[d, "\mu"] \\
\mathbb{T}^2 \arrow[r, "\mu"'] & \mathbb{T}
\end{tikzcd}
$$

Every adjunction $\mathbb{F} \dashv \mathbb{G}$ where $\mathbb{F} : \mathbf{A} \to \mathbf{B}$ and $\mathbb{G} : \mathbf{B} \to \mathbf{A}$ gives rise to a monad $\langle \mathbb{G}\mathbb{F}, \eta, \mathbb{G}\epsilon\mathbb{F} \rangle$ on $\mathbf{A}$.

Consider three examples of this construction that will be used throughout the thesis.

1. Consider $\mathbb{P}_{\mathrm{fin}} \dashv \mathbb{U}$ as an adjunction between finite sets and semilattices. Then it gives rise to the monad $\langle \mathbb{P}_s, \eta, \mu \rangle$ where $\mathbb{P}_s$ is the powerset functor from **FSet** to itself, and for each finite set X we have:

$$
\begin{array}{rcr}
\eta_X : X \to \mathbb{P}_s(X) & \qquad & x \stackrel{\eta_X}{\longmapsto} \{x\} \\
\mu_X : \mathbb{P}_s(\mathbb{P}_s(X)) \to \mathbb{P}_s(X) & \qquad & \{X_1, \ldots, X_n\} \stackrel{\mu_X}{\longmapsto} X_1 \cup \ldots \cup X_n
\end{array}
$$

2. Consider $\mathcal{P}^\flat \dashv \mathbb{U}$ as an adjunction between posets and semilattices. It gives rise to the monad $\langle \mathcal{P}^\flat, \eta, \mu \rangle$ where $\mathcal{P}^\flat$ is now considered as a functor from **Poset** to itself, and for each poset A we have:
$$\eta_A : A \to \mathcal{P}^\flat(X) \qquad a \overset{\eta_A}{\longmapsto} \{a\}$$
$$\mu_A : \mathcal{P}^\flat(\mathcal{P}^\flat(X)) \to \mathcal{P}^\flat(X) \qquad \{X_1, \ldots, X_n\} \overset{\mu_A}{\longmapsto} X_1 \sqcup^\flat \ldots \sqcup^\flat X_n = \max(X_1 \cup \ldots \cup X_n)$$
3. The construction for $\mathcal{P}^\sharp \dashv \mathbb{U}$ is similar except that $\sqcap^\sharp$ is used instead of $\sqcup^\flat$.

It is also known that the converse is true as well, that is, every adjunction comes from a monad, The construction is due to Eilenberg and Moore and it is out of the scope of this thesis. In the rest of this section we define another construction giving an alternative description of monads that inspired some of the programming primitives we will be working with.

Let $\mathcal{T} = \langle \mathbb{T}, \eta, \mu \rangle$ be a monad on $\mathbf{A}$. Then the *Kleisli category* for $\mathcal{T}$, denoted by $Kl(\mathcal{T})$, has the same objects as $\mathbf{A}$, and its arrows are arrows $A \longrightarrow \mathbb{T}(B)$ in $\mathbf{A}$. The composition is obtained by using the properties of the monad. To compose $A \xrightarrow{f} \mathbb{T}(B)$ and $B \xrightarrow{g} \mathbb{T}(C)$ in $Kl(\mathcal{T})$, we obtain an arrow from A to $\mathbb{T}(C)$ by

$$A \xrightarrow{\quad f \quad} \mathbb{T}(B) \xrightarrow{\mathbb{T}(g)} \mathbb{T}^2(C) \xrightarrow{\quad \mu_C \quad} \mathbb{T}(C)$$

The identity is simply $\eta_A : A \to \mathbb{T}(A)$.

There are two functors associated with the Kleisli category. One of them, $\mathbb{G} : Kl(\mathcal{T}) \to \mathbf{A}$ coincides with $\mathbb{T}$ on objects and, given a morphism $A \xrightarrow{f} \mathbb{T}(B)$, produces a morphism $\mathbb{T}(A) \longrightarrow \mathbb{T}(B)$ in $\mathbf{A}$ as follows:

$$\mathbb{T}(A) \xrightarrow{\mathbb{T}(f)} \mathbb{T}^2(B) \xrightarrow{\quad \mu_B \quad} \mathbb{T}(B)$$

The other one, $\mathbb{F} : \mathbf{A} \to Kl(\mathcal{T})$, is the identity on objects, and for $A \xrightarrow{g} B$ in $\mathbf{A}$ it produces a morphism $A \longrightarrow \mathbb{T}(B)$ is $Kl(\mathcal{T})$ as follows:

$$A \xrightarrow{\quad \eta_A \quad} \mathbb{T}(A) \xrightarrow{\mathbb{T}(g)} \mathbb{T}(B)$$

The reason $Kl(\mathcal{T})$ can be called an alternative representation of a monad is the following. For $\mathbb{F}$ and $\mathbb{G}$ just constructed, $\mathbb{F} \dashv \mathbb{G}$, $\mathbb{G}\mathbb{F} = \mathbb{T}$ and the monad associated with $\mathbb{F} \dashv \mathbb{G}$ is $\mathcal{T}$.

Let us apply the Kleisli constructions to the three main examples of this section. In those examples, for the reasons that should emerge shortly, we shall use the notation *ext* for the action of $\mathbb{G}$ on morphisms.

1. Consider $\langle \mathbb{P}_s, \eta, \mu\rangle$ associated with the adjunction $\mathbb{P} \dashv \mathbb{U}$. Given a function $f : X \to \mathbb{P}_{\text{fin}}(Y)$, $ext(f)$ is a function $\mathbb{P}_{\text{fin}}(X) \to \mathbb{P}_{\text{fin}}(Y)$ given by $ext(f) = \mu \circ \mathbb{P}_s(f)$ or equivalently
$$ext(f)(Z) = \bigcup_{z \in Z} f(z)$$

2. Consider $\langle \mathcal{P}^\flat(\cdot), \eta, \mu\rangle$ given by $\mathcal{P}^\flat(\cdot) \dashv \mathbb{U}$. For a monotone map $f : A \to \mathcal{P}^\flat(B)$, we have $ext(f) = \mu \circ \mathcal{P}^\flat(f)$, or,
$$ext(f)(C) = \sqcup^\flat_{c \in C} f(c) = \max(\bigcup_{c \in C} f(c))$$

3. Similarly, for $\langle \mathcal{P}^\sharp(\cdot), \eta, \mu\rangle$ arising from $\mathcal{P}^\sharp(\cdot) \dashv \mathbb{U}$ and a monotone $f : A \to \mathcal{P}^\flat(B)$, we have
$$ext(f)(C) = \sqcap^\sharp_{c \in C} f(c) = \min(\bigcup_{c \in C} f(c))$$

This justifies calling $\mathbb{G}$ on morphisms *ext*: it extends the action of $f : A \to \mathbb{T}(B)$ to $\mathbb{T}(A)$. If *ext* is given, it is also possible to reconstruct the functor $\mathbb{T}$ and the natural transformation $\mu : \mathbb{T}^2 \to \mathbb{T}$ for a given η. If $A \xrightarrow{g} B$ in $\mathbf{A}$, then $\mathbb{T}(g)$ is given by $ext(A \xrightarrow{g} B \xrightarrow{\eta_B} \mathbb{T}(B))$. For any object A, μ_A is obtained as $ext(\mathbb{T}^2(A) \xrightarrow{\mathsf{id}} \mathbb{T}^2(A))$. The reader can easily check that in all three examples above, if we start with *ext*, we obtain the corresponding functor and the natural transformation μ.

There have been two primary motivations for using monads in computer science. One is application in the language design, which will be considered in detail later. Another one is using monads to define a general notion of computation. This idea is due to Moggi [118] who defined $\mathbb{T}(A)$ as "computations of type A", where A could be a set or a domain or any other semantic object representing a type. In fact, Moggi used a slightly more general construction that also accommodates terminal objects and binary products. The use of monads to structure functional programs was discussed in Wadler [176]. A dual construction – comonad – was used by Brookes and Van Stone [29] in their work on intensional semantics of programming languages.

2.4 Rewrite systems

We shall need some basic facts about abstract and term rewrite systems, namely Newman's lemma and the critical pair lemma. For more information on rewrite systems, see Dershowitz and Jouannand [49] and Wechler [177].

An *asbtract rewrite system* (or reduction system) is a pair $\langle A, \to \rangle$ where A is a set and $\to$ is a binary relation. The transitive-reflexive closure of $\to$ will be denoted by either $\twoheadrightarrow$ or $\stackrel{*}{\to}$. The symmetric closure of $\to$, that is, $\to \cup \to^{-1}$ is denoted by $\longleftrightarrow$, and we occasionally use $\leftarrow$ for $\to^{-1}$.

An element $a \in A$ is said to be a *normal form* if there is no $b \in A$ such that $a \to b$. An element $a \in A$ *admits* a normal form if there exists a normal form a' such that $a \twoheadrightarrow a'$. We will mostly be interested in systems in which every element admits a unique normal form.

A rewrite system is called *terminating* or *strongly normalizing* if there is no infinite sequence of rewritings $a_1 \to a_2 \to \ldots a_n \to \ldots$. It is called *confluent* or *Church-Rosser* if for any $a \twoheadrightarrow a_1$ and $a \twoheadrightarrow a_2$ there exists $a' \in A$ such that $a_1 \twoheadrightarrow a'$ and $a_2 \twoheadrightarrow a'$. Diagramically,

$$\begin{tikzcd} & a \arrow[dl, ->>] \arrow[dr, ->>] & \\ a_1 \arrow[dr, ->>] & & a_2 \arrow[dl, ->>] \\ & a' & \end{tikzcd}$$

In a terminating Church-Rosser rewrite system every element admits a unique normal form. However, the property of being Church-Rosser is usually hard to verify, and this is due to the fact that the condition $a_1 \twoheadleftarrow a \twoheadrightarrow a_2$ is rather complicated. Replacing it by $a_1 \longleftarrow a \longrightarrow a_2$ we obtain *weak Church-Rosser* systems. Precisely, a rewrite system is called weakly Church-Rosser if for any $a_1 \longleftarrow a \longrightarrow a_2$ there exists $a' \in A$ such that $a_1 \twoheadrightarrow a'$ and $a_2 \twoheadrightarrow a'$. Diagramically,

$$\begin{tikzcd} & a \arrow[dl] \arrow[dr] & \\ a_1 \arrow[dr, ->>] & & a_2 \arrow[dl, ->>] \\ & a' & \end{tikzcd}$$

Fortunately, in many cases verifying this suffices, because

Lemma 2.4 (Newman) *A terminating rewrite system is Church-Rosser iff it is weakly Church-Rosser.* □

Term rewrite systems constitute the most important example of abstract rewrite systems. Let Ω be a signature and X a set of variables. Then $T_\Omega(X)$ denotes the set of all terms that can be constructed from X by using operations from Ω. A (term) rewrite rule is an expression $t_1 \to t_2$ where $t_1, t_2 \in T_\Omega(X)$. A (term) *rewrite system* is a finite collection R of term rewrite rules. Associated with R, there is a binary relation $\longrightarrow_R$ on $T_\Omega(X)$ defined as follows. Assume $s \to t \in R$, w is a term and σ is a substitution. Then $w[s\sigma] \longrightarrow_R w[t\sigma]$ where $s\sigma$ and $t\sigma$ are substituted at the same position in w, and no other terms are related by $\longrightarrow_R$.

We shall need two results that establish when a term rewrite system is terminating and Church-Rosser.

Lemma 2.5 *Assume that there exists a function $\varphi : T_\Omega(X) \to \mathbb{N}$ such that $u \longrightarrow_R v$ implies $\varphi(u) > \varphi(v)$. Then $\longrightarrow_R$ is terminating.* $\square$

Let $s \longrightarrow t$ and $u \longrightarrow v$ be two rewrite rules in R, with no variable in common (this can be done by renaming variables appropriately). Suppose that a subterm of s at position p is not a variable and is unifiable with u, and let σ be the most general unifier. Then $(t\sigma, s\sigma[v\sigma])$ (where $v\sigma$ is substituted at the position p) is called a *critical pair*. $CP(R)$ stands for the set of all critical pairs between the rules of R.

Lemma 2.6 (Critical pair lemma) *Let R be a term rewrite system. If $s \longleftarrow_R u \longrightarrow_R t$, then either there exists a term v such that $s \twoheadrightarrow_R v \twoheadleftarrow_R t$, or else $s \longleftrightarrow_{CP(R)} t$.* $\square$

Applying the Newman lemma, we get

Corollary 2.7 *A terminating rewrite system R is Church-Rosser iff $CP(R) \subseteq \twoheadrightarrow_R \circ \twoheadleftarrow_R$. In other words, for every critical pair (s, t) there exists a term v such that $s \twoheadrightarrow_R v \twoheadleftarrow_R t$.* $\square$

Chapter 3

Preliminaries

This chapter covers the foundation for the study of the semantics of partiality and languages to work with partial information. As we have observed earlier, the unifying theme for various kinds of partial information is using ordered sets as their semantics, where meaning of the order is "being more informative". There exist standard mathematical models for flat and nested relations without partial information. Once orderings on values come into play, there is a need in new basic models for incomplete databases. The first attempt to come up with such a model was done by Buneman, Jung and Ohori [33] and it was further developed in Libkin [99], Jung, Libkin and Puhlmann [88] and Levene and Loizou [95]. We present the model in the first section and study some of its properties. In particular, we show how to redefine the notions of scheme, functional and multivalued dependencies and operations of the relational algebra. Ordered sets that we use for modeling partiality are *domains* typically used in the programming semantics.

Secondly, we must develop a framework for the query language design. In the second section, we explain Cardelli's *data-oriented* programming [35], in particular, the idea of using introduction and elimination operations for programming with data. We then go on and explain the approach of Buneman, Breazu-Tannen and Naqvi [25] that suggests to derive data-oriented languages from operations naturally associated with the data. Those operations come from the *universality properties*. Their approach was further developed in Breazu-Tannen, Buneman and Wong [26] where a simple reformulation of the nested relational algebra was found. Even more importantly, [26] suggested a uniform way of getting rid of non-well-definedness of programs. This way, in categorical terms, is going from an adjunctions to the associated monad. Before applying this approach to partial data in chapter 5, we demonstrate its usefulness by showing how it can be used with bags (multisets). Some of the results are taken from Libkin and Wong [105, 106, 107, 108].

3.1 Databases with partial information and domain theory

As we have said many times in the introduction, most models of partiality of data can be represented via orderings on values. In this section we study a new approach to databases which treats relations not as subsets of a Cartesian product but as subsets of some domain – a partially ordered space of descriptions. This approach permits generalizations of relations that admit null values and variants. We show how to define the notion of a relation *scheme* in such a setting. We study properties of schemes. Then we show that operations analogous to projection, selection and join retain the desired properties. Schemes also allow us to develop dependency theory for such generalized relations. They play an essential role in an extension of this model which admits a set constructor and is therefore useful for the study of higher-order relations and their generalizations.

Throughout this section, we consider only Scott domains.

3.1.1 Order on objects and partiality

It has recently been discovered by Buneman et al. [33] that a representation of the underlying principles of relational database theory can be found in the theory of domains which has been developed as the basis of the denotational semantics of programming languages. This representation does not take into account the details of the data structure and, therefore, allows us to extend the main principles of relational databases to much more general constructions. Use of domain theory in the generalization of relational databases may also help to establish the connection between data models and types, that is, to represent database objects (not necessarily relational databases) as typed objects in programming languages.

In denotational semantics of programming languages expressions denote values, and the domains of values are partially ordered. A database is a collection of objects having descriptions and meanings. The meaning is the set of all possible objects described by a description. The meaning having been defined as a set, we can order descriptions by saying that a description d_1 is better than a description d_2 if it describes fewer objects, i.e. if it is a more precise description.

Let $[\![d]\!]$ stand for the meaning of d. Suppose that d_1 and d_2 are the records in a relational database and

$$d_1 = [\text{Dept: CIS}, \text{Office: 176}],$$

$$d_2 = [\text{Name: John}, \text{Dept: CIS}, \text{Office: 176}]$$

Assume that name, department and office are the only attributes. Then the meaning of d_1 is the set of all possible records that refer to CIS people in office 176, in particular, d_2. Therefore,

d_2 is better than d_1 because $[\![d]\!]_2 \subseteq [\![d]\!]_1$.

If all descriptions of objects come from the same domain D which is partially ordered by $\leq$, then we can give the following definition of $[\![d]\!]$:

$$[\![d]\!] \stackrel{\text{def}}{=} \{d' \in D \mid d' \geq d\} = \uparrow d$$

Now the following is immediate:

Lemma 3.1 *$d_1 \leq d_2$ iff $[\![d_2]\!] \subseteq [\![d_1]\!]$.* □

The above ordering corresponds to the usual one in the theory of databases with incomplete information. For relations, it was used by Biskup [23], Imielinski and Lipski [78] and others. The same idea of ordering was used for complex objects in Bancilhon and Khoshafian [20].

Before we go any further, let us fix the notation for records. A record with field names $l_1, \ldots, l_k$ and corresponding values $v_1, \ldots, v_k$ will be denoted by $[l_1\colon v_1, \ldots, l_k\colon v_k]$. We use the [] brackets as others will be used for various collections later on. Fortunately, until chapter 6 when we switch to the ML notation, lists are not used, and we are able to avoid confusion. We shall denote the l_ith field of a record r by $r(l_i)$ or $r.l_i$.

Let $\mathcal{V}_\perp = \mathcal{V} \cup \{\perp\}$ where $\mathcal{V}$ is a set of non-partial values, $\perp$ corresponds to incomplete information (it is a generic null) and $\forall v \in \mathcal{V} : \perp \leq v$ while all elements of $\mathcal{V}$ are incomparable. In other words, $\mathcal{V}_\perp$ is a flat domain. Let $\mathcal{L}$ be a set of attributes (in the above example $\mathcal{L} = \{\text{Name}, \text{Dept}, \text{Office}\}$). Then the set of functions from $\mathcal{L}$ to $\mathcal{V}_\perp$, denoted by $\mathcal{L} \to \mathcal{V}_\perp$, is ordered by $d_1 \leq d_2$ iff $d_1(l) \leq d_2(l)$ for all $l \in \mathcal{L}$ where $d_1, d_2 : \mathcal{L} \to \mathcal{V}_\perp$. For example, if d_1 and d_2 are as in the above example, $\mathcal{L} = \{\text{Name}, \text{Dept}, \text{Office}\}$ and $\mathcal{V}$ contains names of departments, people and numbers of offices, then $d_1, d_2 \in \mathcal{L} \to \mathcal{V}_\perp$ since $d_1 = [\text{Name: } \perp, \text{Dept: CIS}, \text{Office: } 176]$. Obviously $d_1 \leq d_2$.

There is an alternative way of giving semantics of partial description by using *maximal* elements of D. Recall that for every Scott domain D there exists a set $D^{\max} \subseteq D$ such that for every $d \in D$ there is $d_m \in D^{\max}$ such that $d \leq d_m$. In other words, $D^{\max}$ is the antichain of maximal elements. For example, in the case of $\mathcal{L} \to \mathcal{V}_\perp$, maximal elements are precisely records without nulls, that is, records without incomplete information. Therefore, it was proposed by some [31, 33, 78] to redefine semantics as

$$[\![d]\!]_{\max} \stackrel{\text{def}}{=} \{d' \in D^{\max} \mid d' \geq d\} = \uparrow d \cap D^{\max}$$

Let us briefly outline some problems with this approach (we shall see more when we study approximations). First, consider (just informally) recursive values with nulls. For instance, if we have a type declaration `person = [name:string, father:person]`, then elements of this type are potentially infinite sequences of names. In fact, if C is a domain of strings, then the semantics of type `person` is given by a solution to the recursive domain equation $D = C \times D$.

Maximal elements of D are then *infinite* sequences of maximal elements of C and it is unlikely we would be interested in approximating those. In fact, we are interested in descriptions of finite length ending with infinitely many bottom elements, *i.e.* generic nulls.

Unfinished experiments are another example. They are just sequences of observations made, say, every day. Formally, such experiments are partial functions from $\mathbb{N}$ to some domain C, and these are ordered by $f_1 \leq f_2 \Leftrightarrow \forall n : f_1(n)$ is defined $\Rightarrow f_2(n)$ is defined and $f_1(n) \leq f_2(n)$. In this example maximal elements are totally defined functions f with $\mathsf{im}(f) \subseteq C^{\max}$. Again, we see that partiality of information does not necessarily mean trying to approximate maximal elements, which are never reached.

Finally, using $[\![]\!]_{\max}$ we lose the nice connection between the ordering and semantics. It is no longer the case that $[\![d_1]\!]_{\max} \supseteq [\![d_2]\!]_{\max} \Leftrightarrow d_1 \leq d_2$. A simple counterexample is a finite chain: for all elements their $[\![]\!]_{\max}$ is the top element.

Looking at these examples gives us another important observation. Elements of type `person` that can be stored in a database are precisely finite sequences of names. Unfinished experiments that can be stored are precisely partial functions with finite domains whose values can be stored. Mathematically speaking, these are *compact elements.* This fits very well with the semantics of compact elements proposed by Dana Scott: in his approach they are "computable" functions; in our approach they are *objects that can be stored in a database.*

We are now in the position to explain the main idea of Buneman et al. [33]. Consider the domain $\mathcal{L} \to \mathcal{V}_\perp$. Its elements are records whose attributes are elements of $\mathcal{L}$ and values are taken from $\mathcal{V}_\perp$. The relations are finite sets of records, that is, finite subsets of $\mathcal{L} \to \mathcal{V}_\perp$. However, not every finite subset of $\mathcal{L} \to \mathcal{V}_\perp$ corresponds to a relation. If we have a subset containing both d_1 and d_2 from our example, d_1 is less informative than d_2 and should be removed (notice that we can not argue this way for bags. Indeed, as we will show later, the ordering on bags of partial descriptions is quite different from the ordering on sets). Less informative here means that $d_1 \leq d_2$. Therefore, relations correspond to finite subsets of domains that do not contain comparable elements, that is, to *antichains.* Combining this with the idea of the previous paragraph that elements that can be stored correspond to compact elements of domains, we arrive at the (slightly changed) principle proposed by Buneman, Jung and Ohori [33]:

Generalized relations are finite antichains of compact elements

Example 3.1 Let $\mathcal{L}$ and $\mathcal{V}$ be as in the above examples. Let

$$d_3 = [\text{Name: Ann}, \text{Dept: Math}, \text{Office: } 628],$$
$$d_4 = [\text{Name: Ann}, \text{Dept: Math}, \text{Office: } \perp]$$

(d_4 shows that the person has not been assigned an office yet). Then $\{d_2, d_3\}$ is a generalized relation but neither $\{d_1, d_2\}$ nor $\{d_3, d_4\}$ is because $d_1 \leq d_2$ and $d_4 \leq d_3$. □

In the examples above we considered only one generic null value. We have seen already other kinds of null values and orders on them. To order more complex structures, it is necessary to *lift* orders. To lift order to records is easy: it is just done componentwise. To lift order to sets is more problematic as domain theory does not provide us with a universal way to do so. We shall briefly discuss lower and upper orderings in this chapter, and later in chapter 4 we justify using the lower ordering for lifting to relations.

3.1.2 Schemes

In the subsequent sections we shall develop dependency theory and a simple query language for the model presented above. To do so, we need to find an analog of the concept of *scheme*. Recall that in the relational database theory, a *scheme* is just a subset of attributes. We need this concept to formulate the definitions of functional and multivalued dependencies and later to define analogs of the relational algebra operations such as projection and selection. In this subsection we consider two definitions: one due to Bunemanet al. [33] and the other due to Libkin [99].

Consider the usual relational algebra projection, where relations are allowed to have nulls. Let $\mathcal{L}'$ be a subset of the set of attributes $\mathcal{L}$. If $y \leq p_{\mathcal{L}'}(x)$, then y has nulls in the positions corresponding to attributes in $\mathcal{L} - \mathcal{L}'$. Hence, $p_{\mathcal{L}'}(y) = y$, and this shows that $\mathcal{I}_{\mathcal{L}'} = \{p_{\mathcal{L}'}(x) \mid x \in \mathcal{L} \to \mathcal{V}_\perp\}$ is an ideal. Furthermore, if $p_{\mathcal{L}'}(x) \vee p_{\mathcal{L}'}(y)$ exists, it still has nulls in all positions corresponding to $\mathcal{L} - \mathcal{L}'$, and hence it belongs to $\mathcal{I}_{\mathcal{L}'}$.

Ideals which are closed under existing suprema are called *strong ideals*. The observation we have just made shows that as the first approximation to the definition of scheme we can take strong ideals. However, this is not good enough as the following example shows:

Example 3.2 Let $\mathcal{L}$ and $\mathcal{V}$ be as in the examples of the previous subsection. Let

$$\mathcal{I}_1 = \{[\text{Name: } v, \text{Dept: } \perp, \text{Office: } \perp] \mid v \in \mathcal{V}_\perp\}.$$

Then $\mathcal{I}_1$ is a strong ideal and for any $d = [\text{Name: } v_1, \text{Dept: } v_2, \text{Office: } v_3]$ its projection onto $\mathcal{I}_1$ is $p_{\mathcal{I}_1}(d) = [\text{Name: } v_1, \text{Dept: } \perp, \text{Office: } \perp]$. The set of maximal elements of $\mathcal{I}_1$ is $\{[\text{Name: } v,$ Dept: $\perp$, Office: $\perp] \mid v \in \mathcal{V}\}$.

Let $\mathcal{I}_2 = \downarrow d$ where $d \in \mathcal{L} \to \mathcal{V}_\perp$. Then $\mathcal{I}_2$ is a strong ideal with unique maximal element d and for any $d' \in \mathcal{L} \to \mathcal{V}_\perp : p_{\mathcal{I}_2}(d') = d \wedge d'$. □

Therefore, we need more for the analogy of projection in relational algebra than being a projection onto a strong ideal. In fact, that ideal must satisfy some additional properties. In $\mathcal{L} \to \mathcal{V}_\perp$ schemes correspond to subsets of $\mathcal{L}$. That is, a projection onto the scheme corresponding to

$S \subseteq \mathcal{L}$ is given by

$$p_S(x) = x' \text{ where } x'(l) = x(l) \text{ if } l \in S \text{ and } x'(l) = \perp \text{ otherwise.}$$

These projections will be called *canonical.* It is a natural requirement for the definition of scheme and projection in an arbitrary domain that the projections be canonical when restricted to $\mathcal{L} \to \mathcal{V}_\perp$. One can easily see that for every $x \in \mathcal{L} \to \mathcal{V}_\perp$ the ideal $\downarrow x$ is strong while the projection $p_{\downarrow x}$ is not canonical.

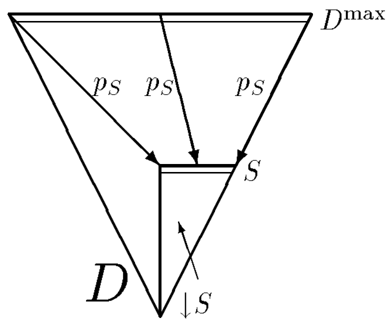
In $\mathcal{L} \to \mathcal{V}_\perp$ schemes correspond to the subsets of $\mathcal{L}$ and projections to the canonical projections. It is natural to define the concept of scheme such that, being applied to $\mathcal{L} \to \mathcal{V}_\perp$, it will give rise exactly to canonical projections. Also, schemes should be significant parts of a domain that reflect the structure of the whole domain. This means that if the elements of a domain are treated as database objects (for example, records in relations), then projection into an ideal generated by a scheme should correspond to losing some piece of information and the same pieces of information are lost for all the elements of the domain. This means that projections generated by schemes are in a way homogeneous.

If we have two maximal elements of a domain (complete descriptions) and they are projected into a scheme (i.e. the same pieces of information are ignored) then the projections can not be comparable. This observation leads us to the following definition.

Definition 3.1 *Let D be a domain and S an antichain in D such that $\downarrow S$ is a strong ideal. Then S is called a* scheme *in D if projection $p_{\downarrow S}(x)$ of any element of $x \in D^{\max}$ is a maximal element in $\downarrow S$.*

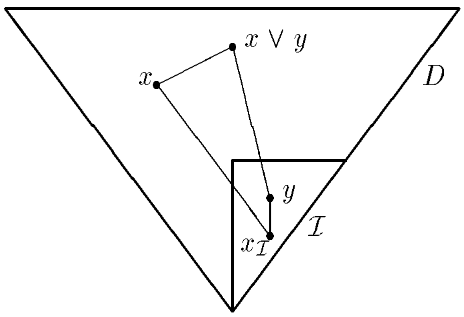
If $S \subseteq D$ is a scheme, then $\downarrow S$ is called a *scheme-ideal* and $p_{\downarrow S}$ is called a *scheme-projection.* We shall write p_S instead of $p_{\downarrow S}$.

The picture below illustrates these concepts.



In the reasonings that led us to the above definition we took into account only *how* we lose information by projecting into a scheme. In Buneman et al. [33] another aspect of the problem was considered : what can be said about the lost information? Can we consider it independently and "add" to another object (element of a domain)?

The idea of [33] was that, given a scheme S, there is its complement (as there is a complement $\mathcal{L}-S$ for every $S \subseteq \mathcal{L}$ for the domain $\mathcal{L} \to \mathcal{V}_\perp$), and projecting into S is simply losing information corresponding to the complement of S. Assuming that the pieces of information contained in projections into the scheme and its complement are independent, we can combine them. To be more precise, if we have an object and its projection into a scheme is less than an element of this scheme, we can add lost information to the latter element. This corresponds precisely to the *slide condition* of Buneman et al. [33]. We say that a strong ideal $\mathcal{I}$ satisfies the slide condition if for any $x \in D$ and $y \in \mathcal{I}$, $p_\mathcal{I}(x) \leq y$ implies that $x \vee y$ exists. This property obviously holds for canonical projections in $\mathcal{L} \to \mathcal{V}_\perp$. The following picture illustrates the slide condition ($x_\mathcal{I}$ stands for $p_\mathcal{I}(x)$):



Definition 3.2 ([33, 88]) *Let D be a domain and S an antichain such that $\downarrow S$ is a strong ideal. Then S is called a* semi-factor *if $\downarrow S$ satisfies the slide condition, that is, if $x \in D$ and $y \in \downarrow S$ are such that $p_S(x) \leq y$, then $x \vee y$ exists. $\downarrow S$ is called a semi-factor ideal, and p_S is called a semi-factor projection.*

Every semi-factor is a scheme; the converse is not true in general. If it were true, it would mean (informally) that for all the schemes their complements exist, because we could consider the paragraph before the definition of semi-factor as an informal proof. In a certain class of domains this can be formally proved, and we will finish this section with such a result.

Example 3.3 Let d_2, d_3 be as in the examples 1 and 2. Let

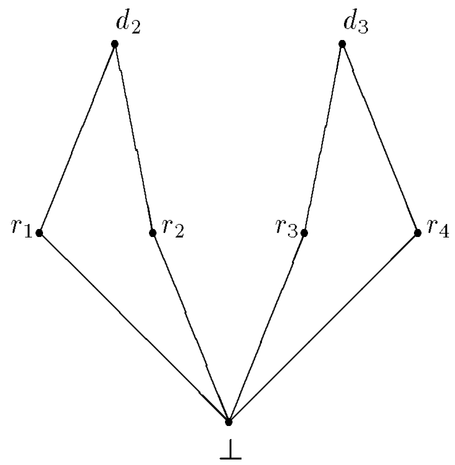
$$r_1 = [\text{Name: John}, \text{Dept: } \perp, \text{Office: } \perp],$$

$$r_2 = [\text{Name: } \perp, \text{Dept: CIS}, \text{Office: 176}],$$

$$r_3 = [\text{Name: Ann}, \text{Dept: Math}, \text{Office: } \perp],$$

$$r_4 = [\text{Name: } \perp, \text{Dept: } \perp, \text{Office: } 628].$$

Let $D = \{d_2, d_3, r_1, r_2, r_3, r_4, \perp\}$ where $\perp$ is the tuple with all null values. The diagram of D is shown below:



This domain has no semi-factors except $\{\perp\}$ and $D^{\max}$ while it has eight proper schemes: $\{r_1, r_3\}$, $\{r_2, r_3\}$, $\{r_1, r_4\}$, $\{r_2, r_4\}$, $\{d_2, r_3\}$, $\{d_2, r_4\}$, $\{d_3, r_1\}$, $\{d_3, r_2\}$. □

In order to justify both definitions we must prove that they describe exactly canonical projections when applied to the domain $\mathcal{L} \to \mathcal{V}_\perp$.

Proposition 3.2 *S is a scheme (or a semi-factor) of $\mathcal{L} \to \mathcal{V}_\perp$ iff p_S is a canonical projection.*

Proof. We prove the proposition for schemes only. (See [33] for semi-factors). The *'if'* part is immediate. To prove the *'only if'* part, consider a scheme S. Define $L \subseteq \mathcal{L}$ as $L = \{l \in \mathcal{L} : r(l) \neq \perp$, where $r = p_S(r'')$ for some $r'' \in (\mathcal{L} \to \mathcal{V}_\perp)^{\max}\}$. Now we are to show that p_S is the canonical projection onto L. That is, we are to show that $r(l) = r'(l)$ for all $l \in L$ and $r'(l) = \perp$ for $l \notin L$ provided that $r' = p_S(r)$.

If $l \notin L$, consider any $r'' \geq r$, $r'' \in (\mathcal{L} \to \mathcal{V}_\perp)^{max}$. Then $p_S(r'')(l) = \perp \geq p_S(r)(l) = r(l)$. Thus $r(l) = \perp$. If $l \in L$, consider two cases. If $r(l) = \perp$, then $r' \leq r$ and $r'(l) = \perp$ too. If $r(l) = v$, by definition of L there is a maximal element r'' such that $p_S(r'')(l) = v' \neq \perp$. If $v' \neq v$, consider another maximal element r_v that differs from r'' only in its lth component which is v. If $p_S(r_v)(l) = \perp$, then $r'' \vee r_v$ exists, which contradicts the definition of scheme. Thus, $p_S(r_v)(l) = v$, and a record r^v whose only nonbottom component is $r^v(l) = v$ is in $\downarrow S$. Since $r^v \leq r$, r^v is also below r', which proves $r'(l) = v = r(l)$. □

If $\mathcal{L}$ is finite, $\mathcal{L} \to \mathcal{V}_\perp$ is isomorphic to $\mathcal{V}_\perp^n$, where $n = |\mathcal{L}|$. Therefore, in direct products of flat domains all schemes are semi-factors. Theorem 3.10 below will generalize this fact.

We shall mostly use schemes rather than semi-factors because the definition of schemes is more general and does not make use of any additional assumptions and, as we are going to show, schemes satisfy almost all properties that were proved in order to justify the definition of semi-factor in Buneman et al. [33]. In the rest of the subsection we prove some properties of schemes and state a result characterizing qualitative domains in which the concepts of scheme and semi-factor coincide.

Let $A, B \subseteq D$ be two sets. We define $A \vee B$ as the pointwise supremum, *i.e.* $A \vee B = \{a \vee b : a \in A, b \in B,\ a \vee b \text{ exists}\}$.

Proposition 3.3 *Let D be a distributive domain. Then*
1) *If A, B are scheme-ideals, then so is $A \vee B$.*
2) *The set of scheme-ideals over D is a complete lattice.*

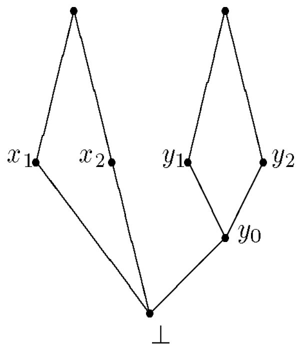
Proof. We are going to prove a more general fact, namely that for any indexed set of scheme-ideals $A_i, i \in I$, $A = \bigvee A_i$ is a scheme-ideal (the supremum is defined pointwise). Then both 1) and 2) will easily follow.

Prove that A is a strong ideal. Of course, it is closed under all existing joins since so are all A_is. To show that it is an ideal, consider $x = \vee x_i$, where $x_i \in A_i$, and $y \leq x$. Since D is a domain, y is the join of all compact elements below it. Let $a \leq y$ be compact. Then $a \leq x_{i_1} \vee \ldots \vee x_{i_k}$, and by distributivity there are $x'_{i_j} \leq x_{i_j}$, $j = 1, ..., k$, such that $a = x'_{i_1} \vee \ldots \vee x'_{i_k}$. Thus, $a \in A$, and since A is closed under existing joins, $y \in A$. Therefore, A is a strong ideal.

Our next step is to show that $p_A(x) = \bigvee p_{A_i}(x)$. Let $p_A(x) = \vee a_i$, where $a_i \in A_i$. Then each $a_i \leq p_A(x) \leq x$. Since $a_i \in A_i$, $a_i \leq p_{A_i}(x)$. Thus, $p_A(x) \leq \bigvee p_{A_i}(x)$. The equality now follows from $p_{A_i}(x) \leq p_A(x)$.

To finish the proof, show that A is a scheme-ideal. Let $x, y \in D^{\max}$ and $p_A(x) \leq p_A(y)$. Then for each index i: $p_{A_i}(x) \leq p_A(y) \leq y$. Since A_i is a scheme-ideal, $p_{A_i}(x) = p_{A_i}(y)$, hence $p_A(x) = p_A(y)$. Thus, A is a scheme-ideal. □

The same results have been proved for semi-factors in [33]. Notice that scheme-ideals may not be closed under intersection in contrast to the case of semi-factor ideals. For example, in the domain shown below, both $\{x_2, y_0, \perp\}$ and $\{x_1, y_1, y_0, \perp\}$ are scheme-ideals, but their intersection $\{y_0, \perp\}$ is not.



Proposition 3.3(2) says that schemes ordered by $\sqsubseteq^\flat$ form a lattice if D is distributive. A question arises : what can be said about other powerdomain orderings $\sqsubseteq^\sharp$ and $\sqsubseteq^\natural$? The following result shows that these orderings coincide for schemes in any domain. The same result for semi-factors was proved in [33].

Theorem 3.4 *Let D be an arbitrary domain and S_1, S_2 two schemes. Then $S_1 \sqsubseteq^\flat S_2$ iff $S_1 \sqsubseteq^\sharp S_2$ iff $S_1 \sqsubseteq^\natural S_2$.*

Proof. According to the definition of $\sqsubseteq^\natural$, it is enough to prove that $S_1 \sqsubseteq^\flat S_2$ iff $S_1 \sqsubseteq^\sharp S_2$. Let $S_1 \sqsubseteq^\flat S_2$. Consider any $x \in S_2$. We have to show the existence of an element $z \in S_1$ such that $z \leq x$. Let $x' \in D^{\max}, x' \geq x$. Since $S_1 \sqsubseteq^\flat S_2$, there exists $y \in S_2$ such that $z = p_{S_1}(x') \leq y$. If $y = x$, we are done. Otherwise, z and x are incomparable, and since $z \in \downarrow S_2$ (because $z \leq y$), $z \vee x$ exists. Thus, $z \vee x > x$ and $z \vee x \in \downarrow S_2$ (because S_2 is a scheme), a contradiction.

Let, conversely, $S_1 \sqsubseteq^\sharp S_2$. We are to prove that for every $x \in S_1$ there exists $z \in S_2$ such that $x \leq z$. Let $x' \in D^{\max}, x' \geq x$. Let $z = p_{S_2}(x')$. Since $S_1 \sqsubseteq^\sharp S_2$, there exists $y \in S_1$ such that $y \leq z$. If $y = x$, we are done. Otherwise, $y \vee x$ exists, since y and x are bounded by x', which contradicts the fact that S_1 is a scheme. Theorem is proved. □

Proposition 3.5 *Let $D = D_1 \times D_2$ (or $D = D_1 + D_2$). Then S is a scheme in D iff $S = S_1 \times S_2$ (or $S = S_1 + S_2$) for some schemes S_1 and S_2 in D_1 and D_2, respectively.* □

At this point, let us consider restrictions of our main definitions to the compact elements. First, to be able to speak of projections of compact elements onto compact elements of strong ideals, one must restrict the class of domain as the following lemma shows. Recall that ACC stands for the "ascending chain condition" which states that there are no infinite ascending chains $x_1 \lneq x_2 \lneq x_3 \lneq \ldots$.

Lemma 3.6 *If D is a domain, then the following are equivalent:*

(i) *For any strong ideal* $\mathcal{I}$ *and* $x \in \mathbf{K}D$, $p_{\mathcal{I}}(x) \in \mathbf{K}D$.
(ii) $\mathbf{K}D$ *is downward closed,* i.e. $\downarrow\mathbf{K}D = \mathbf{K}D$.
(iii) $\downarrow x$ *satisfies ACC for any* $x \in \mathbf{K}D$.

Proof. (i) $\Rightarrow$ (ii). Let $x \in \mathbf{K}D$ and $y \leq x$. Then by (i) $y = p_{\downarrow y}(x)$ is in $\mathbf{K}D$.

(ii) $\Rightarrow$ (iii). Assume $\mathbf{K}D = \downarrow\mathbf{K}D$. If $x \in \mathbf{K}D$ is such that $\downarrow x$ does not satisfy ACC, consider a chain $x_1 \lneq x_2 \lneq x_3 \lneq \ldots \leq x$. Then $x' = \sqcup_i x_i$ is not a compact element, but $x' \leq x$. This contradiction shows that $\downarrow x$ satisfies ACC.

(iii) $\Rightarrow$ (i). Let $\mathcal{I}$ be a strong ideal and $x \in \mathbf{K}D$. Then $p_{\mathcal{I}}(x) = \bigvee(y \mid y \leq x \text{ and } y \in \mathbf{K}D \cap \mathcal{I})$. Since $\downarrow x$ satisfies ACC, $p_{\mathcal{I}}(x)$ is a join of only finitely many y's and as such is compact. $\square$

Therefore, if D satisfies any condition of the lemma, we can restrict our attention to compact elements only. A projection is now defined as

$$p_{\mathcal{I}}(x) = \bigvee_{y \leq x \text{ and } y \in \mathcal{I}} y \qquad \text{for } x \in \mathbf{K}D$$

That allows us to redefine semi-factors at the level of compact elements. We say that $\mathcal{I} \subseteq \mathbf{K}D$ is a *compact semi-factor ideal* if it is downward closed, closed under *finite* least upper bounds and satisfies the slide condition for compact elements.

Proposition 3.7 *Let* D *be a domain such that* $\downarrow\mathbf{K}D = \mathbf{K}D$. *Then* $\mathcal{I} \longrightarrow \mathcal{I} \cap \mathbf{K}D$ *establishes a bijection between semi-factor ideals and compact semi-factor ideals.*

Proof. It is easy to see that $\mathcal{I} \cap \mathbf{K}D$ is a compact semi-factor ideal whenever $\mathcal{I}$ is a semi-factor ideal since compact elements are projected into compact elements. To see that the correspondence is bijective, we must prove that $\mathcal{I} = \{\bigvee X \mid X \subseteq \mathcal{I} \cap \mathbf{K}D,\ \bigvee X \text{ exists}\}$. The $\supseteq$ inclusion is obvious. Conversely, let $x \in \mathcal{I}$ and let $X = \downarrow x \cap \mathbf{K}D$. Then $X \subseteq \mathcal{I} \cap \mathbf{K}D$ and $x = \bigvee X$, which proves the reverse inclusion and the proposition. $\square$

Thus, one can reason about semi-factors entirely on the level of compact elements. In this aspect semi-factors are better suited for developing the database concepts in the domain-theoretic model. Another advantage of semi-factors will be seen when multivalued dependencies are studies. However, schemes are more general than semi-factors and in most cases the desired results can be stated for schemes.

We finish this section by two results of the same spirit. Both of them relate the properties of schemes that one would expect in a domain like $\mathcal{L} \to \mathcal{V}_{\perp}$ to the internal structure of the domain.

Observe that in $\mathcal{L} \to \mathcal{V}_{\perp}$ no element of a scheme can be replaced by another element such that the resulting set is still a scheme. To capture this property, we say that a scheme S in an arbitrary domain D is *saturated* if, for any $x \in S$, there is no $y \in D, y \neq x$ such that $(S-x) \cup y$ is a scheme. We say that D is *coatomic* if every element is a meet of maximal elements. Notice that $\mathcal{L} \to \mathcal{V}_\perp$ is coatomic and all schemes in $\mathcal{L} \to \mathcal{V}_\perp$ are saturated.

Proposition 3.8 *If D is coatomic, then all schemes in D are saturated.*

Proof. Let D be a coatomic domain. Assume that S a non-saturated scheme in D, *i.e.* $(S-x) \cup y$ is a scheme for some $x \in S$ and $y \neq x$. If $S = \{x\}$, then $p_S(z) = x$ for any $z \in D^{\max}$ and $x \leq \bigwedge_{z \in D^{\max}} z = \perp$ since D is coatomic. Hence, if $(S-x) \cup y$ were a scheme, y would equal bottom yielding $y = x$. This contradiction shows that S has at least two elements. Now consider three cases.

Case 1: $y < x$. Since D is coatomic, there exists $y_m \in D^{\max}$ such that $y_m \geq y$ but $y_m \not\geq x$. Let $z = p_S(y_m)$. Since $y_m \not\geq x$, $z \neq x$. Therefore, $z, y \in (S-x) \cup y$ and $z \uparrow y$ which contradicts the definition of scheme.

Case 2: $y > x$. Now we can find $x_m \in D^{\max}$ such that $x_m \geq x$ and $x_m \not\geq y$. Let $z = p_{(S-x) \cup y}(x_m)$. Since $x_m \not\geq y$, $x \in S - x$ and $x \uparrow z$ which again contradicts the definition of scheme.

Case 3: y and x are not comparable. Similarly, we can find $x_m \in D^{\max}$ such that $x_m \geq x$ and $x_m \not\geq y$ and the proof proceeds as in the second case. Thus, all three cases lead to contradiction which proves the proposition. □

Corollary 3.9 *Let D be a coatomic domain and S a scheme. If $S' \subset S$, then S is not a scheme.*
□

The reader can easily establish a number of properties of saturated schemes. For instance, even in distributive domains it is possible to find examples of saturated schemes which are not semi-factors and examples of semi-factors which are not saturated. Saturated scheme-ideals may fail to be closed under intersection. The converse of proposition 3.8 is not true: there exists a domain in which all schemes are saturated but which is not coatomic.

Our next result is a precise characterization of those qualitative domains in which the concepts of scheme and semi-factor coincide. Informally, this results states that in a certain class of domains the concepts of scheme and semi-factor coincide iff the domain looks like $\mathcal{L} \to \mathcal{V}_\perp$. The proof is not given here. It relies on the theory of decomposition of domains developed by Jung, Libkin and Puhlmann [88].

Theorem 3.10 *Let D be a qualitative domain. Every scheme of D is a semi-factor iff*

$$D \simeq \prod_{i \in I} D_i$$

where each D_i has no proper scheme; the schemes of D are in 1-1 correspondence with subsets of I. □

3.1.3 Dependency theory

The purpose of this section is to develop some basic dependency theory for our domain model of databases. Functional dependencies were introduced in Buneman et al. [33] for semi-factors; here we show that all the results remain true for schemes. The major contribution of this section is introduction of multivalued dependencies for generalized relations and proving the decomposition theorem. However, some work must be done before we can define multivalued dependencies. In the relational theory these dependencies establish relationship not only between projections into two schemes, but also between projections into a complement of one of them, i.e. all the operations of Boolean algebra – intersection, union and complement – are involved. From proposition 3.3 we know that analogs of only two operations – intersection and union – have been defined for the schemes so far. Therefore, we need to define complements for schemes. In order to do that, it is necessary to restrict the class of domains. It will turn out that this class consists of the qualitative domains.

Functional dependencies

Having introduced the notion of scheme, we can define functional dependencies. If S_1, S_2 are schemes in a domain D, then a *functional dependency* is an expression of the form $S_1 \rightarrow S_2$. Usually in the theory of databases with incomplete information dependencies are defined only on the schemes projections into which do not contain tuples with null values. This condition can be equivalently expressed as: for any record in a relation there is a record in a scheme which is less informative than the relation record. In other words, if R is a relation and S is a scheme, then $S \sqsubseteq^\sharp R$.

Now we can define satisfiability for functional dependencies. Let $R \subseteq D$ be a relation. We say that R satisfies a functional dependency $S_1 \rightarrow S_2$ if $S_1, S_2 \sqsubseteq^\sharp R$ and $p_{S_2}(x) = p_{S_2}(y)$ whenever $p_{S_1}(x) = p_{S_1}(y)$ for every $x, y \in R$.

Functional dependencies in distributive domains have been investigated in [33] for the particular case of semi-factors, and the following analogs of the Armstrong axioms are due to [33], where F is a set of functional dependencies and $\langle \text{Schemes}(D), \leq \rangle$ is the complete lattice of schemes over distributive domain D (cf. proposition 3.3).

(a) If $S_1, S_2 \in \text{Schemes}(D)$, $S_1 \leq S_2$ and $S_2 \rightarrow S_2 \in F$, then $S_2 \rightarrow S_1 \in F$.

(b) If for any $i \in I : S \rightarrow S_i \in F$ where $S, S_i \in \text{Schemes}(D)$, then $S \rightarrow \bigvee_{i \in I} S_i \in F$.

(c) If $S_1 \to S_2 \in F$ and $S_2 \to S_3 \in F$, where $S_1, S_2, S_3 \in \text{Schemes}(D)$, then $S_1 \to S_3 \in F$.

We need the additional condition $S_2 \to S_2 \in F$ to guarantee consistency since generally it may not be the case that $S_2 \sqsubseteq^\sharp R$. The result of [33] proved for semi-factors is also true for schemes:

Proposition 3.11 *The Armstrong Axioms (a)–(c) are consistent and complete for relations in distributive domains.*

Proof. Prove consistency first. (a) Let $S_2 \to S_2 \in F$. Then $S_2 \sqsubseteq^\sharp R$. Since $S_1 \leq S_2$ in $\text{Schemes}(D)$, $\downarrow S_1 \subseteq \downarrow S_2$ and $S_1 \sqsubseteq^\flat S_2$. According to theorem 3.4, $S_1 \sqsubseteq^\sharp S_2$, hence $S_1 \sqsubseteq^\sharp R$. Thus, $S_2 \to S_1 \in F$.

(b) Let $S_i \sqsubseteq^\sharp R$, $i \in I$. Prove that $S_I = \bigvee_{i \in I} S_i \sqsubseteq^\sharp R$. Let $r \in R$. Then for any $i \in I$ there is such $s_i \in S_i$ that $s_i \leq r$. Therefore, $s_I = \vee_{i \in I} s_i \leq r$. Since D is distributive, $s_I \in S_I$ (cf. proposition 3.3), and $S_I \sqsubseteq^\sharp R$. Now let $p_S(x) = p_S(y)$ for $x, y \in R$. Then

$$p_{S_I}(x) = \bigvee_{i \in I} p_{S_i}(x) = \bigvee_{i \in I} p_{S_i}(y) = p_{S_I}(y).$$

Thus, $S \to S_I \in F$.

(c) is obvious. Completeness follows from the fact that our model is a generalization of the standard relational model. Therefore, we have more relations available. $\square$

Complements of schemes

Our goal is to introduce multivalued dependencies for generalized relations. A multivalued dependency $X \twoheadrightarrow Y$, where X, Y are sets, uses the projection onto $X \cup \overline{Y}$. While $\cup$ corresponds to $\vee$ in the domain model, there is no analog for the complement. More precisely, the poset of schemes is a lattice if the domain is distributive, but schemes may fail to have complements in contrast to the case of $\mathcal{L} \to \mathcal{V}_\perp$. Thus, our goal is twofold. First, we define complements of schemes and the proceed to introduce multivalued dependencies and prove the decomposition theorem.

Consider the domain $\mathcal{L} \to \mathcal{V}_\perp$. Its schemes correspond to subsets of $\mathcal{L}$, with scheme-projections being canonical projections. The complement of a scheme corresponds to projecting onto the complementary subset of $\mathcal{L}$. Suppose that we have defined the concept of a complement. Let p be a scheme-projection and $\overline{p}$ the projection corresponding to the scheme's complement. What should the properties of $\overline{p}$ be? First, if we have any element $x \in D$, then $p(x) \wedge \overline{p}(x) = \perp$. Suppose that $x \in D$. Then $\overline{p}(x)$ "forgets" about information contained in $p(x)$. The fact that $\overline{p}$ is the complement of p means that all information contained in x can be reconstructed from $p(x)$ and $\overline{p}(x)$, i.e. $x = p(x) \vee \overline{p}(x)$. That means that in order to introduce complements, we have to require that all principal ideals $\downarrow x$ in D be complemented lattices. Moreover, they must be uniquely complemented since we want to speak about *the* complement. The following proposition shows why we restrict our attention only to qualitative domains.

Proposition 3.12 *Any principal ideal of a domain D is a uniquely complemented lattice iff D is a qualitative domain.*

Proof follows immediately from the definition of qualitative domains and the fact that any uniquely complemented algebraic lattice is Boolean, see Salii [154]. □

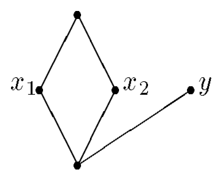
There is another elegant way to define complements due to Jung [87]. Let $S \subseteq D$ be a scheme in any domain D . We define $\mathcal{I}_{\overline{S}}$ as the set of maximal elements of $\{x \in D : p_S(x) = \bot\}$. It can be shown that $\mathcal{I}_{\overline{S}}$ is not generally a scheme. In order to be able to operate with complements, we have to make two observations.

Proposition 3.13 *Let D be a qualitative domain and S any scheme. Then $\downarrow\overline{\mathcal{I}}_S = \downarrow\mathcal{I}_{\overline{S}}$. That is, $\mathcal{I}_{\overline{S}}$ is the set of maximal elements of $\overline{\mathcal{I}}_S$. Moreover, $\downarrow\mathcal{I}_{\overline{S}}$ is a strong ideal.*

Proof. $\downarrow\overline{\mathcal{I}}_S \subseteq \downarrow\mathcal{I}_{\overline{S}}$ easily follows from the observation that $y \leq \overline{p}_S(x)$ implies $p_S(y) = \bot$. To prove the renverse inclusion, consider an element $x \in \downarrow\mathcal{I}_{\overline{S}}$, i.e. $p_S(x) = \bot$. Let $x' \geq x$ be a maximal element. We finish the proof by showing that $x \leq \overline{p}_S(x')$. If x is not under $\overline{p}_S(x')$, then $z = x \wedge p_S(x') \neq \bot$ since $\downarrow x'$ is Boolean. Then $z \in \downarrow S$ and $z \leq x$, hence $\bot = p_S(x) \geq z$, a contradiction. Thus, $x \leq \overline{p}_S(x')$.

To show that $\downarrow\mathcal{I}_{\overline{S}}$ (and therefore $\downarrow\overline{\mathcal{I}}_S$) is a strong ideal, consider an indexed family $x_i \in D, i \in I$ such that $p_S(x_i) = \bot$ for all i and $x = \bigvee_{i \in I} x_i$ exists. By [33] $p_S(x) = \bigvee_{i \in I} p_S(x_i)$. Therefore, $p_S(x) = \bot$ and $x \in \downarrow\mathcal{I}_{\overline{S}}$, which proves strongness. □

The following example shows why $\mathcal{I}_{\overline{S}}$ may fail to be a scheme even in a qualitative domain: $S = \{x_1, y\}$ is a scheme, but $\mathcal{I}_{\overline{S}} = \{x_2\}$ is not.



Given a scheme S in a qualitative domain, we can correctly define its complement as $\mathcal{I}_{\overline{S}}$. As we mentioned above, the complement of a scheme may fail to be a scheme. However, complements of semi-factors are schemes, as the following result shows.

Proposition 3.14 *The complement of a semi-factor is a scheme in any qualitative domain.*

Proof. Let S be a semi-factor in a qualitative domain D. Denote the projection onto $\downarrow\mathcal{I}_{\overline{S}}$ as $\overline{p}$. (This projection is correctly defined since $\downarrow\mathcal{I}_{\overline{S}}$ is a strong ideal.) Suppose that there are $x_1, x_2 \in D^{\max}$ such that $\overline{p}(x_1) > \overline{p}(x_2)$. Then $p_S(\overline{p}(x_1)) = \perp \leq p_S(x_2)$, and $y = \overline{p}(x_1) \vee p_S(x_2)$ exists since S is a semi-factor. Since $p_S(x_2) \leq y$ and $\overline{p}(x_2) < \overline{p}(x_1) \leq y$, $x_2 \leq y$. Thus, $y = x_2$ since $x_2 \in D^{\max}$. The lattice $\downarrow x_2$ is Boolean, therefore $z = p_S(x_2) \wedge \overline{p}(x_1) \neq \perp$ since $\overline{p}(x_1)$ is greater than $p_S(x_2)$'s complement in $\downarrow x_2$. However, $z \leq \overline{p}(x_1)$, thus $p_S(z) = \perp$, which is impossible since $z \leq p_S(x_2)$ and hence is in $\downarrow S$. This contradiction shows that projections of maximal elements can not be comparable; thus the complement of S is a scheme. □

If $\mathcal{I}_{\overline{S}}$ is a scheme, we say that S has the complement (which is $\mathcal{I}_{\overline{S}}$) and denote it by $\overline{S}$. Of course, any semi-factor is complemented by proposition 3.14.

Multivalued dependencies

Now that the complements have been defined, the definition of multivalued dependencies in qualitative domains can be given.

Definition 3.3 *Let D be a qualitative domain and S a scheme having the complement $\overline{S}$. Let S' be a scheme. We say that a relation $R \subseteq D$ satisfies* multivalued dependency $S' \twoheadrightarrow S$ *if for every $x, y \in R$ with $p_{S'}(x) = p_{S'}(y)$ there exists $z \in R$ such that $p_{S'}(z) \vee p_S(z) = p_{S'}(x) \vee p_S(x)$ and $p_{S'}(z) \vee p_{\overline{S}}(z) = p_{S'}(y) \vee p_{\overline{S}}(y)$.*

If D is $\mathcal{L} \to \mathcal{V}_\perp$, this is the usual definition of multivalued dependency in a relational database. Notice that multivalued dependencies, like functional dependencies, should be considered only on schemes the projections into which do not contain null values. As it was shown above, it means that a scheme is less than a relation in the Smyth ordering $\sqsubseteq^\sharp$. Therefore, in the above definition the following should hold: $S' \vee S \sqsubseteq^\sharp R$ and $S' \vee \overline{S} \sqsubseteq^\sharp R$. It can be easily concluded from these inclusions that $R \subseteq D^{\max}$. Therefore we will consider only relations without incomplete information when speaking of multivalued dependencies.

The above introduced functional and multivalued dependencies satisfy two standard properties. The proof is immediate from the definitions.

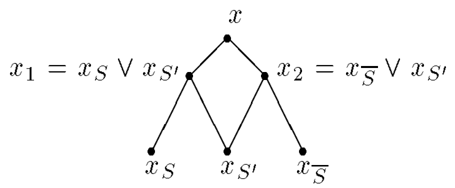
Proposition 3.15 *Let D be a qualitative domain, and S a scheme having complement $\overline{S}$. Let S' be a scheme, and R a relation without incomplete information, i.e. a finite subset of $D^{\max}$. Then*
1) If R satisfies $S' \to S$ then R satisfies $S' \twoheadrightarrow S$;
2) If R satisfies $S' \twoheadrightarrow S$, then R satisfies $S' \twoheadrightarrow \overline{S}$. □

It was shown in Buneman et al. [33] that the natural join operation in the relational algebra corresponds to the join $\sqcup^\sharp$ in the Smyth ordering. We have shown that the complement of a semi-factor in a qualitative domain is a scheme and defined multivalued dependencies. Now we are in the position to formulate the decomposition theorem.

Theorem 3.16 *Let D be a qualitative domain, and R a relation without incomplete information (that is, a finite subset of $D^{\max}$). Let S and S' be semi-factors of D. Then R satisfies multivalued dependency $S' \twoheadrightarrow S$ iff $R = [p_{S'} \vee p_S(R)] \bowtie [p_{S'} \vee p_{\overline{S}}(R)]$, where join $\bowtie$ is $\sqcup^\sharp$.*

Proof. To simplify the notation, we will write x_S and R_S instead of $p_S(x)$ and $p_S(R)$. $S' \vee S$ will be denoted by S_1 and $S' \vee \overline{S}$ by S_2; the corresponding projections are x_1 and x_2.

Since $\overline{S}$ is the complement of S, $x_S \vee x_{\overline{S}} = x_1 \vee x_2 = x$ for any x. By proposition 3.3, $x_1 = x_S \vee x_{S'}$ and $x_2 = x_{\overline{S}} \vee x_{S'}$. We will also need the following fact: $x_{S'} = x_1 \wedge x_2$. Indeed, since $\downarrow x$ is a Boolean lattice and x_S and $x_{\overline{S}}$ complement each other in $\downarrow x$, $x_1 \wedge x_2 = (x_S \vee x_{S'}) \wedge (x_{\overline{S}} \vee x_{S'}) = x_{S'}$. The following picture illustrates the relationship between different projections of x.



According to the definitions introduced above, R obeys $S' \twoheadrightarrow S$ iff $x_{S'} = y_{S'}$ implies the existence of such z that $z_1 = x_1$ and $z_2 = y_2$.

Recall the definition of the join: $R' \bowtie R'' = \min\{x \in D | \exists r' \in R', r'' \in R'' : r' \vee r'' \leq x\}$. For example, if $x = x' \vee x''$, then $x = x' \bowtie x''$. In particular, $x = x_1 \bowtie x_2$ for any x. Having done the preliminary work, we can now proceed to prove the theorem.

- Let R obey the dependency $S' \twoheadrightarrow S$. We must show that $R = R_1 \bowtie R_2$. Suppose $x \in R$. Then $x = x_1 \vee x_2$, and x is not in $R_1 \bowtie R_2$ iff there exist $t, t' \in R$ such that $x > t_1 \vee t'_2$. Assume such t, t' exist. Let $v = t_S \vee t'_{\overline{S}}$ (it exists since it is bounded by x). Suppose $v \notin D^{\max}$. Then there is $v' \in D^{\max}, v' > v$. Since both S and $\overline{S}$ are schemes, $v'_S = t_S$ and $v'_{\overline{S}} = t'_{\overline{S}}$. Therefore, $v' = t_S \vee t'_{\overline{S}} = v$. This shows $v \in D^{\max}$. Since $v \leq x$, $v = x$. But then $x = v \leq t_1 \vee t'_2 < x$. This contradiction shows that $x \in R_1 \bowtie R_2$, i.e. $R \subseteq R_1 \bowtie R_2$.

  Let, conversely, $x \in R_1 \bowtie R_2$, i.e. for some $t, t' \in R : x = t_1 \vee t'_2$. As we have shown above, $x \in D^{\max}$. Since S_1 is a scheme, $x_1 = t_1$; projecting both parts into S' we get $x_{S'} = t_{S'}$. Analogously, $x_{S'} = t'_{S'}$. Thus $t_{S'} = t'_{S'}$. Since R obeys $S' \twoheadrightarrow S$, there is such $v \in R$ that $v_1 = t_1$ and $v_1 = t'_2$. Hence $v = v_1 \vee v_2 = t_1 \vee t'_2 = x$, i.e. $x \in R$. Thus, $R = R_1 \bowtie R_2$.

- Let, conversely, $R = R_1 \bowtie R_2$. We have shown above that for any x and y the element $x_1 \vee y_2$ is maximal if it exists; therefore, if $x_1 \vee y_2$ exists for some $x, y \in R$, it must belong to R according to the definition of join.

  Consider $x, y \in R$ such that $x_{S'} = y_{S'}$. Then $p_{S' \vee S}(y_2) = p_{S'}(y_2) \vee p_S(y_2) = y_{S'} \vee p_S(y_{S'}) \vee p_S(y_{\overline{S}}) = y_{S'} = x_{S'} \leq x_1$. Since both S and S' are semi-factors, so is $S \vee S'$ [33]. Hence $z = y_2 \vee x_1$ exists and is an element of R.

  We will finish the proof that R obeys $S' \twoheadrightarrow S$ by showing that $z_1 = x_1$ and $z_2 = y_2$. Calculate z_1: $z_1 = p_{S' \vee S}(y_2 \vee x_1) = p_{S'}(y_2) \vee p_S(y_2) \vee p_{S'}(x_1) \vee p_S(x_1) = p_{S'}(y_{S'} \vee y_{\overline{S}}) \vee p_S(y_{S'} \vee y_{\overline{S}}) \vee p_{S'}(x_{S'} \vee x_S) \vee p_S(x_{S'} \vee x_S) = y_{S'} \vee p_S(y_{S'}) \vee p_S(y_{\overline{S}}) \vee x_{S'} \vee x_S = y_{S'} \vee x_{S'} \vee x_S = x_{S'} \vee x_S = x_1$. Similarly $z_2 = y_2$. Theorem is proved. □

Let us finish this section by an observation that supports the reasonings that led us to two alternative definitions of scheme. It was said before that only a very natural assumption that complete descriptions are projected into complete descriptions is behind the definition of scheme, while in the definition of semi-factors it is implicitly assumed that each scheme is complemented and projecting is just throwing away those pieces of information which belong to this complement. So it did not appear as a complete surprise that a scheme may fail to have a complement even in a qualitative domain while a semi-factor is always complemented in such a domain. The above theorem that relates multivalued dependencies and decompositions of relations in a qualitative domain holds for semi-factors but not for schemes because we do need complements and the possibility to work with the information "thrown away". Notice, however, that the 'only if' part remains true if S' is an arbitrary scheme and S is a scheme having complement $\overline{S}$.

3.1.4 Queries

In this section we shall find analogs of the main operations of relational algebra for generalized relations. Schemes introduced before will be used to define projections. Generalized relations will be considered as finite antichains in database domains. Let us describe the operations of the algebra as in Libkin [99].

1. *Union*. Let D be a domain and R_1, R_2 two relations. Then their union is defined as $R_1 \tilde{\cup} R_2 = \max(R_1 \cup R_2)$. Observe that this is the join in the Hoare powerdomain. That is,

$$R_1 \tilde{\cup} R_2 = R_1 \sqcup^\flat R_2$$

We need the max operation because $R_1 \cup R_2$ may fail to be an antichain, but $R_1 \tilde{\cup} R_2$ always is. $R_1 \tilde{\cup} R_2$ can be interpreted as the set of the most informative elements from R_1 and R_2.

2. *Difference*. Let D be a domain and R_1, R_2 two relations. Then $R_1 - R_2$ is the usual set difference. Since $R_1 - R_2 \subseteq R_1$, it is a relation.

Intersection can be expressed as $R_1 \cap R_2 = R_1 - (R_1 - R_2)$.

3. *Cartesian (direct) product*. Let D_1, D_2 be two domains and R_1, R_2 relations in D_1, D_2 respectively. Then $R_1 \times R_2$ is a relation in $D_1 \times D_2$ defined as $R_1 \times R_2 = \{\langle r_1, r_2 \rangle \mid r_1 \in R_1, r_2 \in R_2\}$.

4. *Projection*. Given a (database) domain D, we define *projection* as projection into a scheme-ideal $\downarrow S$ in D. If D is $\mathcal{L} \to \mathcal{V}_\perp$, then projections thus defined coincide with projections in the relational algebra.

If $R \subseteq D$ is a relation and S is a scheme, $p_S(R)$ may fail to be an antichain. Therefore, we need two operations of projection:

$$p_S^{min}(R) = \min p_S(R), \;\; p_S^{max}(R) = \max p_S(R).$$

If R is a one-element relation, these two projections coincide and we will write simply $p_S(R)$. The above defined operations also coincide for relations without incomplete information, i.e. subsets of $D^{\max}$.

5. *Selection*. We can also define *selection* using the concept of scheme. First we have to define conditions. As usually, if c_1, c_2 are conditions, then so are $c_1 \vee c_2, c_1 \& c_2$ and $\neg c_1$. Schemes are necessary to define conditions we start with. Let $S, S' \subseteq D$ be schemes, $a \in \downarrow S$, $x \in D$. Then the elementary conditions are $p_S(x)\theta a$, $p_S(x)\theta p_{S'}(x)$, where $\theta \in \{<, \leq, =, \neq, \geq, >\}$.

Let $R \subseteq D$ be a relation. that is, an antichain in D. If $c : D \to \{\mathit{true}, \mathit{false}\}$ is a condition, then the *selection* is defined as

$$\sigma_c(R) = \{x \in R : c(x) = \mathit{true}\}.$$

If we do not know what the class $\mathcal{B}$ of basic domains is and how D was constructed from the basic domains, the above defined selection is all we can get. However, if we know a concrete procedure of construction of D (for example, a term in the signature $\langle \times, + \rangle$ with variables from $\mathcal{B}$), then we can define more complex conditions. For example, if the database domain is $D \times D \times D$, then we are able to select those elements whose first and third projections coincide.

We can give the selection more power if we introduce binary relations on domains from $\mathcal{B}$. For example, if P is a binary relation on $D_1 \in \mathcal{B}$ and $\downarrow S = D_1$, then we can introduce conditions like $(p_S(x), a) \in P$. This is necessary because, for example, domain of natural numbers is represented in domain theory as a flat domain $\mathbf{N}_\perp = \{\perp, n_0, n_1, n_2, \ldots\}$ where n_i corresponds to the natural number i, and the ordering of $\mathbf{N}_\perp$ is given by letting $\perp$ be less than all n_i's. We can not conclude that $1 < 2$ from this information. Therefore, we need a binary relation P on $\mathbf{N}_\perp$ describing the ordering of natural numbers as comparing values stored in a database is one of the most typical operations used in queries over relational databases. Therefore, it is essential that the selection on database domains be powerful enough to be able to carry out various comparisons.

To define such powerful selection we first need the definition of *similar* schemes and a 1-1 correspondence between their scheme-ideals. In the above example of $D \times D \times D$ schemes

$D \times \{\perp\} \times \{\perp\}$ and $\{\perp\} \times \{\perp\} \times D$ should be similar and 1-1 correspondence between their scheme-ideals associates the first and the third projections of any record. This gives us a possibility to compare projections on different schemes. As it was said earlier, we may want, for example, to select records with coinciding first and third projections.

In what follows, assume that only record and variant constructors are allowed. That is, D can be represented as $t(D_1, \ldots, D_n)$ where t is a term in the signature $\langle \times, + \rangle$ and $D_1, \ldots, D_n \in \mathcal{B}$ (for example, $\mathbf{N}_\perp \times \mathbf{N}_\perp \times (\mathbf{N}_\perp + (\mathbf{Bool} \times \mathbf{N}_\perp)))$. We now define *similarity* of two schemes S, S' and mapping $\varphi_{S \to S'} : \downarrow S \to \downarrow S'$.

- If S is a scheme in $D \in \mathcal{B}$, then S is similar to itself and $\varphi_{S \to S}$ is the identity mapping on $\downarrow S$.
- Let $D = t(D_1, \ldots, D_n)$, where $D_i \in \mathcal{B}$, $i = 1, \ldots, n$. Suppose S, S' are two schemes in D. Assume that the last operation of t is $\times$, i.e. $t(\cdot) = t_1(\cdot) \times \ldots \times t_k(\cdot)$ and the last operation of each t_i is not $\times$. Then $S = S_1 \times \ldots \times S_k$ and $S' = S'_1 \times \ldots \times S'_k$ where S_i, S'_i are schemes in $t_i(D_1, \ldots, D_n)$. Then S is similar to S' iff there are i and j such that $t_i = t_j$, S_i is similar to S'_j in $t_i(D_1, \ldots, D_n) = t_j(D_1, \ldots, D_n)$ and $S_l = \{\perp_{t_l(D_1,\ldots,D_n)}\}$, $S'_p = \{\perp_{t_p(D_1,\ldots,D_n)}\}$, $l \neq i, p \neq j$. $\varphi_{S \to S'}$ maps a record $x \in \downarrow S$ with only nonbottom ith component $x_i \in \downarrow S_i$ to the record whose only nonbottom jth component is $\varphi_{S_i \to S'_j}(x_i)$.
- If the last operation of the term is $+$, then $S = S_1 + \ldots + S_k$ and $S' = S'_1 + \ldots + S'_k$ where S_i, S'_i are schemes in $t_i(D_1, \ldots, D_n)$. Then S is similar to S' iff each S_i is similar to S'_i in $t_i(D_1, \ldots, D_n)$, and for any $x \in \downarrow S : \varphi_{S \to S'}(x) = \varphi_{S_i \to S'_i}(x)$ if $x \in S_i$.

Example 3.4 Let $S = \{\perp\} \times \{\perp\} \times D$ and $S' = D \times \{\perp\} \times \{\perp\}$ be two scheme-ideals in $D \times D \times D$. Then S and S' are similar and $\varphi_{S \to S'}(\{\perp, \perp, x\}) = \{x, \perp, \perp\}$. Scheme-ideals $D + (\{\perp\} \times D)$ and $D + (D \times \{\perp\})$ are similar in $D + (D \times D)$. □

Now we can extend the list of possible elementary conditions by adding the conditions of form $\varphi_{S \to S'}(p_S(x)) \theta p_{S'}(x)$ where S, S' are two similar schemes in a database domain D.

As we said before, one may also want to define some binary relations on basic domains. Let $P_i^k, k \in I_i$ be a family of binary relations on $D_i \in \mathcal{B}$, where I_i is (possibly empty) set of indices. We say that a scheme S of a database domain $D = t(D_1, \ldots, D_n)$ is also a scheme in a basic domain D_i if $S = t(\{\perp\}, \ldots, S_i, \ldots, \{\perp\})$ where $S_i \subseteq D_i$ is a scheme. In this case we can identify elements of $\downarrow S$ and $\downarrow S_i$.

The third type of elementary conditions includes the conditions $(p_S(x), a) \in P_i^k$ and $(p_S(x), p_{S'}(x)) \in P_i^k$ where S, S' are schemes in D_i identified with S_i, $a \in S_i$ and $k \in I_i$.

With such extensions being added, selection covers the usual selection in the relational algebra.

Example 3.5 Consider a relation with variants describing companies. Each record contains the following information: name, total donations for non-profit companies, gross revenue and costs for profit companies. Below are examples of records with variants:

$$r_1 = [\text{Name: X}, \text{Status: } \langle \text{Non}-\text{profit: } [\text{Donations: } 1,000,000]\rangle],$$

$$r_2 = [\text{Name: Y}, \text{Status: } \langle \text{Profit: } [\text{Revenue: } 2,000,000, \text{Costs: } 1,000,000]\rangle].$$

Let D be a domain of names. Then the above records are elements of a database domain $D \times (\mathbf{N}_\perp + (\mathbf{N}_\perp \times \mathbf{N}_\perp))$. Consider the following schemes:

$$S_1 = \{\perp_D\} \times (\mathbf{N}_\perp + (\{\perp_{\mathbf{N}_\perp}\} \times \{\perp_{\mathbf{N}_\perp}\})),$$

$$S_2 = \{\perp_D\} \times (\{\perp_{\mathbf{N}_\perp}\} + (\mathbf{N}_\perp \times \{\perp_{\mathbf{N}_\perp}\})),$$

$$S_3 = \{\perp_D\} \times (\{\perp_{\mathbf{N}_\perp}\} + (\{\perp_{\mathbf{N}_\perp}\} \times \mathbf{N}_\perp)).$$

Then S_1, S_2, S_3 are also schemes in $\mathbf{N}_\perp$ and S_2 is similar to S_3.

Let P be a binary relation on $\mathbf{N}_\perp$ such that $(n_i, n_j) \in P$ iff $i \leq j$, $(\perp, x) \in P$ for all $x \in \mathbf{N}_\perp$. Consider the following conditions: $c_1 \equiv (p_{S_1}(x) \neq \perp_{\mathbf{N}_\perp})$ (to be more precise, we should compare $p_{S_1}(x)$ with an element of $\downarrow S_1$, that is, with $\{\perp_D, \perp_{\mathbf{N}_\perp} \times \{1\}\}$), $c_2 \equiv ((p_{S_3}(x), p_{S_2}(x)) \in P)$. Let R be a relation in the above database domain. Then $\sigma_{c_1}(R)$ selects non-profit companies from R while $\sigma_{c_2}(R)$ selects companies working well, that is, whose gross revenue exceeds costs. $\square$

6. *Natural join*. Join was introduced in [33] as the supremum in the Smyth ordering. That is, given two relations (antichains) $R_1, R_2 \subseteq D$, their join is $R_1 \sqcup^\sharp R_2$. It was proved that for domain $\mathcal{L} \to \mathcal{V}_\perp$ the above defined operation coincides with the natural join in relational algebra. We use more convenient and customary symbol $\bowtie$ instead of $\sqcup^\sharp$.

There is another way to think of the join operation. Given two generalized relations $R_1, R_2 \subseteq D$, their join $R_1 \bowtie R_2$ is the set of minimal (in D) elements which are greater than some element of R_1 and some element of R_2 :

$$R_1 \bowtie R_2 = \min\{x \in D \mid \exists r_1 \in R_1, r_2 \in R_2 \,:\, r_1 \leq x, r_2 \leq x\}.$$

This formula follows from the definition of $R_1 \sqcup^\sharp R_2$ and basic properties of the Smyth powerdomain ordering $\sqsubseteq^\sharp$ [33, 157].

Several conditions were given in Tanaka and Chang [164] that the analog of the natural join in object-oriented model should satisfy. Informally, they are: 1) if there are no common attributes of two relations, the result of the join is isomorphic to their direct (Cartesian) product; 2) if two relations are defined over the same sets of attributes, the result of the join is their intersection;

3) the join of two relations can be obtained as union of pairwise joins of their elements (if these exist). Join is also known to be associative in relational algebra, see Ullman [168].

Let us formalize these properties.

1) *Let $R_1 \subseteq D_1, R_2 \subseteq D_2$ be two relations, and $D_1 \cap D_2 = \emptyset$. Let $R_1' = R_1 \times \{\perp_2\}$ and $R_2' = R_2 \times \{\perp_1\}$ be two relations in $D_1 \times D_2$. Then $R_1' \bowtie R_2' = R_1 \times R_2$.*

2) *Let $R_1, R_2 \subseteq D^{\max}$ be two relations. Then $R_1 \bowtie R_2 = R_1 \cap R_2$.*

Formalizing property 3) we must keep in mind that the union of pairwise joins may contain comparable elements while relations are antichains. Therefore, after finding union of individual joins we have to eliminate some elements in order to obtain an antichain. According to Imielinski and Lipski [78], there is no "semantically correct" way to do it. Since joining relations with null values may often yield counter-intuitive results (cf. [78, 123]) we think that formalizing the third property we have to eliminate nonminimal elements, i.e. to leave the least informative elements among pairwise joins.

Let us illustrate it by the following example. Consider two relations:

R_1 :

Name	Room	Phone
John	076	$\perp$
John	$\perp$	1595

R_2 :

Name	Room	Salary
John	076	12K

Taking element-wise joins gives us two records over attributes Name, Room, Phone, Salary. One is $r_1 = \boxed{\text{John} \mid 076 \mid \perp \mid \text{12K}}$ and the other is $r_2 = \boxed{\text{John} \mid 076 \mid 1595 \mid \text{12K}}$. Clearly, $r_1 \leq r_2$. Hence, taking maximal records into the result of the join operation tells us that John is in the room 076, makes 12K *and* has the telephone number 1595, even though there is no indication in R_1 and R_2 that this should be the case. Taking the minimal record r_1 as the result is indeed consistent with the information stored in R_1 and R_2. Summing up, the third property of the join operation is the following.

3) *Let $R, R' \subseteq D$ be two relations, and $R = \{r_1, \ldots, r_n\}, R' = \{r_1', \ldots, r_m'\}$. Then $R \bowtie R' = \min(\bigcup(\{r_i\} \bowtie \{r_j'\} : i = 1, ..., n, j = 1, ..., m))$.*

4) *If $R_1, R_2, R_3 \subseteq D$ are three relations, then $R_1 \bowtie (R_2 \bowtie R_3) = (R_1 \bowtie R_2) \bowtie R_3$.*

Proposition 3.17 *The join operation $\sqcup^\sharp$ satisfies* 1) - 4).

Proof. Let us first rewrite the definition of the join operation as $R_1 \bowtie R_2 = \min\{r_1 \vee r_2 | r_1 \in R_1, r_2 \in R_2, r_1 \vee r_2 \text{ exists}\}$.

1) If $r_1' \in R_1'$, then r_1' is of form $\langle r_1, \perp \rangle$ for some $r_1 \in R_2$. Similarly each $r_2 \in R_2$ is of

form $\langle \bot, r_2 \rangle$ for some $r_2 \in R_2$. Thus for any $r_1' \in R_1'$ and $r_2' \in R_2'$ their join exists; in fact, $\langle r_1, \bot \rangle \vee \langle \bot, r_2 \rangle = \langle r_1, r_2 \rangle$. Notice that $\{\langle r_1, r_2 \rangle | r_1 \in R_1, r_2 \in R_2\}$ is an antichain provided that so are R_1 and R_2. Thus $R_1' \bowtie R_2' = R_1 \times R_2$.

2) If both R_1 and R_2 are subsets of $D^{\max}$, then $r_1 \vee r_2$ exists iff $r_1 = r_2$. Hence $R_1 \bowtie R_2 = R_1 \cap R_2$.

3) follows immediately from the formula for the join operation given above and an observation that $\{r'\} \bowtie \{r''\}$ is $r' \vee r''$ if this join exists and empty otherwise.

4) follows from the basic properties of the powerdomain ordering $\sqsubseteq^\sharp$, see [33, 67, 157]. □

Recall that the join operation in the algebra of Zaniolo [181] is defined only if there are no occurrences of null values for attributes which are common for both relations. In this case, starting with antichains, we always obtain an antichain as the result. Therefore, even though Zaniolo suggests taking maximal elements among individual joins of records and the Smyth join operation takes minimal elements, they coincide in the limited setting where Zaniolo's operation is defined.

It is known that in relational algebra join can be expressed via projection, selection and Cartesian product. This is not true for generalized relations. However, if the underlying domain is the direct product of domains, then such a representation for join exists. Let $D = D_1 \times \ldots \times D_n$ and R_1, R_2 be two relations in D. For any $x \in D$ by x_i we mean its ith component, i.e. projection to D_i. Let $R \subseteq D$ be a relation, and $I(R) = \{i \,|\, \exists r \in R : r_i \neq \bot_{D_i}\}$. Let $S_i = \{\bot\} \times \ldots \times D_{k(i)} \times \ldots \times \{\bot\}$ where $k(i) = i$ if $i \leq n$ and $n - i$ otherwise and $D_{k(i)}$ is the ith factor among $2n$ factors. Then S_i is a scheme in $D \times D$. Let S be the direct product of such S_is that $i \in I(R_1)$ for $i \leq n$ and $i - n \notin I(R_1)$ for $i > n$. Let c be the conjunction of conditions $p_{S_i}(x) = p_{S_{n+i}}(x)$ for all $i \in I(R_1) \cap I(R_2)$. Then

$$R_1 \bowtie R_2 = p_S^{min}(\sigma_c(R_1 \times R_2)).$$

We finish this section by observation that the above defined operations indeed form an algebra, that is, generalized relations are closed under union $\tilde{\cup}$, difference, Cartesian product, projections, selection and join. The proof is immediate from the definitions.

Theorem 3.18 *Generalized relations are closed under the operations* $\tilde{\cup}, -, \times, p^{min}, p^{max}, \sigma, \bowtie$.
□

Summing up, we have seen how relational algebra can be reconstructed in the domain model. However, we shall not use this algebra as the basis for our languages. In section 3.2 we describe a new formalism for design of relational query languages which will generalize smoothly to many kinds of collections, ordered or not. We shall use that formalism as a foundation for query languages for partial information.

3.2 Languages for programming with collections

3.2.1 Data-oriented programming

In this section we give an overview of the *data-orientation* as a new programming language paradigm (cf. Cardelli [35]) and demonstrate some important instances such as languages for sets and bags. In particular, we cover a new approach that uses universality properties of collections as a source of operations that are to be included in a language.

It was observed by Cardelli [35] that while traditional programming languages are mostly algorithmic and procedure-oriented and pay little attention to handling of data, dealing with information systems in general and databases in particular requires more emphasis on the data. Databases are designed using some data models, e.g. relational, complex object, etc. To make it possible to program with data, it is necessary to represent the concept of a data model in a programming language. The best way to do it is to use *type systems* as a representation of data models.

Representing data models via type systems often allows *static type-checking* of programs which is particularly important in handling large data as run-time errors are very costly. To make sure that the type system is not too restrictive and does not limit the programmer's freedom, some form of polymorphism must be allowed. We allow all type constructs to be polymorphic, *e.g.* a set type constructor can be applied to any type, a product type constructor can be applied to any pair of types etc.

It was suggested by Cardelli [35] that one use *introduction* and *elimination* operations associated with a type constructor as primitives of a programming language. The introduction operations are needed to construct objects of a given type whereas the elimination operations are used to deconstruct them, or rather to do some computation with them. For example, for records, the introduction operation is forming a record with given fields, and the elimination operations are projections.

Since databases work with various kinds of collections, it is important to look at the introduction and elimination operations associated with those collections. One way to do it is to find operations that are *naturally* associated with collections. To do so, we define semantics of a collection type and try to characterize it by finding out if it has a *universality property*.

Universality properties immediately tell us what are the introduction and the elimination operations. Assume we have a collection type constructor that we denote by $\mathsf{C}(\cdot)$ and a type t. Recall that by universality property we mean that it is possible to find a set Ω of operations on the semantic domain of $\mathsf{C}(t)$, which we denote by $[\![\mathsf{C}(t)]\!]$, and a map $\eta : [\![t]\!] \to [\![\mathsf{C}(t)]\!]$ such that for any other Ω-algebra $\langle X, \Omega\rangle$ and a map $f : [\![t]\!] \to X$ there exists a unique Ω-homomorphism f^+ such that

$$\begin{tikzcd}
\llbracket t \rrbracket \arrow[r, "\eta"] \arrow[dr, "f"'] & \langle \llbracket \mathsf{C}(t) \rrbracket, \Omega \rangle \arrow[d, "f^+"] \\
& \langle X, \Omega \rangle
\end{tikzcd}$$

Hence, the introduction operations are η and those in Ω as we can use them to construct any object of type $\mathsf{C}(t)$ from objects of type t. The elimination operations are given by the universality property. In fact, the general elimination operation is the one that takes f into f^+. It is often called the *structural recursion*.

Notice, however, that the structural recursion has as its parameters the interpretation of the operations of Ω on X. Should it happen that in a particular application those do not satisfy the intended axioms (usually equations), the resulting program f^+ may not be well-defined. (We shall see some examples shortly). Therefore, it is particularly important to ensure well-definedness. One way to do it is to require that $\langle X, \Omega \rangle$ be $\langle \llbracket \mathsf{C}(s) \rrbracket, \Omega \rangle$ for some type s. Then for any function f of type $t \to \mathsf{C}(s)$, the unique completing homomorphism of the diagram below, f^+, is of type $\mathsf{C}(t) \to \mathsf{C}(s)$ and it is always well-defined.

$$\begin{tikzcd}
\llbracket t \rrbracket \arrow[r, "\eta"] \arrow[dr, "f"'] & \langle \llbracket \mathsf{C}(t) \rrbracket, \Omega \rangle \arrow[d, "f^+ = ext(f)"] \\
& \langle \llbracket \mathsf{C}(s) \rrbracket, \Omega \rangle
\end{tikzcd}$$

The reader who chose not to skip the optional section on adjunctions and monads can now be rewarded. He can see now that there is no mysticism in what we have been doing. In fact, the general form of the structural recursion corresponds to the adjunction given by the universality property while the restricted form is precisely the Kleisli category of the corresponding monad! Indeed, f^+ in that case is what we called $ext(f)$.

More precisely, assume that semantic domains of all types are objects in some category $\mathbf{A}$ and that C is a functor from $\mathbf{A}$ to Ω-$\mathbf{Alg}$. Since every $\llbracket t \rrbracket$ is an object of $\mathbf{A}$, there exists a forgetful functor $\mathsf{U} : \Omega\text{-}\mathbf{Alg} \to \mathbf{A}$. In fact, U simply "forgets" the additional structure given by Ω, that is, $\mathsf{U}(\langle \llbracket \mathsf{C}(t) \rrbracket, \Omega \rangle) = \llbracket \mathsf{C}(t) \rrbracket$. Further assume that η is a natural transformation between id and CU (this will be the case on all applications). Then the universality property stated above means that C is left adjoint to U, that is, $\mathsf{C} \dashv \mathsf{U}$.

Let $\langle \mathbb{T}, \eta, \mu \rangle$ be the monad associated with the adjunction $\mathsf{C} \dashv \mathsf{U}$, where $\mathbb{T} = \mathsf{UC}$. Then, for any type t, $\eta_{[\![t]\!]}$ is an arrow from $[\![t]\!]$ to $[\![\mathsf{C}(t)]\!]$. In other words, we can regard η as a *polymorphic function* of type $t \to \mathsf{C}(t)$. Similarly, $\mu_{[\![t]\!]}$ is an arrow from $[\![\mathsf{C}(\mathsf{C}(t))]\!]$ to $[\![\mathsf{C}(t)]\!]$. Thus, μ can be understood as a polymorphic function of type $\mathsf{C}(\mathsf{C}(t))$ to $\mathsf{C}(t)$.

Finally, $\mathbb{T} = \mathsf{UC}$ is a functor on $\mathbf{A}$. Given a function $f : s \to t$ and its semantic interpretation $[\![f]\!]$ which a function from $[\![s]\!]$ to $[\![t]\!]$ in $\mathbf{A}$, $\mathbb{T}([\![f]\!])$ is a function from $[\![\mathsf{C}(s)]\!] \to [\![\mathsf{C}(t)]\!]$. That is, $\mathbb{T}$ can be regarded as a polymorphic constructor that takes a function of type $s \to t$ and returns a function of type $\mathsf{C}(s) \to \mathsf{C}(t)$.

Associated with a monad, there is its Kleisli category. In particular, there is a functor from the Kleisli category of a monad to the original category $\mathbf{A}$ whose action on an arrow $A \to \mathbb{T}(B)$ in the Kleisli category is an arrow $\mathbb{T}(A) \to \mathbb{T}(B)$ in $\mathbf{A}$. In our terminology, this can be represented as a polymorphic constructor that takes a function of type $t \to \mathsf{C}(s)$ and produces a function of type $\mathsf{C}(t) \to \mathsf{C}(s)$. This constructor corresponds to taking f into f^+ in the universality diagram when the target is $\langle [\![\mathsf{C}(s)]\!], \Omega \rangle$. In our terminology, this constructor is called *ext*. Its examples for various adjunctions have been given in section 2.3.

The fact that the Kleisli category describes a monad can be translated into certain equations on the polymorphic functions and constructs defined above. It is a simple exercise to go through the constructions of section 2.3 and see that the following hold:

$$ext(f) = \mu \circ \mathbb{T}(f) \qquad \mu = ext(\mathsf{id}) \qquad \mathbb{T}(f) = ext(\eta \circ f)$$

Therefore, there are two equivalent presentations of the restricted form of structural recursion: one is $\langle \eta, ext \rangle$ and the other is $\langle \mathbb{T}, \eta, \mu \rangle$.

In two subsequent sections we apply this approach to sets and bags. Before we proceed with the technical development, let us offer some remarks on the origins of this approach and some of its features that are out of the scope of this thesis.

This approach finds its origins in functional languages like Machiavelli [127] which use special constructs to work with sets. It was first proposed in Breazu-Tannen, Buneman and Naqvi [25] (a related language was studied almost simultaneously by Immerman, Stemple and Patnaik [86]). Its various restrictions, properties and generalizations to other collections were studied in Breazu-Tannen, Buneman and Wong [26], Wong [179], Libkin and Wong [104, 105, 106, 107, 108] and Suciu [161].

There are a few logical languages for complex objects, e.g. COL of [6] and f-logic of [90]. However, recently the idea of using functional languages rather than logical ones for database programming has been advocated by many researchers. A survey of functional languages for databases can be found in Buneman [30]. Mathematical foundations for development of such languages for relational databases have been studied in Hillebrand, Kanellakis and Mairson [72] and Hillebrand and Kanellakis [73]. Atkinson et al. [16] point out that one of the advantages of using functional languages is having a simple comprehension syntax associated with them that closely resembles conventional query languages like SQL. It is important to note that for the languages studied here there is an associated comprehension syntax that gives us the languages of exactly the same expressive power, see Buneman et al. [34]. Initially, the idea of using comprehensions in functional programming appeared in Wadler [175]. Immerman, Patnaik and Stemple [86] and Stemple and Sheard [159] studied languages closely related to those to be presented shortly. There is an important distinction between their approach and the one that we are using here: the main computing engine of their language, the *set-reduce* operation, is based on nondeterministic choice of elements from a set, whereas there is no nondeterminism in any of the languages we study. A functional language for sets based on the operations coming from the consideration of the Plotkin powerdomain was studied in Poulovassilis and Small [140].

In the next section we describe two forms of structural recursion on sets. We discuss problems with them such as non-well-definedness, and show how to overcome these problems by imposing simple syntactic restrictions which correspond to the *ext* constructor. The language thus obtained turns out to be equivalent to what is known in database theory as *the nested relational algebra*. Strictly speaking, there are several nested relational algebras and calculi: of Thomas and Fischer [167], of Schek and Scholl [156], of Colby [41] and of Abiteboul et al. [2]. But since all of them are known to be equivalent, we speak of *the* nested relational algebra.

The methodology of using structural recursion and monads has the advantage of being easily applied to any kind of collections for which a universality property is known. We show how to use the approach to design the language for nested bags. We shall also discuss some properties of query language for bags and its representation in a set language. These results will play an important role when it comes to choosing primitives to be used in the implementation of a language for sets and or-sets.

3.2.2 Sets

The language being described is designed to work with nested sets and records. For simplicity of exposition, we assume only products (these are sufficient to simulate records). Types of objects (object types) are given by the following grammar:

$$t ::= b \mid unit \mid bool \mid t \times t \mid \{t\}$$

Here b ranges over an unspecified collection of base types (like *int, string* etc.) and *unit* is a type whose domain consists of a unique element denoted by ().

Semantics of the product type is as usual: $[\![t \times s]\!] = \{(x, y) \mid x \in [\![t]\!], y \in [\![s]\!]\}$. Semantics of the set type is the finite powerset. That is, $[\![\{t\}]\!] = \{X \mid X \subseteq_{\text{fin}} [\![t]\!]\}$.

Expressions of the language have type $s \rightarrow t$ where s and t are object types. Let us consider the question of what should be included into such language. For each type constructor there must the introduction and the elimination operations. For products these are pair formation and two projections. Since all expressions are functions, we include $(f, g) : s \times t \rightarrow r$ if $f : s \rightarrow r$ and $g : t \rightarrow r$ and $\pi_1 : s \times t \rightarrow s, \pi_2 : s \times t \rightarrow t$. For type *unit* there is only one introduction operation $! : t \rightarrow unit$ which always returns the unique element $()$ of type *unit*.

To see what must be included for sets, recall that the semantic constructor of the set type, the finite powerset $\mathbb{P}_{\text{fin}}$, can be regarded as a functor from **Set**, the category of sets, to $\mathbf{SL}_0$, the category of join-semilattices with zero. Moreover, $\mathbb{P}_{\text{fin}}$ is left adjoint to the forgetful functor $\mathbb{U} : \mathbf{SL}_0 \rightarrow \mathbf{Set}$ and η defined by $\eta_X : X \rightarrow \mathbb{P}_{\text{fin}}(X)$ where $\eta_X(x) = \{x\}$ is a natural transformation from id to $\mathbb{U}\mathbb{P}_{\text{fin}}$. This tell us that for any join-semilattice with zero $\langle A, \vee, 0\rangle$ and a function $f : X \rightarrow A$ there is a unique homomorphism f^+ such that the following diagram commutes:

$$\begin{tikzcd}
\mathrm{X} \arrow[r, "\eta"] \arrow[dr, "f"'] & \langle \mathbb{P}_{\text{fin}}(X), \cup, \emptyset\rangle \arrow[d, "f^+"] \\
& \langle A, \vee, 0\rangle
\end{tikzcd}$$

Therefore, the introduction operations for the set type constructor are $\emptyset$, the singleton formation η and union $\cup$. To represent any constant c of type t as a function, we make it a function $Kc : unit \rightarrow t$. Thus, $\emptyset$ is represented in the language as a function $empty : unit \rightarrow \{t\}$.

The universality property also tells us what the decomposition operation is. The following function is uniquely defined, provided e and u supply its range with the structure of a semilattice with zero:

$$\begin{array}{lllll}
fun & s_sru[e, h, u](\emptyset) & = & e \\
\mid & s_sru[e, h, u](\{x\}) & = & h(x) \\
\mid & s_sru[e, h, u](A \cup B) & = & u(s_sru[e, h, u](A), s_sru[e, h, u](B))
\end{array}$$

Here *s_sru* stands for the "structural recursion on the union presentation of sets". So, one possibility to deal with sets is to add the empty set, singleton formation, union and *s_sru* as operations on sets.

However, if e and u do not supply the range of *s_sru* with the structure of a semilattice with zero, then *s_sru* may not be well-defined. For example, if e is 0 of type *int*, h always returns

1, and u is $+$, one may think that $s\_sru[0, \lambda x.1, +]$ is the cardinality of a set. But this is false as the following example shows: $1 = s\_sru[0.\lambda x.1, +](\{1\}) = s\_sru[0.\lambda x.1, +](\{1, 1\}) = 2$. Unfortunately, Breazu-Tannen and Subrahmanyam [27] showed that checking if $s\_sru[e, h, u]$ is well-defined is undecidable.

To ensure well-definedness, we have to go to the monad or its Kleisli category as it was explained in section 3.2.1. Going back to the examples from section 2.3, we can see what the operations $\mathbb{T}$, μ and *ext* are. $\mathbb{T}$ simply maps a function of type $s \to t$ over a set of type $\{s\}$ returning a set of type $\{t\}$. For example, $\mathbb{T}(\lambda x.x + 1)\{1, 2, 3\} = \{2, 3, 4\}$. From now on, we shall call it *map*. μ takes a set of sets of type s and returns their union. For example, $\mu(\{\{1, 3, 5\}, \{2, 4, 6\}, \{1, 5\}\}) = \{1, 2, 3, 4, 5, 6\}$. And $ext(f)$ is defined as $\mu \circ map(f)$.

Thus, at this time we can add $map(\cdot)$ and μ as the elimination operations to the language. Note that there is still no way to interact between sets and products and to compare objects. So, we add an operator $\rho_2 : s \times \{t\} \to \{s \times t\}$ whose semantics is $\rho_2(x, \{y_1, \ldots, y_n\}) = \{(x, y_1), \ldots, (x, y_n)\}$ and the equality test. The operator ρ_2 comes from the notion of a *strong monad*, see Moggi [118]. Finally, to make the language compositional, we allow composition of functions.

The language we have obtained is shown in the figure 3.1 below. It is denoted by $\mathcal{NRL}$ (nested relational language). We have added the type of booleans and the *if-then-else* construct. For all expressions in the figure 3.1 we showed their most general types in the superscripts. In the future, those superscripts will be usually omitted as the most general type of any expression can be inferred.

Writing $\mathcal{NRL}$ expressions we shall occasionally use one level of λ-abstraction (no higher order functions) and application of a $\lambda-$term to an object. This is possible because there is a calculus equivalent to $\mathcal{NRA}$ which allows such operations, see Breazu-Tannen, Buneman and Wong [26] and Libkin and Wong [105].

The following was proved in Breazu-Tannen, Buneman and Wong [26], Paredaens and Van Gucht [132] and Wong [179].

Theorem 3.19 *1) $\mathcal{NRL}$ has precisely the expressive power of the nested relational algebra. Moreover, if eq is replaced by either of membership test, subset test, intersection or difference together with an emptiness test, the expressive power remains the same.*
2) $\mathcal{NRA}$ is conservative over relational algebra. That is, the expressive power of the sublanguage of $\mathcal{NRA}$ obtained by restricting input and output types to flat types (that is, sets of products of base types) is precisely that of the relational algebra. □

This theorem tells us about limitations of the language. Since it has essentially the power of the first order logic, it can not express recursive queries or parity of cardinality. There are various tools for analyzing the expressiveness of the first order logic, such as Ehrenfaucht-Fraïssé games,

Category with products

$$\frac{g : u \to s \quad f : s \to t}{f \circ g : u \to t} \qquad \frac{c : bool \;\; f : s \to t \quad g : s \to t}{if\ c\ then\ f\ else\ g : s \to t} \qquad \frac{f : u \to s \quad g : u \to t}{(f, g) : u \to s \times t}$$

$$\frac{}{\pi_1^{s,t} : s \times t \to s} \qquad \frac{}{\pi_2^{s,t} : s \times t \to t} \qquad \frac{}{!^t : t \to unit}$$

$$\frac{}{Kc : unit \to Type(c)} \qquad \frac{}{id^t : t \to t} \qquad \frac{}{eq^s : s \times s \to bool}$$

Set monad

$$\frac{}{\rho_2^{s,t} : s \times \{t\} \to \{s \times t\}} \qquad \frac{}{\eta^t : t \to \{t\}} \qquad \frac{}{\cup^t : \{t\} \times \{t\} \to \{t\}}$$

$$\frac{}{\mu^t : \{\{t\}\} \to \{t\}} \qquad \frac{}{empty^t : unit \to \{t\}} \qquad \frac{f : s \to t}{map\ f : \{s\} \to \{t\}}$$

Figure 3.1: Expressions of $\mathcal{NRL}$

0/1 laws (see Fagin [51]), Hanf's lemma (see Fagin et al. [52]). Here we demonstrate another tool, *the bounded degree*, which was proposed by Libkin and Wong [108]. It has an advantage of being more uniform than other techniques.

Let $G = \langle V, E\rangle$ be a graph. Define $\textit{in-deg}(v) = \textit{card}(\{v' \mid (v', v) \in E\})$ and $\textit{out-deg}(v) = \textit{card}(\{v' \mid (v, v') \in E\})$. The *degree set* of G, $\textit{deg}(G)$, is defined as $\{\textit{in-deg}(v) \mid v \in V\} \cup \{\textit{out-deg}(v) \mid v \in V\} \subseteq \mathbb{N}$. One of the reasons why most recursive queries are not first-order definable is that they may take in a graph[1] whose degree set contains only small integers and may return a graph whose degree set is large. The definition below captures this intuition.

Definition 3.4 *Let $\mathcal{L}$ be a language. It is said to have the* bounded degree property *(at type s) if, for any $f : \{s \times s\} \to \{s \times s\}$ that is definable in $\mathcal{L}$ and for any number k there exists a number c, depending on f and k only, such that $\textit{card}(\textit{deg}(f(G))) \leq c$ for any graph G satisfying $\textit{deg}(G) \subseteq \{0, 1, \ldots, k\}$.*

First, let us show how the bounded degree property can be used to prove various inexpressibility results. We consider the following queries:

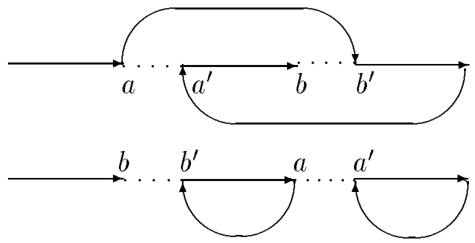
- $\textit{chain} : \{s \times s\} \to \textit{bool}$ is a query that takes a graph and returns *true* iff the graph is a chain, that is, a tree such that the out-degree of each node is at most 1.
- $\textit{bbtree} : \{s \times s\} \to \textit{bool}$ is a query that takes a graph and returns *true* iff the graph is a balanced binary tree, that is, a binary tree in which all paths from the root to the leaves have the same length.
- $\textit{dtc} : \{s \times s\} \to \{s \times s\}$ is the deterministic transitive closure. That is, if $G = \langle V, E\rangle$ is a digraph, then $\textit{dtc}(G) = \langle V, E'\rangle$ where $(v_1, v_k) \in E'$ iff there is a path $(v_1, v_2) \in E, \ldots, (v_{k-1}, v_k) \in E$ such that v_{i+1} is a unique descendant of v_i, $i = 1, \ldots, k-1$. See Immerman [84].

The deterministic transitive closure is a first-order complete problem for DLOGSPACE [84]. It is not hard to show that *chain* and *bbtree* are at most as hard as *dtc*. That is, if $\mathcal{L}$ is a language that has at least the power of the first order logic (relational algebra), then both *chain* and *bbtree* are expressible in $\mathcal{L}$ augmented with *dtc*, see Libkin and Wong [108].

Proposition 3.20 *Let $\mathcal{L}$ be a language that has at least the power of the relational algebra. Then, if $\mathcal{L}$ has the bounded degree property at type s, then neither $\textit{chain} : \{s \times s\} \to \textit{bool}$ nor $\textit{bbtree} : \{s \times s\} \to \textit{bool}$ is expressible in $\mathcal{L}$.*

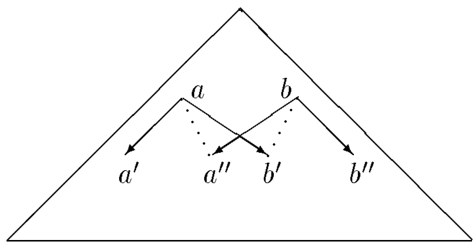
[1] We use graphs for the simplicity of exposition. Relational structures of arbitrary finite arity can be used.

Proof. We offer a proof by picture. Assume *chain* is definable; then it is possible to define an expression that, when given a chain as an input, returns its transitive closure. As shown below, using chain it is possible to determine if a precedes b by re-arranging two edges and checking if the resulting graph is a chain. First, edges from a and b to their successors a' and b' are removed and then two edges are added: one from a to b' and the other from the node with no outcoming edges to a':



But this contradicts the bounded degree property as we started with an $n-$node graph whose degree set is $\{0, 1\}$ and ended up with $\{0, 1, \ldots, n\}$.

If *bbtree* is definable, it is possible to determine if two nodes in a balanced binary tree are at the same level by re-arranging two edges as follows and checking if the result is still a balanced binary tree:



Again, we start with an $n-$node graph whose degree set is $\{0, 1, 2\}$ and, making cliques of the nodes at the same level, end up with a graph whose degree set has cardinality $\log_2(n + 1)$. □

The main reason we study this property is that it holds in $\mathcal{NRL}$.

Theorem 3.21 *$\mathcal{NRL}$ has the bounded degree property at base types.*

Proof sketch. Let $f : \{b \times b\} \to \{b \times b\}$ be an $\mathcal{NRC}$ expression where b is a base type. Then, by conservativity, f is equivalent to a relational algebra expression. Let E be an input to f and $E' = f(E)$; both E and E' are sets of pairs of elements of type b. Then for some first-order expression F we have $\forall a \forall b : (a,b) \in E' \leftrightarrow F(a,b,E)$ where E appears in F as a predicate of form $E(x,y)$.

By a neighborhood of radius r of x in E we mean the set of all nodes whose distance from x (that is, the length of a minimal path in E) does not exceed r. We denote the r-neighborhood of x by $N_r(x)$. By $N_r(X)$ we mean $\bigcup_{x \in X} N_r(x)$. According to Gaifman [55], F is a Boolean combination of certain sentences and formulae with a, b as free variables in which all quantifiers are bounded to some neighborhoods of a and b. Moreover, the maximal radius of those neighborhoods, r, is determined by F. If $deg(G) \subseteq \{0, \ldots, k\}$, then it is possible to find the number q_r of all nonisomorphic neighborhoods of radius up to r. In fact, $q_r \leq p_r 2^{p_r^2}$ where $p_r = (2k+1)^r$ is an upper bound on the size of $N_r(x)$. (Whenever we speak of a neighborhood, we assume we also know its "center". This is the reason for multiplying by p_r, which represents a choice of the center element).

Now consider a partition $X_1, \ldots, X_{q_{2r+1}}$ of the set of nodes into subsets of nodes having isomorphic neighborhoods of radius $2r+1$. Let a_1, a_2 belong to the same class X_i. If $b \notin N_{2r+1}(a_1) \cup N_{2r+1}(a_2)$, then $N_r(a_1, b)$ and $N_r(a_2, b)$ are isomorphic. In particular, $(a_1, b) \in E'$ iff $(a_2, b) \in E'$.

Let $Y_a = \{b \mid (a,b) \in E'\}$. Then there exists a constant d_i that depends on r and k only such that $\mid card(Y_{a_1}) - card(Y_{a_2}) \mid \leq d_i$ whenever $a_1, a_2 \in X_i$. Indeed, for elements b outside of $N_{2r+1}(a_1) \cup N_{2r+1}(a_2)$, (a_1, b) iff (a_2, b), and hence the only difference is in the edges either inside or between those neighborhoods. But the upper bound on the number of those is determined by k and r. In fact, it is at most $2p_{2r+1}^2 + 2p_{2r+1}$. Now assume that a_1 and a_2 are such elements in the class in the partition that the cardinality of Y_{a_1} is minimal and the cardinality of Y_{a_2} is maximal. Then we derive that the number of different outdegrees restricted to targets outside of respective $2r+1$ neighborhoods is at most d_i. Since the number of possible outdegrees inside $2r+1$ neighborhoods is bounded above by p_{2r+1}, we obtain that the number of different outdegrees in a given partition class X_i is at most $p_{2r+1} + d_i$. Since the number of elements in the partition is at most q_{2r+1}, this tells us that the number of distinct outdegrees in E' depends only on k and r. In fact, it is bounded above by $q_{2r+1} \sum_{i=1}^{q_{2r+1}} (p_{2r+1} + d_i)$. The proof for indegrees is similar. □

Corollary 3.22 *None of the following are expressible in $\mathcal{NRL}$: dtc, transitive closure, tests for connectivity of directed and undirected graphs, testing whether a graph is a tree, testing for acyclicity.* □

Therefore, there is a need in primitives that enrich the expressive power of the language. We have seen one of them - the structural recursion on the union presentation. Alternatively, one can construct sets using "insert presentation", and define *s_sri*, structural recursion on the insert presentation, as follows:

$$\begin{array}{llcl} fun & s\_sri[e,i](\emptyset) & = & e \\ \mid & s\_sri[e,i](insert(x,X)) & = & i(x, s\_sri[e,i](X)) \end{array}$$

The typing rules for both structural recursion constructs are as follows:

$$\frac{e : t \qquad h : s \to t \qquad u : t \times t \to t}{s\_sru[e,h,u] : \{s\} \to t} \qquad \frac{e : t \qquad i : s \times t \to t}{s\_sri[e,i] : \{s\} \to t}$$

The semantics of s_sri is given by $s\_sri[e,i](\{x_1, \ldots, x_n\}) = i(x_1, i(x_2, \ldots i(x_n, e) \ldots))$. Unfortunately, s_sri retains the major of problem of s_sru. It is well-defined iff $i(x, i(x,a)) = i(x,a)$ and $i(x, i(y,a)) = i(y, i(x,a))$. That is, it must be irrelevant in which order elements of a set are processed and how many duplicates are found. It was shown by Breazu-Tannen and Subrahmanyam [27] that these conditions are generally undecidable.

So, both forms of the structural recursion can express recursive queries like transitive closure, but they are not necessarily well-defined. The question arises: is there a well-defined construct that adds sufficient power to the language?

One solution proposed by Abiteboul and Beeri [1] and Gyssens and Van Gucht [70] was to include *powerset* as a primitive. The type of *powerset* is $\{t\} \to \{\{t\}\}$ and it returns the set of all subsets of a given set. It was shown by Abiteboul and Beeri that many recursive queries, such as the transitive closure, can be expressed in $\mathcal{NRL}(powerset)$. Moreover, Breazu-Tannen, Buneman and Wong [26] and independently Gyssens and Van Gucht [70] showed that

Theorem 3.23 $\mathcal{NRL}(s\_sri) \simeq \mathcal{NRL}(powerset)$. □

However, using *powerset* has a big disadvantage: it has exponential complexity. For example, to compute transitive closure of a relation, it is necessary to take the powerset of the total relation of the domain. Moreover, it was shown recently by Suciu and Paredaens [162] that any expression for transitive closure in $\mathcal{NRA}(powerset)$ needs exponential space to be evaluated. Thus, using *powerset* as an alternative to the structural recursion is unsatisfactory.

Another alternative was proposed by Libkin and Wong [105][2]. It is the *loop* construct given by

$$\frac{f : s \to s}{loop(f) : \{t\} \times s \to s}$$

[2] I was informed recently that Saraiya [155] studied the same construct and proved one direction of theorem 3.24.

with the following semantics: given an n-element set X and an object $x : s$, then $loop(f)(X, x) = f^n(x)$. Then the following holds:

Theorem 3.24 $\mathcal{NRL}(s\_sri) \simeq \mathcal{NRL}(loop)$. $\square$

We shall prove a similar theorem for bags later. The proof of theorem 3.24 is essentially the same. Note that simulation of *loop* with *s_sri* is efficient, while the reverse simulation requires exponential time. In the subsection dealing with bags we shall demonstrate an efficient simulation.

3.2.3 Bags

Sets and bags are closely related structures. While sets have been studied intensively by the theoretical database community, bags have not received the same amount of attention. However, real implementations frequently use bags as the underlying data model. For example, the "select distinct" construct and the "select average of column" construct of SQL can be better explained if bags instead of sets are used.

To use our approach, we first change the type system to

$$t ::= b \mid unit \mid bool \mid t \times t \mid \{\!|t|\!\}$$

where the $\{\!|\,|\!\}$ brackets are used for bags. To see what the bag constructs are, we must exhibit a universality property for bags.

Let X be a set and $\mathbb{P}_b(X)$ the set of all finite bags of elements of X. Define $\uplus$ as the additive union on bags. For example, $\{\!|a, a, b|\!\} \uplus \{\!|a, b, b, b|\!\} = \{\!|a, a, a, b, b, b, b|\!\}$. Then $\langle \mathbb{P}_b(X), \uplus, \{\!|\,|\!\}\rangle$ is the free commutative monoid generated by X. That is, for any other commutative monoid $\langle A, \star, e\rangle$, any map f from X to A and $\eta : X \to \mathbb{P}_b(X)$ defined by $\eta(x) = \{\!|a|\!\}$, there exists a unique monoid homomorphism f^+ such that the following diagram commutes:

$$\begin{tikzcd}
\mathrm{X} \arrow[r, "\eta"] \arrow[dr, "f"'] & \langle \mathbb{P}_b(X), \uplus, \{\!|\,|\!\}\rangle \arrow[d, "f^+"] \\
& \langle A, \star, e\rangle
\end{tikzcd}$$

Therefore, the introduction operations for the bag type constructor are the empty bag $\{\!|\,|\!\}$, the singleton formation which we denote by b_η to distinguish it from the corresponding set construct, and the additive union $\uplus$.

The universality property also tells us what the elimination operation is. The following function is uniquely defined, provided e and u supply its range with the structure of a commutative monoid:

$$\begin{array}{llcl} fun & b\_sru[e,h,u](\emptyset) & = & e \\ \mid & b\_sru[e,h,u](\{x\}) & = & h(x) \\ \mid & b\_sru[e,h,u](A \cup B) & = & u(b\_sru[e,h,u](A), b\_sru[e,h,u](B)) \end{array}$$

Note that calculation of cardinality of bag as $b\_sru[0, \lambda x.1, +]$ is now correct as 0 and $+$ do supply $\mathbb{N}$ with the structure of a commutative monoid. However, $b\_sru[0, \mathsf{id}, -]$ is not well-defined because $-1 = b\_sru[0, \mathsf{id}, -](\{\!|1,2|\!\}) = b\_sru[0, \mathsf{id}, -](\{\!|2,1|\!\}) = 1$. The reason of course is that $-$ is not commutative. Moreover, it was shown by Breazu-Tannen and Subrahmanyam [27] that checking preconditions for b_sru to be well-defined is generally undecidable.

There is an insert presentation of the bag structural recursion given by the construct

$$\frac{e : t \quad i : s \times t \to t}{b\_sri(i,e) : \{\!|s|\!\} \to t}$$

Its semantics is similar to the semantics of s_sri. Moreover, it has the same expressive power as b_sru. However, it is required that i satisfy the commutativity precondition: $i(a, i(b, X)) = i(b, i(a, X))$, which again can not be automatically verified [27].

Therefore, we need to impose syntactic restriction to ensure well-definedness, that is, we must go from the adjunction to the monad. In this case it means adding mapping of a function over bags, b_map, and flattening bag of bags, b_μ. For example,

$$b\_map(\lambda x.x+1)(\{\!|1,1,2,3,3|\!\}) = \{\!|2,2,3,4,4|\!\}$$
$$b\_\mu\{\!|\{\!|1,1|\!\}, \{\!|1,1|\!\}, \{\!|1,2,2|\!\}|\!\} = \{\!|1,1,1,1,1,2,2|\!\}$$

Note that unlike mapping over sets, b_map always preserves the cardinality of a bag.

Now we can add the bag monad constructs shown in the table below to the general categorical constructs (composition, pairing etc) to obtain the language that we call $\mathcal{NBL}$ – the nested bag language.

Bag monad

$$\frac{}{b\_\rho_2^{s,t} : s \times \{\!|t|\!\} \to \{\!|s \times t|\!\}} \qquad \frac{}{b\_\eta^t : t \to \{\!|t|\!\}} \qquad \frac{}{\uplus^t : \{\!|t|\!\} \times \{\!|t|\!\} \to \{\!|t|\!\}}$$

$$\frac{}{b\_\mu^t : \{\!|\{\!|t|\!\}|\!\} \to \{\!|t|\!\}} \qquad \frac{}{b\_empty^t : unit \to \{\!|t|\!\}} \qquad \frac{f : s \to t}{b\_map\ f : \{\!|s|\!\} \to \{\!|t|\!\}}$$

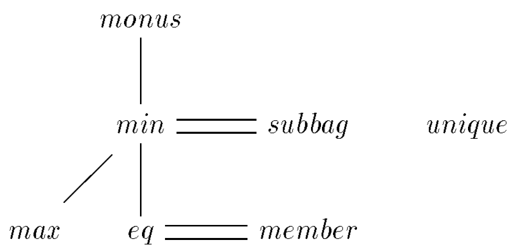
Recall that the equality test was included in $\mathcal{NRL}$, and we showed that it was enough to define various other tests (membership, subset), difference, intersection etc. However, this is not the case with bags. Moreover, with bags we have a new important construct: duplicate elimination. Our first goal is to study the relative expressive power of the following operations (see Grumbach and Milo [60] and Libkin and Wong [105]) with respect to $\mathcal{NBL}$. In what follows, $count(d, B)$ is the number of occurrences of an element d in a bag B.

- $monus : \{\!|s|\!\} \times \{\!|s|\!\} \to \{\!|s|\!\}$. $monus(B_1, B_2)$ evaluates to a B such that for every $d : s$, $count(d, B) = count(d, B_1) - count(d, B_2)$ if $count(d, B_1) > count(d, B_2)$; and $count(d, B) = 0$ otherwise.
- $max : \{\!|s|\!\} \times \{\!|s|\!\} \to \{\!|s|\!\}$. $max(B_1, B_2)$ evaluates to a B such that for every $d : s$, $count(d, B) = \max(count(d, B_1), count(d, B_2))$.
- $min : \{\!|s|\!\} \times \{\!|s|\!\} \to \{\!|s|\!\}$. $min(B_1, B_2)$ evaluates to a B such that for every $d : s$, $count(d, B) = \min(count(d, B_1), count(d, B_2))$.
- $eq : s \times s \to bool$ – equality test.
- $member : s \times \{\!|s|\!\} \to bool$ – membership test.
- $subbag : \{\!|s|\!\} \times \{\!|s|\!\} \to bool$ – subbag test.
- $unique : \{\!|s|\!\} \to \{\!|s|\!\}$. $unique(B)$ eliminates duplicates from B. That is, for every $d : s$, $count(d, B) > 0$ if and only if $count(d, unique(B)) = 1$.

The following result of Wong (see Libkin and Wong [105]) gives a precise characterization of expressive power of these constructs relative to $\mathcal{NBL}$.

Theorem 3.25 *monus can express all primitives other than unique. unique is independent of the rest of the primitives. min is equivalent to subbag and can express both max and eq. member and eq are interdefinable and both are independent of max.* □

The results of theorem 3.25 can be visualized in the following diagram.



We therefore work with the strongest combination of those primitives: *monus* and *unique*. The language $\mathcal{NBL}(monus, unique)$ will be denoted by $\mathcal{BQL}$ *(Bag Query Language)*.

How can we study the expressiveness of $\mathcal{BQL}$? One idea is to find a set language equivalent to $\mathcal{BQL}$ in terms of expressive power. Here we exhibit such a language. Add natural numbers, $\mathbb{N}$, as a base type equipped with the following: addition $+$, multiplication $\cdot$, modified subtraction (monus) $\dot{-}$ and summation $\sum$:

$$\frac{f : s \to \mathbb{N}}{\sum f : \{s\} \to \mathbb{N}}$$

with semantics $\sum f(\{x_1, \ldots, x_n\}) = f(x_1) + \ldots + f(x_n)$. Observe that $+$ can be expressed with $\sum$.

Theorem 3.26 $\mathcal{BQL} \simeq \mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$. $\square$

Of course, in order to speak of the equivalence of the languages with different type systems, one has to give a translation between those type systems. For theorem 3.26, sets are translated into bags in a straightforward manner and bags are represented as sets of pairs "element-number of occurrences".

One of the reasons this equivalence is useful is that the set language equivalent to $\mathcal{BQL}$ possesses what is called the *conservative extension property.* That is, its expressive power is independent from the set height of the intermediate data, see Libkin and Wong [105]. As a consequence,

Theorem 3.27 *Let $\mathcal{U}$ be a property of natural numbers. That is, $\mathcal{U} \subseteq \mathbb{N}$. Then membership in $\mathcal{U}$ can be expressed in $\mathcal{BQL}$ iff either $\mathcal{U}$ or $\mathbb{N} - \mathcal{U}$ is finite.*

Proof sketch. Assume that $\mathcal{U}$ and $\mathbb{N} - \mathcal{U}$ are both infinite and that membership in $\mathcal{U}$ is definable. Then the following function $p : \mathbb{N} \to \mathbb{N}$ is definable in $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$: $p(n) = 1$ if $n \in \mathcal{U}$ and $p(n) = 0$ if $n \notin \mathcal{U}$. By conservativity, p can be defined without using any set constructs, i.e. it is constructed from the arithmetic functions, constants and *if-then-else*. It is not hard to show that in this case p coincides with a polynomial almost everywhere. Since it has infinitely many roots, it must then be zero almost everywhere, contradiction. $\square$

Corollary 3.28 *None of the following functions is expressible in $\mathcal{BQL}$:*

- *parity test;*
- *division by a constant;*
- *bounded summation;*
- *bounded product;*
- $gen : \mathbb{N} \to \{\!|\mathbb{N}|\!\}$ *given by* $gen(n) = \{\!|0, 1, \ldots, n|\!\}$. $\square$

We still would like to know if the queries of corollary 3.22 are definable in $\mathcal{BQL}$ or equivalently in $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$. One way to show they are not definable is to prove that $\mathcal{BQL}$ possesses the bounded degree property. This approach is very problematic as, to the best of our knowledge, there is no known logic capturing the language $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$ nor its flat fragment. The proof of the bounded degree property for $\mathcal{NRL}$ is based on Gaifman's result about local formulae [55]. That result was proved by the quantifier elimination. This poses a problem if we try to prove the bounded degree property for flat types in $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$ or $\mathcal{BQL}$.

It was shown by Libkin and Wong [106] that adding operations to $\mathcal{NRL}$ that capture the expressive power of $\mathcal{BQL}$ amounts essentially to adding aggregate functions. Inexpressibility of recursive queries in languages with aggregates was studied by Consens and Mendelzon [42]. They showed that the transitive closure is not expressible in a first-order language with aggregate functions, provided DLOGSPACE is strictly included in NLOGSPACE.

However, there is no simple proof of inexpressibility results we want to show based on this kind of complexity arguments. For example, the deterministic transitive closure is a DLOGSPACE-complexity query. If it can be shown that the complexity of $\mathcal{BQL}$ queries is in a class that is strictly lower than DLOGSPACE, then we would have shown that the deterministic transitive closure is not definable in $\mathcal{BQL}$. It is known that $\mathrm{AC}^0 \subset$ DLOGSPACE [54]. Queries written in $\mathcal{NRL}$ have AC^0 data complexity [163]. This inclusion implies that the parity test (is the cardinality of a set even?) and the transitive closure cannot be expressed in $\mathcal{NRC}$ because they can not be done within AC^0 [54].

If $\mathcal{BQL}$ had AC^0 data complexity, the same argument would work for it. However, it is not hard to see that there are non-AC^0 queries that one can write in $\mathcal{BQL}$ since multiplication is not in AC^0 [54]. As a more interesting example of a non-AC^0 query, consider the restriction of $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$ with just two base types: $\mathbb{N}$ and *unit*. We are going to show that in such a restriction parity of the cardinality of a set is definable. First, we need

Theorem 3.29 *If a linear order $\leq_b$ is given at each base type b, then a linear order $\leq_s$ at each type s can be expressed in $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$.* $\square$

The proof of this result is based on the following lemma (see Libkin and Wong [107] for details):

Lemma 3.30 *Given a partially ordered set $\langle A, \leq \rangle$, define an ordering $\precsim$ on its finite powerset $\mathbb{P}_{\text{fin}}(A)$ as follows: $X \precsim Y$ iff $\max((X-Y) \cup (Y-X)) \subseteq Y$, or, equivalently, if $\forall x \in X - Y \exists y \in Y - X : x \leq y$. Then $\precsim$ is a partial order. Moreover, if $\leq$ is linear, then so is $\precsim$.* □

Since the usual ordering on naturals is definable ($n \leq m$ iff $n \dot{-} m = 0$), by theorem 3.29 the linear ordering $\leq_s$ is available at any type. Then the cardinality of a set $X : \{s\}$ is odd iff there is $x \in X$ such that $\{y \in X \mid y \leq_s x\}$ and $\{y \in X \mid x \leq_s y\}$ have equal cardinality. Since testing for equal cardinality can be done in $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$, one can test whether a set has odd number of elements. Thus, we exhibited another non-AC^0 query that can be defined in $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$. Note that this does not mean that parity of cardinality can be defined at any unordered type.

Therefore, one needs new techniques to study expressiveness of bag languages. Such techniques were proposed recently in Libkin and Wong [108] where the following was proved:

Theorem 3.31 *None of the following are expressible in $\mathcal{BQL}$ (or equivalently $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$): dtc, chain, bbtree, transitive closure, tests for connectivity of directed and undirected graphs, testing whether a graph is a tree, testing for acyclicity.* □

However, it remains open whether $\mathcal{BQL}$ has the bounded degree property.

Summing up, going from sets to bags buys us aggregate functions, but we still can not express recursive queries. Of course they can be expressed with structural recursion, but then verification of preconditions becomes undecidable. Hence, one needs other ways to enhance the expressive power.

Following Abiteboul and Beeri [1], Grumbach and Milo [60] introduced the *powerbag* operator into their nested bag language. The semantics of *powerbag* is the function that produces a bag of all subbags of the input bag. For example,

$$powerbag\{\!|1,1,2|\!\} = \{\!|\{\!||\!\}, \{\!|1|\!\}, \{\!|1|\!\}, \{\!|2|\!\}, \{\!|1,1|\!\}, \{\!|1,2|\!\}, \{\!|1,2|\!\}, \{\!|1,1,2|\!\}|\!\}$$

They also defined the *powerset* operator on bags as *unique* ∘ *powerbag*. For example,

$$powerset\{\!|1,1,2|\!\} = \{\!|\{\!||\!\}, \{\!|1|\!\}, \{\!|2|\!\}, \{\!|1,1|\!\}, \{\!|1,2|\!\}, \{\!|1,1,2|\!\}|\!\}$$

We do not consider *powerset* on bags further because of the following result.

Proposition 3.32 *$\mathcal{BQL}(powerbag) \simeq \mathcal{BQL}(powerset)$.*

Proof sketch. Suppose a bag B is given; then another bag B' can be constructed such that for any $a \in B$, B' contains a pair $(a, \{\!|a, \ldots, a|\!\})$ where the cardinality of the second component is $count(a, B)$. Let $B'' = unique(B')$; then B'' can be computed by $\mathcal{BQL}$. Now observe that changing the second component of every pair to its *powerset* and then $b\_map(b\_\rho_2)$ followed by flattening will give us a bag where each element $a \in B$ will be given a unique label. Now applying *powerset* to this bag followed by elimination of labels produces $powerbag(B)$. □

In contrast to the set languages, the structural recursion for bags is strictly stronger than *powerbag*.

Theorem 3.33 $\mathcal{BQL}(powerbag) \subsetneq \mathcal{BQL}(b\_sri)$.

Proof sketch. First, *powerbag* can be expressed using *b_sri*, cf. [25]. Then it can be shown that any function in $\mathcal{BQL}(powerbag)$ produces outputs whose sizes are bounded by an elementary function on the size of the input, but in $\mathcal{BQL}(b\_sri)$ it is possible to define a function that on the input of size n produces the output of the hyperexponential size (where the height of the stack of powers depends on n) and hence can not be bounded by an elementary function. □

As an illustration of theorem 3.33, we characterize precisely the classes of arithmetic functions that both languages express. It also gives an alternative proof of theorem 3.33.

Theorem 3.34 *a) The class of functions* $f : \mathbb{N} \times \ldots \times \mathbb{N} \to \mathbb{N}$ *definable in* $\mathcal{BQL}(b\_sri)$ *coincides with the class of primitive recursive functions.*
b) The class of functions $f : \mathbb{N} \times \ldots \times \mathbb{N} \to \mathbb{N}$ *definable in* $\mathcal{BQL}(powerbag)$ *coincides with the class of Kalmar-elementary functions.* □

Similar results for other languages for bags or sets with built-in natural numbers were proved in Grumbach and Milo [60] and Immerman et al. [86].

The bounded loop construct for bags is given by

$$\frac{f : s \to s}{loop^t(f) : \{\!|t|\!\} \times s \to s}$$

Its semantics is as follows: $loop(f)(\{\!|o_1, \ldots, o_n|\!\}, o) = f(\ldots f(o) \ldots)$ where f is applied n times to o.

Similarly to the set case, we have

Theorem 3.35 $\mathcal{BQL}(loop) \simeq \mathcal{BQL}(b\_sri)$.

Proof. For the $\mathcal{BQL}(loop) \subseteq \mathcal{BQL}(b\_sri)$ part, it suffices to observe that $loop(f)(n, e) = b\_sri(f \circ \pi_2,\ e)(n)$, where n is a shorthand for the bag of n units.

To prove $\mathcal{BQL}(b\_sri) \subseteq \mathcal{BQL}(loop)$, we first define a function $g : \{\!|t|\!\} \to \{\!|\{\!|t \times \mathbb{N}|\!\}|\!\}$ where $\mathbb{N}$, as usual, is an abbreviation for $\{\!|unit|\!\}$. This function g, when applied to a bag B, produces the bag whose elements are bags of pairs, such that mapping π_1 over such a bag gives B and mapping π_2 gives a bag of numbers from 1 to n where n is the cardinality of B. Moreover, $g(B)$ contains all possible labeling of elements by numbers. For example, $g\{\!|a, b|\!\} = \{\!|\{\!|(a, 1), (b, 2)|\!\}, \{\!|(a, 2), (b, 1)|\!\}|\!\}$.

To show that such g is definable, first notice that *powerbag* is definable in $\mathcal{BQL}(loop)$. Indeed, it is easy to define an expression that, given a bag, produces all subbags of cardinality one less than the cardinality of the bag. Now using the *loop* construct with such an expression gives us *powerset* and therefore *powerbag*. If n is the cardinality of B (which is obtained by applying $b\_map(!)$ to B), then *powerset* applied to it produces the bag of all numbers from 0 to n. Hence, we can construct a bag of all numbers from 1 to n. Now take the cartesian product of this bag and B and denote it by B'. Then $powerbag(B')$ contains all bags whose elements are pairs, the first component being an element of B and the second component being a number from 1 to n. Such a bag B'' makes it to the output of g iff the two conditions are satisfied: first, $b\_map(\pi_1)(B'') = B$ and second, $b\_map(\pi_2)(B'') = \{\!|1, \ldots, n|\!\}$. Since equality test and selection are available, g can be defined in $\mathcal{BQL}(loop)$.

Now we must define $b\_sri(i, e) : \{\!|s|\!\} \to t$ in $\mathcal{BQL}(loop)$. Given a bag $B : \{\!|s|\!\}$, to determine the value of $b\_sri(i, e)$ on B first apply g to B to obtain B_0. Define $h : \{\!|s \times \mathbb{N}|\!\} \times t \to \{\!|s \times \mathbb{N}|\!\} \times t$ as follows. $h(B', a)$ selects the pair (b, k) from B' with the maximal k and returns $(B' monus \{\!|(b, k)|\!\}, i(b, a))$. Now $loop(h)$ applied to $(B, (B', e))$, where B' is an element of B_0, returns a pair whose second component is the value of $sri(i, e)$ on B if elements of B are enumerated for applying the structural recursion as they are labeled in B'. Threfore, mapping this *loop* over B_0 we obtain all possible outcomes of $b\_sri(i, e)(B)$ depending on in which order i was applied. If $b\_sri(i, e)$ is well-defined, then the order does not matter and applying *unique* gives us a singleton bag that contains $b\_sri(i.e)(B)$. This shows that *b_sri* is expressible in $\mathcal{BQL}(loop)$. $\Box$

Note that as in the set case, the simulation of *loop* with *b_sri* is efficient, while the reverse simulation requires exponential time. However, if linear orderings are given at base types, one can efficiently lift them to arbitrary types (cf. theorem 3.29) and define a function $sort : \{s\} \to \{s \times \mathbb{N}\}$ such that $sort(X) = \{(x_1, 1), \ldots, (x_n, n)\}$ whenever $x_1 \leq_s \ldots \leq_s x_n$ by counting the numebr of elements in a set which are less than a given element. Using *sort* we can make both translations efficient: going from *loop* to *b_sri* we use *sort* to pick an order in which elements are given to *b_sri* for processing.

Theorem 3.35 also sheds some light on theorem 3.34 by showing that its statement is very intuitive and well expected. There are two classical results in recursion theory [122]. One, due to Meyer and Ritchie, states that the functions computable by the language that has assignment statement and *for* n *do* S, are precisely the primitive recursive functions. The semantics of *for* n *do* S is to repeat S n times. A similar result by Robinson, later improved by Gladstone, says that the primitive recursive functions are functions built from the initial functions by composition and iteration. That is, $f(n, \vec{x}) = g^{(n)}(\vec{x})$, see [122]. The structural recursion for bags is essentially the *for–do* construct and, not surprisingly, it expresses precisely the primitive recursive functions.

We have seen the equivalence $\mathcal{BQL} \simeq \mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-})$. Now it is natural to ask whether it continues to hold (under the translations of theorem 3.26) when set and bag languages are augmented with *powerset* and *powerbag* or structural recursion. Consider the following primitive in the set language (cf. corollary 3.28):

$$gen : \mathbb{N} \to \{\mathbb{N}\}, \qquad gen(n) = \{0, 1, \ldots, n\}$$

Under translations of theorem 3.26, it corresponds to the bag language primitive that takes a bag of n units and returns bag of bags containing i units for each $i = 0, 1, \ldots, n$. In other words, it is $powerset^{unit} = unique \circ powerbag^{unit}$.

Having made this observation, we can show the separation result.

Theorem 3.36 *a)* $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-}, powerset) \subsetneq \mathcal{BQL}(powerbag)$*;*
b) $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-}, s\_sri) \subsetneq \mathcal{BQL}(b\_sri)$*.* □

Now we have a problem of filling the gap between set and bag languages with power operators or structural recursion. It turns out that the *gen* primitive is sufficiently powerful to do the job.

Theorem 3.37 *a)* $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-}, powerset, gen) \simeq \mathcal{BQL}(powerbag)$*;*
b) $\mathcal{NRL}(\mathbb{N}, \Sigma, \cdot, \dot{-}, s\_sri, gen) \simeq \mathcal{BQL}(b\_sri)$*.* □

We shall use these equivalences later for making decision about adding power to the implementation of the language for sets and or-sets.

This conclues our discussion of the background we need in order to develop the semantics of partiality and to design query languages for partial data.

WHERE ARE WE NOW AND WHERE ARE WE GOING?

It is time to pause for a moment and see where we have arrived to and where we should go from here. In the introduction. we formulated two main themes of this thesis: *partiality of data is represented via orderings on values* and *semantics suggests programming constructs.*

In this chapter, we have developed the background necessary to put these ideas to work. First, we have studied the domain-theoretic model that accommodates various collections of partial values. Then we have seen how universality properties of semantics of datatypes can be turned into the programming language syntax.

Our first task is to specialize the general theory of section 3.1 to various collections of partial data. These include sets under both closed and open world assumptions, or-sets and the approximation constructs. Keeping our second goal of developing query languages in mind, not only do we have to come up with semantic models for those, but we also must find their universality properties. Having developed the semantics of collections and proved their universality properties, we can use the general techniques of section 3.2 to design languages to work with partial information.

Semantics of partial data is studied in the next chapter. We exhibit orderings and semantic domains for all kinds of collections we have seen and, furthermore, prove the universality properties for those semantic domains.

We then proceed in chapter 5 to design languages for sets and or-sets (possibly with null values) and approximation constructs. We shall show that the language for sets and or-sets possesses many intersting properties. Two are of special importance. First, semantics of objects can be incorporated into the language by means of normalization of objects. The process of normalization will be studied in details. Second, we show that the language has adequate expressive power to encode approximation constructs and program with them.

Finally, in chapter 6 we describe a practical system based on the language for sets and or-sets and show how it can be used for querying incomplete databases and producing approximate answers to queries.

Chapter 4

Semantics of Partial Information

The purpose of this chapter is to study the semantics of partial data. Our first goal is to choose orderings on various kinds of collections. To do so, we formalize elementary updates on collections which improve our knowledge about the real world situation represented by that data, that is, add information. Then we characterize transitive closures of those updates, thus obtaining the orderings. We carry out this program for OWA and CWA sets and bags, or-sets and all approximations.

We use the orderings to define the semantics of collections of partial objects. It will be shown that the semantics and the orderings agree naturally. Furthermore, we establish an intimate connection between approximation constructs and certain objects obtained by combination of OWA sets and or-sets. This semantic connection will be used extensively in chapter 5 to design languages for giving approximate answers to queries.

Our approach to the programming language design is based on turning universality property of semantics of types into syntax. In the second half of this chapter we describe various collections as free ordered algebras. These include OWA and CWA sets, or-sets and two iteration constructs, that correspond to sets of or-sets and or-sets of sets.

Furthermore, we show that most approximations arise as free constructions. To do so, we first define formal models of approximations and propose a classification of those. The proposed classification gives rise to ten possible approximation constructs. We study them thoroughly and prove that some of them possess universality properties. Some of them are shown not to be free ordered algebras generated by posets in a "naive" way, but we find a way to repair it by showing that they do possess universality properties with respect to different generating posets and restricted classes of maps. It will be seen in chapter 5 that such characterizations are sufficient for defining the general structural recursion based language and certain sublanguages thereof.

4.1 Order and Semantics

There are several goals we want to pursue in this section. First, we show how sets under both OWA and CWA, or-sets and bags should be ordered. We then use the orderings to give the semantics of collections of partial data. Second, we analyze the approximation constructs and propose a classification of those. Having done this, we define and study orderings and semantics of the approximation constructs in the same way as we did it for the other collections. Finally, we show how to represent the approximation constructs using sets and or-sets.

4.1.1 Orderings on collections

In this section we study the following general problem. Given a poset $\langle A, \leq \rangle$ and the family of all collections (sets, bags, or-sets etc.) over A, how do we order those? As usual, our interpretation of the partial order is "being more informative". What does it mean to say that one collection of partial descriptions is more informative than another?

The technique we use to answer this question is the following. We try to define "elementary updates" that add information. For example, for CWA databases such updates should add information to individual records. For OWA we may have additional updates that add records to a database. For or-sets, reducing the number of possibilities adds information as an or-sets denotes one of its elements. We formalize those updates and then look at their transitive closure. That is, a collection C_1 is more informative than C_2 if C_1 can be reached from C_2 by a sequence of elementary updates that add information. We characterize five orderings that arise this way: for OWA sets, CWA sets, or-sets and bags under both CWA and OWA.

As we mentioned in section 3.1, redundancies represented by comparable elements can usually be removed. That is, we often represent database objects as antichains. Therefore, there are two ways to perform updates that add information. One way is to keep all elements, even those that are comparable. The other way is to remove redundancies, that is, to make sure that the result of each elementary update is an antichain again. These two ways lead to some orderings on either antichains of ordered sets or arbitrary subsets thereof. We shall consider both and show that they coincide.

Ordering CWA databases

In a closed world database, it is possible to update individual records but it is impossible to add new records. To understand what the elementary updates are, let us consider again the example we used in chapter 1.

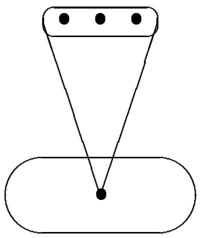
Name	Salary	Room
$\perp$	$\perp$	076
Mary	17K	$\perp$

$\xrightarrow{\text{CWA}}$

Name	Salary	Room
John	15K	076
Ann	$\perp$	076
Mary	17K	561

In these relations, we use generic nulls. The first relation says that there exists room 076, and that Mary makes 17K. Note that there could be more than one person in 076. To see why, it might be easier to consider the first relation as obtained from the second one by *losing* information. Assume we had information about two people in 076 and then lost information about their names and salaries. As the result, there are two copies of the record $\boxed{\perp \mid \perp \mid 076}$. However, we are dealing with sets and duplicates are always removed. Therefore, losing information contained in two records would result in getting just one record in the new database. In other words, an incomplete record can be updated in various ways that give rise to a number of new records, and this is consistent with the closed world assumption.

The third record in the updated database is obtained from the second record in the initial database by adding the salary value. Thus, we see that the way the closed world databases are made more informative is via getting more information about individual records. The following picture illustrates those updates. We simply remove an element (record) from a database and replace it by a number of more informative elements (records).



There are two ways to formalize those updates, depending on whether arbitrary sets or only antichains are allowed. Let $X \subseteq A$ be a finite subset of the poset A. Let $x \in X$ and $X' \subseteq A$ be a finite nonempty subset of A such that $x \leq x'$ for all $x' \in X'$. Then we allow the following update:

$$X \overset{\text{CWA}}{\longmapsto} (X - x) \cup X'$$

For antichains, we need to impose two additional restrictions. First, X' must be an antichain, and second, the result must be an antichain. To ensure that the second requirement is satisfied, we keep only maximal elements. That is, in the case of antichains the legitimate updates are

$$X \overset{\text{CWA}}{\longmapsto}_a \max((X - x) \cup X')$$

We now say that $X \sqsubseteq^{\text{CWA}} Y$ if $X, Y \subseteq A$ and Y can be obtained from X by a sequence of updates $\overset{\text{CWA}}{\longmapsto}$, that is, $\sqsubseteq^{\text{CWA}}$ is the transitive closure of $\overset{\text{CWA}}{\longmapsto}$ on $\mathbb{P}_{\text{fin}}(A)$. Similarly, $X \sqsubseteq_a^{\text{CWA}} Y$ if X, Y are finite antichains of A and Y can be obtained from X by a sequence of updates $\overset{\text{CWA}}{\longmapsto}_a$, that is, $\sqsubseteq_a^{\text{CWA}}$ is the transitive closure of $\overset{\text{CWA}}{\longmapsto}_a$ on $\mathbb{A}_{\text{fin}}(A)$.

Our claim is the following.

The closed world databases must be ordered by the Plotkin ordering.

We justify it by proving

Theorem 4.1 a) *Let $X, Y \in \mathbb{P}_{\text{fin}}(A)$. Then $X \sqsubseteq^{\text{CWA}} Y$ iff $X \sqsubseteq^{\natural} Y$.*
b) *Let $X, Y \in \mathbb{A}_{\text{fin}}(A)$. Then $X \sqsubseteq_a^{\text{CWA}} Y$ iff $X \sqsubseteq^{\natural} Y$.*

Proof. The proof of part a) is easy. First, $X \overset{\text{CWA}}{\longmapsto} Y$ implies $X \sqsubseteq^{\natural} Y$ and hence $X \sqsubseteq^{\text{CWA}} Y$ implies $X \sqsubseteq^{\natural} Y$. Conversely, if $X \sqsubseteq^{\natural} Y$, let $Y_x = \{y \in Y \mid y \geq x\}$. Then updates $X \overset{\text{CWA}}{\longmapsto} (X - x) \cup Y_x$ give a way from X to Y.

To prove part b) first observe that $X \overset{\text{CWA}}{\longmapsto}_a Y$ implies $X \sqsubseteq^{\natural} Y$ and hence $X \sqsubseteq_a^{\text{CWA}} Y$ implies $X \sqsubseteq^{\natural} Y$.

Assume $X, Y \in \mathbb{A}_{\text{fin}}(A)$ and $X \sqsubseteq^{\natural} Y$. We prove by induction on the cardinality of $X \cup Y$ that there exists a family $\{X_1, \ldots, X_l\}$ of subsets of $X \cup Y$ such that $X \overset{\text{CWA}}{\longmapsto}_a X_1 \overset{\text{CWA}}{\longmapsto}_a \ldots \overset{\text{CWA}}{\longmapsto}_a X_l \overset{\text{CWA}}{\longmapsto}_a Y$. In the case when either X or Y is a singleton, we need just one $\overset{\text{CWA}}{\longmapsto}_a$ arrow. Assume that $card(X) = m$, $card(Y) = k$, $m, k > 1$ and for any sets of cardinalities less than m and k the statement above is true.

Let X^0 be a minimal (with respect to inclusion) subset of X such that $X \sqsubseteq^{\natural} Y$. We first show that X^0 and Y are $\overset{\text{CWA}}{\longmapsto}_a$ related by a sequence of subsets of $X^0 \cup Y$. If X^0 is a singleton, this is immediate. If X^0 has more than one element, consider $x \in X^0$. Then $X^0 - x \not\sqsubseteq^{\natural} Y$. Therefore, there exists an element $y \in Y$ such that $y \not\geq z$ for any $z \in X^0 - x$ (otherwise we would have $X^0 - x \sqsubseteq^{\natural} Y$). Let Y^0 be the set of all $y \in Y$ with this property; we know $Y^0 \neq \emptyset$. Then $X^0 - x \sqsubseteq^{\natural} Y - Y^0$. Indeed, if $x' \in X^0 - x$, then there exists $y \in Y$ such that $x' \leq y$. Moreover, $y \notin Y^0$ by the definition of Y^0. Hence, $X^0 - x \sqsubseteq^{\flat} Y - Y^0$. If $y \in Y - Y^0$, then there exists $x' \in X^0$ such that $x' \leq y$. If x is the only element in X^0 that is under y, then $y \in Y^0$. Hence, we can pick $x' \in X^0 - x$. This shows $X^0 - x \sqsubseteq^{\sharp} Y - Y^0$ and hence $X^0 - x \sqsubseteq^{\natural} Y - Y^0$.

Now by induction hypothesis we can find a sequence $Z_1, \ldots, Z_p$ of subsets of $(X^0 - x) \cup (Y - Y^0)$ such that $X^0 - x \overset{\text{CWA}}{\longmapsto}_a Z_1 \overset{\text{CWA}}{\longmapsto}_a \ldots \overset{\text{CWA}}{\longmapsto}_a Z_p \overset{\text{CWA}}{\longmapsto}_a Y - Y^0$. Since for any $Z \subseteq (X^0 - x) \cup (Y - Y^0)$, $Z \cup Y^0$ is an antichain, we obtain $X^0 \overset{\text{CWA}}{\longmapsto}_a (X^0 - x) \cup Y^0 \overset{\text{CWA}}{\longmapsto}_a Z_1 \cup Y^0 \overset{\text{CWA}}{\longmapsto}_a \ldots \overset{\text{CWA}}{\longmapsto}_a Z_p \cup Y^0 \overset{\text{CWA}}{\longmapsto}_a (Y - Y^0) \cup Y^0 = Y$. To see that $X \sqsubseteq_a^{\text{CWA}} Y$, we apply exactly the same updates to X. The only difference with the sequence of updates above is that now at any stage there are possibly some elements of $X - X^0$ added. However, they disappear at the last stage as $X \sqsubseteq^\natural Y$ and we always apply max. This shows $X \sqsubseteq_a^{\text{CWA}} Y$. Theorem is proved. $\square$

Corollary 4.2 *Let X and Y be finite antichains in A such that $X \sqsubseteq^\natural Y$. Then it is possible to find a sequence of antichains $X_1, \ldots, X_n$ such that $X_1, \ldots, X_n \subseteq X \cup Y$ and $X \overset{\text{CWA}}{\longmapsto}_a X_1 \overset{\text{CWA}}{\longmapsto}_a \ldots \overset{\text{CWA}}{\longmapsto}_a X_n \overset{\text{CWA}}{\longmapsto}_a Y$.* $\square$

Ordering OWA databases

In an open world database, it is possible to update individual records and add new records. As in the case of the CWA databases, consider a simple example to understand what the elementary updates are.

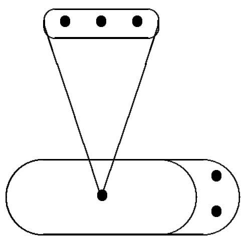
Name	Salary	Room
$\perp$	$\perp$	076
Mary	17K	$\perp$

$\xrightarrow{\text{OWA}}$

Name	Salary	Room
John	15K	076
Ann	$\perp$	325
Mary	17K	561

Some of the records in the second relation, that we view as a more informative one, are obtained by modifying records of the original relation. However, one record, $\boxed{\text{Ann} \mid \perp \mid 325}$ can not be obtained by modifying any record in the original database. The reason it was put there is that the database is open for new records. Under this interpretation, we view adding records as an update that adds information. In the above example, adding that record improves our knowledge about what can be a university or a company database of employees.

The following picture illustrates updates that are used to improve information stored in an open world database. Not only do we allow replacing an element (record) by a number of more informative elements (records), but we also allow adding new records.



Similarly to the CWA case, there are two ways to formalize these updates, depending on whether arbitrary sets or only antichains are allowed. Let $X \subseteq A$ be a finite nonempty subset of the poset A. Let $x \in X$ and $X' \subseteq A$ be a finite subset of A such that $x \leq x'$ for all $x' \in X'$. Let X'' be an arbitrary finite subset of A. Then we allow the following updates:

$$X \stackrel{\text{OWA}}{\longmapsto} (X - x) \cup X' \quad \text{ and } \quad X \stackrel{\text{OWA}}{\longmapsto} X \cup X''$$

For antichains, we impose an additional restriction that the result always be an antichain. We do it by keeping only maximal elements in the results, see section 3.1. Another reason for keeping only maximal elements will be seen shortly. Therefore, in the case of antichains the legitimate updates are

$$X \stackrel{\text{OWA}}{\longmapsto}_a \max((X - x) \cup X') \quad \text{ and } \quad X \stackrel{\text{OWA}}{\longmapsto} \max(X \cup X'')$$

We say that $X \sqsubseteq^{\text{OWA}} Y$ if $X, Y \subseteq A$ and Y can be obtained from X by a sequence of updates $\stackrel{\text{OWA}}{\longmapsto}$, that is, $\sqsubseteq^{\text{OWA}}$ is the transitive closure of $\stackrel{\text{OWA}}{\longmapsto}$ on $\mathbb{P}_{\text{fin}}(A)$. Similarly, $X \sqsubseteq_a^{\text{OWA}} Y$ if X, Y are finite antichains of A and Y can be obtained from X by a sequence of updates $\stackrel{\text{OWA}}{\longmapsto}_a$, that is, $\sqsubseteq_a^{\text{OWA}}$ is the transitive closure of $\stackrel{\text{OWA}}{\longmapsto}_a$ on $\mathbb{A}_{\text{fin}}(A)$.

Our main claim about ordering of OWA databases is the following.

The open world databases must be ordered by the Hoare ordering.

We justify it by proving

Theorem 4.3 a) *Let $X, Y \in \mathbb{P}_{\text{fin}}(A)$. Then $X \sqsubseteq^{\text{OWA}} Y$ iff $X \sqsubseteq^\flat Y$.*
b) *Let $X, Y \in \mathbb{A}_{\text{fin}}(A)$. Then $X \sqsubseteq_a^{\text{OWA}} Y$ iff $X \sqsubseteq^\flat Y$.*

Proof. The proof of part a) is very similar to the proof of a) in theorem 4.1. To prove b), first observe that the inclusion $\sqsubseteq^{\text{OWA}} \subseteq \sqsubseteq^\flat$ is immediate. Let $X, Y \in \mathbb{A}_{\text{fin}}(A)$ and $X \sqsubseteq^\flat Y$. Let $Y_X = \{y \in Y \mid \exists x \in X : x \leq y\}$. Then $X \sqsubseteq^\natural Y_X$ and by theorem 4.1 we can find a family $X_1, \ldots, X_n$ of subsets of $X \cup Y_X$ such that $X \stackrel{\text{CWA}}{\longmapsto}_a X_1 \stackrel{\text{CWA}}{\longmapsto}_a \ldots \stackrel{\text{CWA}}{\longmapsto}_a X_n \stackrel{\text{CWA}}{\longmapsto}_a Y_X$. Since $\stackrel{\text{CWA}}{\longmapsto}$ updates are a particular case of $\stackrel{\text{OWA}}{\longmapsto}$ updates, we obtain $X \stackrel{\text{OWA}}{\longmapsto}_a X_1 \stackrel{\text{OWA}}{\longmapsto}_a \ldots \stackrel{\text{OWA}}{\longmapsto}_a X_n \stackrel{\text{OWA}}{\longmapsto}_a Y_X \stackrel{\text{OWA}}{\longmapsto}_a \max(Y_X \cup (Y - Y_X)) = Y$ which proves $X \sqsubseteq_a^{\text{OWA}} Y$. $\square$

Corollary 4.4 *Let X and Y be finite antichains in A such that $X \sqsubseteq^\flat Y$. Then it is possible to find a sequence of antichains $X_1, \ldots, X_n$ such that $X_1, \ldots, X_n \subseteq X \cup Y$ and $X \stackrel{\text{OWA}}{\longmapsto}_a X_1 \stackrel{\text{OWA}}{\longmapsto}_a \ldots \stackrel{\text{OWA}}{\longmapsto}_a X_n \stackrel{\text{OWA}}{\longmapsto}_a Y$.* $\square$

Ordering or-sets

We now define update rules for or-sets. We start with a simple example.

X_1 : $\langle$

Name	Salary	Room
John	$\perp$	076
Ann	$\perp$	$\perp$
Mary	17K	$\perp$

$\rangle$ $\xrightarrow{\text{or}-\text{set}}$ X_2 : $\langle$

Name	Salary	Room
John	$\perp$	076
Ann	13K	$\perp$

$\rangle$

There are two reasons why we view X_2 as a more informative or-set than X_1. First, additional information about Ann was obtained. It is now known that her salary is 13K. Second, one of the records was removed. Note that removing an element from an or-set makes it more informative. Indeed, while $\langle 1, 2, 3\rangle$ is an integer which is either 1 or 2 or 3, $\langle 1, 2\rangle$ is an integer which is 1 or 2, so we have additional information that it can not be 3. Finally, $\langle 1\rangle$ is an example of perfect knowledge as it stands for the integer 1.

Therefore, we consider two types of updates on or-sets: improving information about individual records and removing elements:

$$X \overset{\text{or}}{\longmapsto} (X - x) \cup X' \qquad \text{if } x \in X \text{ and } x \leq x' \text{ for all } x' \in X' \text{ and } X' \neq \emptyset$$

$$X \overset{\text{or}}{\longmapsto} X - x \qquad \text{if } x \in X \text{ and } X - x \neq \emptyset$$

To redefine these updates for antichains, we must decide how redundancies in or-sets are removed. We suggest that only minimal elements be kept in the results. To see why, consider the following or-set with two comparable records:

$\langle$

Name	Room
John	076
John	**un**

$\rangle$

This or-set denotes a person whose name is John and who is either in room 076 or in an unknown room. The semantics of this is exactly as having one record for John in an unknown room. (This will be made precise in the next section.) Hence, we prefer to retain the minimal elements. Then the updates for antichains become

$$X \overset{\text{or}}{\longmapsto} \min((X - x) \cup X') \qquad \text{if } x \in X \text{ and } x \leq x' \text{ for all } x' \in X' \text{ and } X' \neq \emptyset$$

$$X \overset{\text{or}}{\longmapsto} X - x \qquad \text{if } x \in X \text{ and } X - x \neq \emptyset$$

Our next claim about orderings on collections is the following.

The or-sets must be ordered by the Smyth ordering.

To make it formal, we define $\sqsubseteq^{\mathrm{or}}$ and $\sqsubseteq^{\mathrm{or}}_a$ as the transitive closure of $\stackrel{\mathrm{or}}{\longmapsto}$ and $\stackrel{\mathrm{or}}{\longmapsto}_a$ respectively.

Theorem 4.5 a) *Let $X, Y \in \mathbb{P}_{\mathrm{fin}}(A)$, $X, Y \neq \emptyset$. Then $X \sqsubseteq^{\mathrm{or}} Y$ iff $X \sqsubseteq^{\sharp} Y$.*
b) *Let $X, Y \in \mathbb{A}_{\mathrm{fin}}(A)$, $X, Y \neq \emptyset$. Then $X \sqsubseteq^{\mathrm{or}}_a Y$ iff $X \sqsubseteq^{\sharp} Y$.*

Proof. The proof of a) is similar to proofs of a) in theorems 4.1 and 4.3. To prove b), first observe that $X \stackrel{\mathrm{or}}{\longmapsto}_a Y$ implies $X \sqsubseteq^{\sharp} Y$, and hence $X \sqsubseteq^{\mathrm{or}}_a Y$ implies $X \sqsubseteq^{\sharp} Y$.

Now we prove the following claim. If $X \sqsubseteq^{\natural} Y$, $X \cap Y = \emptyset$ and $X' \sqsubseteq^{\natural} Y$ for no proper subset $X' \subset X$, then $X \sqsubseteq^{\mathrm{or}}_a Y$ and moreover only elements of $X \cup Y$ are used in the $\stackrel{\mathrm{or}}{\longmapsto}_a$ transformations. We prove it by induction on $card(X \cup Y)$. When one set is a singleton, the statement is immediate. Assume cardinalities of both X and Y are bigger than one. Let $Y_x = \{y \in Y \mid y \geq x\}$. We claim that there exists $x \in X$ such that $X - x \sqsubseteq^{\natural} Y - Y_x$. Assume that this is not the case. Then for any x, $X - x \not\sqsubseteq^{\natural} Y - Y_x$. That is, there exists $x_1 \in X$ such that x_1 is not under any element of $Y - Y_x$. In other words, $Y_{x_1} \subseteq Y_x$. Continuing, we obtain $Y_x \supseteq Y_{x_1} \supseteq Y_{x_2} \supseteq \ldots$. Since X and Y are finite, we have $Y_{x_i} = Y_{x_j}$ for some distinct x_i and x_j. But in this case $X - x_i \sqsubseteq^{\natural} Y$ which contradicts the minimality of X. Hence, $X - x \sqsubseteq^{\natural} Y - Y_x$ for some x. By the induction hypothesis, $X - x \sqsubseteq^{\mathrm{or}}_a Y - Y_x$. Since only elements of $(X - x) \cup (Y - Y_x)$ were used in the transformations, $X \sqsubseteq^{\mathrm{or}}_a (Y - Y_x) \cup x \stackrel{\mathrm{or}}{\longmapsto}_a Y$ which finishes the proof of the claim.

Now it is easy to see that the condition $X \cap Y = \emptyset$ can be dropped as adding $X \cap Y$ to any transformation does not interfere with its result. Hence, $X \sqsubseteq^{\natural} Y$ implies $X \sqsubseteq^{\mathrm{or}}_a Y$ if X is minimal such with respect to inclusion.

Let $X \sqsubseteq^{\sharp} Y$. Define $X_Y = \{x \in X \mid \exists y \in Y : x \leq y\}$. Then $X_Y \sqsubseteq^{\natural} Y$. Let X'_Y be a minimal with respect to inclusion subset of X_Y such that $X'_Y \sqsubseteq^{\natural} Y$. Then $X \sqsubseteq^{\mathrm{or}}_a X'_Y \sqsubseteq^{\mathrm{or}}_a Y$ finishes the proof. □

Corollary 4.6 *Let X and Y be finite antichains in A such that $X \sqsubseteq^{\sharp} Y$. Then it is possible to find a sequence of antichains $X_1, \ldots, X_n$ such that $X_1, \ldots, X_n \subseteq X \cup Y$ and $X \stackrel{\mathrm{or}}{\longmapsto}_a X_1 \stackrel{\mathrm{or}}{\longmapsto}_a \ldots \stackrel{\mathrm{or}}{\longmapsto}_a X_n \stackrel{\mathrm{or}}{\longmapsto}_a Y$.* □

Ordering bags

We now use similar techniques to define orderings for *bags*. Even though the orderings appear somewhat awkward, we demonstrate effective algorithms to test whether two bags are comparable.

First of all, let us see why the naive approach would not work. Bags over a poset A are often represented as sets of pairs (a, n) where a is an element of A and n is the number of occurrences. Pairs could be ordered in the usual way: $(a, n) \leq (b, m)$ iff $a \leq b$ and $n \leq m$. While this ordering has many nice properties, it is counterintuitive from the practical point of view. Having a bag rather than a set means that each element of a bag represents an object and if there are many occurrences of some element, then at the moment certain objects are indistinguishable. For example, initially we might have a bag of three null values, representing our knowledge about three objects. Suppose this bag $\{\!|\bot, \bot, \bot|\!\}$ is later updated to $\{\!|a, b, c|\!\}$. We want to say that the latter is more informative than the former. But that is not in the above ordering because it requires that the three nulls be replaced by three identical objects; that is, $\{\!|a, a, a|\!\}$, $\{\!|b, b, b|\!\}$, or $\{\!|c, c, c|\!\}$. Each of them is more informative than $\{\!|\bot, \bot, \bot|\!\}$ but $\{\!|a, b, c|\!\}$ is unfortunately not!

Mathematical aspects of partial information represented by bags were studied by Vickers [174]. He defined the concept of refinements which, among other instances, includes both the ordering that we shall propose shortly and the ordering that we have just seen. Therefore, his approach is too general to be adopted here.

To extend the update idea to bags, recall again that each element of a bag represents an object and if there are many occurrences of some element, then at the moment certain objects are indistinguishable. This justifies the following definition. We say that a bag B_2 is more informative than a bag B_1 if B_2 can be obtained from B_1 by a sequence of updates of the following form: (1) an element a is removed from B_1 and is replaced by an element b such that b is more informative than a, and under OWA in addition (2) an element b is added to B_1.

Formally, let $\langle A, \leq \rangle$ be a partially ordered set. Let $\mathbb{P}_b(A)$ be the set of all finite bags whose elements are in A. Then we define the following updates for elements of $\mathbb{P}_b(A)$. Under both CWA and OWA we have

$$B \overset{\text{CWA}}{\rightsquigarrow} (B\,monus\{\!|a|\!\}) \uplus \{\!|b|\!\} \qquad \text{and} \qquad B \overset{\text{OWA}}{\rightsquigarrow} (B\,monus\{\!|a|\!\}) \uplus \{\!|b|\!\} \qquad \text{where } a \in B.$$

In addition, under OWA we add a new update

$$B \overset{\text{OWA}}{\rightsquigarrow} B \uplus \{\!|b|\!\}$$

As usual, by $\trianglelefteq^{\text{CWA}}$ and $\trianglelefteq^{\text{OWA}}$ we denote the transitive closure of $\overset{\text{CWA}}{\rightsquigarrow}$ and $\overset{\text{OWA}}{\rightsquigarrow}$ respectively. To describe these relations, let $\mathbb{N}^{\shortparallel}$ denote the totally unordered poset whose elements are natural numbers (the superscript is used to distinguish it from $\mathbb{N}$ which typically denotes natural numbers with the usual ordering). For a finite bag B and an injective map $\phi : B \to \mathbb{N}^{\shortparallel}$, which is sometimes called *labeling*, by $\phi(B)$ we denote the set $\{(b, \phi(b)) \mid b \in B\}$. In other words, ϕ assigns a unique label to each element of a bag. If $B \in \mathbb{P}_b(A)$, the ordering on pairs (b, n) where $b \in B$ and $n \in \mathbb{N}^{\shortparallel}$ is the usual pair ordering; that is, $(b, n) \leq (b', n')$ iff $b \leq b'$ and $n = n'$.

Proposition 4.7 *The binary relations $\trianglelefteq^{\text{CWA}}$ and $\trianglelefteq^{\text{OWA}}$ on bags are partial orders. Given two bags B_1 and B_2, $B_1 \trianglelefteq^{\text{CWA}} B_2$ ($B_1 \trianglelefteq^{\text{OWA}} B_2$) iff there exist labelings ϕ and ψ on B_1 and B_2 respectively such that $\phi(B_1) \sqsubseteq^{\natural} \psi(B_2)$ (respectively $\phi(B_1) \sqsubseteq^{\flat} \psi(B_2)$).*

Proof. We prove the statement about $\trianglelefteq^{\mathrm{OWA}}$; the statement about $\trianglelefteq^{\mathrm{CWA}}$ is proved similarly. We write $B_1 \preccurlyeq^\flat B_2$ if there exist ϕ and ψ such that $\phi(B_1) \sqsubseteq^\flat \psi(B_2)$. First demonstrate that $\preccurlyeq^\flat$ is a partial order. It is obviously reflexive.

To prove transitivity, let $B_1 \preccurlyeq^\flat B_2$ and $B_2 \preccurlyeq^\flat B_3$. That is, $\alpha(B_1) \sqsubseteq^\flat \beta(B_2)$ and $\phi(B_2) \sqsubseteq^\flat \psi(B_3)$. Let γ be a bijection on $\mathbb{N}$ such that $\gamma \circ \beta = \phi$. Define δ as $\gamma \circ \alpha$. Then for every $b \in B_1$ there is $b' \in B_2$ such that $b \leq b'$ and $\alpha(b) = \beta(b')$. Therefore, $\delta(b) = \phi(b')$ and there exists $b'' \in B_3$ such that $\psi(b'') = \phi(b')$ and $b'' \geq b'$. This shows $\delta(B_1) \sqsubseteq^\flat \psi(B_3)$ and hence $B_1 \preccurlyeq^\flat B_3$.

To show that $\preccurlyeq^\flat$ is anti-symmetric, let $B_1 \preccurlyeq^\flat B_2$ and $B_2 \preccurlyeq^\flat B_1$. As was shown above, there exist a, ϕ and ψ such that $\alpha(B_1) \sqsubseteq^\flat \phi(B_2) \sqsubseteq^\flat \psi(B_1)$. In particular, if we define $g : \alpha(B_1) \to \psi(B_1)$ by $g(b, n) = (b', n)$ where $\psi(b') = n$, it is easy to see that g is one-to-one, monotone and inflationary. Since B_1 is finite, it is the identity map. If $b'' \in B_2$ and $\phi(b'') = n$, then $b \leq b'' \leq b' = b$, so $b = b''$ where $\alpha(b) = \psi(b') = n$. Therefore, every element of B_1 is in B_2 and vice versa, i.e. $B_1 = B_2$. This shows that $\preccurlyeq^\flat$ is a partial order.

Since $B_1 \overset{\mathrm{OWA}}{\leadsto} B_2$ implies $B_1 \preccurlyeq^\flat B_2$, we conclude $\trianglelefteq^{\mathrm{OWA}} \subseteq \preccurlyeq^\flat$. Conversely, if $B_1 \preccurlyeq^\flat B_2$, i.e. $\phi(B_1) \sqsubseteq^\flat \psi(B_2)$, then, according to 4.3, $\psi(B_2)$ can be obtained from $\phi(B_1)$ by a sequence of $\overset{\mathrm{OWA}}{\longmapsto}$ updates which, if we drop indices, are translated into $\overset{\mathrm{OWA}}{\leadsto}$ updates on bags. Therefore, $B_1 \trianglelefteq^{\mathrm{OWA}} B_2$, which proves $\trianglelefteq^{\mathrm{OWA}} = \preccurlyeq^\flat$. □

The Hoare ordering $\sqsubseteq^\flat$ of sets can be effectively verified. Indeed, if two sets are given, there is an $O(n^2)$ time complexity algorithm to check if they are comparable. The description of $\trianglelefteq^{\mathrm{OWA}}$ given above seems to be somewhat awkward algorithmically. However, it is not much harder to test for.

Proposition 4.8 *There exists an $O(n^{5/2})$ time complexity algorithm that, given two bags B_1 and B_2 in $\mathbb{P}_b(A)$, returns* true *if $B_1 \trianglelefteq^{\mathrm{OWA}} B_2$ ($B_1 \trianglelefteq^{\mathrm{OWA}} B_2$) and* false *otherwise.*

Proof. The proof is almost the same for both $\trianglelefteq^{\mathrm{OWA}}$ and $\trianglelefteq^{\mathrm{OWA}}$. Given B_1 and B_2, consider two labelings ϕ and ψ on B_1 and B_2. Assume without loss of generality that the codomains of ϕ and ψ are disjoint. Define a bipartite graph $G = \langle V, E \rangle$ by $V := \phi(B_1) \cup \psi(B_2)$ and $E := \{((b, n), (b', n)) \mid (b, n) \in \phi(B_1), (b', n') \in \psi(B_2), b \leq b'\}$. It can be easily concluded from proposition 4.7 that $B_1 \trianglelefteq^{\mathrm{OWA}} B_2$ iff there is a matching in G that contains all $\phi(B_1)$. In other words, $B_1 \trianglelefteq^{\mathrm{OWA}} B_2$ iff the cardinality of the maximal matching in G is that of B_1. The proposition now follows from the facts that all maximal matching in G have the same cardinality (as bases of a matroid) and that the Hopcroft-Karp algorithm finds a maximal matching in $O(n^{5/2})$ where n is the cardinality of V (see [75]). □

There is a big difference between orders on sets and bags. While $X \sqsubseteq^\flat Y$ does not say anything about cardinality of X and Y, $B_1 \trianglelefteq^{\mathrm{OWA}} B_2$ implies that the cardinality of B_1 is less than or equal to the cardinality of B_2. This reflects our point of view that having a bag rather than a set means that each element of a bag represents a distinct object. Therefore, the cardinality can not be reduced in the process of obtaining more information. In particular, in the set case the Hoare ordering can be obtained as the transitive closure of the following binary relation: $X \mapsto (X - X') \cup \{x\}$ where $x \geq x'$ for all $x' \in X'$ and $X \mapsto X \cup \{x\}$. However, applying the same idea to bags amounts to the loss of information about the number of occurrences of each element in a bag. Precisely, let $\blacktriangleleft$ be defined as the transitive closure of $\twoheadrightarrow$, where $B_1 \twoheadrightarrow (B_1 monus B_1') \uplus \{\!|b|\!\}$, $b \geq b'$ for any $b' \in B_1'$, and $B_1 \twoheadrightarrow B_1 \uplus \{\!|b|\!\}$. It can be easily shown that $B_1 \blacktriangleleft B_2$ iff $unique(B_1) \sqsubseteq^\flat unique(B_2)$. And, in our opinion, this is not the right ordering on bags as it loses information about duplicates.

It can also be shown easily that, unlike $\sqsubseteq^\flat$ and $\sqsubseteq^\sharp$, the orderings $\trianglelefteq^{\mathrm{OWA}}$ and $\trianglelefteq^{\mathrm{CWA}}$ may not have least upper or greatest lower bounds and may fail to take bounded complete posets into bounded complete posets. The reader is invited to find simple counterexamples.

4.1.2 Semantics of collections

Recall that in section 3.1 the semantics of a database object d which is an element of an ordered set A was defined as the set of all elements of A that it can possibly denote, that is, the set of all elements in A that are greater than or equal to d:

$$[\![d]\!] = \uparrow d = \{d' \in A \mid d' \geq d\}$$

Following this definition and the results of the previous section, we can define the semantics of sets under OWA and CWA. Assume that elements of sets are taken from a partially ordered set A. Then we define the semantic functions $[\![\cdot]\!]^{\mathrm{OWA}}_{\mathrm{set}}, [\![\cdot]\!]^{\mathrm{OWA}}, [\![\cdot]\!]^{\mathrm{CWA}}_{\mathrm{set}}, [\![\cdot]\!]^{\mathrm{CWA}}$ where index set stands for the set semantics (as opposed to the antichain semantics for which we do not use an index), as follows:

$$[\![X]\!]^{\mathrm{OWA}}_{\mathrm{set}} = \{Y \in \mathbb{P}_{\mathrm{fin}}(A) \mid X \sqsubseteq^\flat Y\} \qquad [\![X]\!]^{\mathrm{OWA}} = \{Y \in \mathbb{A}_{\mathrm{fin}}(A) \mid X \sqsubseteq^\flat Y\}$$

$$[\![X]\!]^{\mathrm{CWA}}_{\mathrm{set}} = \{Y \in \mathbb{P}_{\mathrm{fin}}(A) \mid X \sqsubseteq^\natural Y\} \qquad [\![X]\!]^{\mathrm{CWA}} = \{Y \in \mathbb{A}_{\mathrm{fin}}(A) \mid X \sqsubseteq^\natural Y\}$$

As we mentioned in section 3.1, sometimes only subsets of maximal elements of A (if such elements exist) are taken into account. In this case we use index *max* instead of set in the semantic function.

In what follows, we shall mostly consider the open world assumption. Hence, if no superscript is used, it is assumed that we deal with the OWA sets or bags. That is, $[\![]\!]$ is the same as $[\![]\!]^{\mathrm{OWA}}$ and $[\![]\!]_{\mathrm{set}}$ is the same as $[\![]\!]^{\mathrm{OWA}}_{\mathrm{set}}$.

There are a number of useful properties of these semantic functions which we summarize in the following proposition. An easy proof is left to the reader.

Proposition 4.9 *1. If $X, Y \subseteq_{\rm fin} A$, then $[\![Y]\!]^{\rm OWA}_{\rm set} \subseteq [\![X]\!]^{\rm OWA}_{\rm set}$ iff $X \sqsubseteq^{\rm OWA} Y$ iff $X \sqsubseteq^{\flat} Y$.*

2. If $X, Y \in \mathbb{A}_{\rm fin}(A)$, then $[\![Y]\!] \subseteq [\![X]\!]$ iff $X \sqsubseteq^{\rm OWA}_a Y$ iff $X \sqsubseteq^{\flat} Y$.

3. If $X \subseteq_{\rm fin} A$, then $[\![X]\!] = [\![\max X]\!]$ and $[\![X]\!]^{\rm OWA}_{\rm set} = [\![\max X]\!]^{\rm OWA}_{\rm set}$.

4. If $X, Y \subseteq_{\rm fin} A$, then $[\![Y]\!]^{\rm CWA}_{\rm set} \subseteq [\![X]\!]^{\rm CWA}_{\rm set}$ iff $X \sqsubseteq^{\rm CWA} Y$ iff $X \sqsubseteq^{\natural} Y$.

5. If $X \subseteq_{\rm fin} A$, then $[\![X]\!]^{\rm CWA}_{\rm set} = [\![\max X \cup \min X]\!]^{\rm CWA}_{\rm set}$ and $[\![X]\!]^{\rm CWA} = [\![\max X \cup \min X]\!]^{\rm CWA}$. □

In chapter 1 we discussed Reiter's work [142] on the CWA databases. He defined a CWA answer to a query as a certain set of complete tuples. In our terminology, this corresponds to finding an answer to a query with respect to the $[\![]\!]^{\rm CWA}_{\max}$ semantic function. Reiter [142] proved that CWA query evaluation distributes over union and intersection, and that whenever a database is consistent with the negations of the facts stored in it, the OWA and the CWA query evaluation algorithms produce the same result. He also proved that the minimal CWA answers contain exactly one tuple.

The following proposition shows that analogs of these results hold in our setting. Note that to say that a database X is consistent with negation of any fact stored in it, is the same as to say that any $y \notin X$ is consistent with some $x \in X$. In other words, if every $z \in A$ lies under some $z_m \in A^{\max}$, then $X \sqsubseteq^{\sharp} A^{\max}$. Finally, a domain of n-ary relations with one kind of null is the product of n copies of an infinite flat domain. In view of this, the proposition below says that the results of [142] are preserved, at least in the spirit.

Proposition 4.10 *Let A be a poset such that each element is under an element of $A^{\max}$. Then 1) If A is a product of n copies of infinite flat domains and $Y \in [\![X_1 \cap X_2]\!]^{\rm CWA}_{\max}$, then $Y = Y_1 \cap Y_2$ where $Y_1 \in [\![X_1]\!]^{\rm CWA}_{\max}$ and $Y_2 \in [\![X_2]\!]^{\rm CWA}_{\max}$.*
2) For any poset A, $[\![X_1 \cup X_2]\!]^{\rm CWA}_{\max} = \{Y_1 \cup Y_2 \mid Y_1 \in [\![X_1]\!]^{\rm CWA}_{\max}, Y_2 \in [\![X_2]\!]^{\rm CWA}_{\max}\}$.
3) If $X \sqsubseteq^{\sharp} A^{\max}$, then $[\![X]\!]^{\rm CWA}_{\max} = [\![X]\!]^{\rm OWA}_{\max}$.
4) If X is bounded above in A, then a minimal nonempty $Y \in [\![X]\!]^{\rm CWA}_{\max}$ is a singleton. □

For or-sets the situation is different. Recall that or-sets can be treated at both structural and conceptual levels. At the structural level we just define $[\![X]\!]^{\rm or} = \{Y \in \mathbb{P}_{\rm fin}(A) \mid X \sqsubseteq^{\sharp} Y\}$ (or using $\mathbb{A}_{\rm fin}(A)$ if we need an antichain semantics.) The following proposition is immediate from the definitions.

Proposition 4.11 *1. If $X, Y \subseteq_{\rm fin} A$, then $[\![Y]\!]^{\rm or} \subseteq [\![X]\!]^{\rm or}$ iff $X \sqsubseteq^{\rm or} Y$ iff $X \sqsubseteq^{\sharp} Y$.*

2. If $X, Y \in \mathbb{A}_{\rm fin}(A)$, then $[\![Y]\!]^{\rm or} \subseteq [\![X]\!]^{\rm or}$ iff $X \sqsubseteq^{\rm or}_a Y$ iff $X \sqsubseteq^{\sharp} Y$.

3. If $X \subseteq_{\rm fin} A$, then $[\![X]\!]^{\rm or} = [\![\min X]\!]^{\rm or}$. □

Similar semantic functions can be defined for bags, depending on whether OWA or CWA is used. Unlike sets, bags are not subject to removal of redundancies as every entry in a bag represents a distinct object and nothing can be deleted.

Note that propositions 4.9 and 4.11 justify using maximal elements to remove redundancies from sets under OWA and using minimal elements to remove redundancies from or-sets. For sets under CWA, it is necessary to retain both minimal and maximal elements; the elements which are strictly in between can be removed as the fifth item in proposition 4.9 suggests.

The semantic functions above could also be used to define the semantic domains of *types*. For example, assume that we have the following type system

$$t ::= b \mid t \times t \mid \{t\}_{\mathrm{OWA}} \mid \{t\}_{\mathrm{CWA}} \mid \langle t \rangle$$

We now define the *structural semantics* $[\![]\!]_s$ that corresponds to the structural interpretation of or-sets.

Suppose that for each base type b its semantic domain $[\![b]\!]_s$ is given. We define the semantic domains of all types inductively. Suppose we want to deal with antichains. Then

- $[\![t \times s]\!]_s = [\![t]\!]_s \times [\![s]\!]_s$.
- $[\![\{t\}_{\mathrm{OWA}}]\!]_s = \langle \mathbb{A}_{\mathrm{fin}}([\![t]\!]_s), \sqsubseteq^{\flat} \rangle = \mathcal{P}^{\flat}([\![t]\!]_s)$.
- $[\![\{t\}_{\mathrm{CWA}}]\!]_s = \langle \mathbb{A}_{\mathrm{fin}}([\![t]\!]_s), \sqsubseteq^{\natural} \rangle$.
- $[\![\langle t \rangle]\!]_s = \langle \mathbb{A}_{\mathrm{fin}}([\![t]\!]_s), \sqsubseteq^{\sharp} \rangle = \mathcal{P}^{\sharp}([\![t]\!]_s)$.

The structural semantics of objects is defined inductively.

- For each base type b and an element x of this type, $[\![x]\!]_s = \uparrow x = \{x' \in [\![b]\!]_s \mid x' \geq x\}$.
- If $x = (x_1, x_2)$, then $[\![x]\!]_s = [\![x_1]\!]_s \times [\![x_2]\!]_s$.
- Let X be a CWA set of type $\{t\}_{\mathrm{CWA}}$, then $[\![X]\!]_s = [\![X]\!]^{\mathrm{CWA}}$. Similarly, for OWA sets, $[\![X]\!]_s = [\![X]\!]^{\mathrm{OWA}}$.
- Let $X = \langle x_1, \ldots, x_n \rangle$ be an or-set of type $\langle t \rangle$. Then $[\![X]\!]_s = [\![X]\!]^{\mathrm{or}}$.

Note that the last clauses in the definitions of type and object semantics say that we have defined the *structural semantics* of or-sets. That is, we viewed or-sets as collections and not as single elements they could represent. Our next goal is to define the *conceptual* semantics of or-sets.

Semantics of sets and or-sets

Our purpose here is to define a semantics to be used when or-sets are dealt with at the conceptual level. This semantic function takes database objects into finitely generated filters of ordered sets. For simplicity, assume that we have the following type system:

$$t ::= b \mid t \times t \mid \{t\} \mid \langle t \rangle$$

and that we are dealing with the open world assumption.

We shall denote the semantic function that deals with the conceptual representation of or-sets by $[\![]\!]_c$. We know that conceptually an or-set is one of its elements. That is, conceptually $X = \langle d_1, \ldots, d_n \rangle$ is one of d_i's. If d_i's themselves are partial descriptions, they may denote other elements. Hence, whatever the semantic function $[\![]\!]$ for the elements of X is, we have $[\![X]\!]_c = \bigcup_{x \in X} [\![x]\!]$.

If the semantic function $[\![]\!]$ satisfies the property that $[\![x]\!] \subseteq [\![y]\!]$ iff $y \leq x$, then $[\![x]\!] = \uparrow x$ and we obtain

$$[\![X]\!]_c = \bigcup_{x \in X} \uparrow x = \uparrow X$$

Hence, from the results of the previous section and the properties of the Smyth order, we conclude that in this particular case $[\![]\!]_c$ for or-sets satisfies all properties listed in proposition 4.11.

To define the conceptual semantics of types, we assume that a semantic domain $[\![b]\!]_c$ is given for each base type b. We now define semantic domains of arbitrary types as follows. Note that there are two possibilities for the semantics of the set type constructor, but the definition of the semantics of objects will work with both of them.

- $[\![t \times s]\!]_c = [\![t]\!]_c \times [\![s]\!]_c$.
- $[\![\{t\}]\!]_c = \langle \mathbb{A}_{\text{fin}}([\![t]\!]), \sqsubseteq^\flat \rangle = \mathcal{P}^\flat([\![t]\!]_c)$ or $[\![\{t\}]\!]_c = \langle \mathbb{P}_{\text{fin}}([\![t]\!]_c), \sqsubseteq^\flat \rangle$.
- $[\![\langle t \rangle]\!]_c = [\![t]\!]_c$.

The last clause corresponds to the fact that conceptually an or-set is just one of its elements. Semantics of each object is now going to be a finitely generated filter $F = \uparrow\{f_1, \ldots, f_n\} = \uparrow f_1 \cup \ldots \cup \uparrow f_n$. Again, we define it inductively.

- For each base type b and an element x of this type, $[\![x]\!]_c = \uparrow x = \{x' \in [\![b]\!]_c \mid x' \geq x\}$.
- If $x = (x_1, x_2)$, then $[\![x]\!]_c = [\![x_1]\!]_c \times [\![x_2]\!]_c$.

- Let $X = \{x_1, \ldots, x_n\}$ be a set of type $\{t\}$. Then $[\![X]\!]_c = \{Y \mid \forall i = 1, \ldots, n : Y \cap [\![x_i]\!]_c \neq \emptyset\}$. Here Y is taken from $\mathbb{P}_{\text{fin}}([\![t]\!]_c)$ or $\mathbb{A}_{\text{fin}}([\![t]\!]_c)$ depending on the definition of the semantics of types.

- Let $X = \langle x_1, \ldots, x_n \rangle$ be an or-set of type $\langle t \rangle$. Then $[\![X]\!]_c = [\![x_1]\!]_c \cup \ldots \cup [\![x_n]\!]_c$.

Before we prove that this semantic function possesses the desired properties, let us make a few observation. First, the definition of the semantics of or-sets agrees with the definition of $[\![]\!]^{\text{or}}$ given above. Second, to understand the semantics of pairs and sets, consider tow simple examples. Let $x_1 = \langle 1, 2 \rangle$, $x_2 = \langle 3, 4 \rangle$. Assume that there is no ordering involved. The semantics of x_1 is then a set $\{1, 2\}$ and the semantics of x_2 is $\{3, 4\}$. Therefore, $[\![(x_1, x_2)]\!]_c = \{(1, 3), (1, 4), (2, 3), (2, 4)\}$. Now consider (x_1, x_2). It is a pair whose first component is 1 or 2 and whose second component is 3 or 4. Hence, it is one of the following pairs: $(1, 3), (1, 4), (2, 3), (2, 4)$. And this is exactly what the semantic function $[\![]\!]_c$ tells us.

For semantics of sets, consider $X = \{x_1, x_2\} = \{\langle 1, 2 \rangle, \langle 3, 4 \rangle\}$. It is is a set that has at least two elements: one is 1 or 2, and the other is 3 or 4. Hence, it must contain one of the following sets (since we believe in OWA): $\{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}$. Now look at $[\![X]\!]_c$. A set Y belongs to $[\![X]\!]_c$ if $Y \cap [\![\langle 1, 2 \rangle]\!]_c = Y \cap \{1, 2\} \neq \emptyset$ and $Y \cap [\![\langle 3, 4 \rangle]\!]_c = Y \cap \{3, 4\} \neq \emptyset$ which happens if and only if Y contains one of the four sets above. This justifies our definition of the conceptual semantics of sets.

Now we can prove the following.

Proposition 4.12 *For every object x of type t, $[\![x]\!]_c$ is a finitely generated filter in $[\![t]\!]_c$. Furthermore, if x and y are of type t and $x \leq y$ in $[\![t]\!]_s$, then $[\![y]\!]_c \subseteq [\![x]\!]_c$.*

Proof. Prove the first statement by induction. For objects of base types it is given by the definition. For pairs, it is easy to show that if x_1 and x_2 are finitely generated filters in $[\![t_1]\!]_c$ and $[\![t_2]\!]_c$ respectively, then $[\![x]\!]_c$ is a finitely generated filter in $[\![t_1 \times t_2]\!]_c$. For sets, let $X = \{x_1, \ldots, x_n\}$ be a set of type $\{t\}$. Let $[\![x_i]\!]_c = \uparrow\{f_1^i, \ldots, f_{n_i}^i\}$. Let $\mathcal{G}$ be the set of maps $g : \{1, \ldots, n\} \to \mathbb{N}$ such that $1 \leq g(i) \leq n_i$ for all i. Define $\mathcal{G}(X) = \min_{\sqsubseteq^\flat}\{\{f_{g(i)}^i \mid i = 1, \ldots, n\} \mid g \in \mathcal{G}\}$. Then $Y \in [\![X]\!]_c$ iff there exists $Y' \in \mathcal{G}(X)$ such that $Y' \sqsubseteq^\flat Y$. Therefore, $[\![X]\!]_c = \uparrow\mathcal{G}(X)$. This shows that $[\![X]\!]_c$ is a finitely generated filter. For arbitrary sets, the proof proceeds similarly but we do not have to take min. The second result will be proved later (see theorem 5.17 in chapter 5.) $\square$

From the properties of the structural and conceptual semantics, we obtain

Corollary 4.13 *If x and y are objects of the same type, then $[\![x]\!]_s = [\![y]\!]_s$ implies $[\![x]\!]_c = [\![y]\!]_c$.* $\square$

The converse is not true: $\langle\langle 1,2\rangle,\langle 3\rangle\rangle$ and $\langle\langle 1\rangle,\langle 2\rangle,\langle 3\rangle\rangle$ are structurally different objects of type $\langle\langle int\rangle\rangle$, but $[\![\langle\langle 1,2\rangle,\langle 3\rangle\rangle]\!]_c = [\![\langle\langle 1\rangle,\langle 2\rangle,\langle 3\rangle\rangle]\!]_c = \{1,2,3\}$.

The importance of the conceptual semantics will be seen in the next chapter when we show that normalization of or-objects does not change the meaning.

Relationship between CWA sets, OWA sets and or-sets

There is a naturally arising question: do we really need all three kinds of collections – OWA sets, CWA sets and or-sets? Can not we just represent some of them using the others? The answer to this question is that we do need all three kinds of collections and no such representations exist. First, let us see what could be a representation of, say, OWA sets with or-sets. It could be a procedure that, given a poset A and $X \in \mathbb{A}_{\text{fin}}(A)$, calculates $Y \in \mathbb{A}_{\text{fin}}(A)$ such that $Z \in [\![X]\!]$ iff $Z \in [\![Y]\!]^{\text{or}}$. The following proposition tells us that it is impossible to do so.

Proposition 4.14 *For every poset A which is not a chain, there exists $X \in \mathbb{A}_{\text{fin}}(A)$ such that for no $Y \in \mathbb{A}_{\text{fin}}(A)$ the following holds: 1) $[\![X]\!] = [\![Y]\!]^{\text{or}}$; 2) $[\![X]\!]^{\text{or}} = [\![Y]\!]$; 3) $[\![X]\!] = [\![Y]\!]^{\text{CWA}}_{\text{set}}$; 4) $[\![X]\!]^{\text{CWA}}_{\text{set}} = [\![Y]\!]$; 5) $[\![X]\!]^{\text{or}} = [\![Y]\!]^{\text{CWA}}_{\text{set}}$; 6) $[\![X]\!]^{\text{CWA}}_{\text{set}} = [\![Y]\!]^{\text{or}}$.*

Proof. 1) Assume A has two incomparable elements x and y and let $X = \{x\}$. Assume Y is such that $[\![\{x\}]\!] = [\![Y]\!]^{\text{or}}$. Then $\{x\} \sqsubseteq^{\flat} \{x,y\} \sqsubseteq^{\sharp} \{y\}$ and hence $\{y\} \in [\![\{x\}]\!]$, contradiction. For 2), consider the same poset by take X to be $\{x,y\}$. For 3), take the same poset and take $X = \{x\}$. Assume there is Y such that $[\![\{x\}]\!] = [\![Y]\!]^{\text{or}}$. Then $\{x,y\} \in [\![\{x\}]\!]$ and hence $Y \sqsubseteq^{\sharp} \{x,y\}$. We have $\{x\} \sqsubseteq^{\flat} Y$, so there is an element $z \leq y$ such that $x \leq z$, contradiction. The proof of 4) is similar. We invite the reader to find similar easy proofs for 5) and 6). $\square$

4.1.3 Formal models of approximations

In this section we re-examine the approximation constructs such as sandwiches, mixes and snacks introduced in chapter 1. We do it by applying the idea of representing database objects with partial information as elements of certain ordered sets, and then getting all approximation constructs as families of antichains in those posets.

Recall the definition of a *sandwich*. It is given by an upper approximation U and a lower approximation L which satisfy the following consistency condition: for every $u \in U$ there is $l \in L$ such that u and l are consistent. The notion of consistency here is the same as consistency in posets. If there are two records, then they are consistent if there is a record that is above both of them in the ordering. For example, $\begin{array}{|c|c|c|}\hline 1 & 2 & \bot \\ \hline\end{array}$ and $\begin{array}{|c|c|c|}\hline \bot & 2 & 3 \\ \hline\end{array}$ are consistent as they have a common upper bound $\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline\end{array}$, but $\begin{array}{|c|c|c|}\hline 1 & 2 & \bot \\ \hline\end{array}$ and $\begin{array}{|c|c|c|}\hline \bot & 4 & 3 \\ \hline\end{array}$ are not consistent as there are no common upper bounds. Recall that we use the notation $x \uparrow y$ to denote the fact that x and y are consistent. Now we can give a formal definition of sandwiches.

Definition 4.1 *Given a poset $\langle A, \leq \rangle$, a* sandwich *over A is a pair of finite antichains (U, L) satisfying the following consistency condition:*

$$\forall l \in L \; \exists u \in U : \; u \uparrow l$$

U is usually referred to as the upper approximation and L as the lower approximation. The family of all sandwiches over A is denoted by $\mathcal{P}^{\forall\wedge}(A)$ (the reason for this notation will be seen shortly).

For example,

Name	Salary	Room
John	15K	⊥
Ann	17K	⊥
Mary	12K	⊥
Michael	14K	⊥

and

Name	Salary	Room
John	⊥	076
Michael	⊥	320

form a sandwich. First, each relation can be considered as a subset of $\mathcal{V}_\perp \times \mathcal{V}_\perp \times \mathcal{V}_\perp$ as explained in section 3.1. Moreover, since

$$\boxed{\text{John} \mid \perp \mid 076} \uparrow \boxed{\text{John} \mid 15\text{K} \mid \perp} \quad \text{and} \quad \boxed{\text{Michael} \mid \perp \mid 320} \uparrow \boxed{\text{Michael} \mid 14\text{K} \mid \perp}$$

the pair satisfies the consistency condition and hence forms a sandwich in $\mathcal{V}_\perp \times \mathcal{V}_\perp \times \mathcal{V}_\perp$, where the first relation is the upper approximation and the second relation is the lower approximation.

The consistency condition for sandwiches can be equivalently stated in the following way. A pair of finite antichains (U, L) is a sandwich if there exists a set W such that $U \sqsubseteq^\sharp W$ and $L \sqsubseteq^\flat W$. Observe that $U = \emptyset$ implies $L = \emptyset$.

Recall that in chapter 1 we used the assumption that the Name field is a key to infer additional information about the relations shown above. It led us to the following relations:

Name	Salary	Room
John	15K	076
Ann	17K	⊥
Mary	12K	⊥
Michael	14K	320

and

Name	Salary	Room
John	15K	076
Michael	14K	320

The difference is that now for each record in the second relation there is a record in the first relation that is less or equally informative. This is the definition of mixes.

Definition 4.2 *Given a poset $\langle A, \leq \rangle$, a* mix *over A is a pair of finite antichains (U, L) satisfying the following consistency condition:*

$$\forall l \in L\ \exists u \in U:\ u \leq l$$

U is usually referred to as the upper approximation and L as the lower approximation. The family of all mixes over A is denoted by $\mathcal{P}^{\forall}(A)$ (the reason for this notation will be seen shortly).

Observe that the consistency condition for mixes can be also stated as $U \sqsubseteq^{\sharp} L$. Again, $U = \emptyset$ implies $L = \emptyset$.

Now recall the definition of scones. In a scone, the lower approximation is a family of sets (relations), as shown below.

Name	Salary	Room
John	15K	$\perp$
Ann	17K	$\perp$
Michael	14K	$\perp$

Name	Salary	Room
John	$\perp$	076
Jim	$\perp$	$\perp$

Name	Salary	Room
Michael	$\perp$	320

The consistency condition that relates the upper and the lower approximations now says that for every set in the lower approximation, there exists an element in that set that is consistent with an element of the upper approximation. Therefore, we can formalize the notion of a scone as follows.

Definition 4.3 *Given a poset $\langle A, \leq \rangle$, a* scone *over A is a pair $(U, \mathcal{L})$ where U is a finite antichain, and $\mathcal{L} = \{L_1, \ldots, L_k\}$ is a family of finite nonempty antichains which is itself an antichain with respect to $\sqsubseteq^{\sharp}$. That is, $L_i \not\sqsubseteq^{\sharp} L_j$ if $i \neq j$. In addition, a scone is required to satisfy the consistency condition:*

$$\forall L \in \mathcal{L}\ \exists l \in L\ \exists u \in u:\ u \uparrow l$$

We refer to U as the upper approximation and to $\mathcal{L}$ as the lower approximation. The family of all scones over A is denoted by $\mathcal{P}^{\exists\wedge}(A)$.

Note that the consistency condition for scones can be reformulated as $\uparrow L \cap \uparrow U \neq \emptyset$ for any $L \in \mathcal{L}$.

The last construction that we have seen in chapter 1 was a snack. Snacks are obtained from scones in the same way as mixes are obtained from sandwiches: by using the assumption about keys, additional information is inferred. Moreover, the record for Jim disappears as it is now inferred that Jim is not a TA. In our example, assuming that Name is a key, this yields:

Name	Salary	Room
John	15K	076
Ann	17K	$\perp$
Michael	14K	320

Name	Salary	Room
John	15K	076

Name	Salary	Room
Michael	14K	320

Thus, now we know that every record in every relation in the lower approximation is at least as informative as some record in the upper approximation. This leads us to the following definition.

Definition 4.4 *Given a poset $\langle A, \leq \rangle$, a* snack *over A is a pair $(U, \mathcal{L})$ where U is a finite antichain, and $\mathcal{L} = \{L_1, \ldots, L_k\}$ is a family of finite nonempty antichains which is itself an antichain with respect to $\sqsubseteq^\sharp$. That is, $L_i \not\sqsubseteq^\sharp L_j$ if $i \neq j$. In addition, a snack is required to satisfy the consistency condition:*

$$\forall L \in \mathcal{L}\ \forall l \in L\ \exists u \in u :\ u \leq l$$

We refer to U as the upper approximation and to $\mathcal{L}$ as the lower approximation. The family of all snacks over A is denoted by $\mathbb{P}^\forall(A)$.

The consistency condition for snacks can be equivalently stated as $U \sqsubseteq^\sharp L$ for any $L \in \mathcal{L}$.

Now let us look at these constructs again. There are three main parameters that may vary and give rise to new constructs.

1. The lower approximation is either a set or a set of sets.
2. The consistency condition is of form

   $$\mathbf{Q} l \in L \quad \exists u \in U \quad C(u,l) \qquad \text{for simple lower approximations and}$$

   $$\forall L \in \mathcal{L} \quad \mathbf{Q} l \in L \quad \exists u \in U \quad C(u,l) \qquad \text{for multi-set lower approximations,}$$

   where $\mathbf{Q}$ is a quantifier (either $\forall$ or $\exists$) and $C(u,l)$ is a condition that relates u and l.
3. The condition $C(u,l)$ is either $u \leq l$ or $u \Uparrow l$.

Therefore, we have eight constructions since each of the parameters that may vary – the structure of the lower approximation, the quantifier $\mathbf{Q}$ and the condition $C(u,l)$ – has two possible values. For constructs that have a single set lower approximation we use notation $\mathcal{P}$ and for the constructs with multi-set lower approximation we use $\boldsymbol{\mathcal{P}}$. The rest is indicated in the superscript which consists of one or two symbols. The first is always a quantifier and indicates whether $\forall$ or $\exists$ is used as $\mathbf{Q}$. The second is omitted if the condition is $u \leq l$, and it is $\wedge$ if the condition is $u \Uparrow l$ (to indicate that there is an element above u and l). Moreover, we have seen a need for constructs with no consistency condition, in order to deal with inconsistencies in independent databases. For such constructs we shall use just one superscript $\emptyset$.

Summing up, we have ten possible constructs: $\mathcal{P}^{\forall}, \boldsymbol{\mathcal{P}}^{\forall}, \mathcal{P}^{\forall\wedge}, \boldsymbol{\mathcal{P}}^{\forall\wedge}, \mathcal{P}^{\exists}, \boldsymbol{\mathcal{P}}^{\exists}, \mathcal{P}^{\exists\wedge}, \boldsymbol{\mathcal{P}}^{\exists\wedge}, \mathcal{P}^{\emptyset}, \boldsymbol{\mathcal{P}}^{\emptyset}$. Some of them we have seen already: $\mathcal{P}^{\forall}(A)$ is the family of mixes over A, $\mathcal{P}^{\forall\wedge}(A)$ is the family of sandwiches over A, $\boldsymbol{\mathcal{P}}^{\forall}(A)$ is the family of snacks over A and $\boldsymbol{\mathcal{P}}^{\exists\wedge}(A)$ is the family of scones over A. This is summarized in the table below.

	type of consistency condition (quantifier–condition)				
L-part	$\forall \quad u \leq l$	$\forall \quad u \Uparrow l$	$\exists \quad u \leq l$	$\exists \quad u \Uparrow l$	no condition
one set	$\mathcal{P}^{\forall}$ (mix)	$\mathcal{P}^{\forall\wedge}$ (sandwich)	$\mathcal{P}^{\exists}$	$\mathcal{P}^{\exists\wedge}$	$\mathcal{P}^{\emptyset}$
family of sets	$\boldsymbol{\mathcal{P}}^{\forall}$ (snack)	$\boldsymbol{\mathcal{P}}^{\forall\wedge}$	$\boldsymbol{\mathcal{P}}^{\exists}$	$\boldsymbol{\mathcal{P}}^{\exists\wedge}$ (scone)	$\boldsymbol{\mathcal{P}}^{\emptyset}$

Our next goal is to define orders on all approximation constructs and their semantics.

Ordering approximations

Our approach to ordering approximations is the same as the one we used for ordering collections. We define elementary updates that add information and then define orderings as transitive closure of those updates. It is important to mention that we use the open world assumption for the lower approximation as it describes the approximated collection only partially.

Let us first introduce the rules for constructions with one-set lower approximation (like mixes and sandwiches). The idea behind these rules is that there are three ways to make a pair more informative: to obtain additional information about elements already in one of the sets; to make the lower approximation more informative by adding new elements and to make the upper approximation more informative by reducing the number of possibilities, i.e. by removing some elements. This is formalized as follows:

1. $(U, L) \mapsto (U - u, L)$.
2. $(U, L) \mapsto (\min((U - u) \cup V), L)$ where $v \geq u$ for all $v \in V$.
3. $(U, L) \mapsto (U, \max((L - l) \cup L'))$ where $l \leq l'$ for all $l' \in L'$.
4. $(U, L) \mapsto (U, \max(L \cup l))$.

Similarly, updates 1 and 2 will work for approximation constructs with the multi-set lower approximation. However, we need new rules for the lower approximation. Recall that in a multi-set lower approximation each set contributes at least one element into the result (an element of the semantics) and elements of that set list possible choices of elements to be included in the results. Hence, adding new sets into $\mathcal{L}$ as well as deleting elements from $L \in \mathcal{L}$ add information. Now we can formalize updates as follows. We use symbol $\max^\sharp$ to denote maximum with respect to $\sqsubseteq^\sharp$.

1. $(U, \mathcal{L}) \leadsto (U - u, \mathcal{L})$.
2. $(U, \mathcal{L}) \leadsto (\min((U - u) \cup V), \mathcal{L})$ where $v \geq u$ for all $v \in V$.
3. $(U, \mathcal{L}) \leadsto (U, \max^\sharp(\mathcal{L} \cup L))$.
4. $(U, \mathcal{L}) \leadsto (U, \max^\sharp((\mathcal{L} - L) \cup (L - l)))$ if $L - l \neq \emptyset$.
5. $(U, \mathcal{L}) \leadsto (U, \max^\sharp((\mathcal{L} - L) \cup \min((L - l) \cup L')))$ where $l \leq l'$ for all $l' \in L'$.

We now define two orderings, called *the Buneman orderings*, see [33, 66]. For pairs (U, L) and (U', L'), let

$$(U, L) \sqsubseteq^{\mathbb{B}} (U', L') \quad \text{iff} \quad U \sqsubseteq^\sharp U' \text{ and } L \sqsubseteq^\flat L'$$

In other words, $\sqsubseteq^{\mathbb{B}} = \sqsubseteq^\sharp \times \sqsubseteq^\flat$. For pairs $(U, \mathcal{L})$ and $(U', \mathcal{L}')$, let

$$(U, \mathcal{L}) \sqsubseteq^{\mathbb{B}}_f (U', \mathcal{L}') \quad \text{iff} \quad U \sqsubseteq^\sharp U' \text{ and } \forall L \in \mathcal{L}\ \exists L' \in \mathcal{L}' :\ L \sqsubseteq^\sharp L'$$

In other words, $\sqsubseteq^{\mathbb{B}}_f = \sqsubseteq^\sharp \times (\sqsubseteq^\sharp)^\flat$. The index f is ised in $\sqsubseteq^{\mathbb{B}}_f$ to indicate that the ordering deals with families of sets in the lower approximations, whereas $\sqsubseteq^{\mathbb{B}}$ deals with simple lower approximations.

Our main claim about orderings on approximations is the following.

The approximations must be ordered by the Buneman orderings.

We justify it by proving the following theorem. Recall that $*$ over an arrow is used as a notation for the transitive-reflexive closure.

Theorem 4.15 *a) Let (U, L) and (V, M) be two approximations with one-set lower approximation (e.g. mixes, sandwiches etc.) Then $(U, L) \overset{*}{\mapsto} (V, M)$ iff $(U, L) \sqsubseteq^{\mathbb{B}} (V, M)$.*
b) Let $(U, \mathcal{L})$ and and $(V, \mathcal{M})$ be two approximations with multi-set lower approximation (e.g. snacks, scones etc.) Then $(U, \mathcal{L}) \overset{*}{\leadsto} (V, \mathcal{M})$ iff $(U, \mathcal{L}) \sqsubseteq^{\mathbb{B}}_f (V, \mathcal{M})$.

Proof. We prove part b) here; the proof of part a) is similar (and in fact easier). First, observe that whenever $\mathcal{S}_1 \leadsto \mathcal{S}_2$ and both $\mathcal{S}_1$ and $\mathcal{S}_2$ are approximation constructs with the multi-set lower approximation, then $\mathcal{S}_1 \sqsubseteq_f^{\mathbb{B}} \mathcal{S}_2$. Hence, the transitive closure of $\rightsquigarrow$ is included in $\sqsubseteq_f^{\mathbb{B}}$. To prove the converse, let $(U, \mathcal{L}) \sqsubseteq_f^{\mathbb{B}} (V, \mathcal{M})$. Since $\mathcal{L}(\sqsubseteq^\sharp)^\flat\mathcal{M}$, by theorem 4.3 there is a sequence $\mathcal{L} \xmapsto{\text{OWA}}_a \mathcal{L}_1 \xmapsto{\text{OWA}}_a \ldots \xmapsto{\text{OWA}}_a \mathcal{L}_k \xmapsto{\text{OWA}}_a \mathcal{M}$ such that $\mathcal{L}_i \subseteq \mathcal{L} \cup \mathcal{M}$. In particular, each $(U, \mathcal{L}_i)$ is a snack (scone) if $(U, \mathcal{L})$ and $(V, \mathcal{M})$ are snacks (scones). For tranformation $\mathcal{L}_i \xmapsto{\text{OWA}}_a \mathcal{L}_{i+1}$ there are two cases.

Case 1. $\mathcal{L}_{i+1} = \max^\sharp(\mathcal{L} \cup L')$. In this case $(U, \mathcal{L}_i) \rightsquigarrow (U, \mathcal{L}_{i+1})$ follows from the definitions.

Case 2. $\mathcal{L}_{i+1} = \max^\sharp((\mathcal{L}_i - L) \cup L')$ where $L \sqsubseteq^\sharp L'$. Then, by theorem 4.5, there is a sequence $L \xmapsto{\text{or}}_a L_1 \xmapsto{\text{or}}_a L_2 \xmapsto{\text{or}}_a \ldots \xmapsto{\text{or}}_a L_p \xmapsto{\text{or}}_a L'$ such that each L_j is a subset of $L \cup L'$. In particular, this shows that $(U, \max^\sharp((\mathcal{L}_i - L_j) \cup L_{j+1}))$ is a snack or a scone respectively. Now there are two subcases. In the first subcase, $L_{j+1} = \min(L_j - l)$ and then $(U, \max^\sharp((\mathcal{L}_i - L) \cup L_j)) \rightsquigarrow (U, \max^\sharp((\mathcal{L}_i - L) \cup L_{j+1}))$ follows from the definition. Similarly, it holds for the second subcase when $L_{j+1} = \min((L_j - l) \cup L')$.

Therefore, $(U, \mathcal{L}_i) \overset{*}{\rightsquigarrow} (U, \mathcal{L}_{i+1})$ which implies $(U, \mathcal{L}) \overset{*}{\rightsquigarrow} (U, \mathcal{M})$. Now from theorem 4.5 we have $U \xmapsto{\text{or}}_a U_1 \xmapsto{\text{or}}_a U_2 \xmapsto{\text{or}}_a \ldots \xmapsto{\text{or}}_a U_r \xmapsto{\text{or}}_a V$ such that each U_i is a subset of $U \cup V$. Since $\uparrow V \subseteq \uparrow U$, this implies consistency condition for each $(U_i, \mathcal{M})$. Each $U_i \rightsquigarrow U_{i+1}$ is either $U_i \rightsquigarrow U_i - u$ or $U_i \to \min((U_i - u) \cup U')$ where $u' \geq u$ for all $u' \in U'$. In both cases, $(U_i, \mathcal{M}) \rightsquigarrow (U_{i+1}, \mathcal{M})$. Therefore, $(U, \mathcal{M}) \overset{*}{\rightsquigarrow} (V, \mathcal{M})$ which finishes the proof of $(U, \mathcal{L}) \overset{*}{\rightsquigarrow} (V, \mathcal{M})$. The result for mixes and sandwiches is easily proved along the same lines. $\square$

Thus, when we consider approximation constructs $\mathcal{P}^i(A)$ and $\mathbb{P}(A)$, where $i \in \{\forall, \exists, \forall\wedge, \exists\wedge, \emptyset\}$, we assume that they are ordered by $\sqsubseteq^{\mathbb{B}}$ and $\sqsubseteq_f^{\mathbb{B}}$ respectively.

Semantics of approximations

To understand the semantics of the approximation constructs, recall the example of querying independent databases from chapter 1. We used two relations, Employees and CS1, to approximate the set of teaching assistants. We assumed that a set TA is approximated by Employees and CS1 if every record in CS1 represents (is less than) a record in TA and every record in TA is represented (is greater than) by a record in Employees. In other words, CS1 $\sqsubseteq^\flat$ TA and TA $\sqsubseteq^\sharp$ Employees.

For scones and snacks, where CS1 was subdivided into a family of relations CS1_i, we assumed that at least one element from each CS1_i represents an element in TA. That is, TA $\sqsubseteq^\sharp$ Employees, and for all i, there exists an element in CS1_i that represents an element of TA. In other words, $\uparrow\text{CS1}_i \cap \uparrow\text{TA} \neq \emptyset$.

To formalize it, we introduce two semantic functions. For constructions with one-element lower approximations (like mixes and sandwiches) we have

$$[\![(U, L)]\!] = \{X \in \mathbb{P}_{\text{fin}}(A) \mid U \sqsubseteq^\sharp X \text{ and } \ L \sqsubseteq^\flat X\}$$

$$[\![(U, L)]\!]_{\max} = \{X \in \mathbb{P}_{\text{fin}}(A^{\max}) \mid U \sqsubseteq^\sharp X \text{ and } \ L \sqsubseteq^\flat X\}$$

For constructions with multi-element lower approximations (like snacks and scones) we have

$$[\![(U, \mathcal{L})]\!] = \{X \in \mathbb{P}_{\text{fin}}(A) \mid U \sqsubseteq^\sharp X \text{ and } \ \forall i : \uparrow L_i \cap X \neq \emptyset\}$$

$$[\![(U, \mathcal{L})]\!]_{\max} = \{X \in \mathbb{P}_{\text{fin}}(A^{\max}) \mid U \sqsubseteq^\sharp X \text{ and } \ \forall i : \uparrow L_i \cap X \neq \emptyset\}$$

Note that for both mixes and sandwiches, it is guaranteed that there semantics is not empty. (Of course for $[\![]\!]_{\max}$ we have to require that every $x \in A$ be bounded above by $x_m \in A^{\max}$.) The same is true for any $S \in \mathcal{P}^{\exists\wedge}(A)$. However, it is easy to see that for $S \in \mathcal{P}^{\emptyset}(A)$, $[\![S]\!] \neq \emptyset$ iff $S \in \mathcal{P}^{\exists\wedge}(A)$.

The semantics of mixes and sandwiches has been studied in Buneman et al. [33] and Gunter [66]. Here we concentrate on the constructs with the multi-element L-part.

Let A be a three element chain $a < b < c$ and $\mathcal{S}_1 = (a, b)$ and $\mathcal{S}_2 = (a, c)$ two snacks over A. Then $[\![\mathcal{S}_1]\!]_{\max} = [\![\mathcal{S}_2]\!]_{\max}$ but $\mathcal{S}_1$ is strictly below $\mathcal{S}_2$ in the snack order. A more complicated example of incomparable $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $[\![\mathcal{S}_1]\!]_{\max} \subset [\![\mathcal{S}_2]\!]_{\max}$ can also be found. Thus, the semantics in terms of maximal elements does not agree very well with the ordering of snacks which is supposed to mean being more partial. However, we can show (cf. Ngair [121]) that

Proposition 4.16 *If $\mathcal{S}_1$ and $\mathcal{S}_2$ are two snacks, then $\mathcal{S}_1 \sqsubseteq_f^{\mathbb{B}} \mathcal{S}_2$ iff $[\![\mathcal{S}_2]\!] \subseteq [\![\mathcal{S}_1]\!]$.*

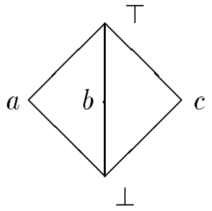
Proof. Let $\mathcal{S}_1 = (U, \mathcal{L})$ and $\mathcal{S}_2 = (V, \mathcal{M})$. Prove the *'if'* part first. Assume $[\![\mathcal{S}_2]\!] \subseteq [\![\mathcal{S}_1]\!]$. Pick arbitrarily an element m_M from each $M \in \mathcal{M}$. Then $V' = V \cup \{m_M | M \in \mathcal{M}\} \in [\![\mathcal{S}_2]\!]$ and therefore $V' \in [\![\mathcal{S}_1]\!]$ which means $U \sqsubseteq^\sharp V' \sqsubseteq^\sharp V$. Hence, $U \sqsubseteq^\sharp V$.

Let $\mathcal{M} = \emptyset$. Then $\mathcal{L} = \emptyset$ because if $\mathcal{L} \neq \emptyset$, then $\emptyset \in [\![\mathcal{S}_2]\!]$ but $\emptyset \notin [\![\mathcal{S}_1]\!]$. Hence, in this case $\mathcal{S}_1 \sqsubseteq_f^{\mathbb{B}} \mathcal{S}_2$.

Assume $\mathcal{M} \neq \emptyset$ and $\mathcal{S}_1 \not\sqsubseteq_f^{\mathbb{B}} \mathcal{S}_2$; then $\exists L\, \forall M\, \exists m \in M\, \forall l \in L : l \not\leq m$. Let $L \in \mathcal{L}$ be a set for which the statement above is true; then, selecting appropriate m for each $M \in \mathcal{M}$ we obtain a set Q such that $Q \cap M \neq \emptyset$ for all $M \in \mathcal{M}$ and $\forall l \in L \forall q \in Q : l \not\leq q$. In other words, $\uparrow L \cap Q = \emptyset$. On the other hand, $Q \in [\![\mathcal{S}_2]\!] \subseteq [\![\mathcal{S}_1]\!]$ and therefore $\uparrow L \cap Q \neq \emptyset$ for all $L \in \mathcal{L}$. This contradiction shows $\mathcal{S}_1 \sqsubseteq_f^{\mathbb{B}} \mathcal{S}_2$.

To show the *'only if'* part, assume $\mathcal{S}_1 \sqsubseteq_f^{\mathbb{B}} \mathcal{S}_2$ and $Q \in [\![\mathcal{S}_2]\!]$. Then $U \sqsubseteq^\sharp V \sqsubseteq^\sharp Q$ and, given $L \in \mathcal{L}$, there exist $M \in \mathcal{M}$ such that $\uparrow M \subseteq \uparrow L$ and therefore $Q \cap \uparrow L \neq \emptyset$. This shows $Q \in [\![\mathcal{S}_1]\!]$. Proposition is proved. $\square$

Unfortunately, this is no longer true for scones because, given the following A:



let $\mathcal{S}_1 = (a, b)$ and $\mathcal{S}_2 = (a, c)$ be two scones over A. Then $\{\{\top\}, \{a, \top\}\} = [\![\mathcal{S}_1]\!] = [\![\mathcal{S}_2]\!]$ but $\mathcal{S}_1$ and $\mathcal{S}_2$ are incomparable.

However, there is a very close connection between semantics of scones and snacks and the ordering. In some sense, the family of snacks over A is the maximal subclass of scones over A on which the semantics and the orderings agree. To formulate this rigorously, let $\mathcal{S}_1 \preccurlyeq \mathcal{S}_2$ iff $[\![\mathcal{S}_2]\!] \subseteq [\![\mathcal{S}_1]\!]$. Then $\preccurlyeq$ is a preorder and the induced equivalence relation is denoted by $\varepsilon_{\preccurlyeq}$.

Proposition 4.17 *For a bounded complete poset A, $\langle \mathcal{P}^{\exists\wedge}(A), \preccurlyeq \rangle / \varepsilon_{\preccurlyeq} \cong \mathcal{P}^{\forall}(A)$.*

Proof. If A is bounded complete, then for two finite sets U and L the set $\min(\uparrow U \cap \uparrow L)$ is also finite. Hence, we define $\psi : \mathcal{P}^{\exists\wedge}(A) \to \mathcal{P}^{\forall}(A)$ by $\psi((U, \mathcal{L})) = (U, \{\min(\uparrow U \cap \uparrow L)\} | L \in \mathcal{L})$. Clearly, $[\![\mathcal{S}]\!] = [\![\psi(\mathcal{S})]\!]$ and $\psi(\psi(\mathcal{S})) = \psi(\mathcal{S})$. According to proposition 4.16, $\psi(\mathcal{S})$ is the only snack in the $\varepsilon_{\preccurlyeq}$-equivalence class of $\mathcal{S}$. Moreover, ψ is monotone because, if $U \sqsubseteq^{\sharp} V$ and $L \sqsubseteq^{\sharp} M$, then $\min(\uparrow L \cap \uparrow U) \sqsubseteq^{\sharp} \min(\uparrow M \cap \uparrow V)$. This finishes the proof of the proposition. □

The following result follows directly from the definitions.

Proposition 4.18 *Given $\mathcal{S} \in \mathcal{P}^{\emptyset}(A)$, $[\![\mathcal{S}]\!] \neq \emptyset$ iff $\mathcal{S} \in \mathcal{P}^{\exists\wedge}(A)$.* □

Summing up, scones are the maximal class of approximation constructs with multi-set L-part that has well-defined semantics, and snacks are the maximal subclass of scones over on which the semantics and the orderings agree.

Using or-sets to encode approximations

We have seen already that orderings on approximations are obtained by combining orderings that were suggested for OWA sets and or-sets. This brings up the idea of using or-sets in encoding approximations. We show an intimate connection between the semantics of sets, or-sets and approximations that suggests a clean way of encoding approximations with sets and or-sets.

First, consider the semantics of a mix (or a sandwich) (U, L). Let $X \in [\![(U, L)]\!]$. Then $L \sqsubseteq^{\flat} X$ which means $X \in [\![L]\!]_s$ where L is considered as an ordinary set. Furthermore $U \sqsubseteq^{\sharp} X$ means $X \in [\![U]\!]_s$ where U is considered as an or-set. Thus, we have

Proposition 4.19 *For any mix (sandwich)* (U, L) *where* $U = \{u_1, \ldots, u_n\}$ *and* $L = \{l_1, \ldots, l_k\}$, $X \in [\![(U, L)]\!]$ *iff* $X \in [\![\{l_1, \ldots, l_k\}]\!]_s$ *and* $X \in [\![\langle u_1, \ldots, u_n\rangle]\!]_s$. □

Furthermore, assume that all elements of U and L come from a poset A. Then $X \in [\![(U, L)]\!]$ means that $X \in [\![\{l_1, \ldots, l_k\}]\!]_c$ and $X \subseteq [\![\langle u_1, \ldots, u_n\rangle]\!]_c$. This suggests that the lower approximation be encoded as a set and the upper as an or-set.

Now consider constructions like snacks and scones. Then the following is immediate from the definitions.

Proposition 4.20 *Assume that* $U \in \mathbb{A}_{\text{fin}}(A)$ *and* $\mathcal{L}$ *is an antichain (with respect to* $\sqsubseteq^\sharp$*) of finite antichains of* A. *Let* $(U, \mathcal{L})$ *be an element of* $\mathbb{P}^i(A)$, *where* $i \in \{\forall, \exists, \forall\wedge, \exists\wedge, \emptyset\}$, *where* $U = \{u_1, \ldots, u_n\}$ *and* $\mathcal{L} = \{L_1, \ldots, L_k\}$, $L_i = \{l_1^i, \ldots, l_{m_i}^i\}$. *Then* $X \in [\![(U, \mathcal{L})]\!]$ *iff*

$$X \subseteq [\![\langle u_1, \ldots, u_n\rangle]\!]_c \text{ and } X \in [\![\{\langle l_1^1, \ldots, l_{m_1}^1\rangle, \ldots, \langle l_1^k, \ldots, l_{m_k}^1\rangle\}]\!]_c$$

This proposition suggests that the lower approximation be encoded as a set of or-sets and the upper as an or-set. Summing up, we have the following correspondence between types of approximations over type t and sets and or-sets:

Approximations	Encoding
$\mathcal{P}^i([\![t]\!])$, $i \in \{\forall, \exists, \forall\wedge, \exists\wedge, \emptyset\}$	$\langle t\rangle \times \{t\}$
$\mathbb{P}^i([\![t]\!])$, $i \in \{\forall, \exists, \forall\wedge, \exists\wedge, \emptyset\}$	$\langle t\rangle \times \{\langle t\rangle\}$

It will be seen in chapter 5 that these encodings provide a convenient way of programming with approximations, which has a number of advantages over the approach based on structural recursion and monads.

4.2 Universality properties of partial data

The goal of this section is to demonstrate the universality properties of various collections that later will be used as a basis for the programming syntax design. We have seen examples of turning universality properties into syntax in section 3.2.

The collections we study include sets and or-sets. We concentrate on sets under the open world assumption. We also look at the iterated constructions which correspond to the objects of types $\langle\{t\}\rangle$ and $\{\langle t\rangle\}$. These will be of special importance when we study normalization of or-objects. Finally, we characterize approximation constructs as free algebras.

To explain the setting before we embark on a lengthy technical development (and thus save time for the reader who does not want to read the proofs and just want to look at the theorems), we always start with a partially ordered set A and characterize various constructions as free ordered algebras generated by A. That is, the general form of the results is finding the signature Ω of an ordered algebra on each construction $C(A)$ and an embedding $\eta : A \to C(A)$ such that for each ordered Ω-algebra $\langle X, \Omega\rangle$ and each monotone map $f : A \to X$, there exists a unique monotone Ω-homomorphism that makes the following diagram commute:

$$\begin{tikzcd} A \arrow[r, hook, "\eta"] \arrow[dr, "f"'] & \langle C(A), \Omega\rangle \arrow[d, dashed, "\exists! f^+"] \\ & \langle X, \Omega\rangle \end{tikzcd}$$

Constructions C that we consider are the following. For sets (under OWA) we use $\mathcal{P}^\flat(A)$; for or-sets we use $\mathcal{P}^\sharp(A)$. We consider two iterated constructions $\mathcal{P}^\flat(\mathcal{P}^\sharp(A))$ and $\mathcal{P}^\sharp(\mathcal{P}^\flat(A))$. And we study approximations $\mathcal{P}^i(A)$ and $\boldsymbol{\mathcal{P}}^i(A)$ where $i \in \{\forall, \exists, \forall\wedge, \exists\wedge, \emptyset\}$.

4.2.1 Universality properties of collections

Universality properties of $\mathcal{P}^\flat(A)$ and $\mathcal{P}^\sharp(A)$ have been demonstrated already. In lemma 2.2 it was proved that $\mathcal{P}^\flat(A)$ is the join-semilattice with bottom element freely generated by A and $\mathcal{P}^\sharp(A)$ is the meet-semilattice with top element freely generated by A. In other words, if we consider $\mathcal{P}^\flat$ as a functor from **Poset** to $\mathbf{SL}_0$ and $\mathcal{P}^\sharp$ as a functor from **Poset** to $\mathbf{SL}_1$, then we have the following adjunctions:

$$\mathcal{P}^\flat \dashv \mathbb{U}_{\mathbf{SL}_0 - \mathbf{Poset}} \qquad \mathcal{P}^\sharp \dashv \mathbb{U}_{\mathbf{SL}_1 - \mathbf{Poset}}$$

where $\mathbb{U}$s are forgetful functors. This adjunction cuts down to an adjunction of categories in which all objects are finite. The monads corresponding to these adjunctions have been shown in section 2.3. We shall return to them again in chapter 5.

4.2.2 The iterated construction

We have seen that or-sets correspond to the Smyth powerdomain and sets correspond to the Hoare powerdomain. If we would like to see how sets and or-sets can interact, we should look at a combination of these two constructions. (This is similar to the way the definition of a strong monad was introduced. One needed an interaction of a functor $\mathbb{T}$ with products. In the case of the languages from section 3.2, this resulted in adding ρ_2 which provides interaction between sets or bags and products.)

We have two ways of combining the the semantic constructions corresponding to sets and or-sets: $\mathcal{P}^{\flat\sharp}(A) = \mathcal{P}^{\flat}(\mathcal{P}^{\sharp}(A))$ and $\mathcal{P}^{\sharp\flat}(A) = \mathcal{P}^{\sharp}(\mathcal{P}^{\flat}(A))$. The question that arises is which one to consider. The answer is: either one. This is possible because Flannery and Martin [53] proved that $\mathcal{P}^{\flat\sharp}(A)$ and $\mathcal{P}^{\sharp\flat}(A)$ are isomorphic. However, from their proof it is impossible to derive the isomorphism we would be able to use, as they proved the isomorphism at the level of *information systems,* cf. C. Gunter [67].

Later Heckmann [71] tried to simplify the proof. His proof, however, was based on a number of universality properties which postulated existence and uniqueness of certain mappings, and a combination of some of those was shown to be the desired isomorphism. This again is not satisfactory. Finally, in Libkin [100], an elementary proof was given in which the isomorphism was explicitly constructed. We state the result here, and later use the isomorphism to add a primitive providing interaction between sets and or-sets to the language for those collections.

An element of $\mathcal{P}^{\sharp\flat}(A)$ is a finite antichain, with respect to $\sqsubseteq^{\flat}$, of finite antichains of elements of A, and a element of $\mathcal{P}^{\flat\sharp}(A)$ is a finite antichain, with respect to $\sqsubseteq^{\sharp}$, of finite antichains of elements of A. Given a finite set of finite sets $\mathcal{X} = \{X_1, \ldots, X_n\}$ where $X_i = \{x^i_1, \ldots, x^i_{k_i}\}$, let $F_{\mathcal{X}}$ be the set of functions $f : \{1, ..., n\} \to \mathbb{N}$ such that for any i: $1 \leq f(i) \leq k_i$. For $f \in F_{\mathcal{X}}$, let $f(\mathcal{X}) = \{x^i_{f(i)} \mid i = 1, \ldots, n\}$. If all X_i's are subsets of A, define two maps α and β as follows:

$$\alpha(\mathcal{X}) = \min_{f \in F_{\mathcal{X}}} {}_{\sqsubseteq^{\flat}}(\max f(\mathcal{X}))$$

$$\beta(\mathcal{X}) = \max_{f \in F_{\mathcal{X}}} {}_{\sqsubseteq^{\sharp}}(\min f(\mathcal{X}))$$

Theorem 4.21 *$\alpha : \mathcal{P}^{\flat\sharp}(A) \to \mathcal{P}^{\sharp\flat}(A)$ and $\beta : \mathcal{P}^{\sharp\flat}(A) \to \mathcal{P}^{\flat\sharp}(A)$ are mutually inverse isomorphisms between $\mathcal{P}^{\flat\sharp}(A)$ and $\mathcal{P}^{\sharp\flat}(A)$.*

Proof. We have to show that α maps $\mathcal{P}^{\flat\sharp}(A)$ to $\mathcal{P}^{\sharp\flat}(A)$, β maps $\mathcal{P}^{\sharp\flat}(A)$ to $\mathcal{P}^{\flat\sharp}(A)$ and α and β are mutually inverse and monotone. The first two claims follow immediately from the definitions of α and β. To complete the proof, show that α is monotone and $\beta \circ \alpha = \mathsf{id}$. By duality the proof of monotonicity of β and $\alpha \circ \beta = \mathsf{id}$ can be obtained.

Recall that if V and W are finite subsets of an arbitrary poset, then 1) $V \sqsubseteq^{\flat} W$ iff $\max V \sqsubseteq^{\flat} \max W$ and 2) $V \sqsubseteq^{\sharp} W$ iff $\min V \sqsubseteq^{\sharp} \min W$. Notice that both $\mathcal{P}^{\sharp\flat}(A)$ and $\mathcal{P}^{\flat\sharp}(A)$ have bottom and top elements. These are $\emptyset$ and $\{\emptyset\}$, and they are mapped to each other by α and β. Hence, in the rest of the proof we do not consider empty sets.

Throughout this proof, $\mathcal{X}$ is defined as above, i.e. $\mathcal{X} = \{X_1, \ldots, X_n\}$ and each X_i consists of elements x^i_j, $j = 1, \ldots, k_i$.

Claim 1: α is monotone.
Proof of claim 1: Let $\mathcal{X}, \mathcal{Y} = \{Y_1, ..., Y_m\} \in \mathcal{P}^{\flat\sharp}(A)$ and $\mathcal{X} \sqsubseteq^\flat \mathcal{Y}$. We must prove $\alpha(\mathcal{X}) \sqsubseteq^\sharp \alpha(\mathcal{Y})$. In view of the above observations, it is enough to show that for any $f \in \mathcal{F}_\mathcal{Y}$ there exists $g \in \mathcal{F}_\mathcal{X}$ such that $g(\mathcal{X}) \sqsubseteq^\flat f(\mathcal{Y})$. Since for each $i = 1, \ldots, n$ there exists j_i such that $X_i \sqsubseteq^\sharp Y_{j_i}$, there is an element $x^i_{p_i} \in X_i$ such that $x^i_{p_i} \leq y^{j_i}_{f(j_i)}$. Let $g(i) = p_i$. Then for this function g one has $\{x^i_{g(i)} \mid i = 1, \ldots, n\} \sqsubseteq^\flat \{y^i_{f(i)} \mid i = 1, \ldots, m\}$, i.e. $g(\mathcal{X}) \sqsubseteq^\flat f(\mathcal{Y})$. Claim 1 is proved.

Let $\mathcal{X} \in \mathcal{P}^{\flat\sharp}(A)$ and $\mathcal{Y} = \{Y_1, \ldots, Y_m\} = \alpha(\mathcal{X}) \in \mathcal{P}^{\sharp\flat}(A)$. By 1) and 2) above, to show that $\beta \circ \alpha = \mathsf{id}$, i.e. that $\beta(\mathcal{Y}) = \mathcal{X}$, it suffices to prove
Claim 2: For any $f \in \mathcal{F}_\mathcal{Y}$ there exists $X_i \in \mathcal{X}$ such that $f(\mathcal{Y}) \sqsubseteq^\sharp X_i$.
Claim 3: Every X_i is in $\beta(\mathcal{Y})$.

Proof of claim 2: Let $\mathcal{Z}$ be the collection of all sets $f(\mathcal{X})$ where $f \in \mathcal{F}_\mathcal{X}$; $\mathcal{Z} = \{Z_1, \ldots, Z_k\}$. Then for any $g \in F_\mathcal{Z}$, there exists $X_i \in \mathcal{X}$ such that X_i is contained in $g(\mathcal{Z})$ because, if this is not the case, for any $X_i \in \mathcal{X}$ there exists $j_i \leq k_i$ such that $x^i_{j_i} \in X_i$ and, for any $f \in \mathcal{F}_\mathcal{X}$, g on $f(\mathcal{X})$ picks an element different from $x^i_{j_i}$. If we define f_0 such that $f_0(i) = j_i$, g may pick only elements of form $x^i_{j_i}$ on $f_0(\mathcal{X})$, a contradiction. Therefore, $g(\mathcal{Z}) \sqsubseteq^\sharp X_i$ for some i.

Let $f \in \mathcal{F}_\mathcal{Y}$. Let H be the set of functions in $\mathcal{F}_\mathcal{X}$ that correspond to elements of $\mathcal{Y} = \alpha(\mathcal{X})$ or, in other words, $\max h(\mathcal{X}) \in \mathcal{Y}$ for $h \in H$. Then, for any $h' \in \mathcal{F}_\mathcal{X} - H$, there exists a function $h \in H$ such that $\max h(\mathcal{X}) \sqsubseteq^\flat \max h'(\mathcal{X})$, i.e. $h(\mathcal{X}) \sqsubseteq^\flat h'(\mathcal{X})$. Since $h \in H$, $\max h(\mathcal{X}) \in \mathcal{Y}$, i.e. $\max h(\mathcal{X}) = Y_i$. If $f(i) = j$, then there is an element in $h'(\mathcal{X})$ that is greater than y^i_j. Define a function $g \in F_\mathcal{Z}$ to coincide with f on those Z_i's that are given by functions in H. On Z_i that corresponds to $f \in \mathcal{F}_\mathcal{X} - H$, let g pick an element which is greater than some y^i_j where $f(i) = j$ (we have just shown it can be done). Then $f(\mathcal{Y}) \sqsubseteq^\sharp \{z^i_{g(i)} \mid i = 1, \ldots, k\} = g(\mathcal{Z})$. We know that there exists $X_i \in \mathcal{X}$ such that $g(\mathcal{Z}) \sqsubseteq^\sharp X_i$. Thus, $f(\mathcal{Y}) \sqsubseteq^\sharp X_i$. Claim 2 is proved.

Proof of claim 3: Prove that for any $x^i_j \in X_i$ there exists $Y_l \in \mathcal{Y}$ such that $x^i_j \in Y_l$. Consider the set $F^{ij}_\mathcal{X}$ of functions $f \in \mathcal{F}_\mathcal{X}$ such that $f(i) = j$. If for no $f \in F^{ij}_\mathcal{X}$: $x^i_j \in \max f(\mathcal{X})$, then there exists $X_p \in \mathcal{X}$ such that all elements of X_p are greater than x^i_j, i.e. $X_i \sqsubseteq^\sharp X_p$ which contradicts our assumption that $\mathcal{X}$ is an antichain with respect to $\sqsubseteq^\sharp$. Hence, $x^i_j \in \max f(\mathcal{X})$ for at least one function in $F^{ij}_\mathcal{X}$. Since $\mathcal{X}$ is an antichain, for any $p \neq i$ there exists $x^p_q \in X_p$ which is not greater than any element of X_i. Change f to pick such an element for any $p \neq i$. Then x^i_j is still in $\max f(\mathcal{X})$. There exists a function $f' \in \mathcal{F}_\mathcal{X}$ such that $\max f'(\mathcal{X}) \sqsubseteq^\flat \max f(\mathcal{X})$ and $\max f'(\mathcal{X}) \in \alpha(\mathcal{X})$. If $f'(i) = j' \neq j$, then, since $f'(\mathcal{X}) \sqsubseteq^\flat f(\mathcal{X})$ and X_i is an antichain, $x^i_{j'} \leq x^p_q$ for some p and q, where $p \neq i$. But this contradicts the definition of f. Hence, $f'(i) = j$ and $x^i_j \in \max f'(\mathcal{X})$ because $x^i_j \in \max f(\mathcal{X})$. Since $\max f'(\mathcal{X}) = Y_l$ for some index l, $x^i_j \in Y_l \in \mathcal{Y}$.

Let $\mathcal{Y}'$ be the collection of elements of $\mathcal{Y}$ that contain elements of X_i. Then we can define a function $f \in \mathcal{F}_\mathcal{Y}$ on elements of $\mathcal{Y}'$ to pick all elements of X_i. Each $Y_j \in \mathcal{Y} - \mathcal{Y}'$ either contains an element of X_i or contains an element which is greater than some $x^i_p \in X_i$. Let f pick any such element. Then $\min f(\mathcal{Y}) = X_i$. Suppose $X_i \notin \beta(\mathcal{Y})$. Then $X_i \sqsubseteq^\sharp \min g(\mathcal{Y})$ for some function $g \in \mathcal{F}_\mathcal{Y}$ such that $\min g(\mathcal{Y}) \in \beta(\mathcal{Y})$. By claim 2, $g(\mathcal{Y}) \sqsubseteq^\sharp X_j$ for some X_j. Hence, $\min g(\mathcal{Y}) \sqsubseteq^\sharp X_j$ and since $\mathcal{X}$ is an antichain with respect to $\sqsubseteq^\sharp$, $X_i = X_j = \min g(\mathcal{Y}) \in \beta(\mathcal{Y})$. This finishes the proof of claim 3 and the theorem. $\square$

Now, let us see what α does if there is no order involved. In this case an input to α can be considered as a set of or-sets:

$$\mathcal{X} \quad = \quad \{\langle x_1^1, \ldots, x_{k_1}^1 \rangle, \ldots, \langle x_1^n, \ldots, x_{k_n}^n \rangle\}$$

Then $\alpha(\mathcal{X})$ is the or-set of sets

$$\langle \{x_{f(1)}^1, \ldots, x_{f(n)}^n\} \mid f \in F_\mathcal{X} \rangle$$

That is, all possible choices encoded by or-sets are explicitly listed. Notice that we used a very similar construction in the proof of proposition 4.12 to show that the conceptual semantics of any object is a finitely generated filter. We shall use α as a programming primitive extensively in chapter 5.

The iterated construction does possess a universality property.

Theorem 4.22 *For any poset A, $\mathcal{P}^{\flat\sharp}(A)$ is the free distributive lattice with top and bottom generated by A.*

Proof. First, $\mathcal{P}^{\flat\sharp}(A)$ is a distributive lattice with top and bottom since $\mathcal{P}^\flat(A)$ is a distributive lattice for any A, and $\mathcal{P}^{\flat\sharp}(A)$ has top element $\{\emptyset\}$ and bottom element $\emptyset$. Now we must prove the following: for any distributive lattice with bottom and top $\langle D, \bot\top \rangle$, and any monotone map $f : A \to D$, there exists a unique homomorphism of distributive lattices with top and bottom f^+ that makes the following diagram commute (where $\eta(x) = \{\{x\}\}$):

$$\begin{tikzcd}
A \arrow[r, hook, "\eta"] \arrow[dr, "f"'] & \langle \mathcal{P}^{\flat\sharp}(A), \sqcup^\flat, \sqcap^\flat, \{\emptyset\}, \emptyset \rangle \arrow[d, dashed, "\exists! f^+"] \\
& \langle D, \vee, \wedge, \bot, \top \rangle
\end{tikzcd}$$

To define f^+, first notice that $f^+(\emptyset) = \bot$ and $f^+(\{\emptyset\}) = \top$. Other elements of $\mathcal{P}^{\flat\sharp}(A)$ are antichains $\mathcal{X}$, with respect to $\sqsubseteq^\sharp$, of antichains of A. Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ where $X_i = \{x_1^i, \ldots, x_{k_i}^i\}$. Then define

$$f^+(\mathcal{X}) \quad = \quad \bigvee_{i=1}^{n} \bigwedge_{j=1}^{k_i} f(x_j^i)$$

Clearly, $f^+(\eta(x)) = f(x)$. Let us show that f^+ is a homomorphism. Given $\mathcal{X}$ and $\mathcal{Y} = \{Y_1, ..., Y_m\}$, then $\mathcal{X} \sqcup^\flat \mathcal{Y} = \max^\sharp(\mathcal{X} \cup \mathcal{Y})$ and $\mathcal{X} \sqcap^\flat \mathcal{Y} = \max^\sharp\{X \sqcap^\sharp Y \mid X \in \mathcal{X}, Y \in \mathcal{Y}\} = \max^\sharp\{\min(X \cup Y) \mid X \in \mathcal{X}, Y \in \mathcal{Y}\}$. Notice that $V \sqsubseteq^\sharp W$ implies $\bigwedge_{v \in V} f(v) \leq \bigwedge_{w \in W} f(w)$. Moreover, $\mathcal{X}(\sqsubseteq^\sharp)^\flat\mathcal{Y}$ implies $f^+(\mathcal{X}) \leq f^+(\mathcal{Y})$. Hence, writing expressions for f^+ we may leave nonminimal elements in individual antichains and nonmaximal elements in families of antichains. With this in mind, we calculate

$$f^+(\mathcal{X} \sqcup^\flat \mathcal{Y}) \quad = \quad \bigvee_{Z \in \mathcal{X} \cup \mathcal{Y}} \bigwedge_{z \in Z} f(z) = f^+(\mathcal{X}) \vee f^+(\mathcal{Y}) \quad \text{and}$$

$$f^+(\mathcal{X} \sqcap^\flat \mathcal{Y}) = \bigvee_{X \in \mathcal{X}, Y \in \mathcal{Y}} \bigwedge_{z \in X \cup Y} f(z) \quad =$$

$$\Big(\bigvee_{X \in \mathcal{X}} \bigwedge_{x \in X} f(x)\Big) \wedge \Big(\bigvee_{Y \in \mathcal{Y}} \bigwedge_{y \in Y} f(y)\Big) \quad = \quad f^+(\mathcal{X}) \wedge f^+(\mathcal{Y})$$

Thus, f^+ is a homomorphism. Its uniqueness follows from $\mathcal{X} = \sqcup^\flat_i \sqcap^\sharp_j \eta(x^j_i)$. Theorem is proved. $\square$

This result can be generalized for slightly changed iterated constructions. Let $\mathcal{P}^\flat_{\neq\emptyset}(A)$ and $\mathcal{P}^\sharp_{\neq\emptyset}$ be defined as $\mathcal{P}^\flat$ and $\mathcal{P}^\sharp$ except that the empty antichain is not allowed. Let $\mathcal{P}^{\flat\sharp}_{\neq\emptyset}$ and $\mathcal{P}^{\sharp\flat}_{\neq\emptyset}$ be respective compositions of $\mathcal{P}^\flat_{\neq\emptyset}$ and $\mathcal{P}^\sharp_{\neq\emptyset}$. Then analyzing the proofs of theorems 4.21 and 4.22, it is easy to see that the following holds.

Corollary 4.23 *For an arbitrary poset A, $\mathcal{P}^{\flat\sharp}_{\neq\emptyset}(A)$ and $\mathcal{P}^{\sharp\flat}_{\neq\emptyset}(A)$ are isomorphic. Moreover, $\mathcal{P}^{\flat\sharp}_{\neq\emptyset}(A)$ is the free distributive lattice generated by A.* $\square$

In particular, $\mathcal{P}^{\flat\sharp}_{\neq\emptyset} \dashv \mathbb{U}$ form an adjoint pair of functors between categories **Poset** and **DL**, where **DL** is the category of distributive lattices and $\mathbb{U}$ is the forgetful functor **DL** $\to$ **Poset**.

4.2.3 Universality properties of approximations

The main purpose of this section is to describe all approximation constructions, that is, $\mathcal{P}^i(A)$ and $\mathbf{P}^i(A)$, as free ordered algebras generated by A. Of course we have to explain how A is viewed as a subset of those. This is achieved by defining two functions (for which we use the same notation)

$$A \xhookrightarrow{\eta} \mathcal{P}^i(A): \quad \eta(x) = (x, x) \qquad \text{and} \qquad A \xhookrightarrow{\eta} \mathbf{P}^i(A): \quad \eta(x) = (x, \{x\})$$

Notice that we often omit the set brackets $\{\}$ when we deal with singletons. In particular, by $\{x\}$ we meant a family of sets that consists of one singleton. In proofs, we shall also occasionally omit commas separating elements of sets, writing xyz for $\{x, y, z\}$.

It would be ideal if could obtain freeness results for all constructions, but there is one obstacle. Consider a poset A and $x, y \in A$ such that $x \Uparrow y$. Then (x, y) is a sandwich and $(x, \{y\})$ is a scone. Thus, if $\mathcal{P}^{\forall\wedge}(A)$ or $\mathbb{P}^{\exists\wedge}(A)$ were free algebras generated by A, there would be a way to construct (x, y) or $(x, \{y\})$ from the singletons like (x, x) or $(x, \{x\})$. But this way must use the information about consistency in A and therefore can not be "universal"!

We shall make this precise by proving that the approximation constructs with $u \Uparrow l$ used in the consistency condition do *not* arise as free ordered algebras generated by A. But we give a method to repair the failure of certain approximations to be free algebras. The idea is that information about consistency in A must be conveyed by the generating poset. We define the *consistent closure* of A as

$$A \Uparrow A = \{(a, b) \mid a \in A, b \in A, a \Uparrow b\}$$

The consistent closure of A can be embedded into $\mathcal{P}^i(A)$ and $\mathbb{P}^i(A)$ (where $i \in \{\exists\wedge, \forall\wedge\}$) by means of the following functions:

$$A \Uparrow A \xhookrightarrow{\eta^\Uparrow} \mathcal{P}^i(A) \; : \quad \eta^\Uparrow(x, y) = (x, y) \qquad \text{and} \qquad A \Uparrow A \xhookrightarrow{\eta^\Uparrow} \mathbb{P}^i(A) \; : \quad \eta^\Uparrow(x, y) = (x, \{y\})$$

When the structure of an arbitrary free algebra is described, it is assumed that η is an arbitrary map of generators into an algebra of the given signature. This is no longer enough for ordered constructions like $\mathcal{P}^\flat(A)$ and $\mathcal{P}^\sharp(A)$ because those are free *ordered* algebras generated by *ordered* sets. In particular, we always start with a monotone map that is to be extended to a monotone homomorphism. In the case of sandwiches or scones, we go even further and impose additional structure on the generating poset. This structure must be consistent with the resulting algebra. To guarantee it, we put additional restriction on the map f saying that the structure of $A \Uparrow A$ should not be destroyed by f. We call such maps *admissible.* Of course there will be different definitions of admissibility for different kinds of approximations. When we say that an algebra is freely generated by a poset with respect to a class C of maps, we mean that any map f in C can be extended to a monotone homomorphism.

In the rest of this section we prove three kinds of results. The constructs not using $u \Uparrow l, u, l \in A$ in the consistency condition are found to be certain ordered algebras freely generated by A. Those that do use such consistency conditions can not be obtained as free algebras generated by A. However, some of them can be obtained as algebras freely generated by $A \Uparrow A$ with respect to properly restricted (admissible) maps.

Operations used in the free algebra characterizations are either operations similar to the "formal union" such as in the characterization of the Plotkin powerdomain [137], or modal operations in the spirit of Winskel [178] or operations associated with the orderings (such as infimum) or other binary operations that can be viewed as combinations of the above.

Universality of $\mathcal{P}^{\forall}(A)$ (mixes)

The characterization of the mixes as free ordered algebras was given by C. Gunter [66]. For the sake of completeness, we recall it here. We shall also need the same algebras for dealing with sandwiches.

Definition 4.5 *A* mix *algebra* $\langle M, +, \Box, e\rangle$ *has partially ordered carrier* M*, one monotone binary operation* $+$ *and one monotone unary operation* $\Box$*.* $\langle M, +, e\rangle$ *is a semilattice with identity* e*, and in addition the following equations must hold:*
1) $\Box(x+y) = \Box x + \Box y$,
2) $\Box\Box x = \Box x$,
3) $\Box x \leq x$,
4) $x + \Box x = x$,
5) $x + \Box y \leq x$.

A *mix homomorphism* of two mix algebras $\langle M_1, +_1, \Box_1, e_1\rangle$ and $\langle M_2, +_2, \Box_2, e_2\rangle$ is a monotone map $f : M_1 \to M_2$ such that $f(x +_1 y) = f(x) +_2 f(y), f(\Box_1 x) = \Box_2 f(x)$ and $f(e_1) = f(e_2)$. That is, in addition to being homomorphism in the usual sense, f must be monotone as well.

$\mathcal{P}^{\forall}(A)$ can be given the structure of a mix algebra by taking the ordering $\sqsubseteq^{\mathbb{B}}$ and defining

$$(U, L) + (V, M) = (\min(U \cup V), \max(L \cup M)) \qquad \Box(U, L) = (U, \emptyset) \qquad e = (\emptyset, \emptyset)$$

Theorem 4.24 (C. Gunter [66]) $\mathcal{P}^{\forall}(A)$ *is the free mix algebra generated by* A*. That is, for any mix algebra* M *and a monotone map* $f : A \to M$ *there exists a unique mix homomorphism* $f^+ : \mathcal{P}^{\forall}(A) \to M$ *that makes the following diagram commute:*

$$\begin{tikzcd} A \arrow[r, hook, "\eta"] \arrow[dr, "f"'] & \langle \mathcal{P}^{\forall}(A), +, \Box, e\rangle \arrow[d, dashed, "\exists! f^+"] \\ & \langle M, +, \Box, e\rangle \end{tikzcd}$$

$\Box$

Universality of $\mathcal{P}^{\natural}$ (sandwiches)

We would like to characterize sandwiches as a free construction over A. Suppose we start with the same function $\eta : A \to \mathcal{P}^{\natural}(A)$ given by $\eta(x) = (x, x)$. For any pair $x, y \in A$ such that $x \uparrow y$, there is a sandwich (x, y) over A. Thus, if we view $\mathcal{P}^{\natural}(A)$ as a free algebra in a certain signature, there must be a way to construct (x, y) out of pairs with identical components. But this way must use information that $x \uparrow y$ and therefore can not be "universal". To be precise, the following holds.

Theorem 4.25 *It is impossible to find a family Ω of operations on sandwiches such that $\mathcal{P}^{\natural}(\cdot)$ would be left adjoint to the forgetful functor from the category of ordered Ω-algebras to* **Poset**. *In other words, for no Ω is $\mathcal{P}^{\natural}(A)$ the free ordered Ω-algebra generated by A.*

Proof. Assume that there exists a set of operation Ω such that $\mathcal{P}^{\natural}(A)$ the free ordered Ω-algebra generated by A for any poset A. Let $A = \{x, y, z\}$ be an antichain and $A' = \{x', y', z'\}$ be a poset such that $x', y' \precsim z'$ and $x' \not\precsim y'$, $y' \not\precsim x'$. Let $f : A \to \mathcal{P}^{\natural}(A')$ be defined by $f(a) = (a', a'), a \in A$. Now the assumed universality property tells us that f can be extended to a monotone Ω-homomorphism $f^+ : \mathcal{P}^{\natural}(A) \to \mathcal{P}^{\natural}(A')$. Let $\mathcal{S} \in \mathcal{P}^{\natural}(A')$. Since $\mathcal{P}^{\natural}(A')$ is the free Ω-algebra generated by A', we can find a term t in the signature Ω such that $\mathcal{S} = t(\eta(x'), \eta(y'), \eta(z'))$. Since $\eta(x') = f(x) = f^+(\eta(x))$ and similarly for y' and z', we obtain $\mathcal{S} = f^+(t(\eta(x), \eta(y), \eta(z))) = f^+(\mathcal{S}_0)$ for some $\mathcal{S}_0 \in \mathcal{P}^{\natural}(A)$. Therefore, f^+ is onto.

Define $\mathcal{P}^{\natural}_{\neg xy}(A)$ as the set of elements of $\mathcal{P}^{\natural}(A)$ which are not under (x, x) or (y, y). It is easy to check that $\mathcal{P}^{\natural}_{\neg xy}(A)$ includes the following: (z, z), (xz, z), (yz, z), $(z, \emptyset)$, (xz, xz), (yz, yz), (xy, xy), (xyz, xz), (xyz, yz), (xyz, xy), (xyz, z). Similarly, define $\mathcal{P}^{\natural}_{\neg x'y'}(A')$ as the set of elements of $\mathcal{P}^{\natural}(A')$ which are not under (x', x') or (y', y'). These are: (x', y'), (y', x'), $(x'y', z')$, $(z', x'y')$, (x', z'), (z', x'), (y', z'), (z', y'), $(z', \emptyset)$, (z', z'). Since f^+ is monotone, we derive that its restriction on $\mathcal{P}^{\natural}_{\neg xy}(A)$ must be an onto map from a subset of $\mathcal{P}^{\natural}_{\neg xy}(A)$ to $\mathcal{P}^{\natural}_{\neg x'y'}(A')$. Observe that in $\mathcal{P}^{\natural}_{\neg xy}(A)$ the only element that is not above (xyz, z) is $(z, \emptyset)$. Hence, if $f^+((xyz, z)) = \mathcal{S} \in \mathcal{P}^{\natural}_{\neg x'y'}(A')$, then $f^+(\mathcal{P}^{\natural}_{\neg xy}(A) - \{(z, \emptyset)\})$ is a subset of the principal filter of $\mathcal{S}$ in $\mathcal{P}^{\natural}_{\neg x'y'}(A')$. However, $\mathcal{P}^{\natural}_{\neg x'y'}(A')$ has four minimal elements: (x', y'), (y', x'), $(x'y', z')$ and $(z', \emptyset)$ which shows that f^+ can not be an onto monotone map between $\mathcal{P}^{\natural}_{\neg xy}(A)$ and $\mathcal{P}^{\natural}_{\neg x'y'}(A')$. This contradiction shows that $\mathcal{P}^{\natural}(A)$ can not be obtained as the free Ω-algebra generated by A. $\square$

Therefore, as we suggested in the introduction to this section, the information about consistency in A must be conveyed by the generating poset. That is, we use $A \uparrow A$ instead of A. The surprising result now says that sandwiches over A are the free mix algebra generated by the consistent closure of A under the same interpretation of the operations of mix algebras! Of course, we need an admissibility condition.

Definition 4.6 *Let M be a mix algebra. A monotone map $f : A{\upharpoonleft}A \to M$ is called* admissible *(or sandwich-admissible) if $f(x,y) + f(z,y) \leq f(x,y)$ and $\Box f(x,y) = \Box f(x,z)$.*

Theorem 4.26 *Given a poset A, $\mathcal{P}^{\forall\wedge}(A)$ is the free mix algebra generated by $A{\upharpoonleft}A$ with respect to the admissible maps. That is, given a mix algebra M and an admissible map $f : A{\upharpoonleft}A \to M$, there exists a unique mix homomorphism $f^+ : \mathcal{P}^{\forall\wedge}(A) \to M$ such that the following diagram commutes:*

$$\begin{tikzcd}
A{\upharpoonleft}A \arrow[r, hook, "\eta^{\upharpoonleft}"] \arrow[dr, "f"'] & \langle \mathcal{P}^{\forall\wedge}(A), +, \Box, e\rangle \arrow[d, dashed, "\exists! f^+"] \\
& \langle M, +, \Box, e\rangle
\end{tikzcd}$$

Proof. We omit an easy verification that $\mathcal{P}^{\forall\wedge}(A)$ is a mix algebra.

Let us first establish a number of useful properties of admissible maps. In what follows, f is always an admissible map from $A{\upharpoonleft}A$ to M.

1) Assume $v \precsim u$ and $u{\upharpoonleft}l$. Then $f(u,l) + f(v,l) = f(v,l)$.

First, $f(u,l) \geq f(v,l)$. By monotonicity of $+$, $f(v,l) = f(v,l) + f(v,l) \leq f(v,l) + f(u,l)$. But since f is admissible, $f(u,l) + f(v,l) \leq f(v,l)$. Hence, 1) holds.

2) Assume $p \succsim l$, $v{\upharpoonleft}l$ and $q{\upharpoonleft}p$. Then $f(v,l) + f(q,p) = \Box f(v,v) + f(q,p)$.

First show $f(q,p) + f(q,l) = f(q,p)$. By monotonicity, $f(q,p) + f(q,l) \leq f(q,p) + f(q,p) = f(q,p)$. On the other hand, $f(q,p) + f(q,l) \geq f(q,p) + \Box f(q,l) = f(q,p) + \Box f(q,p) = f(q,p)$, which proves the equation. Since $\Box f(v,v) = \Box f(v,l) \leq f(v,l)$, the $\geq$ inequation for 2) holds. Conversely, $f(v,l) + f(q,p) = f(v,l) + f(q,l) + f(q,p) = \Box f(v,l) + f(v,l) + f(q,l) + f(q,p) \leq \Box f(v,l) + f(q,l) + f(q,p) \leq \Box f(v,v) + f(q,p)$ which shows the reverse inequation. 2) is proved.

3) If $l \precsim m$, then $f(v,l) + f(q,m) = \Box f(v,v) + f(q,m)$.

The $\geq$ inequation is obvious. As in the proof of 2), we obtain $f(v,l) + f(q,m) = f(v,l) + f(q,l) + f(q,m) = \Box f(v,l) + f(v,l) + f(q,l) + f(q,m) \leq \Box f(v,l) + f(q,l) + f(q,m) \leq \Box f(v,l) + f(q,m) = \Box f(v,v) + f(q,m)$.

4) Assume $v \precsim u$. Then $f(v,l) = f(u,l) + \Box f(v,v)$.

First, $f(u,l)+f(v,l) \leq f(v,l) = f(v,l)+f(v,l) \leq f(u,l)+f(v,l)$; hence $f(u,l)+f(v,l) = f(v,l)$. Now we have: $f(v,l) = f(v,l) + f(u,l) \geq f(u,l) + \Box f(v,l) = f(u,l) + \Box f(v,v)$. On the other hand, $f(v,l) = f(v,l) + \Box f(v,l) \leq f(u,l) + \Box f(v,v)$, proving 4).

5) If $v \gtrsim u$, then $\Box f(u,u) + \Box f(v,v) = \Box f(v,v)$.

According to the proof of 4), $f(u,v) + f(v,v) = f(v,v)$ and from this 5) follows immediately.

6) Assume $u \upharpoonleft l$ and $v \upharpoonleft l$. Then $f(v,l) + \Box f(u,u) = f(v,l) + \Box f(u,u) + f(u,l)$.

Since $\Box f(u,u) = \Box f(u,l) \leq f(u,l)$, the $\leq$ inequality holds. Since $f(v,l) + f(u,l) \leq f(v,l)$, we obtain the reverse inequality.

Now let us come back to the statement of the theorem. Let $\mathcal{S} = (U,L)$ be a sandwich over A with $U = \{u_1, \ldots, u_n\}$ and $L = \{l_1, \ldots, l_k\}$. Since $\mathcal{S}$ is a sandwich, for every $l_j \in L$ there exists $u_{i_j} \in U$ such that $l_j \upharpoonleft u_{i_j}$. Let $\mathcal{I} \subseteq [n] \times [k]$ be the set of pairs of indices such that $(i,j) \in \mathcal{I} \Leftrightarrow u_i \upharpoonleft l_j$. Then

$$\mathcal{S} = \sum_{(i,j) \in \mathcal{I}} (u_i, l_j) + \Box \sum_{i=1}^{n} (u_i, u_i) \tag{1}$$

From now on we assume that summation over an empty set is the identity for the $+$ operation. It shows that (1) holds even if one of the components of a sandwich is empty.

Using representation (1), define f^+ for an admissible $f : A \upharpoonleft A \to M$ as follows:

$$f^+(S) = \sum_{(i,j) \in \mathcal{I}} f(u_i, l_j) + \Box \sum_{i=1}^{n} f(u_i, u_i) \tag{2}$$

Let us show that f^+ is a homomorphism. Prove that f^+ is monotone first. Let $\mathcal{S}_1 = (U,L)$ and $\mathcal{S}_2 = (V,M)$ be two sandwiches such that $\mathcal{S}_1 \sqsubseteq^{\mathbb{B}} \mathcal{S}_2$, that is, $U \sqsubseteq^{\sharp} V$ and $L \sqsubseteq^{\flat} M$. Let $\mathcal{S} = (U,M)$. Observe that $\mathcal{S}$ is a sandwich. Therefore, the proof of $f^+(\mathcal{S}_1) \leq f^+(\mathcal{S}_2)$ is contained in the following two claims.

Claim 1: $f^+(\mathcal{S}_1) \leq f^+(\mathcal{S})$.

Proof of claim 1: If $L = \emptyset$, then claim follows easily from (1), admissibility and equation 4 of mix algebras. For $L \neq \emptyset$, since $L \sqsubseteq^{\flat} M$, there is a sequence of sets $L_0 = L, L_1, \ldots, L_n = M$ such that each $L_i \subseteq L \cup M$ and either $L_{i+1} = \max(L_i \cup l)$ or $L_{i+1} = \max((L_i - L') \cup l)$ where $l' \precsim l$ for all $l' \in L'$, see theorem 4.3. Then each (U, L_i) is a sandwich. We must show $f^+(U, L_i) \leq f^+(U, L_{i+1})$. Consider the first case, i.e. $L_{i+1} = \max(L_i \cup l)$. To verify $f^+(U, L_i) \leq f^+(U, L_{i+1})$ in this case, it is enough to show $\Box f(u,u) + f(u,l) \geq \Box f(u,u)$ if $u \upharpoonleft l$ and, if there is an element $l' \in L$ such that $l' \leq l$, then $f(u',l') + f(u,l) + \Box f(u,u) \geq f(u',l') + \Box f(u,u)$

if $u'\Uparrow l'$. The first one is easy: $\Box f(u,u) + f(u,l) = \Box f(u,l) + f(u,l) = f(u,l) \geq \Box f(u,u)$. The second one follows from monotonicity of $+$: $f(u,l) + \Box f(u,u) \geq \Box f(u,l) = \Box f(u,u)$.

Consider the second case, i.e. $L_{i+1} = \max((L_i - L') \cup l)$. Assume $u\Uparrow l$. Then $u\Uparrow l'$ for any $l' \in L'$. Therefore, any summand $f(u,l)$ in (2) for (U, L_{i+1}) is bigger than $f(u,l')$ in (2) for (U, L_i). Now suppose there is $l' \in L'$ such that $u'\Uparrow l'$ but u' is not consistent with l. If l is consistent with some $u \in U$, then $u\Uparrow l'$. Therefore, to finish the proof of claim 1, we must show that $f(u',l') + f(u,l') \leq f(u,l)$. But this follows from admissibility of f: $f(u',l') + f(u,l) \leq f(u,l') \leq f(u,l)$. Claim 1 is proved.

Claim 2: $f^+(\mathcal{S}) \leq f^+(\mathcal{S}_2)$.

Proof of claim 2: Again, we assume non-emptiness, since for empty sets the proof of claim 2 readily follows from (1). We start with proving the following. Given a sandwich (W,N) and $n \in N$, let w_n be arbitrarily chosen element of W such that $w_n\Uparrow n$. Then, given an admissible function f, $f^+(W,N)$ defined by (2) equals $\sum_{n\in N} f(w_n,n) + \Box \sum_{w\in W} f(w,w)$. To prove this, assume that there are two elements w_1 and w_2 in W consistent with $n \in N$. Then we must show $f(w_1,n) + f(w_2,n) + \Box f(w_1,w_1) + \Box f(w_2,w_2) = f(w_1,n) + \Box f(w_1,w_1) + \Box f(w_2,w_2)$. That the left hand side is less than the right hand side follows from admissibility. On the other hand, $f(w_1,n) + \Box f(w_1,w_1) + \Box f(w_2,w_2) = f(w_1,n) + \Box f(w_2,n) + \Box f(w_1,w_1) + \Box f(w_2,w_2) \leq f(w_1,n) + f(w_2,n) + \Box f(w_1,w_1) + \Box f(w_2,w_2)$ which proves our claim.

Now, to prove claim 2, consider $\mathcal{S}_2 = (V,M)$ and let v_m be an element of V consistent with $m \in M$. Since $U \sqsubseteq^\sharp V$, let u_m be an element of U under v_m. Then $u_m\Uparrow m$. Also, let u^v be an element of U under $v \in V$. Then $\Box \sum_{u\in U} f(u,u) = \Box \sum_{v\in V} f(u^v,u^v) + \Box \sum_{u\neq u^v} f(u,u) \leq \Box \sum_{v\in V} f(u^v,u^v) \leq \Box \sum_{v\in V} f(v,v)$. Now, by the claim proved above, $f^+(\mathcal{S}) = \sum_{m\in M} f(u_m,m) + \Box \sum_{u\in U} f(u,u) \leq \sum_{m\in M} f(v_m,m) + \Box \sum_{v\in V} f(v,v) = f^+(\mathcal{S}_2)$ which finishes the proof of claim 2 and monotonicity of f^+.

Now we demonstrate that f^+ preserves the operations of the signature of the mix algebras. Since $\Box$ distributes over $+$, $\Box f^+(\mathcal{S}) = \sum_{(i,j)\in\mathcal{I}} \Box f(u_i,l_j) + \sum_i \Box f(u_i,u_i)$. Since $\Box f(u_i,l_j) + \Box f(u_i,u_i) = \Box f(u_i,u_i)$, we obtain $\Box f^+(\mathcal{S}) = \sum_{i=1}^n \Box f(u_i,u_i) = f^+(\Box\mathcal{S})$. Moreover, since $\Box e = e$, this also holds when one of components is empty. In addition, $f^+(\emptyset,\emptyset) = e$.

That f^+ is a $+$-homomorphism easily follows from (2) when one of the components is empty. So in the rest of the proof we assume that the second components of all sandwiches are not empty.

Let $\mathcal{S}_1 = (U,L)$, $\mathcal{S}_2 = (V,M)$. Let $\mathcal{S} = \mathcal{S}_1 + \mathcal{S}_2 = (W,N)$. Consider a pair (u_i,l_j) with $(i,j) \in \mathcal{I}$. There are three cases: this pair is either present in the representation (1) of $\mathcal{S}$ or $u_i \gtrsim v_k$ for some $v_k \in V \cap \min(U \cup V)$ or $l_j \precsim m_k \in M \cap \max(L \cup M)$.

Consider the second case. We have $v_k\Uparrow l_j$. Assume $l_j \precsim p$ and $p \in N$. We know that $p\Uparrow q$ for some $q \in W$. Since $f(v_k,l_j) + f(q,p) + \Box f(v_k,v_k) = f(q,p) + \Box f(v,v)$ by 2), we obtain $f^+(\mathcal{S}) = f^+(\mathcal{S}) + f(v_k,l_j)$. Furthermore, since $\Box f(v_k,v_k) + f(u_i,l_j) + f(v_k,l_j) = \Box f(v_k,v_k) + f(v_k,l_j)$

by 1), we have $f^+(\mathcal{S}) = f^+(\mathcal{S}) + f(v_k, l_j) + f(u_i, l_j)$.

Consider the third case. Assume u_i is greater or equal than some $v \in W$ and $m_k \!\uparrow\! q$ for $q \in W$. Then $f(v, l_j) + f(q, m_k) = \Box f(v, v) + f(q, m_k)$ by 3), and hence $f^+(\mathcal{S}) = f^+(\mathcal{S}) + f(v, l_j)$. Since $f(v, l_j) = f(u, l_j) + \Box f(v, v)$ by 4), we obtain $f^+(\mathcal{S}) = f^+(\mathcal{S}) + f(u_i, l_j)$.

Assume that $u \gtrsim v$. Since $\Box f(u, u) + \Box f(v, v) = \Box f(v, v)$ by 5), we obtain $f^+(\mathcal{S}) = f^+(\mathcal{S}) + \Box f(u_i, u_i)$ for any u_i.

All this shows that $f^+(\mathcal{S})$ can be rewritten as $f^+(\mathcal{S}_1) + f^+(\mathcal{S}_2) + X$ where X is a sum of some elements of form $f(u_i, m_j)$ or $f(v_i, l_j)$. Consider a pair (u_i, m_j) such that $u_i \!\uparrow\! m_j$. There exists v_k such that $v_k \!\uparrow\! m_j$. Since $f(v_k, m_j) + \Box f(u_i, u_i) = f(v_k, m_j) + \Box f(u_i, u_i) + f(u_i, m_j)$ by 6), the summand $f(u_i, m_j)$ can be safely removed from X. Thus, any summand can be removed from X and $f^+(\mathcal{S}) = f^+(\mathcal{S}_1) + f^+(\mathcal{S}_2)$. Therefore, f^+ is a homomorphism.

The uniqueness of f^+ follows from (1). Since $f^+(\eta^\dagger(x, x)) = f(x, x) + \Box f(x, x) = f(x, x)$, we have $f^+ \circ \eta^\dagger = f$. The theorem is proved. $\Box$

Universality of $\mathcal{P}^\exists$

For $\mathcal{P}^\exists(A)$, the situation is analogous to mixes. That is, there exists a family of operations Ω such that $\mathcal{P}^\exists(A)$ is the free ordered Ω-algebra generated by A.

Recall that a *left normal band* is an algebra $\langle B, * \rangle$ such that $*$ is associative, idempotent and $x * y * z = x * z * y$, see Romanowska and Smith [149].

Definition 4.7 *An algebra $\langle B, \oplus, * \rangle$ is called a* distributive bi-LNB algebra *if:*
1) $\oplus$ and $*$ are left normal band operations.
2) All distributive laws between $*$ and $\oplus$ hold.
3) $a \oplus (b * c) = a \oplus b$.
4) $(a * b) \oplus b = (b * a) \oplus a$.

Some useful equalities can be derived from 1) - 4). For example, $a * (b \oplus c) = a * b \oplus a * c = a * b \oplus a = a * b \oplus a * b = a * b$ and $a * b \oplus a * c = a * b \oplus a = a * b \oplus a * b = a * b$. It follows immediately from 3) and 4) that $(a * b) \oplus (b * a) = (b * a) \oplus (a * b)$.

We need not include the order in the signature as it is definable.

Lemma 4.27 *In a distributive bi-LNB algebra, $a \leq b := b \oplus a = a * b$ defines a partial order. Moreover, $\oplus$ and $*$ are monotone with respect to $\leq$.*

Proof. First, let us show that $b \oplus a = a * b$ implies $a \oplus b = a$ and $b * a = b$. If $a * b = b \oplus a$, then $b*a = b*a*b = b*(b \oplus a) = b \oplus b*a = b \oplus b = b$. Moreover, $a = a \oplus a = a \oplus a*b = a \oplus b \oplus a = a \oplus b$.

Because of idempotency, $\leq$ is reflexive. To prove transitivity, let $a \leq b$ and $b \leq c$. We must show $a * c = c \oplus a$. Calculate $c \oplus a = c * b \oplus a \oplus b = (c \oplus b) * b \oplus a = b * c * b \oplus a = b * c \oplus a = (b \oplus a) * (c \oplus a) = a * b * c \oplus a * b * a = a * b * c \oplus a = a * b * c \oplus a * b * c = a * b * c$. On the other hand, $a * c = (a \oplus b) * c * b = a * c * b \oplus b * c * b = a * c * b \oplus b = (a \oplus b) * (c \oplus b) b = a * b * c * b = a * b * c$. Hence, $c \oplus a = a * c$ and $a \leq c$. Finally, if $a \leq b$ and $b \leq a$, then $a \oplus b = a$ and $b * a = b$. Hence, $b = b * a = a \oplus b = a$, which finishes the proof that $\leq$ is a partial order.

Assume that $a \leq b$. To see that $a \oplus c \leq b \oplus c$, calculate $(a \oplus c) * (b \oplus c) = a * b \oplus a * c \oplus c * b \oplus c = a * b \oplus a \oplus c = b \oplus a \oplus c = (b \oplus c) \oplus (a \oplus c)$. Similarly, $\oplus$ is monotone in its second argument. To show $a * c \leq b * c$, calculate $a * c \oplus b * c = (a \oplus b) * c = b * a * c = b * c * a * c$. Similarly, $c * a \oplus c * b = c * (a \oplus b) = c * b * a = c * a * c * b$. Hence, $*$ is monotone. □

Thus, we treat distributive bi-LNB algebras as ordered algebras.

We interpret the operations $*$ and $\oplus$ on $\mathcal{P}^{\exists}(A)$ as follows:

$$(U, L) \oplus (V, M) = (\min(U \cup V), L) \qquad (U, L) * (V, M) = (U, \max(L \cup M))$$

Note that under this interpretation $x + y = (x * y) \oplus y$ where the $+$ operation is the one used for mixes and sandwiches. Hence, 4) is just commutativity of $+$.

Theorem 4.28 *$\mathcal{P}^{\exists}(A)$ is the free distributive bi-LND algebra algebra generated by A. That is, for any distributive bi-LND algebra B and any monotone map $f : A \to B$, there exists a unique homomorphism which completes the following diagram:*

$$\begin{tikzcd}
A \arrow[r, hook, "\eta"] \arrow[dr, "f"'] & \langle \mathcal{P}^{\exists}(A), \oplus, * \rangle \arrow[d, dashed, "\exists! f^{+}"] \\
& \langle B, \oplus, * \rangle
\end{tikzcd}$$

Proof. First observe that if $(U, L) \in \mathcal{P}^{\exists}(A)$, then $U, L \neq \emptyset$. We leave it to the reader to find an easy proof that $\mathcal{P}^{\exists}(A)$ satisfies all equations of the distributive bi-LND algebras under the given interpretation of $\oplus$ and $*$ and that $\mathcal{S}_1 \sqsubseteq^{\mathbb{B}} \mathcal{S}_2$ iff $\mathcal{S}_1 * \mathcal{S}_2 = \mathcal{S}_2 \oplus \mathcal{S}_1$. Given $(U, L) \in \mathcal{P}^{\exists}(A)$, we can find $u \in U$ and $l \in L$ such that $u_1 \precsim l_1$. Then, using $\sum$ for repeated applications of $\oplus$, and $\bigotimes$ for repeated applications of $*$, we can see that

$$(U, L) = \textstyle\sum_{u \in U} \eta(u) * \eta(u_1) * \eta(l_1) * \displaystyle\bigotimes_{l \in L} \eta(l)$$

if in the summation over elements of U the first summand is below an element of L. Now, given a monotone f from A into an algebra B, define $f^+ : \mathcal{P}^{\exists}(A) \to B$ as follows:

$$f^+(U, L) = \textstyle\sum_{u \in U} f(u) * f(u_1) * f(l_1) * \bigotimes_{l \in L} f(l)$$

Our first goal is to show that in the above representation any number of expressions of form $f(u') * f(l')$, where $u' \precsim l'$, can be added after $f(u_1) * f(l_1)$. This is indeed correct, as $f(u') \leq f(l')$ implies $f(u') * f(l') = f(l')$ and $f(l')$ is subsumed by $\bigotimes_{l \in L} f(l)$.

Denote $f(u_1) \oplus \ldots \oplus f(u_n)$ by $\tilde{U}$ for $U = \{u_1, \ldots, u_n\}$ and $f(l_1) * \ldots * f(l_k)$ by $\hat{L}$ for $L = \{l_1, \ldots, l_k\}$. Then $f^+((U, L)) = \tilde{U} * f(u_{i_1}) * \ldots * f(u_{i_m}) * \hat{L}$ for any number of u_{i_j}'s which are under some elements of L.

To show that f^+ is well-defined, we must prove that its value does not change if we pick a different first summand in $\tilde{U}$ as long as it is below an element of L. It suffices to prove the following. Let $u_i \leq l_i$, $i = 1, 2$. Then $(f(u_1) \oplus f(u_2)) * \hat{L} = (f(u_2) \oplus f(u_1)) * \hat{L}$. This can be further reduced to proving $(f(u_1) \oplus f(u_2)) * f(l_1) * f(l_2) = (f(u_2) \oplus f(u_1)) * f(l_1) * f(l_2)$. Again, we calculate

$$\begin{array}{rl} (f(u_1) \oplus f(u_2)) * f(l_1) * f(l_2) = f(u_1) * f(l_1) * f(l_2) \oplus f(u_2) * f(l_1) * f(l_2) & = \\ (f(l_1) \oplus f(u_1)) * f(l_2) \oplus (f(l_2) \oplus f(u_2)) * f(l_1) & = \\ f(l_1) * f(l_2) \oplus f(l_2) * f(l_1) \oplus f(u_1) * f(l_2) \oplus f(u_2) * f(l_1) & \end{array}$$

Similarly,

$$(f(u_2) \oplus f(u_1)) * f(l_1) * f(l_2) = f(l_2) * f(l_1) \oplus f(l_1) * f(l_2) \oplus f(u_1) * f(l_2) \oplus f(u_2) * f(l_1)$$

Now the desired equality follows from the equality $(a * b) \oplus (b * a) = (b * a) \oplus (a * b)$ which is true in all bi-LNB algebras.

Our next goal is to show that any number of nonminimal elements can be added to U and any number of nonmaximal elements can be added to L and that it does not change the value of f^+. That is, writing expressions for f^+ we may disregard min and max operations.

Assume that $u \precsim u'$ and u' is added to U. There are two cases. If $f(u')$ is not the first summand in $\widetilde{U \cup u'}$, then $f(u) \oplus f(u') = f(u)$, so we may disregard $f(u')$. It is also possible that $f(u')$ can be used in the expression for f^+ between $\tilde{U}$ and $\hat{L}$, in which case it can also be disregarded as, if it is below some l, then $f(u') * f(l) = f(l)$. Finally, consider the case when $f(u')$ is the first summand. It is only possible if $u \precsim u' \precsim l$ for some $l \in L$. To prove that $f(u')$ can be dropped and replaced by $f(u)$ in this case, we must show $(f(u') \oplus f(u)) * f(l) = f(u) * f(l)$. Since $f(u) \leq f(u')$ and $f(u') \oplus f(u) = f(u) * f(u')$, we obtain $(f(u') \oplus f(u)) * f(l) = f(u) * f(u') * f(l) = f(u) * f(l) * f(u') = f(u) * f(l)$.

If $l' \precsim l$ is added to L, $f(l')$ does not change the value of f^+ as $f(l) * f(l') = f(l)$. Therefore, we may disregard all max and min operations in expressions for f^+.

At this point we are ready to show that f^+ is a homomorphism. Its uniqueness will follow from the representation of elements of $\mathcal{P}^{\exists}(A)$ from singletons and well-definedness of f^+. Let $\mathcal{S}_1 = (U, L)$ and $\mathcal{S}_2 = (V, M)$. Let $u_1 \precsim l_1$ and $v_1 \precsim m_1$ for $u_1 \in U, l_1 \in L, v_1 \in V, m_1 \in M$. Then $f^+(\mathcal{S}_1) * f^+(\mathcal{S}_2) = \sum_{v \in V}(f^+(\mathcal{S}_1) * f(v) * f(v_1) * \hat{M})$. For two v_i and v_j, consider $f^+(\mathcal{S}_1) * f(v_i) * f(v_1) * \hat{M}$ and $f^+(\mathcal{S}_1) * f(v_j) * f(v_1) * \hat{M}$. Since $L \neq \emptyset$, they are the same, because $a * b \oplus a * c = a * b$ is a derivable equality. Hence, $f^+(\mathcal{S}_1) * f^+(\mathcal{S}_2) = f^+(\mathcal{S}_1) * f(v_1) * \hat{M}$. Since $v_1 \precsim m_1$, we have $f(m_1) * f(v_1) = f(m_1)$ and hence $x * f(v_1) * \hat{M} = x * \hat{M}$ for any x. Thus, $f^+(\mathcal{S}_1) * f^+(\mathcal{S}_2) = \tilde{U} * f(u_1) * \hat{L} * \hat{M} = \tilde{U} * f(u_1) * \widehat{L \cup M} = f^+(\mathcal{S}_1 * \mathcal{S}_2)$. Therefore, f^+ is a $*$-homomorphism.

Now consider $f^+(\mathcal{S}_1) \oplus f^+(\mathcal{S}_2)$. From the equational theory, we immediately have $f^+(\mathcal{S}_1) \oplus f^+(\mathcal{S}_2) = (\tilde{U} * f(u_1) * \hat{L}) \oplus \tilde{V}$. Furthermore, since $(a \oplus c) * b = a * b \oplus c * b = a * b \oplus c$, we have $f^+(\mathcal{S}_1) \oplus f^+(\mathcal{S}_2) = (\tilde{U} \oplus \tilde{V}) * f(u_1) * \hat{L} = \widetilde{U \cup V} * f(u_1) * \hat{L} = f^+(\mathcal{S}_1) \oplus f^+(\mathcal{S}_2)$. Thus, f^+ is a homomorphism. Theorem is proved. $\square$

Universality of $\mathcal{P}^{\emptyset}$

Recall that $\mathcal{P}^{\emptyset}(A)$ is the poset of pairs of finite antichains (U, L) ordered by $\sqsubseteq^{\mathbb{B}}$. Hence, it is isomorphic to the direct product of $\mathcal{P}^{\flat}(A)$ and $\mathcal{P}^{\sharp}(A)$, each of them being a free construction. A product of free algebras is not necessarily a free algebra. However, for the case of $\mathcal{P}^{\emptyset}(A)$ we exhibit a simple way of combining mixes with their "dual" algebras to obtain the universality property for $\mathcal{P}^{\emptyset}(A)$.

Definition 4.8 *An algebra $\langle B, +, \Box, \Diamond \rangle$ is called a* bi-mix *algebra if $\langle B, +, \Box \rangle$ is a mix algebra (see definition 4.5), $x = \Box x + \Diamond x$ and $\langle B, +, \Diamond \rangle$ is a dual mix algebra. By this we mean that $\Diamond$ is a closure, that is, $\Diamond$ is monotone, $\Diamond x \geq x$, $\Diamond\Diamond x = \Diamond x$ and $\Diamond(x + y) = \Diamond x + \Diamond y$, and in addition $x + \Diamond x = x$ and $x + \Diamond y \geq x$.*

We give $\mathcal{P}^{\emptyset}(A)$ the structure of a bi-mix algebra by interpreting $+$ and $\Box$ in the same way as for $\mathcal{P}^{\forall}(A)$ and by putting $\Diamond(U, L) = (\emptyset, L)$.

Theorem 4.29 *$\mathcal{P}^{\emptyset}(A)$ is the free bi-mix algebra generated by A. That is, for any bi-mix algebra B and any monotone map $f : A \to B$, there exists a unique homomorphism which completes the following diagram:*

$$\begin{tikzcd} A \arrow[r, hook, "\eta"] \arrow[dr, "f"'] & \langle \mathcal{P}^{\emptyset}(A), +, \Box, \Diamond \rangle \arrow[d, dashed, "\exists! f^{+}"] \\ & \langle B, +, \Box, \Diamond \rangle \end{tikzcd}$$

Proof. It is easy to check that $\mathcal{P}^{\emptyset}(A)$ is a bi-mix algebra under the given interpretation of the operations. To prove freeness, we first need a few facts about bi-mix algebras.

Let $e = \Box\Diamond x$. We have $y + \Diamond x \geq y$ and hence by monotonicity $\Box y + e \geq \Box y$. Adding $\Diamond y$ to both sides, we get by monotonicity that $\Diamond y + \Box y + e \geq \Diamond y + \Box y$ and hence $y \geq y + e \geq y$ which proves that e is the identity of $+$. Similarly, if we define $e' = \Diamond\Box x$, then e' is the identity of $+$ and therefore $e = e'$. This shows that the identity of $+$ can be correctly defined as $e = \Box\Diamond x = \Diamond\Box y$ for arbitrary x and y. Since $x \geq \Box x$, we have $\Diamond x \geq \Diamond\Box x = e$. Similarly, $\Box x \leq e$. It is also easy to see that $\Box e = \Diamond e = e$.

Now, given $(U, L) \in \mathcal{P}^{\emptyset}(A)$, observe that

$$(U, L) \quad = \quad \Box \sum_{u \in U} \eta(u) \; + \; \Diamond \sum_{l \in L} \eta(l)$$

As usual, summation over $\emptyset$ is assumed to be e. Then, given $f : A \to B$, define $f^{+} : \mathcal{P}^{\emptyset}(A) \to B$ as follows:

$$f^{+}((U, L)) \quad = \quad \Box \sum_{u \in U} f(u) \; + \; \Diamond \sum_{l \in L} f(l)$$

First, $f^{+}(\eta(x)) = f^{+}((x, x)) = \Box f(x) + \Diamond f(x) = f(x)$ and hence $f^{+} \circ \eta = f$. Now we are going to show that f^{+} is a homomorphism. Its uniqueness will then follow from the representation of elements of $\mathcal{P}^{\emptyset}(A)$ given above.

Before we show that f^{+} is monotone, let us check that the value of f^{+} does not change if an element $l' \precsim l \in L$ is added to L or an element $u' \succsim u \in U$ is added to U. Indeed, to prove the former, observe that $f(l') \leq f(l)$ and $\Diamond f(l') + \Diamond f(l) \leq \Diamond f(l) + \Diamond f(l) = \Diamond f(l)$. For the latter, $\Box f(u) \geq \Box f(u) + \Box f(u') \geq \Box f(u)$ and hence $\Box f(u) + \Box f(u') = \Box f(u)$.

To show that f^{+} is monotone, observe that if $(U, L) \sqsubseteq^{\mathbb{B}} (V, M)$, then $U \sqsubseteq_a^{\mathrm{or}} V$ and $L \sqsubseteq_a^{\mathrm{OWA}} M$ and hence V can be obtained from U by a sequence of updates described in theorem 4.5 and M can be obtained from L by a sequence of updates described in theorem 4.3. It is easy to see that updates that replace an element by a number of bigger elements are monotone. Consider removing an element u from U. If $U = \{u\}$, then $\sum_{u' \in \emptyset} \Box f(u') = e \geq \Box f(u)$. If $u' \in U - \{u\}$, then $\Box f(u') \geq \Box f(u') + \Box f(u)$ which proves monotonicity in this case. Finally, if $L = \emptyset$ and an element is l added, then $\Diamond f(l) \geq \sum_{l' \in \emptyset} \Diamond f(l') = e$. If $l \in L$ and l' is added, we have monotonicity because $\Diamond f(l) + \Diamond f(l') \geq \Diamond f(l)$.

Now we are ready to prove that f^+ is preserves $+$, $\Box$ and $\Diamond$. First, $\Box f^+((U, L)) = \Box\Box \sum_{u \in U} f(u) + \Box\Diamond \sum_{l \in L} f(l) = \Box \sum_{u \in U} f(u) + e = f^+(\Box(U, L))$. Similarly, $\Diamond f^+((U, L)) = f^+(\Diamond(U, L))$. The fact that $+$ is preserved follows immediately from the definition of f^+ and the observation that nonminimal elements in U and nonmaximal elements in L do not affect the value of f^+. $\Box$

Universality of $\mathcal{P}^{\forall}$ (snacks)

Snacks were first introduced by Buneman and then studied by Ngair in his dissertation [121]. Later they were characterized by Puhlmann [141] as free distributive bisemilattices [61, 134]. Since Pulhmann's proof is not very complicated and since it exploits an unusual presentation of the equational theory, for the sake of completeness we prove the characterization theorem here. We then shall demonstrate the connection between snacks and theory of Płonka's sums of algebras [149, 135].

First observe that the ordering $\sqsubseteq_f^{\mathbb{B}}$ gives $\mathcal{P}^{\forall}(A)$ the structure of a meet-semilattice [141] where

$$(U, \mathcal{L}) \wedge (V, \mathcal{M}) = (\min(U \cup V), \max^{\sharp}\{\min(L \cup M) \mid L \in \mathcal{L}, M \in \mathcal{M}\})$$

Definition 4.9 (see [61, 134]). *A bisemilattice is an algebra $\langle B, +, \cdot \rangle$ such that $+$ and $\cdot$ are semilattice operations. A bisemilattice B is called* distributive *if both distributive laws hold, that is: $x(y+z) = xy + xz$ and $x + yz = (x+y)(x+z)$. (For convenience, we often omit $\cdot$ in formulas and equations.)*

When we speak of *the* ordering on a bisemilattice B, we mean the ordering associated with $\cdot$, that is, $x \leq y$ iff $xy = x$.

$\mathcal{P}^{\forall}(A)$ can be given the structure of distributive bisemilattice by making $\cdot$ to be the greatest lower bound operation above and by defining $+$ as

$$(U, \mathcal{L}) + (V, \mathcal{M}) = (\min(U \cup V), \max^{\sharp}(\mathcal{L} \cup \mathcal{M}))$$

Observe that the empty snack $e = (\emptyset, \emptyset)$ is the identity for $+$.

Definition 4.10 *A* snack algebra *is a distributive bisemilattice in which $+$ has identity e.*

A homomorphism of snack algebras is a homomorphism in the usual algebraic sense. In other words, there is no need to require monotonicity as we did for mixes, because it is implied: if $x \leq y$, then $h(x) \cdot h(y) = h(x \cdot y) = h(x)$ and $h(x) \leq h(y)$.

Theorem 4.30 *Given a poset A, $\mathcal{P}^{\forall}(A)$ is the free snack algebra generated by A. That is, for any snack algebra Sn and a monotone map $f : A \to Sn$, there exists a unique snack homomorphism $f^{+} : \mathcal{P}^{\forall}(A) \to Sn$ such that the following diagram commutes:*

$$\begin{tikzcd}
A \arrow[r, hook, "\eta"] \arrow[dr, "f"'] & \langle \mathcal{P}^{\forall}(A), +, \cdot, e \rangle \arrow[d, dashed, "\exists! f^{+}"] \\
& \langle Sn, +, \cdot, e \rangle
\end{tikzcd}$$

Proof. We omit verification that $\mathcal{P}^{\forall}(A)$ is a snack algebra (in fact, the distributivity laws will be verified later in the greater generality).

Given a snack $\mathcal{S} = (U, \mathcal{L})$ where $U = \{u_1, \ldots, u_n\}$ and $\mathcal{L} = \{L_1, \ldots, L_k\}$, $L_i = \{l_1^i, \ldots, l_{k_i}^i\}$, we have

$$\mathcal{S} = (\prod_{i=1}^{n} \eta(u_i))e + \sum_{i=1}^{k} \prod_{j=1}^{k_i} \eta(l_j^i) \tag{3}$$

Then, if monotone $f : A \to Sn$ is given, define $f^{+} : \mathcal{P}^{\forall}(A) \to Sn$ by

$$f^{+}(\mathcal{S}) = (\prod_{i=1}^{n} f(u_i))e + \sum_{i=1}^{k} \prod_{j=1}^{k_i} f(l_j^i) \tag{4}$$

Clearly, $f^{+}(\emptyset, \emptyset) = e$ and $f^{+}(\eta(x)) = f(x) \cdot e + f(x) = f(x)$. We must show that f^{+} is a homomorphism.

We start with a few easy observations. First, notice that for a snack algebra $+$ is monotone with respect to $\leq$. Indeed, take $b \geq c$ and observe that $(a+b)(a+c) = a + bc = a + c$, hence $a + b \geq a + c$. Let us now take three elements $a \leq b \leq c$. We have: $ae + c \leq ae + ae + c \leq ae + b + c \leq ae + c + c = ea + c$. Hence, $ae + b + c = ae + c$. Furthermore, consider arbitrary a and b. Since $abe(a+b) = abe$, we have $abe \leq (a+b)e$. On the other hand, $ae + be$ is below a, b and e, and hence $ae + be \leq abe$. Thus, $abe = (a+b)e$.

Let $x \precsim y$ in A. Then $f(x) \leq f(y)$ and hence $f(x) \cdot f(y) = f(x)$. Therefore, if X and Y are two finite subsets of A equivalent with respect to $\sqsubseteq^{\sharp}$, then $\prod_{x \in X} f(x) = \prod_{y \in Y} f(y)$.

Furthermore, assume $U \sqsubseteq^{\sharp} X \sqsubseteq^{\sharp} Y$ for $U, X, Y \in \mathbb{P}_{\mathrm{fin}}(A)$. Then we have $\prod_{u \in U} f(u) \cdot e \leq \prod_{x \in X} f(x) \leq \prod_{y \in Y} f(y)$ and therefore $\prod_{u \in U} f(u) \cdot e + \prod_{x \in X} f(x) + \prod_{y \in Y} f(y) = \prod_{u \in U} f(u) \cdot e + \prod_{y \in Y} f(y)$. This observation shows that writing an expression for $f^{+}(\mathcal{S}_1 + \mathcal{S}_2)$ and $f^{+}(\mathcal{S}_1 \cdot \mathcal{S}_2)$

one may disregard all max and min operations. That is, for $\mathcal{S}_1 = (U, \mathcal{L})$ and $\mathcal{S}_2 = (V, \mathcal{M})$,

$$f^+(\mathcal{S}_1 + \mathcal{S}_2) = \prod_{u \in U} f(u) \cdot \prod_{v \in V} f(v) \cdot e + \sum_{L \in \mathcal{L}} \prod_{l \in L} f(l) + \sum_{M \in \mathcal{M}} \prod_{m \in M} f(m) \tag{5}$$

$$f^+(\mathcal{S}_1 \cdot \mathcal{S}_2) = \prod_{u \in U} f(u) \cdot \prod_{v \in V} f(v) \cdot e + \sum_{L \in \mathcal{L}, M \in \mathcal{M}} \prod_{l \in L} f(l) \cdot \prod_{m \in M} f(m) \tag{6}$$

That $f^+(\mathcal{S}_1 + \mathcal{S}_2) = f^+(\mathcal{S}_1) + f^+(\mathcal{S}_2)$ follows immediately from (5).

Let us denote $\prod_{x \in X}$ by $\tilde{X}$. Then $f^+(\mathcal{S}_1 \cdot \mathcal{S}_2) = \tilde{U}\tilde{V}e + \tilde{U}e \cdot \sum_M \tilde{M} + \tilde{V}e \cdot \sum_L \tilde{L} + \sum_L \tilde{L} \cdot \sum_M \tilde{M}$. The last summand is easily seen to be $\sum_{L,M} \tilde{L} \cdot \tilde{M}$. Since $\sum_M \tilde{M} \geq \tilde{V}$, the last summand is also greater than $\tilde{V}e \cdot \sum_L \tilde{L}$ which can therefore be dropped. Similarly, $\tilde{U}e \cdot \sum_M \tilde{M}$ can be dropped. Thus, $f^+(\mathcal{S}_1 \cdot \mathcal{S}_2) = f^+(\mathcal{S}_1) \cdot f^+(\mathcal{S}_2)$ which shows that f^+ is a homomorphism. Its uniqueness follows from (3). $\square$

We now show that a particular case of this theorem (when A is a discrete order) is well known. If A is discrete, then any subset of A is an antichain, and the consistency condition says that $L_i \subseteq U$ for any snack $(U, \{L_1, \ldots, L_k\})$. To redefine bisemilattice operations, simply remove all min's and replace $\max^\sharp$ with $\min^\subseteq$.

Fix $U \subseteq A$ and consider $\mathbb{L}_U = \{\mathcal{L} \mid (U, \mathcal{L}) \text{ is a snack}\}$. Then it is easy to see that $\langle \mathbb{L}_U, +, \cdot \rangle$ is the free distributive lattice generated by U. It shows that $\mathcal{P}^\forall(A)$ is what is known in universal algebra as the Płonka sum of free distributive lattices over the semilattice $\langle \mathbb{P}_{\text{fin}}(A), \cup \rangle$ which itself is the free semilattice generated by A. Now, the result of Płonka [134] tells us that such construction is isomorphic to a free distributive bisemilattice. Thus, we have shown how to extend the result of [134] to arbitrary generated posets and how to include the identity constant into the signature.

Universality of $\mathcal{P}^{\forall\wedge}$

The difference between elements of $\mathcal{P}^\forall(A)$ and $\mathcal{P}^{\forall\wedge}(A)$ is that in the latter the consistency condition is similar to that in sandwiches: for any $(U, \mathcal{L})$ in $\mathcal{P}^{\forall\wedge}(A)$, and any $L \in \mathcal{L}$, there exists W such that $L \sqsubseteq^\flat W$ and $U \sqsubseteq^\sharp W$. Our goal is to show that $\mathcal{P}^{\forall\wedge}(A)$ can *not* be described as a free ordered algebra generated by A. Recall that we defined $+$ as an operation on snacks by $(U, \mathcal{L}) + (V, \mathcal{M}) = (\min(U \cup V), \max^\sharp(\mathcal{L} \cup \mathcal{M}))$. It is easy to see that elements of $\mathcal{P}^{\forall\wedge}(A)$ are closed under this operation as well.

Theorem 4.31 *Let Ω_+ be a set of operations on elements of $\mathcal{P}^{\forall\wedge}(A)$ such that $+$ is a derived operation. Then $\mathcal{P}^{\forall\wedge}(\cdot)$ is not left adjoint to the forgetful functor from the category of ordered*

Ω_+-algebras to **Poset**. *In other words, for no Ω_+ is $\mathcal{P}^{\forall\wedge}(A)$ the free ordered Ω_+-algebra generated by A.*

Proof. Assume that there exists a set of operation Ω_+ such that $\mathcal{P}^{\forall\wedge}(A)$ the free ordered Ω-algebra generated by A for any poset A and $+$ is a derived operation. Let $A = \{x, y, z\}$ be an antichain and $A' = \{x', y', z'\}$ be a poset such that $x', y' \precsim z'$ and $x' \not\precsim y'$, $y' \not\precsim x'$. Let $f : A \to \mathcal{P}^{\forall\wedge}(A')$ be defined by $f(a) = (a', a'), a \in A$. Now the assumed universality property tells us that f can be extended to a monotone Ω_+-homomorphism $f^+ : \mathcal{P}^{\forall\wedge}(A) \to \mathcal{P}^{\forall\wedge}(A')$. Let $\mathcal{S} \in \mathcal{P}^{\forall\wedge}(A')$. Since $\mathcal{P}^{\forall\wedge}(A')$ is the free Ω_+-algebra generated by A', we can find a term t in the signature Ω_+ such that $\mathcal{S} = t(\eta(x'), \eta(y'), \eta(z'))$. Since $\eta(x') = f(x) = f^+(\eta(x))$ and similarly for y' and z', we obtain $\mathcal{S} = f^+(t(\eta(x), \eta(y), \eta(z))) = f^+(\mathcal{S}_0)$ for some $\mathcal{S}_0 \in \mathcal{P}^{\forall\wedge}(A)$. Therefore, f^+ is an onto $+$-homomorphism.

Using the fact that f^+ is a $+$-homomorphism, we find $f^+((xy, \{x, y\})) = f^+((x, x) + (y, y)) = (x', x') + (y', y') = (x'y', \{x', y'\})$ and $f^+((xz, \{x, z\})) = f^+((x, x) + (z, z)) = (x', x') + (z', z') = (x', z')$. Similarly, $f^+((yz, \{y, z\})) = (y', z')$. Define

$$\begin{aligned} \mathcal{P}_0^{\forall\wedge}(A) &= \mathcal{P}^{\forall\wedge}(A) - \downarrow\{(x, x), (y, y), (xy, \{x, y\}), (xz, \{x, z\}), (yz, \{y, z\})\} \quad \text{and} \\ \mathcal{P}_0^{\forall\wedge}(A') &= \mathcal{P}^{\forall\wedge}(A') - \downarrow\{(x', x'), (y', y'), (x'y', \{x', y'\}), (x', z'), (y', z')\} \end{aligned}$$

Since f^+ maps $\mathcal{P}^{\forall\wedge}(A) - \mathcal{P}_0^{\forall\wedge}(A)$ into $\mathcal{P}^{\forall\wedge}(A') - \mathcal{P}_0^{\forall\wedge}(A')$, there must be an onto map from a subset of $\mathcal{P}_0^{\forall\wedge}(A)$ onto $\mathcal{P}_0^{\forall\wedge}(A')$. Now we can find that $\mathcal{P}_0^{\forall\wedge}(A) = \{(xyz, \{x, y, z\}), (z, z), (z, \emptyset)\}$ and $\mathcal{P}_0^{\forall\wedge}(A') = \{(z', z'), (z', \{x', y'\}), (z', x'), (z', y'), (z', x'y'), (z', \emptyset), (x'y', z')\}$. Therefore, there is no map from a subset of $\mathcal{P}_0^{\forall\wedge}(A)$ onto $\mathcal{P}_0^{\forall\wedge}(A')$. This contradiction proves the theorem. $\square$

Universality of $\mathcal{P}^{\exists}$

The consistency condition for $(U, \mathcal{L}) \in \mathcal{P}^{\exists}(A)$ says that $\uparrow U \cap L \neq \emptyset$ for every $L \in \mathcal{L}$. Therefore, $\mathcal{P}^{\exists}(A)$ is closed under $+$ defined, as usual, by $(U, \mathcal{L}) + (V, \mathcal{M}) = (\min(U \cup V), \max^{\sharp}(\mathcal{L} \cup \mathcal{M}))$. Our goal is to show that $\mathcal{P}^{\exists}(A)$ can *not* be described as a free ordered algebra generated by A. This is more surprising than similar results for approximation constructs using $u \| l$ in the consistency condition. Here no information about consistency is needed, but we still can not find a free algebra characterization.

Theorem 4.32 *Let Ω_+ be a set of operations on elements of $\mathcal{P}^{\exists}(A)$ such that $+$ is a derived operation. Then $\mathcal{P}^{\exists}(\cdot)$ is not left adjoint to the forgetful functor from the category of ordered Ω_+-algebras to* **Poset**. *In other words, for no Ω_+ is $\mathcal{P}^{\exists}(A)$ the free ordered Ω_+-algebra generated by A.*

Proof. Consider two posets: $A = \{x, y, z\}$ and $A' = \{x', y', z'\}$. In A, $x, y \precsim z$ and x and y are incomparable. A' is a chain: $x' \precsim y' \precsim z'$. Define $f : A \to A'$ by $f(x) = x', f(y) = y'$ and $f(z) = z'$. Clearly, f is monotone.

Assume that there exists a signature Ω_+ such that for any poset B, $\langle \mathcal{P}^{\exists}(B), \Omega_+\rangle$ is the free Ω_+ algebra generated by B. Then we would have a monotone $+$-homomorphism $f^+ : \mathcal{P}^{\exists}(A) \to \mathcal{P}^{\exists}(A')$ such that $f^+((x,x)) = (x',x'), f^+((y,y)) = (y',y')$ and $f^+((z,z)) = (z',z')$. Then we have $f^+((xy,\{x,y\})) = f^+((x,x)+(y,y)) = (x',x')+(y',y') = (x',y')$ and $f^+((y,z)) = f^+((y,y)+(z,z)) = (y',y')+(z',z') = (y',z')$.

Since f^+ is monotone and $(x,xy) \leq (x,x)$, we obtain $f^+((x,xy)) = (x',x')$. Similarly, $f^+((xy, xy)) = (x',x')$. Then $(x',x') = f^+((xy,xy)) = f^+((x,xy)+(y,xy)) = (x',x')+f^+((y,xy))$. Since $(y,xy) \leq (y,y)$, $f^+((y,xy))$ can be either (y',y') or (x',y') or (x',x'). The equality above then tells us that $f^+((y,xy)) = (x',x')$.

Now we use these values of f^+ to calculate $(y',z') = f^+((y,z)) = f^+((y,xy)+(y,z)) = f^+((y,xy))+f^+((y,z)) = (x',x')+(y',z') = (x',z')$. This contradiction shows that $f : A \to A'$ can not be extended to a monotone $+$-homomorphism between $\mathcal{P}^{\exists}(A)$ and $\mathcal{P}^{\exists}(A')$ and hence $\mathcal{P}^{\exists}(A)$ is not a free Ω_+-algebra generated by A. $\square$

Universality of $\mathcal{P}^{\exists\curlywedge}$ (scones)

Scones were introduced recently by Jung and a few initial results were proved by Puhlmann [141]. For example, it was shown that scones preserve bounded completeness and distributivity, while snacks preserve the former but not the latter.

If $x, y \in A$ and $x \uparrow y$, then $(x, \{y\})$ is a scone. Thus, we have the same problem as we had with sandwiches: it is no longer enough to start with A itself as a generating poset if we want to represent scones as a free construction. That is, some information about consistency must be incorporated into the generating poset. As we did in the case of sandwiches, we use $A \uparrow A$ as the generating poset.

Let us now describe the algebra. Recall that a left normal band is an algebra $\langle B, *\rangle$ where $*$ is idempotent, associative and $x*y*z = x*z*y$, see Romanowska and Smith [61, 149].

Definition 4.11 *A* scone *algebra is an algebra $\langle Sc, +, *, e\rangle$ where $+$ is a semilattice operation with identity e, $*$ is a left normal band operation, $+$ and $*$ distribute over each other, the absorption laws hold and $e*x = e$. Formally, in addition to $*$ being left normal band and $+$ being semilattice operation, the following hold:*
1) $x + y*z = (x+y)*(x+z)$;
2) $(x+y)*z = x*z + y*z$;
3) $z*(x+y) = z*x + z*y$;
4) $x + x*y = x$;
5) $e + x = x + e = x$;
6) $e*x = e$.

In other words, a scone algebra is "almost distributive lattice" – commutativity of one of the operations is replaced by the law of the left normal bands. Scone algebras are known as idempotent distributive semirings with semilattice and left zero bands reducts. There is no known characterization of such algebras we could benefit from (though the characterization of free idempotent distributive semirings with semilattice reducts is known, see [147]).

If Sc is a scone algebra, define $x \cdot y = x * y + y * x$. It is an easy observation that $\cdot$ is a semilattice operation. An ordering on Sc is defined according to this operation, that is, $x \leq y \Leftrightarrow xy = x$. Similarly to the case of snacks, this implies monotonicity of any homomorphism.

To give $\mathcal{P}^{\exists\wedge}(A)$ the structure of a scone algebra we must show how to define $+$ and $*$. The $+$ operation is defined as for snacks, and

$$(U, \mathcal{L}) * (V, \mathcal{M}) = (U, \max{}^{\sharp}\{\min(L \cup M) \mid L \in \mathcal{L}, M \in \mathcal{M}\})$$

It is easy to check that $(U, \mathcal{L}) * (V, \mathcal{M})$ satisfies the consistency condition. e is the empty scone $(\emptyset, \emptyset)$. Similarly to the case of sandwiches, a definition of admissibility is needed to preserve the additional structure given by the consistent closure of A.

Definition. *Let $\langle Sc, +, *, e\rangle$ be a scone algebra. A monotone map $f : A{\uparrow}A \to Sc$ is called* admissible *if $f(u, l) * f(v, m) = f(u, m) * f(w, l)$ and $f(u, l) * e = f(u, m) * e$.*

Theorem 4.33 *Given a poset A, $\mathcal{P}^{\exists\wedge}(A)$ is the free scone algebra generated by $A{\uparrow}A$ with respect to admissible maps. That is, for any scone algebra Sc and an admissible map $f : A{\uparrow}A \to Sc$, there exists a unique scone homomorphism $f^+ : \mathcal{P}^{\exists\wedge}(A) \to Sc$ which completes the following diagram:*

$$\begin{tikzcd}
A{\uparrow}A \arrow[r, hook, "\eta^{\uparrow}"] \arrow[dr, "f"'] & \langle \mathcal{P}^{\exists\wedge}(A), +, *, e\rangle \arrow[d, dashed, "\exists! f^+"] \\
& \langle Sc, +, *, e\rangle
\end{tikzcd}$$

Proof. We shall verify the distributivity laws in the proof of algebraic characterization of the salads in the next subsection. Distributivity laws for scones then follow from the observation that the second components of $(U, \mathcal{L}) \cdot (V, \mathcal{M})$ and $(U, \mathcal{L}) * (V, \mathcal{M})$ coincide. Equation 4) is immediate. Thus, $\mathcal{P}^{\exists\wedge}(A)$ is a scone algebra.

We now need some observations about the scone algebras. In what follows, f is an admissible map from $A{\uparrow}A$ to a scone algebra Sc. The definition of admissibility can be rewritten to $f(u, l) * f(v, m) = f(u, l) * f(w, m) = f(u, m) * f(v, l)$.

1) $+$ is monotone with respect to the ordering given by $\cdot$.

Let $b \leq a$. Then $(a+c)(b+c) = (a+c)*(b+c)+(b+c)*(a+c) = c+a*b+b*a = c+ab = b+c$, i.e. $b+c \leq a+c$.

2) $\cdot$ distributes over $+$.

$x(y+z) = x*(y+z)+(y+z)*x = x*y+y*x+x*z+z*x = xy+xz$.

3) If $a \leq b$, then $a*e \leq b*e$.

$(a*e)\cdot(b*e) = a*b*e+b*a*e = (a*b+b*a)*e = (ab)*e = a*e$.

4) $f(x,y)+f(z,y) \leq f(x,y)$.

$f(x,y)+f(z,y)\cdot f(x,y) = (f(x,y)+f(z,y))*f(x,y)+f(x,y)*(f(x,y)+f(z,y)) = (f(x,y)+f(x,y)*f(z,y))+f(z,y)*f(x,y) = f(x,y)+f(z,y)*f(z,y) = f(x,y)+f(z,y)$.

5) If $a \precsim b$, then $f(a,a)*e+f(b,b)*e = f(a,a)*e$.

First of all, $f(a,a)*e+f(b,b)*e = f(a,a)*e+f(b,a)*e = (f(a,a)+f(b,a))*e \leq f(a,a)*e$ by 3) and 4). Furthermore, $f(a,a) = f(a,a)+f(a,a) \leq f(a,a)+f(b,b)$ by 1) and therefore $f(a,a)*e \leq (f(a,a)+f(b,b))*e$ which finishes the proof.

6) If $a \precsim b$ and $b \Uparrow x$, then $f(x,a)*f(b,b) = f(x,a)$.

We have $f(x,a)*f(b,b) = f(x,a)*f(x,b) = f(x,b)*f(x,a)$. Hence $f(x,a)*f(b,b) = f(x,a)*f(x,b)+f(x,b)*f(x,a) = f(x,a)\cdot f(x,b) = f(x,a)$ because $f(x,a) \leq f(x,b)$.

7) For any $a \Uparrow b$, $f(a,b)*f(b,a) \leq f(a,b)$.

It is easy to see that $(f(a,b)*f(b,a))\cdot f(a,b) = f(a,b)*f(b,a)$.

8) If $a \precsim b$, then $f(b,b)*f(a,a) = f(b,a)$.

By admissibility and 7), $f(b,b)*f(a,a) = f(b,a)*f(a,b) \leq f(b,a)$. On the other hand, $f(b,a)\cdot(f(b,b)*f(a,a)) = f(b,a)*f(b,b)*f(a,a)+f(b,b)*f(a,a)*f(b,a) = f(b,a)*f(b,b)*f(b,a)+f(b,b)*f(b,a)*f(b,a) = f(b,a)*f(b,b)+f(b,b)*f(b,a) = f(b,a)\cdot f(b,b) = f(b,a)$. Hence, $f(b,a) \leq f(b,b)*f(a,a)$ which proves 8).

Since $\prod$ is already used to denote repeated applications of $\cdot$, for many applications of $*$ we shall use $\otimes$.

Let $\mathcal{S} = (U,\mathcal{L})$ be a scone over A. Since $\uparrow U \cap \uparrow L \neq \emptyset$ for all $L \in \mathcal{L}$, there exists a pair $(u_i, l^j_{k_i})$ for every j such that $u_i \Uparrow l^j_{k_i}$. Let $i(j)$ and $k(j)$ be some indices such that $u_{i(j)} \Uparrow l^j_{k(j)}$. Then $\mathcal{S}$ can be represented as

$$\mathcal{S} = \sum_{u \in U} \eta^{\uparrow}(u,u) * e + \sum_{L_j \in \mathcal{L}} (\eta^{\uparrow}(u_{i(j)}, l^j_{k(j)}) * \bigotimes_{l \in L_j} \eta^{\uparrow}(l,l)) \tag{7}$$

Recall that summation over $\emptyset$ is the identity. We will never need product over the empty index set for all antichains in the second component are nonempty. Moreover, observe that in (7) it does not matter how pairs $(i(j), k(j))$ are chosen.

Using (7), define

$$f^+(\mathcal{S}) = \sum_{u \in U} f(u,u) * e + \sum_{L_j \in \mathcal{L}} (f(u_{i(j)}, l^j_{k(j)}) * \bigotimes_{l \in L_j} f(l,l)) \tag{8}$$

Our first goal is to verify that f^+ is well-defined, that is, it does not depend on how pairs $i(j), k(j)$ are chosen. To save space, denote $\bigotimes_{l \in L} f(l,l)$ by $\hat{L}$. First observe that any number of applications of f to a consistent pair (u,l) for $l \in L_j$ can be put after $f(u_{i(j)}, l^j_{k(j)})$ because, by admissibility, $f(u_{i(j)}, l^j_{k(j)}) * f(u,l) = f(u_{i(j)}, l^j_{k(j)}) * f(l,l)$ and $*$ is idempotent. To finish the proof of well-definedness, it is enough to show that the following equation holds: $f(u,u) * e + f(u',u') * e + f(u,l) * \hat{L} = f(u,u) * e + f(u',u') * e + f(u',l') * \hat{L}$ where $u, u' \in U$ and $l, l' \in L$. By distributivity, this reduces to showing that $f(u,u) * e + f(u',u') * e + f(u,l) * f(l',l') = f(u,u) * e + f(u',u') * e + f(u',l') * f(l,l)$. Because of the symmetry in this equation, it is enough to prove

$$f(u,u) * e + f(u',u') * e + f(u,l) * f(l',l') \leq f(u,u) * e + f(u',u') * e + f(u',l') * f(l,l)$$

Denote $f(u,u) * e + f(u',u') * e$ by p, $f(u,l) * f(l',l')$ by q and $f(u',l') * f(l,l)$ by r. We must show $q + p \leq r + p$. By 2), $(q+p)(r+p) = rq + rp + qp + p$. By monotonicity of $+$ (see 1)), it enough to prove $qp \leq r$. We prove more. In fact, $p \leq r$. First observe that if $a \leq b$, then $a * e \leq b * c$. Indeed, $(a * e) \cdot (b * c) = a * e + b * e = a * e$ by the same argument as in 5). Thus, we must show $p \leq f(u,l)$. Calculate $p \cdot f(u,l) = (f(u,u) + f(u',u')) * e \cdot f(u,l) = (f(u,u) + f(u',u')) * e * f(u,l) + f(u,l) * (f(u,u) + f(u',u')) * e = (f(u,u) + f(u',u')) * e + f(u,l) * e = f(u,u) * e + f(u',u') * e = p$. Thus, $p \leq r$ and this finishes the proof of well-definedness.

Our next goal is to show, as we did for snacks, that if we drop max and min in defining operations on scones, formula (7) will remain true. That will make it much easier to prove that f^+ is a homomorphism.

First observe that if $u \in U$ and $v \succsim u$, then $\tilde{U} * e = \widetilde{U \cup v} * e$ (we use notation $\tilde{U}$ as a shorthand for $\sum_{u \in U} f(u,u)$). This follows immediately from 5).

Consider the $\mathcal{L}$-part. In order to show that for $l' \succsim l \in L$, the corresponding summand of (8) remains the same if $f(l',l')$ is added, we must show $f(u,l_0) * f(l,l) * f(l',l') = f(u,l_0) * f(l,l)$. The left hand side is equal to $f(u,l_0) * f(l,l) * f(l,l')$ and by 6) $f(l,l) * f(l,l') = f(l,l)$. Therefore, the left hand side is equal to $f(u,l_0) * f(l,l)$.

Finally, it must be shown that adding $M \sqsubseteq^\sharp L \in \mathcal{L}$ does not change the value of the right hand side of (8). Assume $u \in U$, $m \in M$ and $l \in L$ are such that $m \leq l$ and $u \uparrow l$ (we can find such because of the consistency condition and $M \sqsubseteq^\sharp L$). Let $a = \hat{L}$ and $b = \hat{M}$. We must show $f(u,l) * a + f(u,m) * b = f(u,l) * a$ (it was already shown that it does not matter which consistent pair is chosen in representation (8)). Let $a' = f(u,l) * a$ and $b' = f(u,m) * b$. First, $a' \cdot b' = (f(u,l) * f(u,m) + f(u,m) * f(u,l)) * a * b = (f(u,l) \cdot f(u,m)) * a * b = f(u,m) * a * b$. Since $L \sqsubseteq^\sharp M$ and $f(c,c) * f(d,d) = f(d,c)$ for $d \succsim c$ by 8), we obtain $a' \cdot b' = f(u,m) * b = b'$. Hence $b' \leq a'$ and $a' + b' \leq a'$ by 1). To prove the reverse inequality, $a' \leq a' + b'$, calculate $a' \cdot (a' + b') = a' + (a' \cdot b') = a' + a' * b' + b' * a' = f(u,l) * a + f(u,l) * f(u,m) * a * b + f(u,m) * f(u,l) * a * b$. By admissibility, $f(u,l) * f(u,m) = f(u,m) * f(u,l)$. Therefore, $a' \cdot (a' + b') = f(u,l) * a + f(u,l) * a * f(u,m) * b = a' + a' * b' = a'$. Thus, $a' \leq a' + b'$ and this finishes the proof that the summand corresponding to $M \sqsubseteq^\sharp L$ can be added to (8).

Now we are ready to prove that f^+ is a homomorphism. First, $f^+(\emptyset, \emptyset) = e * e + e = e$.

Let $\mathcal{S}_1 = (U, \mathcal{L}_1)$ and $\mathcal{S}_2 = (V, \mathcal{M})$. Writing expression (8) for $f^+(\mathcal{S}_1 + \mathcal{S}_2)$ we can use $U \cup V$ as the first component and $\mathcal{L} \cup \mathcal{M}$ as the second. We know that it does not matter how we pick an element from $U \cup V$ to be consistent with some element of a set from $\mathcal{L} \cup \mathcal{M}$. For every $L \in \mathcal{L}$ choose $u_L \in U$ which is consistent with some $l_L \in L$ and similarly for every $M \in \mathcal{M}$ choose $v_M \in V$ which is consistent with some $m_M \in M$. Then we have

$$f^+(\mathcal{S}_1 + \mathcal{S}_2) = \sum_{u \in U \cup V} f(u,u) * e + \sum_{L \in \mathcal{L}} (f(u_L, l_L) * \hat{L}) + \sum_{M \in \mathcal{M}} (f(v_M, m_M) * \hat{M}) = f^+(\mathcal{S}_1) + f^+(\mathcal{S}_2)$$

Clearly, this also holds if either $\mathcal{L}$ or $\mathcal{M}$ or both are empty.

Let $a_L = f(u,l) * \hat{L}$, $c_M = f(v,m) * \hat{M}$ where $u \uparrow l$, $v \uparrow m$, $v \in V$, $u \in U$, $l \in L \in \mathcal{L}$ and $m \in M \in \mathcal{M}$. Let $b = \tilde{U} * e$ and $d = \tilde{V} * e$. Then $f^+(\mathcal{S}_1) * f^+(\mathcal{S}_2) = (\sum_{L \in \mathcal{L}} (a_L + b)) * (\sum_{M \in \mathcal{M}} (c_M + d)) = \sum_{L \in \mathcal{L}, M \in \mathcal{M}} (a_L * c_M + a_L * d + b * c_M + b * d)$. Since $d = \tilde{V} * e$, $a_L * d = a_L * e$ and $a_L * c_M + a_L * d = a_L * c_M + a_L * e = a_L * c_M$. Similarly, $b * d = b * e$. Since $b = \tilde{U} * e$, $b = b * e$. Therefore, $b * c_M = b * e = b$ and $b * d = b * e = b$. Therefore, $f^+(\mathcal{S}_1) * f^+(\mathcal{S}_2) = \sum_{L \in \mathcal{L}, M \in \mathcal{M}} (a_L * c_M) + b$. Consider $a_L * c_M$. Since $f(v,m)$ occurs inside the expression, by admissibility it can be changed to $f(m,m)$. Therefore, $a_L * c_M = f(u,l) * \hat{L} * \hat{M}$. Thus,

$$\begin{aligned} f^+(\mathcal{S}_1) * f^+(\mathcal{S}_2) \ = \ b + \sum_{L \in \mathcal{L}, M \in \mathcal{M}} f(u,l) * \hat{L} * \hat{M} \ = \\ \sum_{u \in U} f(u,u) * e + \sum_{N \in \{L \cup M \mid L \in \mathcal{L}, M \in \mathcal{M}\}} f(u,l) * \hat{N} \ = \ f^+(\mathcal{S}_1 * \mathcal{S}_2) \end{aligned}$$

Now, to finish that proof that f^+ is a homomorphism, it is enough to show that $f^+(\mathcal{S}_1) * f^+(\mathcal{S}_2) = f^+(\mathcal{S}_1 * \mathcal{S}_2)$ if one of the components is empty. Assume $\mathcal{L} = \emptyset$. Then the equation follows from $x * e * y = x * e$ and the fact that $\mathcal{S}_1 * \mathcal{S}_2 = \mathcal{S}_1$. If $\mathcal{M} = \emptyset$, then $f^+(\mathcal{S}_1) * f^+(\mathcal{S}_2) = (\tilde{U} * e + \sum_{L \in \mathcal{L}} f(u_L, l_L) * \hat{L}) * \tilde{V} * e = \tilde{U} * e + \sum_{L \in \mathcal{L}} f(u_L, l_L) * e = \tilde{U} * e = f^+(U, \emptyset) = f^+(\mathcal{S}_1 * \mathcal{S}_2)$. Thus, f^+ is a homomorphism.

The uniqueness of f^+ follows from (7) and well-definedness of (8). Finally, $f^+(\eta^\uparrow(x,y)) = f(x,x)*e+f(x,y)*f(y,y) = f(x,y)*e+f(x,y) = f(x,y)$. This shows $f^+\circ\eta^\uparrow = f$. Theorem is proved. □

Now we prove that it is impossible to characterize $\mathcal{P}^{\exists\wedge}(A)$ as a free ordered algebra generated by A, such that all operations of the scone algebras are present. That is, they are derived operations of the signature.

Theorem 4.34 *Let Ω_{Sc} be a set of operations on scones such that $+,*$ and e are derived operations. Then $\mathcal{P}^{\exists\wedge}(\cdot)$ is not left adjoint to the forgetful functor from the category of ordered Ω_{Sc}-algebras to* **Poset**. *In other words, for no Ω_{Sc} is $\mathcal{P}^{\exists\wedge}(A)$ the free ordered Ω_{Sc}-algebra generated by A.*

Proof. Let $x,y\precsim z$ in A. Then $((x,x)*(\emptyset,\emptyset)+(z,z))*(y,y) = (x,y)$. Now consider the following poset $A=\{x,y,z,v\}$. In this poset $x,y\precsim z$, $x,y\precsim v$ and $\{x,y\}$ and $\{z,v\}$ are antichains. Now consider the following scone algebra $Sc_1 = \langle B,+,*,e\rangle$. Its carrier is a four-element chain $p_1>p_2>p_3>p_4$. We interpret $+$ as minimum of two elements, $*$ as maximum, and $e=p_1$. It is easy to see that Sc_1 is a scone algebra as it is a distributive lattice.

Define $f: A\to B$ as follows: $f(z)=p_1, f(v)=p_2, f(x)=p_3$ and $f(y)=p_4$. Now suppose that f can be extended to a homomorphism $f^+:\mathcal{P}^{\exists\wedge}(A)\to Sc$. Then

$$f^+((x,y)) = f^+((\eta(x)*e+f(z))*\eta(y)) \quad = \\ (f(x)*e+f(z))*f(y) = \max\{\min\{\max\{p_1,p_3\},p_1\},p_4\} = p_1$$

On the other hand,

$$f^+((x,y)) = f^+((\eta(x)*e+f(v))*\eta(y)) \quad = \\ (f(x)*e+f(v))*f(y) = \max\{\min\{\max\{p_1,p_3\},p_2\},p_4\} = p_2$$

Hence, $p_1=p_2$, which contradicts the definition of B. This shows that f can not be extended to a homomorphism of scone algebras. □

The main observation we used in the proof of theorem 4.34 was the following. If $x\uparrow y$, then the scone (x,y) can be obtained as follows: $(x,y)=(\eta^\uparrow(x)*e+\eta^\uparrow(z))*\eta^\uparrow(y)$, provided $x,y\precsim z$. Therefore, the question arises: is it possible to restrict the class of maps from A to scone algebras in such a way that the universality diagram will be obtained for such maps. The next theorem we are going to prove gives us a way to do so. But first we need a new definition of admissibility.

Definition 4.12 *A monotone function $f: A\to Sc$ from a poset A to a scone algebra Sc is called* scone-admissible *if, for any two consistent pairs $x\uparrow y_1$ and $x\uparrow y_2$ such that $x,y_i\le z_i, i=1,2$, the following holds:*

$$(f(x)*e+f(z_1))*f(y_1)*f(y_2) = (f(x)*e+f(z_2))*f(y_1)*f(y_2)$$

Theorem 4.35 *For any poset A, $\mathcal{P}^{\exists\wedge}(A)$ is the free scone algebra generated by A with respect to scone-admissible maps. That is, for any scone algebra Sc and a scone-admissible map $f : A \to Sc$, there exists a unique scone homomorphism which completes the following diagram:*

$$\begin{tikzcd}
A \arrow[r, hook, "\eta"] \arrow[dr, "f"'] & \langle \mathcal{P}^{\exists\wedge}(A), +, *, e \rangle \arrow[d, dashed, "\exists! f^{+}"] \\
 & \langle Sc, +, *, e \rangle
\end{tikzcd}$$

Proof. Let $f : A \to Sc$ be a scone-admissible map. Define $\varphi_f : A{\uparrow}A \to Sc$ by

$$\varphi_f((x, y)) = (f(x) * e + f(z)) * f(y) \qquad \text{if } x, y \precsim z$$

It follows from the definition of scone-admissible maps that φ_f is well-defined. That is, if $x, y \precsim z_1, z_2$, then $(f(x)*e+f(z_1))*f(y) = (f(x)*e+f(z_1))*f(y)*f(y) = (f(x)*e+f(z_2))*f(y)*f(y) = (f(x)*e+f(z_1))*f(y)$ and hence the value of $\varphi_f((x, y))$ does not depend on the choice of z above x and y.

Let $\Delta : A \to A{\uparrow}A$ be given by $\Delta(a) = (a, a)$. Our next goal is to prove two claims.

Claim 1. φ_f is admissible (according to definition before theorem 4.33).
Claim 2. $\varphi_f \circ \Delta = f$.

Before we prove these two claims, let us show how the theorem follows from them. Consider the following diagram.

$$\begin{tikzcd}
A \arrow[r, "\Delta"] & A{\uparrow}A \arrow[r, "\eta^{\uparrow}"] \arrow[dr, "\varphi_f"'] & \mathcal{P}^{\exists\wedge}(A) \arrow[d, dashed, "\exists! f^{+}"] \\
 & & Sc
\end{tikzcd}$$

Since φ_f is admissible and $\eta^{\uparrow} \circ \Delta = \eta$, we can find a homomorphism f^{+} such that $f^{+} \circ \eta = f^{+} \circ \eta^{\uparrow} \circ \Delta = \varphi_f \circ \Delta = f$. Assume f^{-} is another homomorphism $\mathcal{P}^{\exists\wedge}(A) \to Sc$ such that $f^{-} \circ \eta = f$. Consider $(x, y) \in A{\uparrow}A$, $x, y \precsim z$. Then $\eta^{\uparrow}(x, y) = (\eta(x) * e + \eta(z)) * \eta(y)$. Hence, $f^{-}(\eta^{\uparrow}(x, y)) = (f(x) * e + f(z)) * f(y) = \varphi_f((x, y))$ which shows that $f^{-} \circ \eta^{\uparrow} = \varphi_f$. Then, by claim 2 and theorem 4.33, we obtain $f^{-} = f^{+}$ and thus there is a unique homomorphic extension of f.

Proof of claim 1. First, we must show $\varphi_f((x, y_1)) * e = \varphi_f((x, y_2)) * e$ if $x, y_1 \precsim z_1$ and $x, y_2 \precsim z_2$. From the properties of scone algebras, it follows that $a*e+b*e = a*e$ if $a \le b$. Since $f(x) \le f(z_1)$, we obtain $\varphi_f((x,y_1))*e = (f(x)*e+f(z_1))*f(y_1)*e = f(x)*e+f(z_1)*e = f(x)*e$. Similarly, $\varphi_f((x,y_2))*e = f(x)*e = \varphi_f((x,y_1))$.

For the second condition in the definition of admissibility, assume $u, l \precsim x_{ul}$ and $v, m \precsim x_{vm}$. Moreover, let $u, m \precsim x_{um}$ and $w, l \precsim x_{wl}$. We must show $\varphi_f((u,l))*\varphi_f((v,m)) = \varphi_f((u,m))*\varphi_f((w,l))$. Observe that $b \geq c$ implies $a*b*c = a*c$ in a scone algebra. Hence, $f(x_{ul})*f(x_{vm})*f(m) = f(x_{ul})*f(m)$. Moreover, as we saw already, $f(u)*e+f(x_{ul})*e = f(u)*e$. Now we calculate:

$$\begin{aligned}\varphi_f((u,l))*\varphi_f((v,m)) = (f(u)*e+f(x_{ul}))*f(l)*(f(v)*e+f(x_{vm}))*f(m) &= \\ (f(u)*e+f(x_{ul})*e+f(x_{ul})*f(x_{vm}))*f(l)*f(m) &= \\ (f(u)*e+f(x_{ul})*f(x_{vm}))*f(l)*f(m) = (f(u)*e+f(x_{ul}))*f(l)*f(m)\end{aligned}$$

Similarly,

$$\varphi_f((u,m))*\varphi_f((w,l)) = (f(u)+f(x_{um}))*f(l)*f(m)$$

Now the desired equality follows from scone-admissibility of f. Claim 1 is proved.

Proof of claim 2. $\varphi_f((x,x)) = (f(x)*e+f(x))*f(x) = f(x)*e+f(x) = f(x)$. Claim 2 and the theorem are proved. □

Universality of $\mathbfcal{P}^{\emptyset}$

In this section we describe $\mathbfcal{P}^{\emptyset}(A)$ – a construction which can be seen as "all others put together with no restrictions". This justifies the name of the *salad*. Salads can be viewed as snacks or scones without the consistency condition.

Similarly to the case of $\mathcal{P}^{\emptyset}(A)$, $\mathbfcal{P}^{\emptyset}(A)$ is isomorphic to the direct product of $\mathcal{P}^{\sharp}(A)$ and the iterated construction from section 4.2.2. Both possess universality property, but, as we mentioned already, a product of two free algebras need not be a free algebra. However, similarly to the case of $\mathcal{P}^{\emptyset}(A)$, we find a way to combine the two in a way that gives us a characterization of $\mathbfcal{P}^{\emptyset}(A)$ as a free ordered algebra.

Definition 4.13 *A* salad algebra $\langle Sd, +, \cdot, \Box, \Diamond\rangle$ *is an algebra with two semilattice operations* $+$ *and* $\cdot$ *and two unary operation* $\Box$ *and* $\Diamond$ *such that the following equations hold:*
1) $x\cdot(y+z) = x\cdot y + x\cdot z$.
2) $x = \Box x + \Diamond x$.
3) $\Box(x+y) = \Box x + \Box y = \Box x\cdot\Box y = \Box(x\cdot y)$.
4) $\Diamond(x+y) = \Diamond x + \Diamond y$.
5) $\Diamond(x\cdot y) = \Diamond x\cdot\Diamond y$.
6) $\Box x\cdot\Diamond y = \Box x$.
7) $\Diamond x\cdot\Diamond y + \Diamond x = \Diamond x$.

8) $\Diamond\Diamond x = \Diamond x$.
9) $\Box\Box x = \Box x$.

Define an ordering $\leq$ on a salad algebra according to the $\cdot$ operation: $x \leq y$ iff $xy = x$. Then every homomorphism of salad algebras is monotone with respect to the ordering.

Define $\Box Sd = \{\Box x \mid x \in Sd\}$ and $\Diamond Sd = \{\Diamond x \mid x \in Sd\}$. Some useful properties of salads are summarized in the following proposition.

Proposition 4.36 *Given a salad algebra Sd, the distributivity law $x + yz = (x+y)(x+z)$ holds. Consequently, $+$, $\Box$ and $\Diamond$ are monotone. In addition, the following holds:*
(i) $\Box x \leq x \leq \Diamond x$.
(ii) *$\Diamond Sd$ is a distributive lattice.*
(iii) *$+$ and $\cdot$ coincide on $\Box Sd$.*
(iiii) $\Box\Diamond x = \Diamond\Box y$.

Proof. Using 2) and distributivity law 1) calculate $(x+y)(x+z) = (\Box x + \Box y + \Diamond x + \Diamond y)(\Box x + \Box z + \Diamond x + \Diamond z) =$ (by 1) and 6)) $= \Box x + \Box y + \Box z + \Diamond x + \Diamond x \cdot \Diamond y + \Diamond x \cdot \Diamond z + \Diamond y \cdot \Diamond z =$ (by 7)) $= \Box x + \Box y + \Box z + \Diamond x + \Diamond y \cdot \Diamond z$. Similarly, $x + yz = \Box x + \Diamond x + (\Box y + \Diamond y)(\Box z + \Diamond z) = \Box x + \Diamond x + \Box y + \Box z + \Diamond y \cdot \Diamond z$. Hence, $(x+y)(x+z) = x + yz$. Now monotonicity of $+$ follows from the distributivity laws. That $\Box$ and $\Diamond$ are monotone, follows from 4) and 6). To prove (i), calculate $x \cdot \Box x = (\Box x + \Diamond x)\Box x = \Box x + \Diamond x \cdot \Box x = \Box x + \Box x = \Box x$. Moreover, $x \cdot \Diamond x = (\Box x + \Diamond x)\Diamond x = \Box x \cdot \Diamond x + \Diamond x = \Box x + \Diamond x = x$.

(ii) and (iii) follow immediately from the definitions.

(iiii) By 7), $\Box x \leq \Diamond\Box y$; hence $\Diamond\Box x \leq \Diamond\Box y$ and by symmetry $\Diamond\Box x = \Diamond\Box y$. Similarly, $\Box\Diamond x = \Box\Diamond y$. Define $e_\Diamond = \Diamond\Box x$ and $e_\Box = \Box\Diamond x$. The equations above show that $e_\Diamond$ and $e_\Box$ are well-defined. Now calculate $e_\Diamond + x = \Diamond\Box x + x = \Diamond\Box x + \Diamond x + x = \Diamond(\Box x + x) + x = \Diamond x + x = x$. Similarly, $e_\Box + x = \Box\Diamond x + x = \Box\Diamond x + \Box x + x = \Box(\Diamond x + x) + x = \Box x + x = x$. Thus, both $e_\Diamond$ and $e_\Box$ are identities for $+$. Therefore, $e_\Diamond = e_\Diamond + e_\Box = e_\Box$. $\Box$

This proposition tells us that we can give the following equivalent definition of a salad algebra: A salad algebra is a distributive bisemilattice $\langle Sd, +, \cdot\rangle$ on which a projection $\Box$ and a closure $\Diamond$ are defined such that $\Box Sd$ is a semilattice, $\Diamond Sd$ is a lattice, $x = \Box x + \Diamond x$ and $\forall x \in \Box Sd\, \forall y \in \Diamond Sd$: $x \leq y$.

There is also one property of salad algebras that is worth mentioning and that follows directly from the definitions. Given a semilattice $\langle S, \vee\rangle$ with bottom, a pair of ideals $\mathcal{I}_1$ and $\mathcal{I}_2$ is called a *general decomposition* of S if bottom is the only common element of $\mathcal{I}_1$ and $\mathcal{I}_2$ and every s in S has a unique representation as $s = s_1 \vee s_2$ where $s_1 \in \mathcal{I}_1$ and $s_2 \in \mathcal{I}_2$. If S is a bounded lattice, general decompositions become direct decompositions. For a large class of posets with partially defined lubs general decompositions are in 1-1 correspondence with neutral complemented ideals, see Jung, Libkin and Puhlmann [88].

Proposition 4.37 *Given a salad algebra Sd, $\Box Sd$ and $\Diamond Sd$ form a general decomposition of Sd.*

Proof. Let $\leq_+$ denote the ordering given by $+$, that is, $x \leq_+ y$ iff $x + y = y$. Let $x \leq_+ \Box y$. Then $\Diamond x + \Diamond\Box y = \Diamond\Box y$, i.e. $\Diamond x + e_\Diamond = e_\Diamond$ and $\Diamond x = e_\Diamond$. Now $x = \Diamond x + \Box x = e_\Diamond + \Box x = \Box x$. Hence $x \in \Box Sd$, which shows that $\Box Sd$ is an ideal. Similarly, $\Diamond Sd$ is an ideal. It follows from (iiii) of the lemma that $\Box Sd \cap \Diamond Sd = \{e\}$ where $e = e_\Box = e_\Diamond$. Finally, let $x = \Box y + \Diamond z$. Then $\Box x = \Box y + \Box\Diamond z = \Box y$ and similarly $\Diamond x = \Diamond z$. Hence, $x = \Box x + \Diamond x$ is a unique representation of x as a sum of elements from $\Box Sd$ and $\Diamond Sd$. Thus, $\Box Sd$ and $\Diamond Sd$ form a general decomposition. $\Box$

Let us now show how the salad algebra operations are interpreted on $\mathcal{P}^\emptyset(A)$. Operations $+$ and $\cdot$ are defined precisely as for snacks. For $\Box$ and $\Diamond$,

$$\Box(U, \mathcal{L}) = (U, \emptyset) \qquad \Diamond(U, \mathcal{L}) = (\emptyset, \mathcal{L})$$

Theorem 4.38 *Given a poset A, $\mathcal{P}^\emptyset(A)$ is the free salad algebra generated by A. That is, for every monotone map f from A to a salad algebra Sd there exists a unique salad homomorphism $f^+ : \mathcal{P}^\emptyset(A) \to Sd$ such that the following diagram commutes:*

$$\begin{tikzcd}
A \arrow[r, hook, "\eta"] \arrow[dr, "f"'] & \langle \mathcal{P}^\emptyset(A), +, \cdot, \Box, \Diamond \rangle \arrow[d, dashed, "\exists! f^+"] \\
& \langle Sd, +, \cdot, \Box, \Diamond \rangle
\end{tikzcd}$$

Proof. First verify that $\mathcal{P}^\emptyset(A)$ is a salad algebra. We need to check the distributivity law and 7); all others are straightforward. Let $\mathcal{S}_1 = (U, \mathcal{L}), \mathcal{S}_2 = (V, \mathcal{M})$ and $\mathcal{S}_3 = (W, \mathcal{N})$. Our goal is to show $\mathcal{S}_1 \cdot (\mathcal{S}_2 + \mathcal{S}_3) = \mathcal{S}_1 \cdot \mathcal{S}_2 + \mathcal{S}_1 \cdot \mathcal{S}_3$. The first components of the left hand and the right hand sides coincide. It this case it is easier to work with filters rather than antichains – it allows us to drop max and min operations. In particular, it is enough to show that

$$\begin{aligned}
\{\uparrow(L \cup K) | L \in \mathcal{L}, K \in \mathcal{M} \cup \mathcal{N}\} &= \\
\{\uparrow L_M | L_M \in \{L \cup M | L \in \mathcal{L}, M \in \mathcal{M}\}\} \bigcup \{\uparrow L_N | L_N \in \{L \cup N | L \in \mathcal{L}, N \in \mathcal{N}\}\}
\end{aligned}$$

Let C be an element of the left hand side, i.e. $C = \uparrow(L \cup K)$. Without loss of generality, $K \in \mathcal{M}$. Then C is in the right hand side. Conversely, if C is in the right hand side, say $C = \uparrow L_M$ for $L_M = L \cup M$, then $C = \uparrow(L \cup M)$ and therefore is in the left hand side. This shows the equality above. Now, taking minimal elements for each filter and applying $\max^\sharp$ to both collections would give us second components of the lhs and the rhs of the distributivity equation, which therefore are equal.

Now prove 7), that is, $\Diamond(U, \mathcal{L}) \cdot \Diamond(V, \mathcal{M}) + \Diamond(U, \mathcal{L}) = \Diamond(U, \mathcal{L})$. The first components of both sides are $\emptyset$. The second component of the left hand side is $\max^\sharp(\mathcal{L} \cup \max^\sharp\{\min(L \cup M) | L \in \mathcal{L}, M \in \mathcal{M}\})$. Since $\min(L \cup M) \sqsubseteq^\sharp L$, this expression is equal to $\max^\sharp \mathcal{L} = \mathcal{L}$. Hence, 7) holds. Thus, $\mathcal{P}^\emptyset(A)$ is a salad algebra.

Now show that $\mathcal{P}^\emptyset(A)$ is a free salad algebra. Given a salad $\mathcal{S} = (U, \mathcal{L})$,

$$\mathcal{S} = \Box \sum_{u \in U} \eta(u) + \Diamond \sum_{L \in \mathcal{L}} \prod_{l \in L} \eta(l) \tag{9}$$

To see that this also works for empty components, observe that $\Box e = \Diamond e = e$.

Now, given monotone $f : A \to Sd$, define

$$f^+(\mathcal{S}) = \Box \sum_{u \in U} f(u) + \Diamond \sum_{L \in \mathcal{L}} \prod_{l \in L} f(l) \tag{10}$$

We have: $f^+(\eta(x)) = f^+((x, \{x\})) = \Box f(x) + \Diamond f(x) = x$. Now we must show that f^+ is a homomorphism. First, it follows immediately from the properties of $\Box$ and $\Diamond$ and the fact that $e = \Box\Diamond x = \Diamond\Box y$ is the identity for $+$ (see lemma) that $f^+(\Box \mathcal{S}) = \Box f^+(\mathcal{S})$ and $f^+(\Diamond \mathcal{S}) = \Diamond f^+(\mathcal{S})$.

Assume $X \sqsubseteq^\sharp Y$, $Y \neq \emptyset$, and let x_y be an element in X below $y \in Y$. Then

$$\Box \sum_{x \in X} f(x) \cdot \Box \sum_{y \in Y} f(y) = \Box(\sum_{x \in X} f(x) + \sum_{y \in Y} f(y)) = \Box \sum_{x \in X} f(x) + \Box \sum_{y \in Y} (f(y) + f(x_y)) =$$
$$\Box \sum_{x \in X} f(x) + \Box \sum_{y \in Y} (f(y) \cdot f(x_y)) = \Box \sum_{x \in X} f(x) + \Box \sum_{y \in Y} f(x_y) = \Box \sum_{x \in X} f(x)$$

Therefore, if X and Y are equivalent with respect to $\sqsubseteq^\sharp$, $\Box \sum_{x \in X} f(x) = \Box \sum_{y \in Y} f(y)$. Our next goal is to show that $\Diamond \prod_{x \in X} f(x) + \Diamond \prod_{y \in Y} f(y) = \Diamond \prod_{y \in Y} f(y)$ if $Y \neq \emptyset$. Since $X \sqsubseteq^\sharp Y$, we have $\prod_{x \in X} f(x) \leq \prod_{y \in Y} f(y)$ and then the equation above follows from 7). Finally, let $x' \succsim x \in X$. Then $f(x') \geq f(x)$ and $\prod_{x \in X} f(x) = f(x') \cdot \prod_{x \in X} f(x)$.

These three observations show that max and min operations can be disregarded when one writes an expression for f^+ on $\mathcal{S}_1 + \mathcal{S}_2$ or $\mathcal{S}_1 \cdot \mathcal{S}_2$. Therefore, for $\mathcal{S}_1 = (U, \mathcal{L})$ and $\mathcal{S}_2 = (V, \mathcal{M})$,

$$f^+(\mathcal{S}_1 + \mathcal{S}_2) = \Box \sum_{x \in U \cup V} f(x) + \Diamond(\sum_{L \in \mathcal{L}} \prod_{l \in L} f(l) + \sum_{M \in \mathcal{M}} \prod_{m \in M} f(m)) = f^+(\mathcal{S}_1) + f^+(\mathcal{S}_2)$$

To calculate $f^+(\mathcal{S}_1 \cdot \mathcal{S}_2)$, observe that $\sum_{i\in I} \Box x_i \cdot \sum_{j\in J} \Diamond y_j = \sum_{i\in I, j\in J} \Box x_i \cdot \Diamond y_j = \sum_{i \in I} \Box x_i$ and this is also true if $I = \emptyset$ because $e \cdot \Diamond y = e$. Therefore,

$$
\begin{aligned}
f^+(\mathcal{S}_1 \cdot \mathcal{S}_2) = (\Box \sum_{u\in U} f(u) + \Diamond \sum_{L\in\mathcal{L}} \prod_{l\in L} f(l)) \cdot (\Box \sum_{v\in V} f(v) + \Diamond \sum_{M\in\mathcal{M}} \prod_{m\in M} f(m)) &= \\
(\Box \sum_{u\in U} f(u) \cdot \Box \sum_{v\in V} f(v)) + (\Box \sum_{v\in V} f(v) \cdot \Diamond \sum_{M\in\mathcal{M}} \prod_{m\in M} f(m)) &+ \\
+(\Box \sum_{v\in V} f(v) \cdot \Diamond \sum_{L\in\mathcal{L}} \prod_{l\in L} f(l)) + (\Diamond \sum_{L\in\mathcal{L}} \prod_{l\in L} f(l) \cdot \Diamond \sum_{M\in\mathcal{M}} \prod_{m\in M} f(m)) &= \\
\Box \sum_{u\in U} f(u) + \Box \sum_{v\in V} f(v) + \Diamond \sum_{\substack{L\in\mathcal{L}\\ M\in\mathcal{M}}} (\prod_{l\in L} f(l) \cdot \prod_{m\in M} f(m)) &= \\
\Box \sum_{x\in U\cup V} f(x) + \Diamond \sum_{\substack{L\in\mathcal{L}\\ M\in\mathcal{M}}} \prod_{y\in L\cup M} f(y) = f^+(\mathcal{S}_1) \cdot f^+(\mathcal{S}_2)
\end{aligned}
$$

Thus, f^+ is a homomorphism. Its uniqueness follows from (9). Theorem is proved. $\Box$

Let us summarize the results on the universality properties of approximations in the following table. For each construction with $u \le l$ used in the consistency condition (with one exception) we found a free algebra characterization. For constructions with $u \Uparrow l$ used in the consistency condition, we showed that they do not arize as free algebras generated by the poset itself, but do arize as free constructions generated by $A\Uparrow A$ (with respect to a restricted class of map). We use **dna** (does not apply) for constructions based on the $u \le l$ consistency condition with $A\Uparrow A$ as the generating poset. Notice that there are still three **ni** null values – these questions remain open.

	type of consistency condition (quantifier–condition)				
L-part; generator	$\forall \quad u \le l$	$\forall \quad u \Uparrow l$	$\exists \quad u \le l$	$\exists \quad u \Uparrow l$	no condition
one set; A	mix algebra	**ne**	bi-LNB algebra	**ni**	bi-mix algebra
one set; $A\Uparrow A$	**dna**	mix	**dna**	**ni**	**dna**
family of sets; A	snack algebra	**ne**	**ne**	**ne**	salad algebra
family of sets; $A\Uparrow A$	**dna**	**ni**	**dna**	scone algebra	**dna**

Relationship between the approximations

In this subsection we study the relationship between the four best-known approximations: mixes, sandwiches, scones, and snacks. We also show that we can view them as instances of the most general construction: salads, that is, $\mathcal{P}^{\emptyset}(A)$. We will substantiate the assertion that by their "complexity" the approximation constructs should be places as

$$\text{Salads} \rightarrow \text{Scones} \rightarrow \text{Snacks} \rightarrow \text{Sandwiches} \rightarrow \text{Mixes}$$

and algebras as

$$\text{Salads} \rightarrow \text{Scones} \rightarrow \text{Snacks} \rightarrow \text{Mixes}$$

The reader is invited to see how other constructions studied in this chapter will fit into the general picture. We consider only five approximation constructions to keep the diagrams reasonably small.

Relationship between algebras. The general technique we use is the following. Given an algebra $\langle \mathcal{A}, \Omega \rangle$, let Ω' be a subset of Ω and Ω'' a set of derived operations. Let $\Theta = (\Omega - \Omega') \cup \Omega''$. Then $\mathcal{A}$ can be considered as a Θ-algebra which is called Θ-*reduct of* $\langle \mathcal{A}, \Omega \rangle$, see Grätzer [64]. We denote a map that takes an Ω-algebra $\langle \mathcal{A}, \Omega \rangle$ and returns the Θ-algebra $\langle \mathcal{A}, \Theta \rangle$ by $\varphi^{\Omega \rightarrow \Theta}$.

We now define reductions for the algebras from the previous section. The superscripts of these reductions contain the information about its argument. They are the same as superscipts for the approximations themselves, except that we use index f (family) for $\mathcal{P}^i$'s. For example, a snack reduct of a scone will be denoted by $\varphi^{\exists\wedge \rightarrow \forall_f}$.

Definition. a) *Given a salad algebra* $Sd = \langle \mathcal{A}, +, \cdot, \Box, \Diamond \rangle$, *define its reducts as follows:*

Scone reduct $\varphi^{\emptyset \rightarrow \exists\wedge}(Sd) = \langle \mathcal{A}, +, *, e \rangle$ *where* $x * y = x \cdot \Diamond y$ *and* $e = \Diamond \Box x$.

Snack reduct $\varphi^{\emptyset \rightarrow \forall_f}(Sd) = \langle \mathcal{A}, +, \cdot, e \rangle$ *where* $e = \Diamond \Box x$.

Mix reduct $\varphi^{\emptyset \rightarrow \forall}(Sd) = \langle \mathcal{A}, +, \Box, e \rangle$ *where* $e = \Diamond \Box x$.

b) *Given a scone algebra* $Sc = \langle \mathcal{A}, +, *, e \rangle$, *define its reducts as follows:*

Snack reduct $\varphi^{\exists\wedge \rightarrow \forall_f}(Sc) = \langle \mathcal{A}, +, \cdot, e \rangle$ *where* $x \cdot y = x * y + y * x$.

Mix reduct $\varphi^{\exists\wedge \rightarrow \forall}(Sc) = \langle \mathcal{A}, +, \Box, e \rangle$ *where* $\Box x = x * e$.

c) *Given a snack algebra* $Sn = \langle \mathcal{A}, +, \cdot, e \rangle$, *define its* mix reduct $\varphi^{\forall_f \rightarrow \forall}(Sn)$ *as* $\langle \mathcal{A}, +, \Box, e \rangle$ *where* $\Box x = x \cdot e$.

Our first goal is to show that the concepts above are well-defined, i.e. that a mix reduct is a mix algebra, scone reduct is a scone algebra etc. We then proceed to show that it does not matter which path we choose, i.e. a mix reduct of a scone reduct of a salad is a mix reduct of a salad etc.

Proposition 4.39 *The reducts above are well-defined.*

Proof. We start with reducts of salads. First demonstrate that $\varphi^{\emptyset \rightarrow \exists\wedge}(Sd)$ is a scone algebra. That e is the identity for $+$ was already proved. Distributivity of $*$ over $+$ is obvious. We must show the other distributivity law: $a + x * y = (a + x) * (a + y)$. To prove this, calculate $a + xa = a + (\Box x + \Diamond x)(\Box a + \Diamond a) = a + \Box x \cdot \Box a + \Box x + \Box a + \Diamond x \cdot \Diamond a = a + \Box x + \Diamond x \cdot \Diamond a = a + (\Box x + \Diamond x)\Diamond a = a + a \cdot \Diamond a$. Now, $a + x * y = a + x \cdot \Diamond y = (a + x)(a + \Diamond y) = a + xa + a \cdot \Diamond y + x \cdot \Diamond y = a + x \cdot \Diamond a + a \cdot \Diamond y + x \cdot \Diamond y = (a + x)(\Diamond a + \Diamond y) = (a + x) * (a + y)$. This proves distributivity. That $*$ is a left normal band operation is obvious. We have $e * x = \Diamond \Box x \cdot \Diamond x = \Diamond(\Box x \cdot x) = \Diamond \Box x = e$. Finally, $x + x * y = x + (\Box x + \Diamond x) \cdot \Diamond y = x + \Box x + \Diamond x \cdot \Diamond y = x + \Box x + \Diamond x + \Diamond x \cdot \Diamond y = x + \Box x + \Diamond x = x$. Therefore, $\varphi^{\emptyset \to \exists\wedge}(Sd)$ is a scone algebra.

We have already shown in the previous section that $+$ and $\cdot$ distribute over each other; hence, $\varphi^{\emptyset \to \forall_f}(Sd)$ is a snack algebra. To check that $\varphi^{\emptyset \to \forall}(Sd)$ is a mix algebra, verify the equations of the mix algebra. The first two are also equations of the salad algebras, and we have shown already that $x + \Box x = x$ and $\Box x \le x$. Thus, we must show $x + \Box y \le x$. Calculate $(x + \Box y)x = x + \Box y \cdot x = x + \Box y \cdot \Box x + \Box y \cdot \Diamond x = x + \Box x + \Box y = x + \Box y$. Hence, $x + \Box y \le x$.

Now consider reducts of scones. To show that $\varphi^{\exists\wedge \to \forall_f}(Sc)$ is a scone algebra, we must verify the distributivity laws. One of them was verified in the proof of the characterization of scones. The other one is also easy: $x + y \cdot z = x + y * z + z * y = (x + y) * (x + z) + (x + z) * (x + y) = (x + y)(x + z)$. The next step is to verify that $\Box x = x * e$ satisfies the equations of the mix algebras. We have $x + \Box x = x + x * e = (x + x) * (x + e) = x$ and $x \cdot \Box x = x * x * e + x * e * x = x * e = \Box x$, hence $\Box x \le x$. Finally, $(x + y * e)x = (x + y * e) * x + x * (x + y * e) = x + y * e + x * e = x + y * e$. Therefore, $x + \Box y \le x$ and $\varphi^{\exists\wedge \to \forall}(Sc)$ is a mix algebra.

Finally, if in a snack algebra $\Box x$ is defined as xe, then $x + xe = (x + x)(x + e) = x$, $xxe = xe$ and $(x + ye)x = x + yex \le x + x = x$. Thus, $\varphi^{\forall_f \to \forall}(Sn)$ is a mix algebra and this finishes the proof of the proposition. $\Box$

Our next goal is to show path independence, that is, it does not matter if we perform reduction from one algebra to another directly or via a number of steps. This can be formalized as follows.

Theorem 4.40 *The following diagram commutes (where the arrow from Sd to Sn is $\varphi^{\emptyset \to \forall_f}$ and the arrow from Sc to Mix is $\varphi^{\exists\wedge \to \forall}$):*

$$
\begin{tikzcd}
Sd \arrow[r, "\varphi^{\emptyset \to \exists\wedge}"] \arrow[d, "\varphi^{\emptyset \to \forall}"'] \arrow[dr] & Sc \arrow[d, "\varphi^{\exists\wedge \to \forall_f}"] \arrow[dl] \\
Mix & Sn \arrow[l, "\varphi^{\forall_f \to \forall}"]
\end{tikzcd}
$$

Proof. We have already shown that reductions are well-defined. Consider $\varphi^{\exists\wedge\to\forall} \circ \varphi^{\emptyset\to\exists\wedge} : Sd \to Mix$. The identity for $+$ is $e = \Diamond\Box x$ and the box operation of the result, $\Box' x$, is defined as $\Box' x = x * e = x \cdot \Diamond\Diamond\Box x = (\Box x + \Diamond x) \cdot \Box\Diamond x = \Box x \cdot \Box\Diamond x + \Diamond x \cdot \Box\Diamond x = \Box x + \Box\Diamond x + \Box\Diamond x = \Box x + e = \Box x$. Hence, $\varphi^{\exists\wedge\to\forall} \circ \varphi^{\emptyset\to\exists\wedge} = \varphi^{\exists\wedge\to\forall}$. Now consider $\varphi^{\forall_f\to\forall} \circ \varphi^{\emptyset\to\forall_f} : Sd \to Mix$. The box operation of the result is $\Box' x = xe = (\Box x + \Diamond x)\Diamond\Box x = \Box x + e = \Box x$, hence $\varphi^{\forall_f\to\forall} \circ \varphi^{\emptyset\to\forall_f} = \varphi^{\emptyset\to\forall}$. Then consider $\varphi^{\forall_f\to\forall} \circ \varphi^{\exists\wedge\to\forall_f} \circ \varphi^{\emptyset\to\exists\wedge} : Sd \to Mix$. The box operation of the result is $\Box' x = x \cdot_{Sn} e = x * e + e * x = x \cdot \Diamond\Diamond\Box x + \Diamond\Box x \cdot \Diamond x = x \cdot \Box\Diamond x + \Box\Diamond x \cdot \Diamond x = \Box x + e = \Box x$. Thus, $\varphi^{\forall_f\to\forall} \circ \varphi^{\exists\wedge\to\forall_f} \circ \varphi^{\emptyset\to\exists\wedge} = \varphi^{\emptyset\to\forall}$. To show $\varphi^{\emptyset\to\forall_f} = \varphi^{\exists\wedge\to\forall_f} \circ \varphi^{\emptyset\to\exists\wedge}$, it is enough to show that $x \cdot y = x \cdot \Diamond y + y \cdot \Diamond x$. But this is easy: $x \cdot y = (\Box x + \Diamond x) \cdot (\Box y + \Diamond y) = \Box x \cdot \Box y + \Box x + \Box y + \Diamond x \cdot \Diamond y = \Box x + \Box y + \Diamond x \cdot \Diamond y$ and $x \cdot \Diamond y + y \cdot \Diamond x = (\Box x + \Diamond x) \cdot \Diamond y + (\Box y + \Diamond y) \cdot \Diamond x = \Box x + \Box y + \Diamond x \cdot \Diamond y$. Finally, to show that $\varphi^{\exists\wedge\to\forall} = \varphi^{\forall_f\to\forall} \circ \varphi^{\exists\wedge\to\forall_f}$, observe that $x * e + e * x = x * e + e = x * e$ and therefore $\Box x$ is the same for both reductions. Theorem is proved. $\Box$

Embeddings. We show that the reductions introduced above correspond to the embeddings of the approximation constructions. The general idea is as follows. Assume that a poset A is given and $\mathcal{P}'$ and $\mathcal{P}''$ are two approximation constructions such that $\mathcal{P}'$ is "higher" than $\mathcal{P}''$ in the hierarchy shown in the beginning of the section. That is, there is a reduction φ that takes $\mathcal{P}'(A)$ and makes it an algebra in the signature corresponding to $\mathcal{P}''$. Depending on the generating poset for $\mathcal{P}''(A)$, consider either $\eta(A)$ or $\eta^\dagger(A)$ which is a subset of $\mathcal{P}'(A)$. Then the subalgebra of $\varphi(\mathcal{P}'(A))$ generated by this subset is $\mathcal{P}''(A)$. Moreover, this construction is "path independent" in the sense of theorem 4.40. To formalize it, we use the notation

$$\mathcal{P}'(A) \xrightarrow{[\eta(A)] \circ \varphi} \mathcal{P}''(A) \quad \text{or} \quad \mathcal{P}'(A) \xrightarrow{[\eta^\dagger(A)] \circ \varphi} \mathcal{P}''(A)$$

The meaning of these arrows is: Take $\mathcal{P}'(A)$ and consider it as an algebra corresponding to $\mathcal{P}''$ (by means of φ). Then its subalgebra generated by $\eta(A)$ (or $\eta^\dagger(A)$) is $\mathcal{P}''(A)$.

Theorem 4.41 *In the following diagram all arrows are well-defined and the diagram commutes:*

$$\begin{tikzcd}
\boldsymbol{\mathcal{P}}^{\emptyset}(A) \arrow[rr, "{[\eta^{\dagger}(A)] \circ \varphi^{\emptyset \to \exists\wedge}}"] \arrow[rrrd] \arrow[dd, "{[\eta(A)] \circ \varphi^{\emptyset \to \forall}}"'] \arrow[rrdd] & & \boldsymbol{\mathcal{P}}^{\exists\wedge}(A) \arrow[dr, "{[\eta(A)] \circ \varphi^{\exists\wedge \to \forall_f}}"] \arrow[lldd] \arrow[dd] & \\
& & & \boldsymbol{\mathcal{P}}^{\forall}(A) \arrow[dlll] \arrow[dl, "{[\eta^{\dagger}(A)] \circ \varphi^{\forall_f \to \forall}}"] \\
\mathcal{P}^{\forall}(A) & & \mathcal{P}^{\forall\wedge}(A) \arrow[ll, "{[\eta(A)]}"] &
\end{tikzcd}$$

The arrows not shown on the diagram are:

$$[\eta(A)] \circ \varphi^{\emptyset \to \forall_f} : \boldsymbol{\mathcal{P}}^{\emptyset}(A) \to \boldsymbol{\mathcal{P}}^{\forall}(A) \qquad [\eta^{\dagger}(A)] \circ \varphi^{\emptyset \to \forall} : \boldsymbol{\mathcal{P}}^{\emptyset}(A) \to \mathcal{P}^{\forall\wedge}(A)$$

$$[\eta(A)] \circ \varphi^{\exists\wedge \to \forall} : \boldsymbol{\mathcal{P}}^{\exists\wedge}(A) \to \mathcal{P}^{\forall}(A) \qquad [\eta^{\dagger}(A)] \circ \varphi^{\exists\wedge \to \forall} : \boldsymbol{\mathcal{P}}^{\exists\wedge}(A) \to \mathcal{P}^{\forall\wedge}(A)$$

$$[\eta(A)] \circ \varphi^{\forall_f \to \forall} : \mathcal{P}^{\forall\wedge}(A) \to \mathcal{P}^{\forall}(A)$$

Proof. Full proof requires a lot of easy calculations so we only sketch it here. First observe that all definitions of new operations for reductions agree with their interpretation. For example, given two scones $(U, \mathcal{L})$ and $(V, \mathcal{M})$ in $\boldsymbol{\mathcal{P}}^{\exists\wedge}(A)$, the value of $(U, \mathcal{L}) \cdot (V, \mathcal{M})$ in $\varphi^{\exists\wedge \to \forall_f}(\boldsymbol{\mathcal{P}}^{\exists\wedge}(A))$ is $(U, \mathcal{L}) * (V, \mathcal{M}) + (V, \mathcal{M}) * (U, \mathcal{L}) = (\min(U \cup V), \max^{\sharp}\{L \cup M | L \in \mathcal{L}, M \in \mathcal{M}\})$ which is indeed the infimum operation in $\boldsymbol{\mathcal{P}}^{\forall}(A)$. The verification that other reductions agree with the operations on approximations is also straightforward. Now representations of sandwiches (1), snacks (3), scones (7) and mixes as

$$(U, L) = \Box \sum_{u \in U} \eta(u) + \sum_{l \in L} \eta(l) \tag{11}$$

tell us that all arrows are well-defined. Commutativity follows in a straightforward way from the representations (1), (3), (7), (11) and theorem 4.40. □

This completes our discussion of the semantics of partial data. We have defined orderings on various kinds of collections and used them to define the formal semantics of those. The semantic domains of the collection type constructors have been shown to possess universality properties. We shall use the universality properties in the next chapter to design programming languages for partial information, as described in section 3.2.

Chapter 5

Languages for partial information

In previous chapters we have developed the semantics of partial information that was based on one of the two main principles of this thesis: partiality of data is represented via orderings on objects. In this chapter we use the semantic results to build languages for databases with partial information, following the second principle which says that semantics suggests programming constructs.

We start with languages for sets under the open world assumption. Since the universality properties for arbitrary sets and antichains are essentially the same, we obtain two very close languages, and show that one of them, dealing with antichains, can naturally be viewed as a sublanguage of the other. We give several reasons why it is better to view the language dealing with the ordered semantics as a sublanguage of the language for the set-theoretic semantics. One of them is that in the former it is important to be able to identify the monotone fragment of the language, but this is undecidable. We show that two languages considered so far – the language of Zaniolo and the domain theoretic algebra from section 3.1 – are sublanguages of the language for OWA sets.

We also consider languages for or-sets, viewed structurally, and prove similar results. Having defined languages for sets and or-sets, we combine them to obtain a new language called *or-$\mathcal{NRL}$*. Since it is necessary to distinguish between sets and or-sets, we enhance the language with a primitive that provides interaction between sets and or-sets. This primitive is precisely the isomorphism α from section 4.2.2.

The language *or-$\mathcal{NRL}$* has a number of very important properties. First, it is possible to define a function that lists all possibilities encoded by an or-object. This enables the language to answer *conceptual* queries such as: is there a complete design of a given cost? Moreover, we show that under both set-theoretic and antichain semantics the process of listing all possibilities encoded by an object always yields the same result, no matter what strategy is used. We call this result *normalization theorem.* The process of listing all possibilities is also called *normalization.*

We show that normalization can be quite expensive. In fact, we determine tight upper bounds on the size of normalized objects and the number of possibilities that arbitrary or-objects can encode. Then we observe that it is not always necessary to complete the process of normalization to answer a conceptual query. However, it is not always the case that *partial normalization* is unambiguous. That is, the analog of the normalization theorem need not hold. Nevertheless, we are able to identify very strong sufficient conditions for such an analog to hold, and then prove the partial normalization theorem that unambiguously determines a representation of object of one type at another type. This allows us to answer certain conceptual queries faster.

We also demonstrate a *losslessness* result, which says that the loss of structural information in the process of normalization does not have any effect with respect to the large class of queries.

Finally, we discuss two approaches to programming with approximations. One is based on structural recursion and monads. It is now applicable due to the characterization of approximations as free constructions. However, we show that there are certain problems with using this approach. The other is encoding approximations with sets and or-sets and using the language for sets and or-sets. We show how all monad primitives for approximations can then be encoded in that language and argue that this makes it a better candidate for a programming language for approximations.

5.1 Languages for collections of partial data

5.1.1 Language for sets

In this section we consider a language for sets under the open world assumption. This language is based on the universality property. Since the universality properties of the semantic domains of sets with no partial information involved and of sets under OWA are essentially the same – both are free semilattices, but one is generated by a set and the other by a poset – the languages are essentially similar and the only syntactic difference is replacing equality test by comparability test. The only semantic difference is that in the language for partial information we operate with antichains rather than arbitrary sets, as is suggested by the semantic domain for OWA sets. We shall see that the language we define can be viewed as a sublanguage of $\mathcal{NRL}$ with orders on base types. We study some of its properties and explain how two languages that we have seen (Zaniolo's algebra [181] and the domain algebra of section 3.1) can be viewed as its sublanguages.

The language we are about to describe is based on the universality properties for OWA sets. Recall that for a given set X, $\langle \mathbb{P}_{\text{fin}}(X), \cup, \emptyset \rangle$ is the free semilattice with bottom generated by X. For posets, the result is similar: given a poset A, $\langle \mathcal{P}^\flat(A), \sqcup^\flat, \emptyset \rangle$ is the free *ordered* semilattice with bottom generated by A. Therefore, following section 3.2, we define two variations of the structural recursion, the one dealing with antichains using index a. Since we do not consider structural recursion on bags in this chapter, we use sru and sri instead of s_sru and s_sri.

$$\begin{array}{lll} fun & sru[e,h,u](\emptyset) & = & e \\ \mid & sru[e,h,u](\{x\}) & = & h(x) \\ \mid & sru[e,h,u](A \sqcup^\flat B) & = & u(sru[e,h,u](A), sru[e,h,u](B)) \end{array}$$

$$\begin{array}{lll} fun & sru_a[e,h,u](\emptyset) & = & e \\ \mid & sru_a[e,h,u](\{x\}) & = & h(x) \\ \mid & sru_a[e,h,u](A \sqcup^\flat B) & = & u(sru_a[e,h,u](A), sru_a[e,h,u](B)) \end{array}$$

As we discussed in section 3.2, the general structural recursion need not be well-defined. Hence, we used the operation of the Kleisli category of the corresponding adjunction as primitives of the programming language. For sets, we used

$$map(f)\{x_1,\ldots,x_n\} = \{f(x_1),\ldots,f(x_n)\} \qquad \mu(\{X_1,\ldots,X_n\}) = X_1 \cup \ldots \cup X_n \qquad \eta(x) = \{x\}$$

Similarly, for antichains we would have (cf. section 2.3)

$$map_a(f)\{x_1,\ldots,x_n\} = \max\{f(x_1),\ldots,f(x_n)\} \qquad \eta(x) = \{x\}$$

$$\mu_a(\{X_1,\ldots,X_n\}) \; = \; X_1 \sqcup^\flat \ldots \sqcup^\flat X_n \; = \; \max(X_1 \cup \ldots \cup X_n)$$

In addition to the equality test, which was chosen as a primitive in $\mathcal{NRL}$, we include a new primitive which tests whether two objects of type t are comparable as elements of the semantic domains $[\![t]\!]$[1]. That is, we assume that the ordering on base types is given, and it is lifted to pairs component-wise and to sets by using the Hoare ordering:

- $(x,y) \leq_{s\times t} (x',y)' \Leftrightarrow x \leq_s x'$ and $y \leq_t y'$.
- $x \leq_{\{s\}} y \Leftrightarrow x \leq^\flat_s y$ (i.e. $\forall o \in x\ \exists o' \in y:\ o \leq_s o'$).

Now we give the expressions of the language which we call $\mathcal{NRL}_a$; see figure 5.1.

Let us make a few observations about this language.

Proposition 5.1 *Assume that $\leq_b$ is given for any base type b. Then $\leq_s$ is definable in $\mathcal{NRL}_a$ without using $\leq_s$ as a primitive.*

[1]Since we do not use or-sets, the structural semantics $[\![]\!]_s$ and the conceptual semantics $[\![]\!]_c$ coincide. This justifies using just $[\![]\!]$ in this section.

Category with products

$$\frac{g : u \to s \quad f : s \to t}{f \circ g : u \to t} \qquad \frac{c : bool \quad f : s \to t \quad g : s \to t}{\mathit{if}\ c\ \mathit{then}\ f\ \mathit{else}\ g : s \to t} \qquad \frac{f : u \to s \quad g : u \to t}{(f, g) : u \to s \times t}$$

$$\frac{}{\pi_1^{s,t} : s \times t \to s} \qquad \frac{}{\pi_2^{s,t} : s \times t \to t} \qquad \frac{}{!^t : t \to unit}$$

$$\frac{}{Kc : unit \to Type(c)} \qquad \frac{}{id^t : t \to t} \qquad \frac{}{\leq_s : s \times s \to bool}$$

OWA sets monad (given by $\mathcal{P}^\flat$)

$$\frac{}{\rho_2^{s,t} : s \times \{t\} \to \{s \times t\}} \qquad \frac{}{\eta^t : t \to \{t\}} \qquad \frac{}{\sqcup_t^\flat : \{t\} \times \{t\} \to \{t\}}$$

$$\frac{}{\mu_a^t : \{\{t\}\} \to \{t\}} \qquad \frac{}{empty^t : unit \to \{t\}} \qquad \frac{f : s \to t}{map_a\ f : \{s\} \to \{t\}}$$

Figure 5.1: Expressions of $\mathcal{NRL}_a$

Proof. We only have to check that $\leq_{\{s\}}$ is definable if $\leq_s$ is. Assume X, Y are sets of type $\{s\}$. Then we create an object $\{(x, Y) \mid x \in X\}$ of type $\{s \times \{s\}\}$ and check for every (x, Y) if there exists y in Y such that $x \leq_s y$. This is achieved by first applying ρ_2 to (x, Y) and then mapping $\leq_s$ over the result and testing whether *true* occurs in the output. □

Proposition 5.2 *Under the assumption that $\leq_b$ can be tested in $O(1)$ time, the time complexity of verifying $x \leq_s y$ is $O(n^2)$, where n is the total size of x and y.*

Proof. Define the size of a base type object to be 1 and the size of a set or a pair to be the sum of the sizes of its elements (components). We prove by induction on the structure of objects that testing $\leq_t$ of two objects o_1, o_2 of type t can be done in $O(\mathsf{size}\,(o_1) \cdot \mathsf{size}\,(o_2))$. Then the proposition will follow. Let $X = \{x_1, \ldots, x_k\}$ and $Y = \{y_1, \ldots, y_l\}$ be sets of type $\{s\}$. According to the proof of proposition 5.1, checking whether $X \leq_{\{s\}} Y$ requires some preprocessing that costs at most $O(\mathsf{size}\,(X) \cdot \mathsf{size}\,(Y))$ and, by the induction hypothesis, $O(\sum_{i=1}^{k} \sum_{j=1}^{l} (\mathsf{size}\,(x_i) \cdot \mathsf{size}\,(y_j)))$ for actual comparisons. We have $\sum_{i=1}^{k} \sum_{j=1}^{l} (\mathsf{size}\,(x_i) \cdot \mathsf{size}\,(y_j))) \leq \mathsf{size}\,(Y) \sum_{i=1}^{k} \mathsf{size}\,(x_i) \leq \mathsf{size}\,(X) \cdot \mathsf{size}\,(Y)$, which finishes the proof. □

Now we can show that using $\mathcal{NRL}$ is sufficient because

Theorem 5.3 *$\mathcal{NRL}_a$ is a sublanguage of $\mathcal{NRL}(\leq_b)$.*

Proof. We have already shown in proposition 5.1 how to define $\leq_s$ for any s if $\leq_b$ is given. The rest is to observe that $map_a(f)(X) = \max map(f)(X)$ and $\mu_a(\mathcal{X}) = \max(\mu(\mathcal{X}))$. Hence, definability of max would imply that $\mathcal{NRL}_a$ is a sublanguage of $\mathcal{NRL}(\leq_b)$. It is easy to see that $\max X$ is implementable by deleting such elements $x \in X$ for which there exists $x' \in X$ with $x \leq x'$ and $x \neq x'$. Indeed, if $\leq$ is present, there is a first order formula that is true iff $x \in \max X$ and hence even operations of the relational algebra suffice. □

However, there is one subtle point. Assume that we have two sets X_1 and X_2 of type $\{t\}$ such that $\max X_1 = \max X_2$. That is, X_1 and X_2 represent the same object in $[\![\{t\}]\!]$. Let $f : \{t\} \to s$ be a function definable in $\mathcal{NRL}$. Is it true that $f(X_1)$ and $f(X_2)$ represent the same object in $[\![s]\!]$? Unfortunately, the answer to this question is negative. To see why, consider x and y of type t such that $x \leq_t y$ and $x \neq y$. Assume that $g : t \to s$ is such that $g(x)$ and $g(y)$ are not comparable by $\leq_s$. Then $map(g)(\{y\}) = \{g(y)\}$ and $map(g)(\{x, y\}) = \{g(x), g(y)\}$. Even though $\max\{y\} = \max\{x, y\}$, we have $\max(map(g)(\{y\})) \neq \max(map(g)(\{x, y\}))$.

The reason this happens is that g is not a monotone function. Requiring monotonicity is sufficient to repair this problem. Define the following translation function $(\cdot)^\circ$ on objects that forces objects in the set-theoretic semantics into the objects in the antichain semantics:

- For x of base type b, $x^\circ = x$.

- For $x = (x_1, x_2)$, $x^\circ = (x_1^\circ, x_2^\circ)$.
- For $X = \{x_1, \ldots, x_n\}$, $X^\circ = \max\{x_1^\circ, \ldots, x_n^\circ\}$.

We say that a function $f : s \to t$ definable in $\mathcal{NRL}$ *agrees with the antichain semantics* if $x^\circ = y^\circ$ implies $f(x)^\circ = f(y)^\circ$. We say that it is *monotone* iff $x \leq_s y$ implies $f(x) \leq_t f(y)$.

Proposition 5.4 *A monotone function f definable in $\mathcal{NRL}$ agrees with the antichain semantics. If f is not monotone, then $map(f)$ does not agree with the antichain semantics.*

Proof. First prove that $x \leq_s y$ iff $x^\circ \leq_s y^\circ$ for any x, y of type s and vice versa. Prove it by induction. The only interesting case is the set type constructor. Let $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$ be two sets of type $\{s\}$. Assume $X \leq_{\{s\}} Y$. Then $\forall x_i \in X\ \exists y_j \in Y : x_i \leq_s y_j$ and by induction hypothesis $\forall x_i \in X\ \exists y_j \in Y : x_i^\circ \leq_s y_j^\circ$. Hence, $\{x_1^\circ, \ldots, x_n^\circ\} \leq_{\{s\}} \{y_1^\circ, \ldots, y_m^\circ\}$ and then $X^\circ \leq_{\{s\}} Y^\circ$. Conversely, if $X^\circ \leq_{\{s\}} Y^\circ$, then $\{x_1^\circ, \ldots, x_n^\circ\} \leq_{\{s\}} \{y_1^\circ, \ldots, y_m^\circ\}$ and by induction hypothesis $\forall x_i \in X\ \exists y_j \in Y : x_i \leq_s y_j$, that is, $X \leq_{\{s\}} Y$.

Since x° is an antichain for any x, this observation implies that $x^\circ = y^\circ$ for x, y of type s iff $x \leq_s y$ and $y \leq_s x$.

Now assume $f : s \to t$ is monotone and $x^\circ = y^\circ$. Then $x \leq_s y$ and $y \leq_s x$ and hence $f(x) \leq_t f(y)$ and $f(y) \leq_t f(x)$ which proves $f(x)^\circ = f(y)^\circ$. That is, f agrees with antichain semantics.

Assume $f : s \to t$ is not monotone, i.e. $f(x) \not\leq_t f(y)$ for some $x <_s y$. We have $x^\circ \leq_s y^\circ$ and hence $\{x, y\}^\circ = \{y\}^\circ$. Moreover, $x^\circ \neq y^\circ$ for otherwise we would have $y \leq_s x$. Now, $map(f)(\{x, y\}) = \{f(x), f(y)\}$ and $map(f)(\{y\}) = \{f(y)\}$ and it is easy to see that $\{f(x), f(y)\}^\circ \neq \{f(y)\}^\circ$ if $f(y) <_t f(x)$ or $f(y)$ and $f(x)$ are incomparable. Proposition is proved. $\square$

Therefore, we would like to identify the subclass of monotone functions definable in $\mathcal{NRL}$. Unfortunately, it is not possible to do it algorithmically. Not being able to decide monotonicity is another reason why we prefer to view $\mathcal{NRL}_a$ as a sublanguage of $\mathcal{NRL}$ in which the antichain semantics can be modeled, rather than a separate language.

Theorem 5.5 *It is undecidable whether a function f definable in $\mathcal{NRL}$ is monotone.*

Proof. Assume monotonicity is decidable. Now, given two $\mathcal{NRL}$ functions $f, g : \{s\} \to t$, define a new function $\phi : \{s\} \to \{bool\}$ as follows:

$$\phi(x) \quad := \quad \textit{if } x = \emptyset \textit{ then } \{\textit{true}\} \textit{ else if } f(x) = g(x) \textit{ then } \{\textit{true}\} \textit{ else } \{\textit{false}\}$$

Here $\{x\}$ is syntactic sugar for $\eta(x)$. Now, if want to check whether $f(x) = g(x)$ for all x, check if $f(\emptyset)$ and $g(\emptyset)$ are the same and then check if ϕ is monotone. Thus having a test for monotonicity would give us equality test for functions of type $\{s\} \to t$. Such functions include all functions definable in the relational algebra, and it is known that equality of those is undecidable, see Imielinski and Lipski [79]. This shows that monotonicity of $\mathcal{NRL}$ expressions is undecidable. $\Box$

There are some intersting anomalies of the antichain semantics. The most surprising of all is that $[\![\eta]\!] = [\![powerset]\!]$ or, in other words, $\mathcal{NRL}_a(powerset) = \mathcal{NRL}_a$. Indeed, since for any $Y \in \mathbb{P}_{\text{fin}}(X)$ we have $Y \subseteq X$ and hence $Y \sqsubseteq^\flat X$, then under the antichain semantics $[\![\mathbb{P}_{\text{fin}}(X)]\!] = [\![\max \mathbb{P}_{\text{fin}}(X)]\!] = [\![\{X\}]\!] = [\![\eta(X)]\!]$. There are two lessons we learn from this interesting collapse. First, as we have said already, it is better to view $\mathcal{NRL}_a$ as a sublanguage of $\mathcal{NRL}$ rather than a separate language. Second, *powerset* is *not* a good candidate to enrich expressiveness of the language. (Of course, the theorem of Paredaens and Suciu [162] is a much stronger argument against *powerset*!)

The next question we are going to address is that of conservativity of $\mathcal{NRL}$ over $\mathcal{NRL}_a$. Given a family of primitives $\vec{p}$ interpreted for both set theoretic and antichain semantics, we say that $\mathcal{NRL}(\leq_b, \vec{p})$ is *conservative* over $\mathcal{NRL}_a(\vec{p})$ if for any function f definable in $\mathcal{NRL}(\leq_b, \vec{p})$ and satisfying the condition that $f(x) = f(x)^\circ$ for any $x = x^\circ$, such f is definable in $\mathcal{NRL}_a(\vec{p})$. We do not know if $\mathcal{NRL}(\leq_b)$ is conservative over $\mathcal{NRL}_a$. However, we can show that it is conservative when augmented with aggregate functions as in section 3.2.

Proposition 5.6 *$\mathcal{NRL}(\mathbb{N}, \sum, \cdot, \dot{-}, \leq_b)$ is conservative over $\mathcal{NRL}_a(\mathbb{N}, \sum, \cdot, \dot{-})$.*

Proof sketch. The key observation is that in the language with arithmetic functions it is possible to assign unique numerical ranks to elements in a set if linear orders at base types are given. Indeed, this follows from theorem 3.29 since we can lift the linear order to all types, and then for each element of a set use $\sum$ to count the number of elements not greater than it in the linear order. A careful analysis of the lifting procedure and rank assignment shows that they can be done in $\mathcal{NRL}_a(\mathbb{N}, \sum, \cdot, \dot{-})$ as well.

Now consider $x = x^\circ$. Since all its subobjects of set type are antichains, we can do the following in $\mathcal{NRL}_a(\mathbb{N}, \sum, \cdot, \dot{-})$. For each set subobject of x°, assign unique ranks to its elements. Now we have a new object x_1 such that $x_1 = x_1^\circ$ and all elements in all sets in x have their ranks attached to them. Then we can define the action of f on this object. The only two cases that require special care to make sure information is not lost are union and flattening. For $X \cup Y$, we first create $\{(x, 1) \mid x \in X\}$ and $\{(y, 2) \mid y \in Y\}$ and then union those. For $\mu(\{X_1, \ldots, X_n\})$, assume that the rank of X_i is i. Then create $\{\{(x, 1) \mid x \in X_1\}, \ldots, \{(x, n) \mid x \in X_n\}\}$ and apply μ to it. The equality test also requires some care as it needs to be defined in such a way that it disregards all attached indices, but it also can be done.

At the end, we have essentially $f(x)$ except that many integers are attached to its subobjects. We simply remove those using projections. Since $f(x) = f(x)^\circ$, it is guaranteed that no loss of information occurs while those ranks are projected out, and hence the result is $f(x)$. $\Box$

Now we give two examples of using $\mathcal{NRL}(\leq_b)$, based on the fact that $\mathcal{NRL}_a$ is it sublanguage (see theorem 5.3). First, we explain how Zaniolo's language described in section 1.1 can be viewed as a sublanguage of $\mathcal{NRL}(\leq_b)$. Second, we do it with the language of section 3.1 which is based on the domain model.

Example: Zaniolo's language

Recall that in the language of Zaniolo [181] there is only one kind of nulls – **ni**. The ordering on records was defined component-wise and it was lifted to relations by using the Hoare ordering. Zaniolo's language was initially designed for flat relations only but here we show how to extend it to the nested relations.

The main notion was that of x-relation which was an equivalence class with respect to the Hoare ordering. That is, R_1 and R_2 are equivalent if $R_1 \sqsubseteq^\flat R_2$ and $R_2 \sqsubseteq^\flat R_1$. In our terminology this means that $\downarrow R_1 = \downarrow R_2$. Therefore, we can pick a canonical representative of each equivalence class which is given by the max operation. That is, the canonical representative of the equivalence class of R is $\max R$. Clearly, $\downarrow R_1 = \downarrow R_2$ implies $\max R_1 = \max R_2$.

The next notion used for defining the operations was that of generalized membership: $t \hat{\in} R$ iff $t \leq t'$ for some $t' \in R$. In other words, $t \hat{\in} R$ iff $t \in \downarrow R$. Using this notion, Zaniolo defined the following main operations:

$$R_1 \hat{\cup} R_2 = \max\{t \mid t \hat{\in} R_1 \text{ or } t \hat{\in} R_2\}$$

$$R_1 \hat{\cap} R_2 = \max\{t \mid t \hat{\in} R_1 \text{ and } t \hat{\in} R_2\}$$

$$R_1 \hat{-} R_2 = \max\{t \mid t \hat{\in} R_1 \text{ and } \neg(t \hat{\in} R_2)\}$$

We assume that all base types are Scott domains. This is certainly true in the original Zaniolo's model as he only considered flat domains. If we use nested relations, it is still guaranteed that we only deal with bounded complete posets, that is, gratest lower bounds of consistent pairs are defined at all types. With this in mind, we see how the above operations are translated into the standard order-theoretic language we advocate in this thesis:

$$R_1 \hat{\cup} R_2 \;=\; \max\{t \mid t \in \downarrow R_1 \text{ or } t \in R_2\} \;=\; \max \downarrow R_1 \cup \downarrow R_2 \;=\; R_1 \sqcup^\flat R_2$$

$$R_1 \hat{\cap} R_2 \;=\; \max \downarrow R_1 \cap \downarrow R_2 \;=\; \max\{r_1 \wedge r_2 \mid r_1 \in R_1, r_2 \in R_2\} \;=\; R_1 \sqcap^\flat R_2$$

$$R_1 \hat{-} R_2 \;=\; \max\{t \mid t \hat{\in} R_1 \text{ and } \neg(t \hat{\in} R_2)\} \;=\; R_1 - \downarrow R_2$$

Thus, Zaniolo's union, intersection and difference are order-theoretic analogs of the usual set-theoretic union, intersection and difference. Next we notice that these operations are definable in $\mathcal{NRL}_a$ and hence in $\mathcal{NRL}(\leq_b)$. We have seen already that max is definable, so we only need the following lemma which is proved by an easy induction and definitions of $\sqcup^\flat$ and $\sqcap^\flat$.

Lemma 5.7 *If the least upper bound $\vee_b : b \times b \to b$ and the greatest lower bound $\wedge_b : b \times b \to b$ are given for any base type b, then the least upper bound $\vee_s : s \times s \to s$ and the greatest lower bound $\wedge_s : s \times s \to s$ are definable in $\mathcal{NRL}_a$ for every type s.* $\square$

The last operation of Zaniolo's language is the join (we omit projection and selection as these are standard and of course definable in $\mathcal{NRL}_a$). The join with respect to a set X of attributes was defined as

$$R_1 \bowtie_X R_2 \quad := \quad \max\{t_1 \vee t_2 \mid t_1 \hat{\in} R_1, \quad t_2 \hat{\in} R_2, \quad t_1 \text{ and } t_2 \text{ are total on } X\}$$

Without the condition that t_1 and t_2 must be total on X that translates into $\max\{t_1 \vee t_2 \mid t_1 \in R_1, t_2 \in R_2\}$ and hence is definable in $\mathcal{NRL}_a$ by taking cartesian product of R_1 and R_2 and mapping $\vee$ over it. In the case of flat relations, it is also possible to check if the value of a projection is **ni** since **ni** is available as a constant of base types now. Hence, the totality condition can be checked, and since selection is definable, so is $\bowtie_X$. Summing up, we have

Theorem 5.8 *The language of Zaniolo is a sublanguage of $\mathcal{NRL}_a$, and hence $\mathcal{NRL}$.* $\square$

Notice that in the case of model with one null **ni** we do not have to require orderings on base types as these are definable using just equality test.

Example: Domain-theoretic language

A simple language based on the domain model was introduced in section 3.1. It had six operations: union, difference, selection, projection, cartesian product and join. The reason for having six operations rather than the usual five was that the join was not definable via the rest of the operations for all domains, but only for domains of a special structure. The union operation was $\sqcup^\flat$ which is, as we have just seen, definable in $\mathcal{NRL}_a$. Difference was the usual set difference (which was sufficient to define the difference as in Zaniolo's language). Projection and selection were based on the concept of scheme (see section 3.1). Here we assume that there are only trivial schemes, that is, those given by the fileds of records or components of pairs. Therefore, projection and selection are definable in $\mathcal{NRL}_a$.

The join operation was defined as the Smyth join $\sqcup^\sharp$, that is,

$$\begin{aligned} R_1 \sqcup^\sharp R_2 &= \min\{x \mid \exists r_1 \in R_1\ \exists r_2 \in R_2 :\ r_1 \le x \text{ and } r_2 \le x\} = \\ &= \min\{r_1 \vee r_2 \mid r_1 \in R_1 \text{ and } r_2 \in R_2\} \end{aligned}$$

Therefore, by lemma 5.7, $R_1 \sqcup^\sharp R_2$ is definable in $\mathcal{NRL}_a$. Summing up, we obtain

Proposition 5.9 *The domain-theoretic algebra of section 3.1 is a sublanguage of $\mathcal{NRL}_a$ and hence of $\mathcal{NRL}(\leq_b)$.* □

Notice that here we do have to include $\leq_b$ as the domains of base types could be arbitrary.

5.1.2 Language for or-sets

In this section we follow the method developed in the previous section. The language we are going to describe is based on the universality properties for or-sets. One of the languages, $\mathcal{NRL}^{\mathrm{or}}$, disregards order, and views or-sets structurally, that is, just as subsets of a given set. The other, $\mathcal{NRL}^{\mathrm{or}}_a$, also views or-sets structurally, but takes into account the ordering and regards or-sets as antichains.

Given a poset A, $\langle \mathcal{P}^\sharp(A), \sqcap^\sharp, \emptyset\rangle$ is the free *ordered* semilattice with top generated by A. Recall that $X_1 \sqcap^\sharp X_2 = \min(X_1 \cup X_2)$. Hence, the syntax of two languages $\mathcal{NRL}^{\mathrm{or}}$ and $\mathcal{NRL}^{\mathrm{or}}_a$ is very similar to the syntax of $\mathcal{NRL}$ and $\mathcal{NRL}_a$. In particular, the *category with products* part is just inherited from those languages. So here we only give the monad constructs. Types are given by the following grammar for both $\mathcal{NRL}^{\mathrm{or}}$ and $\mathcal{NRL}^{\mathrm{or}}_a$.

$$t ::= b \mid unit \mid bool \mid t \times t \mid \langle t\rangle$$

The monad primitives are shown in figure 5.2

The only difference between the semantics of two languages is the interpretation of or_μ_a and or_map_a which was shown already in section 2.3:

$$or\_map_a(f)(\langle x_1, \ldots, x_n\rangle) = \min\langle f(x_1), \ldots, f(x_n)\rangle$$

$$or\_\mu_a\langle X_1, \ldots, X_n\rangle = X_1 \sqcap^\sharp \ldots \sqcap^\sharp X_n = \min(X_1 \cup \ldots \cup X_n)$$

Since or-sets are ordered by the Smyth ordering and redundancies are removed by taking minimal elements, we augment the definitions of orderings on complex objects and forcing sets into antichains from the previous section as follows:

- $x \leq_{\langle s\rangle} y \Leftrightarrow x \leq^\sharp_s y$ (i.e. $\forall o' \in y\ \exists o \in x :\ o \leq_s o'$)
- $\langle x_1, \ldots, x_n\rangle^\circ = \min\langle x_1^\circ, \ldots, x_n^\circ\rangle$

Now one can repeat the proofs of the previous section verbatim and arrive at the following theorem.

Theorem 5.10 *1. If $\leq_b$ is given at any base type b, then $\leq_s$ is definable in $\mathcal{NRL}^{\mathrm{or}}_a$ without using $\leq_s$ as a primitive.*

Or-Set monad of $\mathcal{NRL}^{\mathrm{or}}$

$$\frac{}{or\_\rho_2{}^{s,t} : s \times \langle t \rangle \to \langle s \times t \rangle} \qquad \frac{}{or\_\eta^t : t \to \langle t \rangle} \qquad \frac{}{or\_\cup^t : \langle t \rangle \times \langle t \rangle \to \langle t \rangle}$$

$$\frac{}{or\_\mu^t : \langle\langle t \rangle\rangle \to \langle t \rangle} \qquad \frac{}{or\_empty^t : unit \to \langle t \rangle} \qquad \frac{f : s \to t}{or\_map\ f : \langle s \rangle \to \langle t \rangle}$$

Or-Set monad of $\mathcal{NRL}^{\mathrm{or}}_a$ (given by $\mathcal{P}^\sharp$)

$$\frac{}{or\_\rho_2{}^{s,t} : s \times \langle t \rangle \to \langle s \times t \rangle} \qquad \frac{}{or\_\eta^t : t \to \langle t \rangle} \qquad \frac{}{\sqcap^\sharp_t : \langle t \rangle \times \langle t \rangle \to \langle t \rangle}$$

$$\frac{}{or\_\mu^t_a : \langle\langle t \rangle\rangle \to \langle t \rangle} \qquad \frac{}{or\_empty^t : unit \to \langle t \rangle} \qquad \frac{f : s \to t}{or\_map_a\ f : \langle s \rangle \to \langle t \rangle}$$

Figure 5.2: Expressions of $\mathcal{NRL}^{\mathrm{or}}$ and $\mathcal{NRL}^{\mathrm{or}}_a$

2. *Under the assumption that $\leq_b$ can be tested in $O(1)$ time, the time complexity of verifying $x \leq_s y$ is $O(n^2)$, where n is the total size of x and y.*

3. *$\mathcal{NRL}_a^{\text{or}}$ is a sublanguage of $\mathcal{NRL}^{\text{or}}(\leq_b)$.*

4. *Any monotone function f definable in $\mathcal{NRL}^{\text{or}}$ agrees with the antichain semantics. If f is not monotone, then or_map(f) does not agree with the antichain semantics.*

5. *It is undecidable whether a function f definable in $\mathcal{NRL}^{\text{or}}$ is monotone.* □

These languages per se are not of great practical interest. In the next section we combine $\mathcal{NRL}$ with $\mathcal{NRL}^{\text{or}}$ and add a new operation that provides a meaningful interaction between sets and or-sets. Then we show that a great deal of structural and conceptual queries can be expressed in the resulting language.

5.1.3 Language for bags

As we explained in sections 3.2 and 4.1.1, the main difference between having bags and sets as the underlying data model is that in a bag every entry represents a distinct object. Therefore, having equal entries means that at the present time we have only partial information about two objects and they can not be distinguished. Having two objects x and y such that x is less informative than y still means that x and y are distinct and now we know less about x than we know about y. In particular, in bags there are no redundancies arising from having comparable elements, and bags need not be represented as antichains.

This interpretation of bags led us to two orderings $\trianglelefteq^{\text{CWA}}$ and $\trianglelefteq^{\text{OWA}}$ depending on whether we believe in OWA or CWA. These orderings are quite different from $\sqsubseteq^\natural$ and $\sqsubseteq^\flat$ used for CWA sets and OWA sets respectively. We have seen that $\sqsubseteq^\natural$ and $\sqsubseteq^\flat$ are definable in the standard language for sets $\mathcal{NRL}$ or standard language for antichains $\mathcal{NRL}_a$ which is a sublanguage of $\mathcal{NRL}$ if orderings on base types are provided. However, the situation with bags is quite different. In the standard bag language $\mathcal{BQL}$, which is the bag counterpart of $\mathcal{NRL}$, it is impossible to define $\trianglelefteq^{\text{CWA}}$ and $\trianglelefteq^{\text{OWA}}$.

Theorem 5.11 *The orderings $\trianglelefteq^{\text{CWA}}$ and $\trianglelefteq^{\text{OWA}}$ are not definable in $\mathcal{BQL}$.*

Proof. We prove this in two stages. First, consider the following problem called SDR. Given an object o of type $\{\!|\{\!|t|\!\}|\!\}$ such that all bags are in fact sets, that is, all elements occur at most once. Does o have a system of distinct representatives? We also need a slight modification of this problem SDR$^=$ asking whether there exists a system of distinct representatives having the same cardinality as the number of bags in o.

We prove the following.

Claim 1. If $\trianglelefteq^{\text{OWA}}$ is definable in $\mathcal{BQL}$, then SDR is definable in $\mathcal{BQL}$.
Claim 2. If $\trianglelefteq^{\text{CWA}}$ is definable in $\mathcal{BQL}$, then $\text{SDR}^=$ is definable in $\mathcal{BQL}$.
Claim 3. Neither SDR nor $\text{SDR}^=$ is definable in $\mathcal{BQL}$.

In proving these claims, we use theorem 3.26 from section 3.2 which says that instead of $\mathcal{BQL}$ we can consider $\mathcal{NRL}$ with natural numbers and simple arithmetic which we denote by $\mathcal{NRL}_{\text{nat}}$.

Proof of claim 1. If $\trianglelefteq^{\text{OWA}}$ or $\trianglelefteq^{\text{CWA}}$ is definable in $\mathcal{BQL}$, then we can write a function that lifts an order on elements of type t to the order on elements of type $\{\!|t|\!\}$. It is enough to restrict our attention to bags without duplicates.

Assume that a family $\mathcal{S} = \{S_1, \ldots, S_n\}$ of sets of type $\{t\}$ is given. Then we do the following. First, by using μ we find $dom(\mathcal{S}) = S_1 \cup \ldots \cup S_n$ and then assign unique ranks to elements of $dom(S)$ (see the remark after theorem 3.35 which explains how to do it in $\mathcal{NRL}_{\text{nat}}$.) Also assign unique ranks to the sets in $\mathcal{S}$. From now on, assume the indices of the sets are their ranks. Then attach the ranks of elements of $dom(\mathcal{S})$ to elements of S_i's. It is easy to see that this can be done in $\mathcal{NRL}_{\text{nat}}$. Thus, we have an object $\mathcal{S}'$ of type $\{\{t \times \mathbb{N}\}\}$. Now define a new set V which consists of pairs (s, m) such that s is the element of $dom(\mathcal{S})$ with rank 1 and $m = card(dom(\mathcal{S})) + 1, \ldots, card(dom(\mathcal{S})) + 1 + n$. Again, this can be done in $\mathcal{NRL}_{\text{nat}}$. Notice that $V \cap dom(\mathcal{S}') = \emptyset$.

Now define a binary relation on $V \cup dom(\mathcal{S}')$ by letting $(s, m) \leq (s', j)$ iff $s' \in S_{m-card(dom(\mathcal{S}))}$. Then, according to proposition 4.8, $V \trianglelefteq^{\text{OWA}} dom(\mathcal{S}')$ (when these are considered as bags) iff $\mathcal{S}$ has a system of distinct representatives. Hence, runing SDR on $\mathcal{S}$ is reduced to testing $\trianglelefteq^{\text{OWA}}$ between two bags. This completes the proof of claim 1.

Proof of claim 2. We just repeat all the steps of proof of claim 1 and observe $V \trianglelefteq^{\text{CWA}} dom(\mathcal{S}')$ iff $\text{SDR}^=$ has a solution on $\mathcal{S}$.

To prove claim 3, we define a new query called *chain_even*. It takes an input of type $\{t \times t\}$ and returns a boolean. If the input is a chain (i.e. a tree with out-degree at most 1), then it returns *true* if the length of the chain is even and *false* if it is odd. If the input is not chain, the output is arbitrary.

Claim 4. chain_even is not definable in $\mathcal{NRL}_{\text{nat}}$.
Proof. It was shown in Libkin and Wong [108] that in $\mathcal{BQL}$, for every boolean query q on simple circuits there exists a number l such that either $q(c) = \mathit{true}$ for all circuits c of length $\geq l$ or $q(c) = \mathit{false}$ for all circuits c of length $\geq l$. Now consider the following query q' which is definable with *chain_even*. Take in a simple a circuits and consider all chains that are obtained by removing one edge from a circuit, and map *chain_even* over all such chains. It is easy to see that $q'(c) = \{\mathit{true}\}$ if the length of c is odd and $q'(c) = \{\mathit{false}\}$ if the length of c is even. This contradicts the result of [108]. The claim is proved.

Now we need the following lemma which reduces SDR and $\text{SDR}^=$ to *chain_even*. In fact, this is a first order reduction.

Let $X_m = \{x_1, \ldots, x_m\}, m > 2$, be a chain such that x_{i+1} is immediate successor of x_i, $i = 1, \ldots, m-1$. Define $\mathcal{S}_m$ as $\{\{x_1\}, \{x_m\}\} \bigcup \{\{x_{i-1}, x_{i+1}\} \mid i = 2, \ldots, m-1\}$.

Lemma. $\mathcal{S}_m$ has a system of distinct representatives iff m is even.

Proof of lemma. First, fix some notation. Given X_m, let Y_i^m be $\{x_1\}$ for $i = 1$, $\{x_m\}$ for $i = m$ and $\{x_{i-1}, x_{i+1}\}$ for $1 < i < m$. If a family $\{Y_i^m\}$ of sets has a system of distinct representatives, then we use $c(Y_i^m)$ to denote the representative of Y_i^m.

We prove this lemma by induction on m. For $m = 3, 4$ it is easy to see that lemma is true. Now, assume that $m > 4$ and m is even. By induction hypothesis, we know $\mathcal{S}_{m-2}$ has a system of distinct representatives. For any $i < m-2$, $Y_i^m = Y_i^{m-2}$. Furthermore, $Y_{m-2}^{m-2} = \{x_{m-2}\}$ (and hence $c(Y_{m-2}^{m-2}) = x_{m-2}$), $Y_{m-2}^m = \{x_{m-3}, x_{m-1}\}$, $Y_{m-1}^m = \{x_{m-2}, x_m\}$, $Y_m^m = \{x_m\}$. Then $\mathcal{S}_m$ has a system of distinct representatives defined as follows. For $k < m-2$, $c(Y_k^m) = c(Y_k^{m-2})$. For $m-2$, $c(Y_{m-2}^m) = x_{m-1}$, and $c(Y_{m-1}^m) = x_{m-2}$ and $c(Y_m^m) = x_m$. Hence, $\mathcal{S}_m$ has a system of distinct representatives.

Now let $m > 4$ be odd. We know $\mathcal{S}_{m-2}$ does not have a system of distinct representatives. Assume $\mathcal{S}_m$ does have it. Then $c(Y_m^m) = \{x_m\}$ and $c(Y_{m-1}^m) = x_{m-2}$ are forced. For $c(Y_{m-2}^m)$ there are two choices: x_{m-3} and x_{m-1}. If $c(Y_{m-2}^m) = x_{m-3}$, then note that x_{m-1} is not present in any other Y_l^m and hence will never get selected. But since the cardinalities of X_m and $\mathcal{S}_m$ coincide, this means $\mathcal{S}_m$ does not have a system of distinct representatives. This contradiction shows that $c(Y_{m-2}^m) = x_{m-1}$. Therefore, for any $i < m-2$, $c(Y_i^m) = x_j$ where $j < m-2$. Since $Y_i^m = Y_i^{m-2}$ for $i < m-2$, then by taking $c(Y_i^{m-2}) = c(Y_i^m)$ for $i < m-2$ and $c(Y_{m-2}^{m-2}) = x_{m-2}$ we obtain a system of distinct representatives for $\mathcal{S}_{m-2}$, contradiction. Hence, $\mathcal{S}_m$ does not have a system of distinct representatives. This finishes the proof of the lemma.

Now claim 3 follows from the lemma and claim 4. Indeed, if SDR (or even SDR$^=$ since cardinalities of X_m and $\mathcal{S}_m$ coincide) were definable, by the lemma we would be able to test whether a chain has even or odd length. This finishes the proof of the theorem. $\square$

Therefore, any implementation of $\mathcal{BQL}$ that is supposed to deal with the problem of partial information must provide $\trianglelefteq^{\text{CWA}}$ and $\trianglelefteq^{\text{OWA}}$ as additional primitives.

Corollary 5.12 *Neither $\mathcal{NRL}$ nor $\mathcal{NRL}$ with arithmetic functions can define a function of type $\{\{s\}\} \to$ bool that tests whether a family of sets has a system of distinct representatives.* $\square$

Unlike most queries whose inexpressibility has been proved earlier, this one is a truly nested query: it has no first order analog.

5.2 Language for sets and or-sets

In this section we introduce the main theoretical language of this thesis that combines sets and or-sets. We study its properties and later show how it can be used to deal with approximations. This language also serves as the core of the system called OR-SML which will be described in the next chapter.

As we often said, or-sets have emerged from applications within the design and planning areas, and in particular computer aided design. Now we give a simple example of an incomplete design database and use it to illustrate the main problems that arise in querying such databases. We then proceed to solve some of those problems.

Example: Querying incomplete database

Assume that we have a database containing an incomplete design. For example, a part may consist of several subparts and each of them can be chosen from several possibilities with different parameters like price and reliability. To give an example, assume that we have a design which requires two subparts, A and B. An A is either $A1$ or $A2$. The part $A1$ consists of two subparts: $A1.1$ and $A1.2$. An $A1.1$ is either x or y and an $A1.2$ is either z or v. The part $A1.2$ consists of three subparts: $A2.1$, $A2.2$ and $A2.3$. An $A2.1$ is either p or q, an $A2.2$ is either r or s and an $A2.3$ is either t or u. A B consists of $B1$ and $B2$. A $B1$ is either w or k and a $B2$ is either l or m. This incomplete design is shown in figure 5.3. We use dashed lines to represent possible choices.

Now assume that for every subpart that can make it into the completed design (those are denoted by lower case letters) we have two parameters: its cost $c(\cdot)$ and reliability $r(\cdot)$. Below we give examples of structural queries, that is, queries asking questions about the structural representation of an incomplete design, and conceptual queries, that is, queries asking questions about completed designs which are not stored in a database and thus are purely conceptual.

Structural Queries

- List all possible subparts of $A1$.
- What is the cost of w?
- How many possible choices are there for $A2.3$?
- Which choice for $A2.3$ has the minimal cost?

Conceptual Queries

- Is there a complete design that costs less than \$50?

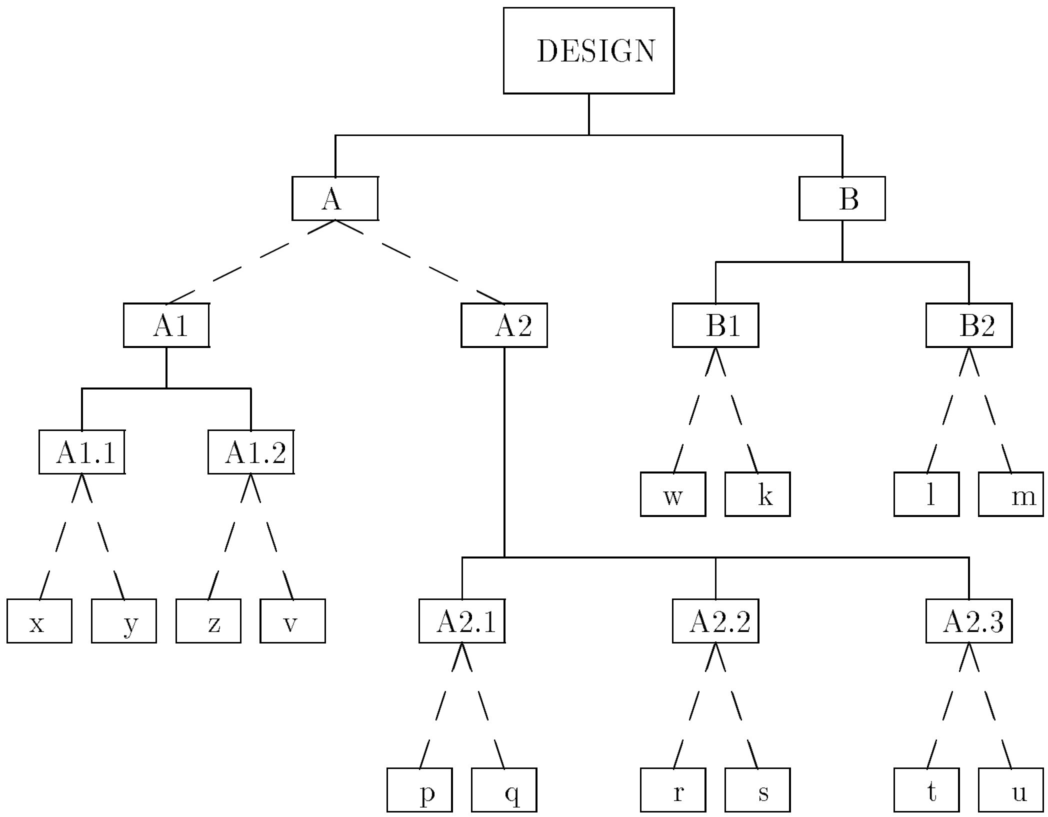


Figure 5.3: An incomplete design

- Is there a complete design that costs less than \$65 and whose reliability is at least 93%?
- What is the least expensive complete design?
- What is the most reliable complete design?
- How many complete designs are there?

We would like to design a language that is capable of supporting both kinds of queries. To do it, we need a way to ask conceptual queries like the ones above. Let us explain, at this point just informally, how this can be done.

First, we must represent DESIGN as an object in the language. We assume that types are built from base types by using the product, set $\{\}$ and or-set $\langle\rangle$ type constructors. We build the design bottom-up. First, we obtain

$$A1.1 = \langle x, y\rangle$$

$$A1.2 = \langle z, v\rangle$$

$$A2.1 = \langle p, q\rangle$$

$$A2.2 = \langle r, s\rangle$$

$$A2.3 = \langle t, u\rangle$$

$$B1 = \langle w, k\rangle$$

$$B2 = \langle l, m\rangle$$

Now $B = (B1, B2)$. The A part requires more care. We see from the diagram that $A = \langle A1, A2\rangle$. Hence, $A1$ and $A2$ must be of the same type. This means that it is impossible to represent $A1$ as $(A1.1, A1.2)$ and $A2$ as $(A2.1, (A2.2, A2.2))$ for then $A = \langle A1, A2\rangle$ would not typecheck. Therefore, we represent $A1$ and $A2$ as sets. That is, we build

$$A1 = \{A1.1, A1.2\}$$

$$A2 = \{A2.1, A2.2, A2.3\}$$

$$A = \langle A1, A2\rangle$$

and finally DESIGN $= (A, B)$. Assuming that all descriptions of the smallest subparts (those that are denoted by the lower case letters) have type t, the type of DESIGN is

$$\langle\{\langle t\rangle\}\rangle \times (\langle t\rangle \times \langle t\rangle)$$

Now consider $A1 = \{\langle x, y\rangle, \langle z, v\rangle\}$ which is of type $\{\langle t\rangle\}$. It is a set which has four possible values: $\{x, z\}, \{x, v\}, \{y, z\}, \{y, v\}$. To obtain or-sets containing these sets from $A1$ one needs essentially the isomorphism between the iterated constructions α described in section 4.2.2. If we apply it to both $A1$ and $A2$, we obtain two objects of type $\langle\{t\}\rangle$. Now A becomes an object of type $\langle\langle\{t\}\rangle\rangle$ and we make it an object of type $\langle\{t\}\rangle$ by applying *or*_μ which, as we remarked earlier, does not change the meaning.

Similarly, B is an object of type $\langle t\rangle \times \langle t\rangle$ and one can list all possibilities encoded by B by taking the cartesian product of $B1$ and $B2$. Hence, B becomes an object of type $\langle t \times t\rangle$. Now the whole DESIGN becomes an object of type $\langle\{t\}\rangle \times \langle t \times t\rangle$. Again, we take the cartesian product and obtain an object of type $\langle\{t\} \times (t \times t)\rangle$.

Intuitively, elements of this object are the complete designs. Therefore, we can write conceptual queries by simply selecting certain elements from this or-set. So, in order to find out if we can ask those conceptual queries, we must answer the following questions:

- Given any object o involving or-sets, is it possible to construct an object o' which is an or-set containing objects not involving or-sets such that o' represents all possibilities encoded by o?
- Does o' depend on the order in which operations like cartesian product and α in our example are performed?

In this section we introduce a language for sets and or-sets and show that using that language we can construct o' from o in a way that is "path independent", that is, does not depend on the order in which operations are applied. That object o' will be called the *normal form* of o, and the language will be capable of expressing a function *normalize* that takes o into o'. Then conceptual queries simply become queries asked against normal forms.

5.2.1 Syntax and semantics

The language we present deals with sets and or-sets. Its type system is given by

$$t ::= b \mid \mathit{unit} \mid \mathit{bool} \mid t \times t \mid \{t\} \mid \langle t\rangle$$

Its expressions simply combine expressions of $\mathcal{NRL}$ and $\mathcal{NRL}^{\text{or}}$. However, if we do just that, there is no way to distinguish between sets and or-set, because all arrows coming out of sets (or-sets) go to sets (or-sets). The way to distinguish between the two is to look at their interaction. That is, we want to know what is the connection between $\{\langle t\rangle\}$ and $\langle\{t\}\rangle$.

Since ordering on sets corresponds to the Hoare ordering, and ordering on or-sets is the Smyth ordering, we would like to see if there is a natural correspondence between the operators $\mathcal{P}^{\flat\sharp}$ and

$\mathcal{P}^{\sharp\flat}$. As we saw in section 4.2.2, these two operators always produce isomorphic domains, so we take one of the isomorphisms as a primitive in the language. Summing up, we have the language for writing structural queries over sets and or-sets, which we call *or-$\mathcal{NRL}$*. Its expressions are shown in figure 5.4.

Syntax of *or-$\mathcal{NRL}_a$* is the same except that $\leq$ is used instead of eq and the following operations have index a; map, or_map, μ, or_μ and α.

Semantics. The semantics of all constructs other than α has been given already. Now define the semantics of α and α_a.

Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ be a set of or-sets where $X_i = \langle x_1^i, \ldots, x_{n_i}^i \rangle$. Let $\mathcal{F}_\mathcal{X}$ be the set of all choice functions on $\mathcal{X}$, that is, the set of all functions $f : \{1, \ldots, n\} \to \mathbb{N}$ such that $1 \leq f(i) \leq n_i$ for all $i = 1, \ldots, n$. Then

$$\alpha(\mathcal{X}) = \langle \{x^i_{f(i)} \mid i = 1 \ldots, n\} \mid f \in \mathcal{F}_\mathcal{X} \rangle$$

$$\alpha_a(\mathcal{X}) = \min_{\sqsubseteq^\flat} \langle \max\{x^i_{f(i)} \mid i = 1 \ldots, n\} \mid f \in \mathcal{F}_\mathcal{X} \rangle$$

Therefore, according to theorem 4.21, α_a^t is an isomorphism between $[\![\{\langle t \rangle\}]\!]_s$ and $[\![\langle \{t\} \rangle]\!]_s$ and in addition $\alpha_a(\mathcal{X}) = \alpha(\mathcal{X})^\circ$.

Recall that objects involving or-sets have two different semantics: the structural semantics $[\![]\!]_s$ and the conceptual semantics $[\![]\!]_c$. Therefore, every expression of *or-$\mathcal{NRL}$* or *or-$\mathcal{NRL}_a$* has interpretation with respect to both $[\![]\!]_s$ and $[\![]\!]_c$. The remark about α used the structural semantics; the conceptual semantics will be studied in the next section.

Combining techniques from the previous section, we can easily show the following properties of *or-$\mathcal{NRL}$* and *or-$\mathcal{NRL}_a$* (see theorem 5.10.)

Theorem 5.13 *1. If $\leq_b$ is given at any base type b, then $\leq_s$ is definable in* or-$\mathcal{NRL}_a$ *without using $\leq_s$ as a primitive.*

2. *Under the assumption that $\leq_b$ can be tested in $O(1)$ time, the time complexity of verifying $x \leq_s y$ is $O(n^2)$, where n is the total size of x and y.*

3. or-$\mathcal{NRL}_a$ *is a sublanguage of* or-$\mathcal{NRL}(\leq_b)$.

4. *For any two objects x, y of type s, $x \leq_s y$ iff $x^\circ \leq_s y^\circ$.*

5. *Any monotone function f definable in* or-$\mathcal{NRL}$ *agrees with the antichain semantics. If f is not monotone, then $map(f)$ and $or\_map(f)$ do not agree with the antichain semantics.*

6. *It is undecidable whether a function f definable in* or-$\mathcal{NRL}$ *is monotone.* □

Operators shared by $\mathcal{NRL}$ and $\mathcal{NRL}^{\mathrm{or}}$

$$\frac{g : u \to s \quad f : s \to t}{f \circ g : u \to t} \qquad \frac{c : bool \;\; f : s \to t \;\; g : s \to t}{\mathit{if}\ c\ \mathit{then}\ f\ \mathit{else}\ g : s \to t} \qquad \frac{f : u \to s \quad g : u \to t}{(f, g) : u \to s \times t}$$

$$\frac{}{\pi_1^{s,t} : s \times t \to s} \qquad \frac{}{\pi_2^{s,t} : s \times t \to t} \qquad \frac{}{!^t : t \to unit}$$

$$\frac{}{Kc : unit \to Type(c)} \qquad \frac{}{id^t : t \to t} \qquad \frac{}{eq^t : t \times t \to bool}$$

Operators from set monad of $\mathcal{NRL}$

$$\frac{}{\rho_2^{s,t} : s \times \{t\} \to \{s \times t\}} \qquad \frac{}{\eta^t : t \to \{t\}} \qquad \frac{}{\cup^t : \{t\} \times \{t\} \to \{t\}}$$

$$\frac{}{\mu^t : \{\{t\}\} \to \{t\}} \qquad \frac{}{empty^t : unit \to \{t\}} \qquad \frac{f : s \to t}{map\ f : \{s\} \to \{t\}}$$

Operators from or-set monad of $\mathcal{NRL}^{\mathrm{or}}$

$$\frac{}{or\_\rho_2{}^{s,t} : s \times \langle t \rangle \to \langle s \times t \rangle} \qquad \frac{}{or\_\eta^t : t \to \langle t \rangle} \qquad \frac{}{or\_\cup^t : \langle t \rangle \times \langle t \rangle \to \langle t \rangle}$$

$$\frac{}{or\_\mu^t : \langle\langle t \rangle\rangle \to \langle t \rangle} \qquad \frac{}{or\_empty^t : unit \to \langle t \rangle} \qquad \frac{f : s \to t}{or\_map\ f : \langle s \rangle \to \langle t \rangle}$$

Interaction of sets and or-sets

$$\frac{}{\alpha^t : \{\langle t \rangle\} \to \langle\{t\}\rangle}$$

Figure 5.4: Syntax of or-$\mathcal{NRL}$

One of *or-*$\mathcal{NRL}$ primitives, α, is essentially a translation of conjunctive normal form into disjunctive normal form. This operation may be very expensive. Indeed, if its argument is a collection of n two-element or-sets, all $2n$ elements being distinct, then α produces an or-set containing 2^n n-element sets. The result that we are going to formulate can be intuitively understood as follows: the expressive power of α is that of *powerset*. However, *powerset* does not use the $\langle\rangle$ type constructor. To be able to speak of the equivalence of expressive power of languages one of which uses or-sets and the other does not, for technical purposes only, we introduce the functions $\mathit{or\_to\_set} : \langle t\rangle \to \{t\}$ and $\mathit{set\_to\_or} : \{t\} \to \langle t\rangle$ with the obvious semantics: $\mathit{or\_to\_set}(\langle x_1,\ldots,x_n\rangle) = \{x_1,\ldots,x_n\}$ and $\mathit{set\_to\_or}(\{x_1,\ldots,x_n\}) = \langle x_1,\ldots,x_n\rangle$. We remark here that, if *or_to_set* and *set_to_or* are given, then $\mathcal{NRL}$ and $\mathcal{NRL}^{\mathrm{or}}$ are interdefinable.

Proposition 5.14 *or-*$\mathcal{NRL}(\mathit{or\_to\_set}, \mathit{set\_to\_or}, \alpha) \cong$ *or-*$\mathcal{NRL}(\mathit{or\_to\_set}, \mathit{set\_to\_or}, \mathit{powerset})$.

Proof. First, *powerset* can be expressed as follows:

$$\mathit{powerset} = \mathit{or\_to\_set} \circ \alpha \circ \mathit{map}(\mathit{or\_}\cup \circ (\mathit{or\_}\eta \circ \mathit{empty} \circ !, \mathit{or\_}\eta \circ \eta))$$

Conversely, we must show that α is definable in *or-*$\mathcal{NRL}(\mathit{or\_to\_set}, \mathit{set\_to\_or}, \mathit{powerset})$. It is known that the test for equal cardinality can be implemented using *powerset* (see [26]). To check whether $\mathit{card}(X) \le \mathit{card}(Y)$, notice that

$$\mu \circ \mathit{map}(\lambda Z.\, \mathit{if}\ \mathit{equal\_card?}(X, Z)\ \mathit{then}\ X\ \mathit{else}\ \{\})(\mathit{powerset}(Y))$$

returns X if $\mathit{card}(X) \le \mathit{card}(Y)$ and $\{\}$ otherwise, thus giving us the test for lesser cardinality.

Now, given an input of type $\{\langle t\rangle\}$, first apply $\mathit{map}(\mathit{or\_to\_set})$ to it and then flatten the result, thus obtaining the set of elements that occur in the input. Applying *powerset* now gives the set of all sets of those elements. A set of elements of the input makes it to the output if and only if two conditions hold: first, its cardinality does not exceed the cardinality of the input (i.e. the number of or-sets) and it has a nonempty intersection with any element of the input, unless the input is $\{\}$. Since selection, lesser cardinality test, intersection and test for nonemptiness are definable in $\mathcal{NRL}$ selection over the powerset followed by an application of *set_to_or* yields the desired result. $\Box$

Example: Membership problem for equality tables in *or-*$\mathcal{NRL}$

As a simple example of applicability of *or-*$\mathcal{NRL}$ to classical problems of incomplete information in relational databases, we show how to use it to solve the membership problem for equality tables. Recall that equality tables are relations where variables can be used as well as nonpartial values, and each variable may occur more than once. The membership problem is to determine, given an equality table and a relation without variables, if the relation is a possible world for the table. That is, if it is possible to instantiate variables to values such that the table will be instantiated into the given relation. It is known that this problem is $\mathcal{NP}$-complete, so we can not hope to give a solution that does not use the expensive α.

For simplicity of exposition, assume that we have a base type b having both variables $x_1, \ldots$ and values $v_1, \ldots$ and that it is possible to distinguish between variables and values. A relation R is an object of type $\{b \times b\}$ such that no variable occurs in it. A table T is also an object of type $\{b \times b\}$ but now variables may occur.

It is possible to find the set of all variables that occur in T using the fact that *select* is defiable in $\mathcal{NRL}$:

$$\textsc{Var}_T \quad := \quad \mathit{select}(\mathit{is\_variable}) \circ \mathit{map}(\pi_1)(T) \;\cup\; \mathit{select}(\mathit{is\_variable}) \circ \mathit{map}(\pi_2)(T)$$

All values that occur in R can be found as

$$\textsc{Val}_R \quad := \quad \mathit{map}(\pi_1)(R) \;\cup\; \mathit{map}(\pi_2)(R)$$

We saw in the proof of proposition 5.14 that $\mathit{powerset}_{\mathrm{or}} : \{t\} \to \langle\{t\}\rangle$ is definable in *or*-$\mathcal{NRL}$. So, the next step is to compute $\mathit{powerset}_{\mathrm{or}}(\mathit{cartprod}(\textsc{Var}_T \times \textsc{Val}_R))$ and select those sets in it in which every variable from $\textsc{Var}_T$ occurs exactly once. We denote this resulting object of type $\langle\{b \times b\}\rangle$ by $\textsc{Assign}$.

Each element of $\textsc{Assign}$ can be viewed as an assignment of values to variables, so it can be applied to T in the following sense. For every x in $\textsc{Assign}$ (which is a set of pairs variable-value, we can write a function that substitutes each variable in T by the corresponding value, and then map this function over $\textsc{Assign}$. The reader is invited to see how such a function can be written in *or*-$\mathcal{NRL}$.

The resulting object is now X of type $\langle\{b \times b\}\rangle$ which is the or-set of all possible relations that can be obtained from T by using valuation maps whose values are in $\textsc{Val}_R$. Therefore, R is a possible world for T if and only if R is a member of X. To verify this, we write $\mathit{or\_map}(\lambda x.\mathit{eq}(x, R))(X)$ and then check if *true* is in the result. This gives us the membership test.

It is interesting to note that the membership problem for Codd tables, while being of polynomial time complexity, requires solving the bipartite matching problem which can be reformulated as a problem of finding a system of distinct representatives, see Abiteboul et al. [8]. Therefore, the power of $\mathcal{NRL}$ is too limited to solve the membership problem even for Codd tables. However, with the power of α, the language can solve a much more complicated membership for equality tables.

5.2.2 Normalization and conceptual programming

The main goal of this section is to show that every object involving or-sets has a unique representation of type $\langle t\rangle$ where t does not involve or-sets. That is, all possibilities encoded by or-objects can be listed and, moreover, in a way that is implementable in *or-$\mathcal{NRL}$*.

We start with a few examples in which we use the set-theoretic semantics. If a pair (x, y) of or-sets is given, say, $(\langle 1, 2\rangle, \langle 3, 4\rangle)$, on conceptual level we must deal with all possible objects it can conceptually stand for, that is, with or-set of pairs $\langle (1, 3), (1, 4), (2, 3), (2, 4)\rangle$. In this case the function that carries out transformation of structural representation to conceptual one can be given as $or\_\mu \circ or\_map(or\_\rho_1) \circ or\_\rho_2$. Another example of the passage from structural to conceptual level is given by the primitive $\alpha^s : \{\langle s\rangle\} \to \langle\{s\}\rangle$, provided that s is in the or-set free fragment.

Let us consider a more sophisticated example. Given an object $x = (\{\langle 1, 2\rangle, \langle 3\rangle\}, \langle 1, 2\rangle)$ of type $\{\langle int\rangle\} \times \langle int\rangle$. Denote the first component by y. Applying or_ρ_2 to x first yields $\langle (y, 1), (y, 2)\rangle$ which is an object of type $\langle\{\langle int\rangle\} \times int\rangle$. Applying $or\_map(\alpha \circ \pi_1, \pi_2)$ yields an object

$$\langle(\langle\{1, 3\}, \{2, 3\}\rangle, 1), (\langle\{1, 3\}, \{2, 3\}\rangle, 2)\rangle$$

of type $\langle\langle\{int\}\rangle \times int\rangle$. Finally, applying $or\_\mu \circ or\_map(or\_\rho_1)$ yields

$$\langle(\{1, 3\}, 1), (\{1, 3\}, 2), (\{2, 3\}, 1), (\{2, 3\}, 2)\rangle$$

of type $\langle\{int\} \times int\rangle$. This can be considered as a conceptual level object for all the possibilities are listed.

However, one could have used another strategy to list all the possibilities. For example, to apply $(\alpha \circ \pi_1, \pi_2)$ first to obtain an object of type $\langle\{int\}\rangle \times \langle int\rangle$ and then $or\_\mu \circ or\_map(or\_\rho_1) \circ or\_\rho_2$ to obtain an object of type $\langle\{int\} \times int\rangle$. It is easy to check that such a strategy results in precisely the same object as the previous one.

In fact, there is a general result saying that each type has a unique representation at the conceptual level – such that no or-set type occurs in the type expression except as the outermost type constructor. For reasons that should emerge shortly we call such a type *a normal form.* Furthermore, for each object of type t there exists its unique representation at the conceptual level whose type is the normal form of t.

To state these results precisely, introduce the rewrite rules for type expressions:

$$t \times \langle s\rangle \longrightarrow \langle t \times s\rangle \qquad\qquad \langle t\rangle \times s \longrightarrow \langle t \times s\rangle$$

$$\langle\langle t\rangle\rangle \longrightarrow \langle t\rangle \qquad\qquad \{\langle t\rangle\} \longrightarrow \langle\{t\}\rangle$$

Proposition 5.15 *The above rewrite system is terminating and Church-Rosser. The normal form $nf(t)$ for type t can be found as follows: If t does not use $\langle\rangle$, then $nf(t) = t$. Otherwise, remove all angle brackets from t. If the resulting type is t', then $nf(t) = \langle t'\rangle$.*

Proof. To show that the rewrite system is terminating, define the following function on types. Considering types as their derivation trees, let k_i be the number of occurrences of $\langle\rangle$ on the ith level of the derivation tree of type t. If the height of the derivation tree is n, define $\varphi(t)$ as $\Sigma_{i=1}^{n} k_i \cdot i$. It is easy to see that if $t \longrightarrow t_0$, then $\varphi(t) > \varphi(t_0)$. Hence, any rewriting terminates.

To prove Church-Rosserness, one has to find the critical pairs, see section 2.4, which in essence are pairs of terms that can give rise to ambiguity in rewriting, and show that for any critical pair (τ_1, τ_2) there exists a term τ such that $\tau_1 \longrightarrow\!\!\!\!\!\rightarrow \tau$ and $\tau_2 \longrightarrow\!\!\!\!\!\rightarrow \tau$. A straightforward analysis of our rewrite system reveals the following critical pairs: 1) $(\langle\{\langle t\rangle\}\rangle, \{\langle t\rangle\})$; 2) $(\langle t \times \langle s\rangle\rangle, t \times \langle s\rangle)$; 3) $(\langle\langle s\rangle \times t\rangle, \langle s \times \langle t\rangle\rangle)$ and 4) $(\langle\langle s\rangle \times t\rangle, \langle\langle s\rangle\rangle \times t)$ and their symmetric analogs. The terms to which both components of critical pairs rewrite are $\langle\{t\}\rangle$ for 1), $\langle t \times s\rangle$ for 2) and $\langle s \times t\rangle$ for 3) and 4). Thus, the rewrite system is Church-Rosser and, therefore, has unique normal forms.

The proof of the last statement is by induction on the structure of a given type. We limit ourselves only to types containing $\langle\rangle$. The base case is immediate. In general case, consider three subcases: 1) $t = t_1 \times t_2$, 2) $t = \{t_1\}$, 3) $t = \langle t_1\rangle$. In subcase 1, $t' = t_1' \times t_2'$, hence, if both t_1 and t_2 contain or-sets, $nf(t_1) = \langle t_1'\rangle, nf(t_2) = \langle t_2'\rangle$ and $t \longrightarrow\!\!\!\!\!\rightarrow \langle t_1'\rangle \times \langle t_2'\rangle \longrightarrow\!\!\!\!\!\rightarrow \langle t_1' \times t_2'\rangle = \langle t'\rangle$ which is a normal form. Thus, $nf(t) = \langle t'\rangle$. The simple proofs of other cases are omitted. □

Having defined rewrite rules for types, we must show how to apply these rules to instances. First, associate a function in *or-$\mathcal{NRL}$* with each rule as follows:

$$or\_\rho_2 : t \times \langle s\rangle \longrightarrow \langle t \times s\rangle \qquad or\_\rho_1 : \langle t\rangle \times s \longrightarrow \langle t \times s\rangle$$

$$or\_\mu : \langle\langle t\rangle\rangle \longrightarrow \langle t\rangle \qquad \alpha : \{\langle t\rangle\} \longrightarrow \langle\{t\}\rangle$$

In the case of using antichain semantics, that is, *or-$\mathcal{NRL}_a$*, we replace or_μ and α by or_μ_a and α_a respectively.

Let t be a type and p a position in the derivation tree for t such that applying a rewrite rule with associated function f to t at p yields type s. Our aim is to define a function $\mathsf{app}(t, p, f) : t \to s$ showing the action of rewrite rules on objects. Define it by induction on the structure of t:

- if p is the root of the derivation of t, then $\mathsf{app}(t, p, f) = f$;
- if $t = t_1 \times t_2$ and p is in t_1 , then $\mathsf{app}(t, p, f) = (\mathsf{app}(t_1, p, f) \circ \pi_1, \pi_2)$;
- if $t = t_1 \times t_2$ and p is in t_2, then $\mathsf{app}(t, p, f) = (\pi_1, \mathsf{app}(t_2, p, f) \circ \pi_2)$;
- if $t = \{t'\}$ then $\mathsf{app}(t, p, f) = map(\mathsf{app}(t', p, f))$;

- if $t = \langle t' \rangle$ then $\mathsf{app}(t, p, f) = or\_map(\mathsf{app}(t', p, f))$.

Notice that the definition of app relies on the fact that the functions associated with the rewrite rules are polymorphic. Again, for $or\text{-}\mathcal{NRL}_a$ we use corresponding operations with index a from $or\text{-}\mathcal{NRL}_a$, and denote the corresponding application function by app_a.

Given a type t and a rewriting strategy $r := t \xrightarrow{f_1} t_1 \xrightarrow{f_2} \ldots \xrightarrow{f_n} t_n = nf(t)$ such that the rewrite rule with associated function f_i is applied at a position p_i, we can extend the function app to $\mathsf{app}(t, r) : t \rightarrow nf(t)$ by

$$\mathsf{app}(t, r) := \mathsf{app}(t_{n-1}, p_n, f_n) \circ \ldots \circ \mathsf{app}(t_1, p_2, f_2) \circ \mathsf{app}(t, p_1, f_1)$$

We now formulate the main theorem which states that it is possible to compute all possibilities a given or-object represents, and that computation is "path independent". We discuss some important consequences of this result before giving a (somewhat lengthy) proof.

Theorem 5.16 (Normalization) *Given a type t, any two rewrite strategies $r_1, r_2 : t \longrightarrow nf(t)$ yield the same result on objects in or-$\mathcal{NRL}$ and or-$\mathcal{NRL}_a$. That is, for any object x of type t,*

$$\mathsf{app}(t, r_1)(x) = \mathsf{app}(t, r_2)(x) \qquad \textit{and} \qquad \mathsf{app}_a(t, r_1)(x) = \mathsf{app}_a(t, r_2)(x) \qquad \square$$

Therefore, all objects with the same meaning at the conceptual level rewrite to the same normal form. The intuitive notion of the conceptual meaning can now be rigorously defined as the normal form. So now we can define the *conceptual query language* $or\text{-}\mathcal{NRL}^+$ by adding the new construct

$$\frac{}{normalize^t : t \rightarrow nf(t)}$$

to $or\text{-}\mathcal{NRL}$. The conceptual query language for the antichain semantics $or\text{-}\mathcal{NRL}_a^+$ can be defined by adding $normalize_a^t : t \rightarrow nf(t)$ to $or\text{-}\mathcal{NRL}_a$.

By the normalization theorem, $normalize^t$ can be implemented as $\mathsf{app}(t, r)$ where $r : t \longrightarrow nf(t)$ and $normalize_a^t$ can be implemented as $\mathsf{app}_a(t, r)$. Notice that, for any given t, $normalize^t$ and $normalize_a^t$ can be expressed in $or\text{-}\mathcal{NRL}$ and $or\text{-}\mathcal{NRL}_a$ (maybe in more than one way) but it is impossible to express them *polymorphically.*

As an illustration of using normalization, consider the example with the incomplete design database. Assuming that the cost function $c(\cdot)$ is given for all pieces, it is possible to calculate the cost function *cost* for the complete designs. Now, to find out if it is possible to complete design using \$50, one would write

$$select(\lambda x.x > 50)(or\_map\ cost\ normalize(DESIGN))$$

(recall that *select* is definable in *or*-$\mathcal{NRL}$). To list the designs which cost exactly \$70, one would write

$$select(\lambda x.cost(x) = 70)(normalize(DESIGN))$$

Moreover, it is possible to express all examples of conceptual queries listed in the beginning of this section. We shall return to this example later in chapter 6 and show how those conceptual queries can be implemented in a practical language OR-SML which is based on *or*-$\mathcal{NRL}$.

Before we prove the normalization theorem, let us make one important observation. The normalization theorem states that $[\![\mathsf{app}(t, r_1)(x)]\!]_s = [\![\mathsf{app}(t, r_2)(x)]\!]_s$, no matter what r_1 and r_2. Of course, this also implies that the conceptual semantics of the two is the same. However, there is a much closer connection between normalization and the conceptual semantics. The slogan is:

Normalization preserves conceptual semantics.

In other words, the following holds.

Theorem 5.17 *For any type t and any object x of type t,*

$$[\![x]\!]_c = [\![normalize(x)]\!]_c \quad and \quad [\![x]\!]_c = [\![normalize_a(x)]\!]_c$$

That is, $[\![normalize]\!]_c = [\![normalize_a]\!]_c = [\![id]\!]_c$.

Proof. We prove this theorem for the antichain semantics; the proof for the set-theoretic semantics is similar (and in fact easier). We must show that all four operations used in the process of normalization do not change the conceptual semantics. We do it by cases. Recall that $[\![x]\!]_c$ is a finitely generated filter in $[\![t]\!]_c$ for any x of type t. First, we need to prove the following.

Claim. If x and y are of type t and $x \leq_t y$, then $[\![y]\!]_c \subseteq [\![x]\!]_c$.
Prove this by cases. The base type case and the product type case are immediate. Let $X = \langle x_1, \ldots, x_n \rangle$, $Y = \langle y_1, \ldots, y_m \rangle$ be of type $\langle t \rangle$ and let $X \leq_{\langle t \rangle} Y$. Then $\forall y_i \in Y \exists x_j \in X : x_j \leq_t y_i$ and hence $\forall y_i \in Y \exists x_j \in X : [\![y_j]\!]_c \subseteq [\![x_i]\!]_c$ and then $[\![Y]\!]_c \subseteq [\![X]\!]_c$.

Let $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_m\}$ be of type $\{t\}$ and let $X \leq_{\{t\}} Y$. Then $\forall x \in X \exists y \in Y : x \leq_t y$, that is, $[\![y]\!]_c \subseteq [\![x]\!]_c$. This also means $\min [\![x]\!]_c \sqsubseteq^\sharp \min [\![y]\!]_c$. Now consider $X' = \{\min [\![x_1]\!]_c, \ldots, \min [\![x_n]\!]_c\}$ and $Y' = \{\min [\![y_1]\!]_c, \ldots, \min [\![y_m]\!]_c\}$. Then $X' (\leq_t^\sharp)^\flat Y'$. Now recall from the proof of proposition 4.12 that $[\![X]\!]_c = \uparrow\alpha_a(X')$ where $\alpha_a(X')$ is considered as a collection of sets, and similarly $[\![Y]\!]_c = \uparrow\alpha_a(Y')$. From the proof of theorem 4.21 we know that $\alpha_a(X') (\leq_t^\flat)^\sharp \alpha_a(Y')$ which means $\alpha_a(X') \leq_{\{t\}}^\sharp \alpha_a(Y')$ and hence $\uparrow\alpha_a(Y') \subseteq \uparrow\alpha_a(X')$. Therefore, $[\![Y]\!]_c \subseteq [\![X]\!]_c$. This finishes the proof of the claim. Now we prove that all operations used in the process of normalization preserve $[\![]\!]_c$.

Case 1: or_ρ_2. Let $[\![x]\!]_c = F_x$ and $[\![y_i]\!]_c = F_i$ for $i = 1, \ldots, n$. Then for $Y = \langle y_1, \ldots, y_n \rangle$ we have $[\![Y]\!]_c = \bigcup_{i=1}^n F_i = F_Y$ and $[\![(x, Y)]\!]_c = F_x \times F_Y$. On the other hand, $[\![or\_\rho_2(x, Y)]\!]_c =$

$[\![\langle (x, y_1), \ldots, (x, y_n) \rangle]\!]_c = \bigcup_{i=1}^n (F_x \times F_i) = F_x \times \bigcup_{i=1}^n F_i = F_x \times F_Y$. Hence, $[\![(x, Y)]\!]_c = [\![or\_\rho_2(x, Y)]\!]_c$. The case of or_ρ_1 is similar.

Case 2: or_μ_a. Let $\mathcal{X} = \langle X_1, \ldots, X_n \rangle$ and $X_i = \langle x_1^i, \ldots, x_{n_i}^i \rangle$ for $i = 1, \ldots, n$. Let $[\![x_j^i]\!]_c = F_j^i$. Then $[\![\mathcal{X}]\!]_c = \bigcup_i \bigcup_j F_j^i$. By monotonicity of $[\![]\!]_c$, we obtain

$$[\![\mathcal{X}]\!]_c = \bigcup_{x_j^i \in \min(X_1 \cup \ldots \cup X_n)} F_j^i$$

and hence $[\![\mathcal{X}]\!]_c = [\![or\_\mu_a(\mathcal{X})]\!]_c$.

Case 3: α_a. Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ where $X_i = \langle x_1^i, \ldots, x_{n_i}^i \rangle$ for $i = 1, \ldots, n$. Let $[\![x_j^i]\!]_c = F_j^i$. Then $[\![\mathcal{X}]\!]_c$ is the filter generated by such finite antichains Y that $Y \cap F^i \neq \emptyset$ for all $i = 1, \ldots, n$ where $F^i = \bigcup_{j=1}^{n_i} F_j^i$. Now, $\alpha_a(\mathcal{X}) = \min_{\sqsubseteq^\flat} \langle \max\{f(x_{f(i)}^i) \mid i = 1, \ldots, n\} \mid f \in \mathcal{F}_{\mathcal{X}} \rangle$. Therefore, by monotonicity of $[\![]\!]_c$, $[\![\alpha_a(\mathcal{X})]\!]_c$ is the filter generated by all finite antichains Y such that for every $i = 1, \ldots, n$, $Y \cap F_{f(i)}^i \neq \emptyset$ for at least one $f \in \mathcal{F}_{\mathcal{X}}$. Now it is easy to see that $[\![\alpha_a(\mathcal{X})]\!]_c = [\![\mathcal{X}]\!]_c$.

Therefore, all operations used in normalization do not change the value of $[\![]\!]_c$ and hence $[\![x]\!]_c = [\![normalize_a(x)]\!]_c$ for any x. □

Proof of the normalization theorem

We start with normalization for the set-theoretic semantics. Let us first explain the strategy for proving the theorem. We define an abstract rewrite system on objects by letting $x \to y$ iff y can be obtained from x by application of one of the rewrite rules for types to x (by means of app). For instance, $(1, \langle\langle 1 \rangle, \langle 2 \rangle\rangle) \to (1, \langle 1, 2 \rangle)$ by applying $\langle\langle t \rangle\rangle \to \langle t \rangle$ in the second position. If x is of type t and y is of type s, then $t \to s$ according to the rewrite system for types. Moreover, normal forms with respect to our new rewrite system are precisely objects whose types are normal form. Therefore, the rewrite system is terminating according to proposition 5.15.

Now our goal is to prove that the new rewrite system is weakly Church-Rosser. Then, by Newman's lemma (see section 2.4) it will follow that it is Church-Rosser and has unique normal forms. Since for a rewriting r from t to s, $y = \mathsf{app}(t, r)(x)$ implies that $x \twoheadrightarrow y$, the uniqueness of normal forms will imply the normalization theorem.

To prove weak Church-Rosserness, we have to show that for $x \to x_1$ and $x \to x_2$, there exists x' such that $x_1, x_2 \twoheadrightarrow x'$. We shall often view types as trees. Assume that $x \to x_1$ by means of rule r_1 in position p_1 in t and $x \to x_2$ by means of rule r_2 in position p_2 in t. We denote the functions that correspond to rules r_1 and r_2 by f_1 and f_2 respectively. Notice that if positions p_1 and p_2 are in two different subtrees determined by a pair formation, then the existence of x' is immediate. Hence, we can assume that one position, say p_1, is closer to the root than p_2 because $\{\}$ and $\langle\rangle$ are unary type constructors.

Now we prove weak Church-Rosserness by cases which are given by the rewrite rules applied in position p_1. Subcases will be given by rewrite rules applied in position p_2.

Case 1. The rule applied in p_1 *is* $s \times \langle t \rangle \to \langle s \times t \rangle$. The object therefore is a pair (x_1, x_2) and the function applied is or_ρ_2. Now we have three subcases.

Subcase 1.1. p_2 occurs inside the tree for s. Assume that $\mathsf{app}(s, p_2, r_2)(x_1) = x_1'$. Then we obtain

$$\begin{tikzcd}
& \langle (x_1, x_2^i) \mid x_2^i \in x_2 \rangle \arrow[dr, "or_map(f_2 \circ \pi_1, \pi_2)"] & \\
(x_1, x_2) \arrow[ur, "or_\rho_2"] \arrow[dr, "(f_2 \circ \pi_1, \pi_2)"'] & & \langle (x_1', x_2^i) \mid x_2^i \in x_2 \rangle \\
& (x_1', x_2) \arrow[ur, "or_\rho_2"'] &
\end{tikzcd}$$

Subcase 1.2. p_2 occurs inside t. That is, rewriting is apllied to elements of or-set x_2. For $x_2 = \langle y_i \rangle_i$, assume that $f_2(y_i) = z_i$. Then we obtain

$$\begin{tikzcd}
& \langle (x_1, y_i) \mid y_i \in x_2 \rangle \arrow[dr, "or_map(\pi_1, f_2 \circ \pi_2)"] & \\
(x_1, x_2) \arrow[ur, "or_\rho_2"] \arrow[dr, "(\pi_1, or_map(f_2) \circ \pi_2)"'] & & \langle (x_1, z_i) \mid y_i \in x_2 \rangle \\
& (x_1, \langle z_i \rangle_i) \arrow[ur, "or_\rho_2"'] &
\end{tikzcd}$$

Subcase 1.3. p_2 coincides with the root of $\langle t \rangle$. Since the root of $\langle t \rangle$ is the or-set type, the only rule that can be applied is $\langle \langle t' \rangle \rangle \to \langle t' \rangle$, that is, $t = \langle t' \rangle$. Now assume $x_2 = \langle X_1, \ldots, X_p \rangle$ where each X_i is an or-set of type $\langle t' \rangle$. Let $V = X_1 \cup \ldots \cup X_p$. Then we obtain

$$\begin{tikzcd}
 & \langle (x_1, X^i) \rangle_{i=1}^{p} \arrow[r, "or_map(or_\rho_2)"] & \langle \langle (x, y) \mid y \in X_i \rangle \mid i = 1, \ldots, p \rangle \arrow[dr, "or_\mu"] & \\
(x_1, x_2) \arrow[ur, "or_\rho_2"] \arrow[dr, "(\pi_1, or_\mu \circ \pi_2)"'] & & & \langle (x_1, y) \mid y \in V \rangle \\
 & (x_1, V) \arrow[r, "id"] & (x_1, V) \arrow[ur, "or_\rho_2"'] &
\end{tikzcd}$$

Case 2. The rule applied in p_1 *is* $\langle\langle t \rangle\rangle \to \langle t \rangle$. The object therefore is an or-set of or-sets $\mathcal{X} = \langle X_1, \ldots, X_p \rangle$ where $X_i = \langle x_1^i, \ldots, x_{n_i}^i \rangle$ and the function applied is or_μ. Now we have two subcases.

Subcase 2.1. p_2 occurs inside the tree for t. Assume that for each element x_j^i in X_i we have $f_2(x_j^i) = y_j^i$. Let $Y_i = or\_map(f_2)(X_i)$. Then we obtain

$$\begin{tikzcd}
 & \langle x_j^i \mid i = 1, \ldots, p, j = 1, \ldots, n_i \rangle \arrow[dr, "or_map(f_2)"] & \\
\mathcal{X} \arrow[ur, "or_\mu"] \arrow[dr, "or_map(or_map(f_2))"'] & & \langle y_j^i \mid i = 1, \ldots, p, j = 1, \ldots, n_i \rangle \\
 & \langle Y_i \mid i = 1, \ldots, p \rangle \arrow[ur, "or_\mu"'] &
\end{tikzcd}$$

Subcase 2.1. p_2 is the root of $\langle t \rangle$, that is, p_2 is the immediate successor of p_1. Hence, the only rule that can be applied at p_2 is $\langle\langle s \rangle\rangle \to \langle s \rangle$. In other words, $t = \langle s \rangle$ and $\mathcal{X}$ has type $\langle\langle\langle s \rangle\rangle\rangle$. Rewriting at p_2 is then $or\_map(or\_\mu)$. Therefore, two reducts of $\mathcal{X}$ are $or\_\mu(\mathcal{X})$ and $or\_map(or\_\mu(\mathcal{X}))$. The case now holds because $or\_\mu \circ or\_map(or\_\mu)(\mathcal{X}) = or\_\mu \circ or\_\mu(\mathcal{X})$.

Case 3. The rule applied in p_1 *is* $\{\langle t \rangle\} \to \langle\{t\}\rangle$. The object therefore is a set of or-sets $\mathcal{X} = \{X_1, \ldots, X_p\}$ where $X_i = \langle x_1^i, \ldots, x_{n_i}^i \rangle$ and the function applied is α. Now we have two subcases.

Subcase 3.1. p_2 is inside t. Assume that applying f_2 to every x^i_j yields y^i_j. The result of applying f_2 (in the sense of app) to $\mathcal{X}$ is $\{\langle y^i_j \mid j = 1, \ldots, n_i\rangle i = 1, \ldots, p\}$. Now we can see that the following diagram commutes and hence the case holds.

$$
\begin{tikzcd}
 & \langle\{x^i_{h(i)} \mid i = 1, \ldots, p\} \mid h \in \mathcal{F}_{\mathcal{X}}\rangle \arrow[dr, "or\_map(map(f_2))"] & \\
\mathcal{X} \arrow[ur, "\alpha"] \arrow[dr, "map(or\_map(f_2))"'] & & \langle\{y^i_{h(i)} \mid i = 1, \ldots, p\} \mid h \in \mathcal{F}_{\mathcal{X}}\rangle \\
 & \{\langle y^i_j \mid j = 1, \ldots, n_i\rangle \mid i = 1, \ldots, p\} \arrow[ur, "\alpha"'] &
\end{tikzcd}
$$

Subcase 3.2. p_2 is the root of $\langle t\rangle$. In this case the only rule that can be applied is $\langle\langle t'\rangle\rangle \to \langle t'\rangle$ and hence $t = \langle t'\rangle$. In particular, applying f_2 now is $map(or\_\mu)$. Now it can be seen that the following diagram commutes which proves the case. In that diagram we only give types of intermediate objects.

$$
\begin{tikzcd}
 & \bullet : \langle\{\langle t'\rangle\}\rangle \arrow[r, "or\_map(\alpha)"] & \bullet : \langle\langle\{t'\}\rangle\rangle \arrow[dr, "or\_\mu"] & \\
\mathcal{X} : \{\langle\langle t'\rangle\rangle\} \arrow[ur, "\alpha"] \arrow[dr, "map(or\_\mu)"'] & & & \bullet : \langle\{t'\}\rangle \\
 & \bullet : \{\langle t'\rangle\} \arrow[r, "id"] & \bullet : \{\langle t'\rangle\} \arrow[ur, "\alpha"'] &
\end{tikzcd}
$$

This finishes the proof that for the set-theoretic semantics the rewrite system is weak Church-Rosser and therefore the normalization theorem holds.

To prove normalization for or-$\mathcal{NRL}_a$, recall the translation from the set-theoretic semantics into the antichain semantics: $x^\circ = x$ for any x of base type, $(x, y)^\circ = (x^\circ, y^\circ)$, $\{x_1, \ldots, x_n\}^\circ = \max\{x_1^\circ, \ldots, x_n^\circ\}$ and $\langle x_1, \ldots, x_n\rangle^\circ = \min\langle x_1^\circ, \ldots, x_n^\circ\rangle$. Now we need two lemmas.

Lemma 5.18 *Any function f in the fragment of* or-$\mathcal{NRL}$ *that does not contain* $\cup$, *or*_$\cup$ *and eq, is monotone.*

Proof is by induction. We consider only a few cases. Most cases, such as projections, pairing, composition, singleton and pair-with are immediate. That α is monotone follows from theorem 4.21. Let $\mathcal{X}, \mathcal{Y}$ be of type $\{\{t\}\}$ and $\mathcal{X} \leq_{\{\{t\}\}} \mathcal{Y}$. Then consider $x \in \mu(\mathcal{X})$. Since $x \in X$ for some $X \in \mathcal{X}$, there exists $Y \in \mathcal{Y}$ such that $X \leq_{\{t\}} Y$ and then there exists $y \in Y$ such that $x \leq_t y$. This shows $\mu(\mathcal{X}) \leq_{\{t\}} \mu(\mathcal{Y})$. The proof for or_μ is similar.

Assume that $g : t \to s$ is monotone and consider $X, Y : \{t\}$ such that $X \leq_{\{t\}} Y$. Let $g(x) \in map(g)(X)$. Then there exists $y \in Y$ such that $x \leq_t y$ and hence $g(x) \leq_s g(y)$ which shows $map(g)(X) \leq_{\{s\}} map_a(g)(Y)$. The proof for or_map is similar. □

Let f be a function definable in $or\text{-}\mathcal{NRL}$. By f_a we denote the corresponding function in $or\text{-}\mathcal{NRL}_a$, that is, the function obtained from f by replacing set-theoretic operations with their antichain counterparts, e.g. by replacing or_map with or_map_a and so on.

Lemma 5.19 *Let f be a function in the fragment of or-$\mathcal{NRL}$ that does not contain $\cup$, $or\_\cup$, equality and comparability tests. Then for any object x, $f_a(x^\circ) = f(x^\circ)^\circ$.*

The proof is again by induction on f. We show a few cases here. The proof for α is easily derived from theorem 4.21. Given $\mathcal{X}$ of type $\{\{t\}\}$, consider $\mu_a(\mathcal{X}^\circ)$. It is easy to see that $\mu_a(\mathcal{X}^\circ) = \max(\mu(\max\{X^\circ \mid X \in \mathcal{X}\})) = \mu(\mathcal{X}^\circ)^\circ$ and so the case holds. The case for map, observe that $map_a(g_a)(X^\circ) = \max(map(g_a)(X)) = \max\{g_a(x^\circ) \mid x^\circ \in X^\circ\} = \max\{g(x^\circ)^\circ \mid x^\circ \in X^\circ\} = map(g)(X^\circ)^\circ$. Similarly, the case for or_map holds. Finally, consider $h = f \circ g$ where $g : s \to t$ and $f : t \to u$. Then by induction hypothesis $h_a(x^\circ) = f(g(x^\circ)^\circ)^\circ$ and $h(x^\circ)^\circ = f(g(x^\circ))^\circ$. Since g and f come from a monotone fragment of $or\text{-}\mathcal{NRL}$, we obtain $g(x^\circ) \leq_t g(x^\circ)^\circ \leq_t g(x^\circ)$ and therefore $f(g(x^\circ)) \leq_u f(g(x^\circ)^\circ) \leq_u f(g(x^\circ))$ which shows $f(g(x^\circ)^\circ)^\circ = f(g(x^\circ))^\circ$ and hence $h_a(x^\circ) = h(x^\circ)^\circ$. This finishes the proof. □

To prove normalization for the antichain semantics, we define the rewrite system on objects in exactly the same way we did it for the set-theoretic semantics. Now our goal is to show that the system is weakly Church-Rosser.

Assume that we have an object x in the antichain semantics, that is, $x = x^\circ$, and assume that it can be rewritten to two objects x_1 and x_2. That is, there exist two functions f and g which are in fact instances of app_a such that $x_1 = f_a(x)$ and $x_2 = g_a(x)$. Let $y_1 = f(x)$ and $y_2 = g(x)$. By the proof of normalization theorem for set-theoretic semantics we know that there exists an object z (in set-theoretic semantics) such that both y_1 and y_2 rewrite to z. That is, for some function f' and g', which are compositions of instances of app, we have $f'(y_1) = g'(y_2) = z$. Now using the fact that $\cup, or\_\cup$ and eq are not present in functions that arise as instances of app and hence these functions are monotone, we apply the previous lemma to obtain $f'_a(x_1) = f'_a(f_a(x)) = f'(f(x))^\circ = z^\circ = g'(g(x))^\circ = g'_a(g_a(x)) = g'_a(x_2)$. Since f'_a and g'_a are compositions of instances of app_a, this means that the rewrite system is weakly Church-Rosser and normalization for the antichain semantics follows.

5.2.3 Partial normalization

We have seen that the normalization process can be quite expensive. Indeed, since α has essentially the expressive power of *powerset* and can be applied several times in the course of normalization, the resulting object may be of at least exponential size. In section 5.2.5 we shall give tight upper bounds for the costs of normalization. Meanwhile, we would like to ask another question. Is it possible to answer conceptual queries faster?

First, we are going to show that even simple existential queries like "is there a complete design that costs less than \$50?" can be very expensive. Then we proceed to suggest a method that occasionally allows to answer queries without completing the normalization process.

The importance of existential queries was emphasized in Imielinski et al. [80, 81]. Essentially, an existential query asks whether there exists a possibility – in the normal form – satisfying a given property. In terms of $or\text{-}\mathcal{NRL}^+$, if $nf(s) = \langle t \rangle$ and $p : t \to bool$ is a predicate, $\exists(p) : \langle t \rangle \to bool$ is a predicate which is true of $y : \langle t \rangle$ if $or\_map(p)(y) : \langle bool \rangle$ is an or-set containing the true value. Given an object y of type s, one may ask a query $\exists(p)(normalize(y))$. Clearly, this query can be answered in time polynomial in the size of $normalize(y)$, but can it be answered in time polynomial in the size of y?

The following example gives a negative answer to this question, provided $\mathcal{P} \neq \mathcal{NP}$. Assume $p_k : \{t\} \to bool$ evaluates to *true* if and only if cardinality of the set is at most k. Let b a base type. For an object x of type $\{\langle b \rangle\}$, one may ask a query $Q(k, x) = \exists(p_k)(normalize(x))$. It is immediately seen that this query evaluates to *true* iff there exists a system of distinct representatives of elements of x (which are or-sets) whose size is at most k. The problem of finding a system of distinct representatives of size $\leq k$ is known to be $\mathcal{NP}$-complete, see [56]. Therefore, the problem whether $Q(k, x)$ evaluates to *true* is $\mathcal{NP}$-complete.

Thus, there is no hope that even simple existential queries can be answered efficiently. Does that mean we always have to go through the whole process of normalization? Not necessarily so. Consider the following query about the incomplete design in figure 5.3. Is it possible to build part A using \$45? Of course we do not have to normalize the whole DESIGN but only the A component. In other words, instead of normalizing an object of type $\langle \{\langle t \rangle\} \rangle \times (\langle t \rangle \times \langle t \rangle)$ and getting an object of type $\langle \{t\} \times (t \times t) \rangle$, it is enough to get an object of type $\langle \{t\} \rangle \times (\langle t \rangle \times \langle t \rangle)$, leaving the B component intact.

The question that naturally arises is whether it is possible to do this unambiguously. That is, if $t \twoheadrightarrow t'$, and r_1 and r_2 are two strategies that perform this rewriting, is it true that $\mathsf{app}(t, r_1)$ and $\mathsf{app}(t, r_2)$ are the same as functions of type $t \to t'$?

It is not hard to see that the answer to this question is negative, as shown in example in figure 5.5.

$$\begin{tikzcd}
\langle\langle\langle 1,2\rangle,\langle 3\rangle\rangle,\langle\langle 4\rangle,\langle 5\rangle\rangle\rangle : \langle\langle\langle int\rangle\rangle\rangle \arrow[d, "or_map(or_\mu)"'] \arrow[dr, "or_\mu"] & \\
\langle\langle 1,2,3\rangle,\langle 4,5\rangle\rangle : \langle\langle int\rangle\rangle & \langle\langle 1,2\rangle,\langle 3\rangle,\langle 4\rangle,\langle 5\rangle\rangle : \langle\langle int\rangle\rangle
\end{tikzcd}$$

Figure 5.5: A counterexample to unambiguity of partial rewriting

However, the result that we are going to prove says that this is essentially the only possible counterexample. We need a couple of definitions first.

Definition 5.1 *A* μ-type *is a type that does not contain a subtype* $\langle\langle t\rangle\rangle$. *A* μ-rewrite strategy $r : t \longrightarrow\!\!\!\!\!\rightarrow s$ *between two* μ*-types* t *and* s *is a rewrite strategy such that, whenever a subtype* $\langle\langle t'\rangle\rangle$ *appears as the result of application of a rewrite rule, the next rewrite rule is* $\langle\langle t'\rangle\rangle \to \langle t'\rangle$.

For example, $\langle t\rangle \times \langle t \times \{\langle t\rangle\}\rangle$ is a μ-type and

$$\langle t\rangle \times \langle t \times \{\langle t\rangle\}\rangle \longrightarrow \langle t\rangle \times \langle t \times \langle\{t\}\rangle\rangle \longrightarrow \langle t\rangle \times \langle\langle t \times \{t\}\rangle\rangle \longrightarrow \langle t\rangle \times \langle t \times \{t\}\rangle$$

is a μ-rewrite strategy. Notice that it does not go all the way to the normal form.

Now the slogan is

The normalization theorem holds for μ*-rewrite strategies between* μ*-types.*

Before we formulate and prove the partial normalization theorem, we need a few results dealing with the structure of types involving the or-set type constructor. Recall that by $t \longrightarrow\!\!\!\!\!\rightarrow s$ we mean that t can be rewritten to s in zero or more steps using the four rules given before proposition 5.15. Now we write $t \prec s$ if s is obtained from t by removing some of the or-set brackets. In other words, s is obtained from t by applying the rules in figure 5.6.

Now define a binary relation $\lhd$ on types by means of four rules in figure 5.7.

Theorem 5.20 *Rules in figure 5.7 are sound and complete for* $\longrightarrow\!\!\!\!\!\rightarrow$. *In other words,* $s \longrightarrow\!\!\!\!\!\rightarrow t$ *iff* $s \lhd t$.

Proof. First prove the following. Let s_{or} be a type obtained from s by inserting a pair of or-set brackets. In terms of trees, it just means inserting a new node marked by $\langle\rangle$ somewhere.

$$
\frac{}{\langle t\rangle \prec t} \qquad \frac{t \prec s}{\langle t\rangle \prec s} \qquad \frac{t \prec s}{\langle t\rangle \prec \langle s\rangle} \qquad \frac{t \prec s}{\{t\} \prec \{s\}}
$$

$$
\frac{t \prec s}{t \times t' \prec s \times t'} \qquad \frac{t \prec s}{t' \times t \prec t' \times s} \qquad \frac{t \prec s \quad t' \prec s'}{t \times t' \prec s \times s'}
$$

Figure 5.6: Rules for $\prec$

$$
\frac{}{t \lhd t} \qquad \frac{t \lhd t' \qquad s \lhd s'}{t \times t' \lhd s \times s'}
$$

$$
\frac{t \lhd s}{\{t\} \lhd \{s\}} \qquad \frac{t \prec t' \qquad t' \lhd s}{t \lhd \langle s\rangle}
$$

Figure 5.7: Rules for $\lhd$

Then $s_{\mathrm{or}} \twoheadrightarrow \langle s\rangle$. We prove this by cases. If s is a base type, then s_{or} could only be $\langle s\rangle$. If $s = s_1 \times s_2$, then in the case when or-set brackets are put around s, we are done. Assume or-set brackets are inserted inside s_1. Then $s_{\mathrm{or}} = s_{1\mathrm{or}} \times s_2 \twoheadrightarrow \langle s_1\rangle \times s_2 \twoheadrightarrow \langle s_1 \times s_2\rangle = \langle s\rangle$. Assume $s = \{s'\}$. Again, if we put or-set brackets around s, we are done. Assume that a new pair of or-set brackets is put in s'. Then $s_{\mathrm{or}} = \{s'_{\mathrm{or}}\} \twoheadrightarrow \{\langle s'\rangle\} \twoheadrightarrow \langle\{s'\}\rangle = \langle s\rangle$. The proof for $s = \langle s'\rangle$ is similar. Therefore, if $t \prec s$, then t was obtained from s by inserting a number of pairs of or-set brackets in s and hence $t \twoheadrightarrow \langle s\rangle$.

Now we prove soundness of the rules in figure 5.7. The first three rules are obvious, so only the last one needs to proved. Assume that we know $t' \twoheadrightarrow s$ and let $t \prec t'$. We must show $t \twoheadrightarrow \langle s\rangle$. By the remark made above we obtain $t \twoheadrightarrow \langle t'\rangle$. Therefore, $t \twoheadrightarrow \langle t'\rangle \twoheadrightarrow \langle s\rangle$, which proves soundness.

To prove completeness, we must show how to derive all four rewrite rules for types from the rules in figure 5.7. First, we obtain

$$
\frac{\langle t\rangle \times s \prec t \times s \qquad \overline{t \times s \lhd t \times s}}{\langle t\rangle \times s \lhd \langle t \times s\rangle} \qquad\qquad \frac{t \times \langle s\rangle \prec t \times s \qquad \overline{t \times s \lhd t \times s}}{t \times \langle s\rangle \lhd \langle t \times s\rangle}
$$

For the rules for sets and or-sets, we have

$$\frac{\{\langle t\rangle\} \prec \{t\} \qquad \overline{\{t\} \lhd \{t\}}}{\{\langle t\rangle\} \lhd \langle\{t\}\rangle} \qquad\qquad \frac{\langle\langle t\rangle\rangle \prec t \qquad \overline{t \lhd t}}{\langle\langle t\rangle\rangle \lhd \langle t\rangle}$$

Finally, we need to show that if a subtype s of a type t rewrites to s', then t rewrites to $t[s'/s]$. In other words, if $s \lhd s'$, then $t \lhd t[s'/s]$. We prove it by induction on the structure of t. If the position of s is the immediate successor of a product or a set node, then this follows immediately from the rules in figure 5.7. Now assume that the position of s is the immediate successor of the or-set node. Then we obtain

$$\frac{\langle s\rangle \prec s \qquad \begin{matrix}\vdots \\ s \lhd s'\end{matrix}}{\langle s\rangle \lhd \langle s'\rangle}$$

as required. This finishes the proof of the theorem. □

The last rule in figure 5.7 resembles the cut rule in the sequent calculus [58] as it introduces a new variable t'. In the sequent calculus it is possible to eliminate the cut rules but the cost is the hyperexponential blow-up in the length of the proof, see Girard [58]. The last rule in figure 5.7 does not suggest an immediate search strategy to prove that $t \lhd \langle s\rangle$ but rather a search for the right t'. Thus, the following question arises. Given two types t and s, how hard is it to check if $t \longrightarrow\!\!\!\!\!\rightarrow s$? One may fear that it is at least exponential in the size of s and t, as suggested by the rules for $\lhd$. Fortunately, we can prove the following result.

Proposition 5.21 *There exists a $O(n^2)$ time complexity algorithm that, given two types s and t, returns* true *if $s \longrightarrow\!\!\!\!\!\rightarrow t$ and* false *otherwise.*

Proof. First, define a *carcass* of type t, denoted by $\hat{t}$, as follows. If $nf(t) = \langle t'\rangle$, then $\hat{t} = t'$, otherwise $\hat{t} = t$. Now, according to proposition 5.15, $s \longrightarrow\!\!\!\!\!\rightarrow t$ implies $\hat{s} = \hat{t}$. Therefore, we assume that the first stage of the algorithm is to check that $\hat{s} = \hat{t}$. This can be done in linear time in the size of t and s.

Assume s and t are given such that $s \longrightarrow\!\!\!\!\!\rightarrow t$. The proof of theorem 5.20 gives us a translation of any rewriting strategy into a proof using the rules for $\lhd$. Analysing these rules, we see that all of them are forced except the case when $t = \langle t'\rangle$. That is, if $t = t_1 \times t_2$, then we should have $s = s_1 \times s_2$ and $s_1 \lhd t_1$ and $s_2 \lhd t_2$ must be proved. If $t = \{t'\}$, then we should have $s = \{s'\}$ and $s' \lhd t'$ must be proved.

Assume that $t = \langle t'\rangle$. Analyzing the translation from $\longrightarrow\!\!\!\!\!\rightarrow$ into $\lhd$ given in the proof of theorem 5.20, we can see that there are only three instances of applying this rule to show $s \lhd \langle t'\rangle$. In three of them, the subproof for the $\lhd$ relation is a one-step proof (equality).

Therefore, when we have to prove $s \lhd \langle t' \rangle$, we do the following. First, we check if $t' = t'_1 \times t'_2$. If this is so, we check if $s = s_1 \times s_2$. If this is again so, we check if $s_1 = \langle t'_1 \rangle$ and $s_2 = t'_2$. If this is so, we stop since we succeeded in proving $s \lhd \langle t' \rangle$. If this is not so, the rule for product could not have been used in the translation of $\twoheadrightarrow$ to $\lhd$.

Next we check if $t' = \{t''\}$. If this is so, we check if $s = \{\langle t'' \rangle\}$ and if this is the case, we stop as we succeeded in proving $s \lhd \langle t' \rangle$. If this is not so, the rule for sets could not have been used in the translation of $\twoheadrightarrow$ to $\lhd$.

Then we check if $s = \langle\langle t'' \rangle\rangle$. If this is so, we stop as $s \lhd \langle t' \rangle$ is proved. If this is not so, the rule for or-sets of or-sets could not have been used in the translation of $\twoheadrightarrow$ to $\lhd$.

If going through these steps the algorithm does not stop, the translation from $\twoheadrightarrow$ to $\lhd$ tells us that the only way to prove $s \lhd \langle t' \rangle$ is to check that $s = \langle s' \rangle$ and to prove $s' \lhd t'$. Hence, we remove one or-set type constructor and then repeat all steps for the simpler types s' and t'. The goal is proved when all its subgoals are proved, that is, in proving each subgoal the algorithm stops returning success.

Analyzing this algorithm, we see that after each step the goal is reduced to a simpler subgoal (or two of them in the case of product) and that the only operations performed are a constant number of equality tests which can be done in linear time. Since the number of equality tests performed is linear in the size of the input, the time complexity of the algorithm is $O(n^2)$. $\square$

Since sizes of types are typically small (as compared to sizes of objects), this $O(n^2)$ algorithm will work very fast. Notice that we assumed that types are represented as trees. This is the case in the implementation called OR-SML which we shall describe in the next chapter. Had types been given as strings, due to the simple grammar for types, they can be parsed by an LR parser to obtain the tree representation in linear time [13]. Hence, the algorithm for checking $s \lhd t$ is still of $O(n^2)$ time complexity.

Our main goal is to prove the normalization theorem for μ-rewrite strategies between μ-types. The first question is how to obtain μ-rewrite strategies and μ-types. Let us see why the naive approach would not work. Given a type t, define $\mathsf{M}t$ as the type obtained by deleting multiple or-set brackets from t. That is, $\mathsf{M}b = b$, $\mathsf{M}s \times t = \mathsf{M}s \times \mathsf{M}t$, $\mathsf{M}\{t\} = \{\mathsf{M}t\}$, $\mathsf{M}\langle\langle t \rangle\rangle = \mathsf{M}\langle \mathsf{M}t \rangle$ and $\mathsf{M}\langle t \rangle = \langle \mathsf{M}t \rangle$ if t is not of form $\langle t' \rangle$. Obviously, for any t, $\mathsf{M}t$ is a μ-type. Now given two types t and s such that $t \lhd s$, is it true that $\mathsf{M}t \lhd \mathsf{M}s$.

The answer to this question is negative. Indeed, take $t = b \times \langle\langle b \rangle\rangle$ and $s = \langle b \times \langle b \rangle\rangle$, where b is a base type. Then $t \lhd s$, but $\mathsf{M}t = b \times \langle b \rangle \not\lhd \mathsf{M}s = \langle b \times \langle b \rangle\rangle$. However, we still can prove the following result.

Proposition 5.22 *If s and t are two μ-types such that $s \lhd t$, then there exists a μ-rewrite strategy that rewrites s to t.*

Proof. Let $s \lhd t$. Then there is a rewrite strategy that rewrites s to t, i.e. $s \longrightarrow\!\!\!\!\!\rightarrow t$. Consider the first step at which the condition for μ-rewrite strategy is violated. That is, in some reduct s' of s a subtype of form $\langle\langle t_0\rangle\rangle$ appeared, but the next rule is not the one that rewrites $\langle\langle t_0\rangle\rangle$ to $\langle t_0\rangle$. Since t is a μ-type and does not have double or-set brackets, this pair of or-set brackets must disappear in the process of rewriting. There are three possible cases.

Case 1. The product rule is used to eliminate the double or-set brackets. That is, t_0 may have been rewritten to some t_0' and then the rule $\langle\langle t_0'\rangle\rangle \times t_1' \rightarrow \langle\langle t_0'\rangle \times t_1'\rangle$ was used. According to the rules for $\lhd$, this means that in s', $\langle\langle t_0\rangle\rangle$ appeared in the context $\langle\langle t_0\rangle\rangle \times t_1$ and that $t_1 \lhd t_1'$. Since s is a μ-type and s' is the first reduct in which a pair of or-set brackets appeared that was not canceled at the next step, there are two possible ways for it to appear.

Subcase 1.1. $t_0 = t_{01} \times t_{02}$ and a pair of or-set brackets around t_0 appeared by applying the rule $t_{01} \times \langle t_{02}\rangle \rightarrow \langle t_{01} \times t_{02}\rangle$. Therefore, the type that was rewritten to $\langle\langle t_0\rangle\rangle \times t_1$ was $\langle t_{01} \times \langle t_{02}\rangle\rangle \times t_1$. Now it can be rewritten to $\langle\langle t_0'\rangle \times t_1'\rangle$ as follows:

$$\langle t_{01} \times \langle t_{02}\rangle\rangle \times t_1 \rightarrow \langle(t_{01} \times \langle t_{02}\rangle) \times t_1\rangle \rightarrow \langle\langle t_{01} \times t_{02}\rangle \times t_1\rangle = \langle\langle t_0\rangle \times t_1\rangle \longrightarrow\!\!\!\!\!\rightarrow \langle\langle t_0'\rangle \times t_1'\rangle$$

Note that the first two rules satisfy the conditions for μ-rewriting.

Subcase 1.2 when $t_0 = t_{01} \times t_{02}$ and the ruled applied is $\langle t_{01}\rangle \times t_{02} \rightarrow \langle t_{01} \times t_{02}\rangle$ is similar to the subcase 1.1.

Subcase 1.3. $t_0 = \{t_{01}\}$ and a pair of or-set brackets around t_0 appeared by applying the rule $\{\langle t_{01}\rangle\} \rightarrow \langle\{t_{01}\}\rangle = \langle t_0\rangle$. Therefore, according to the rules for $\lhd$, the type that was rewritten to $\langle\langle t_0\rangle\rangle \times t_1$ was $\langle\{\langle t_{01}\rangle\}\rangle \times t_1$. Now it can be rewritten to $\langle\langle t_0'\rangle \times t_1'\rangle$ as follows:

$$\langle\{\langle t_{01}\rangle\}\rangle \times t_1 \rightarrow \langle\{\langle t_{01}\rangle\} \times t_1\rangle \rightarrow \langle\langle\{t_{01}\}\rangle \times t_1\rangle = \langle\langle t_0\rangle \times t_1\rangle \longrightarrow\!\!\!\!\!\rightarrow \langle\langle t_0'\rangle \times t_1'\rangle$$

Again, note that the first two rules satisfy the conditions for μ-rewriting.

Case 2. The set rule is used to eliminate the double or-set brackets. That is, t_0 may have been rewritten to some t_0' and then the rule $\{\langle\langle t_0'\rangle\rangle\} \rightarrow \langle\{\langle t_0'\rangle\}\rangle$ was used. According to the rules for $\lhd$, this means that in s', $\langle\langle t_0\rangle\rangle$ appeared in the context $\{\langle\langle t_0\rangle\rangle\}$. Since s is a μ-type and s' is the first reduct in which a pair of or-set brackets appeared that was not canceled at the next step, there are two possible ways for it to appear.

Subcase 2.1. $t_0 = t_{01} \times t_{02}$ and a pair of or-set brackets around t_0 appeared by applying the rule $t_{01} \times \langle t_{02}\rangle \rightarrow \langle t_{01} \times t_{02}\rangle$. Therefore, the type that was rewritten to $\{\langle\langle t_0\rangle\rangle\}$ was $\{\langle t_{01} \times \langle t_{02}\rangle\rangle\}$. Now it can be rewritten to $\langle\{\langle t_0'\rangle\}\rangle$ as follows:

$$\{\langle t_{01} \times \langle t_{02}\rangle\rangle\} \rightarrow \langle\{t_{01} \times \langle t_{02}\rangle\}\rangle \rightarrow \langle\{\langle t_{01} \times t_{02}\rangle\}\rangle = \langle\{\langle t_0\rangle\}\rangle \longrightarrow\!\!\!\!\!\rightarrow \langle\{\langle t_0'\rangle\}\rangle$$

Subcase 2.2. $t_0 = t_{01} \times t_{02}$ and a pair of or-set brackets around t_0 appeared by applying the rule $\langle t_{01} \rangle \times t_{02} \rightarrow \langle t_{01} \times t_{02} \rangle$. This case is similar to 2.1.

Subcase 2.3. $t_0 = \{t_{01}\}$ and a pair of or-set brackets around t_0 appeared by applying the rule $\{\langle t_{01} \rangle\} \rightarrow \langle \{t_{01}\} \rangle = \langle t_0 \rangle$. Therefore, the type that was rewritten to $\{\langle\langle t_0 \rangle\rangle\}$ was $\{\langle\{\{\langle t_{01} \rangle\}\}\rangle\}$. Now it can be rewritten to $\langle\{\{\langle t_0' \rangle\}\}\rangle$ as follows:

$$\{\langle\{\{\langle t_{01} \rangle\}\}\rangle\} \rightarrow \langle\{\{\{\langle t_{01} \rangle\}\}\}\rangle \rightarrow \langle\{\{\langle\{t_{01}\}\rangle\}\}\rangle = \langle\{\{\langle t_0 \rangle\}\}\rangle \twoheadrightarrow \langle\{\{\langle t_0' \rangle\}\}\rangle$$

Note that in all three subcases the new rules we introduce satisfy the conditions for μ-rewriting.

Case 3. In $\langle\langle t_0 \rangle\rangle$, t_0 could be rewritten to $t_0' = \langle t_{01} \rangle$ and then the pair of or-set brackets around to t_0' is canceled by applying the rule $\langle\langle t_{01} \rangle\rangle \rightarrow \langle t_{01} \rangle$. That is, $\langle\langle t_0 \rangle\rangle \twoheadrightarrow \langle\langle t_{01} \rangle\rangle$. This equivalently could be achieved by rewriting $\langle\langle t_0 \rangle\rangle$ as follows:

$$\langle\langle t_0 \rangle\rangle \rightarrow \langle t_0 \rangle \twoheadrightarrow \langle t_0' \rangle = \langle\langle t_{01} \rangle\rangle$$

Notice that the first rule is an instance of μ-rewriting: double or-set brackets are canceled immediately after they appeared.

Now we define a measure of a rewriting from s to t as the total number of instances of $\langle\langle\rangle\rangle$ in all intermediate results of rewritings such that those double or-set brackets are not canceled by applying the μ rule at the next step. If the measure of a rewriting is at least one, we can find an instance of the first appearance of $\langle\langle\rangle\rangle$ that is not canceled immediately afterwards, and use the above algorithm to decrease the measure by at least one. Hence, this algorithm eventually produces a rewrite strategy of measure zero, and such is a μ-strategy. Proposition is proved. □

Now that we know that there exist μ-rewrite strategies between μ-types t_1 and t_2 satisfying $t_1 \lhd t_2$, we can prove the following result.

Theorem 5.23 (Partial Normalization) *Given two μ-types t_1 and t_2 such that $t_1 \lhd t_2$, any two μ-rewrite strategies $r_1, r_2 : t_1 \twoheadrightarrow t_2$ yield the same result on objects in or-$\mathcal{NRL}$ and or-$\mathcal{NRL}_a$. That is, for any object x of type t_1,*

$$\mathsf{app}(t_1, r_1)(x) = \mathsf{app}(t_1, r_2)(x) \qquad \textit{and} \qquad \mathsf{app}_a(t_1, r_1)(x) = \mathsf{app}_a(t_2, r_2)(x)$$

Proof. The proof is going to follow the proof of the normalization theorem, but here we need to do most of the work with types rather than object. Again, we define a rewrite system on objects by letting x of type s_1 rewrite in one step to y of type s_2 if $t_1 \lhd s_1 \lhd s_2 \lhd t_2$ and one of the following holds. Either s_1 is a μ-type and y is obtained by applying one type rewrite rule (in the sense of app) to x, or x has one subobject of type $\langle\langle s' \rangle\rangle$ and y is obtained from x by applying the type rewrite rule $\langle\langle s' \rangle\rangle \rightarrow \langle s' \rangle$. Then, in order to prove the theorem, similarly to the case of the normalization theorem, we must show that thus defined abstract rewrite system is weakly Church-Rosser.

To show this, we go through all the cases considered in the proof of theorem 5.16 and observe that some of them (1.3, 2.1, 2.2, 3.2) can not happen with the new definition of rewriting. Let us list all others, leaving only types in the diagrams. Notice that in all the diagrams, if types we start with are μ-types, and rewritings $s \twoheadrightarrow s'$ and $t \twoheadrightarrow t'$ are μ-rewritings, then all intermediate types are μ-types and all rewritings are μ-rewritings.

Case 1. Two different components of a pair are rewritten.
Case 2. This case corresponds to case 1.1 in the proof of normalization. These two cases are shown in the diagrams below.

$$\begin{tikzcd}
 & s' \times t \arrow[dr, ->>] & \\
s \times t \arrow[ur, ->>] \arrow[dr, ->>] & & s' \times t' \\
 & s \times t' \arrow[ur, ->>] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \langle s \times t \rangle \arrow[dr, ->>] & \\
s \times \langle t \rangle \arrow[ur, ->>] \arrow[dr, ->>] & & \langle s' \times t \rangle \\
 & s' \times \langle t \rangle \arrow[ur, ->>] &
\end{tikzcd}$$

Case 1 Case 2

Case 3. This case corresponds to case 1.2 in the proof of normalization, where $t' \neq \langle t'' \rangle$.
Case 4. This case corresponds to case 1.2 in the proof of normalization, where $t' = \langle t'' \rangle$. It is not hard to see that the diagram below commutes.

$$\begin{tikzcd}
 & \langle s \times t \rangle \arrow[dr, ->>] & \\
s \times \langle t \rangle \arrow[ur, ->>] \arrow[dr, ->>] & & \langle s \times t' \rangle \\
 & s \times \langle t' \rangle \arrow[ur, ->>] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \langle s \times t \rangle \arrow[r, ->>] & \langle s \times \langle t'' \rangle \rangle \arrow[r, equal] & \langle s \times \langle t'' \rangle \rangle \arrow[d, ->>] \\
s \times \langle t \rangle \arrow[ur, ->>] \arrow[dr, ->>] & & & \langle \langle s \times t'' \rangle \rangle \arrow[d, ->>] \\
 & s \times \langle \langle t'' \rangle \rangle \arrow[r, ->>] & s \times \langle t'' \rangle \arrow[r, ->>] & \langle s \times t'' \rangle
\end{tikzcd}$$

Case 3 Case 4

Case 5. This case corresponds to case 3.1 in the proof of normalization, where $t' \neq \langle t'' \rangle$.
Case 6. This case corresponds to case 3.1 in the proof of normalization, where $t' = \langle t'' \rangle$. That the diagram commutes follows from commutativity of diagrams for cases 3.1 and 3.2 in the proof of normalization (see theorem 5.16).

$$\begin{tikzcd}
 & \langle\{t\}\rangle \arrow[dr] & \\
\{\langle t\rangle\} \arrow[ur] \arrow[dr] & & \langle\{t'\}\rangle \\
 & \{\langle t'\rangle\} \arrow[ur] &
\end{tikzcd}$$

Case 5

$$\begin{tikzcd}
 & \langle\{t\}\rangle \arrow[r] & \langle\{\langle t''\rangle\}\rangle \arrow[r, equal] & \langle\{\langle t''\rangle\}\rangle \arrow[d] \\
\{\langle t\rangle\} \arrow[ur] \arrow[dr] & & & \langle\langle\{t''\}\rangle\rangle \arrow[d] \\
 & \{\langle\langle t''\rangle\rangle\} \arrow[r] & \{\langle t''\rangle\} \arrow[r] & \langle\{t''\}\rangle
\end{tikzcd}$$

Case 6

Notice that commutativity of these diagrams is not sufficient to conclude that the rewrite system we defined is weakly Church-Rosser. There is one additional condition, namely that for all intermediate types s_0 it must be the case that $s_0 \lhd t_2$. To show that this is so, we must prove the following.

For case 1, we must show that if $s' \times t \lhd t_2$ and $s \times t' \lhd t_2$, then $s' \times t' \lhd t_2$. For case 2, we must show that of $\langle s \times t\rangle \lhd t_2$ and $s' \times \langle t\rangle \lhd t_2$, then $\langle s' \times t\rangle \lhd t_2$ if $s \lhd s'$. For case 3 we must prove that $\langle s \times t\rangle \lhd t_2$ and $s \times \langle t'\rangle \lhd t_2$ imply that $\langle s \times t'\rangle \lhd t_2$ if $t \lhd t'$. For case 4 it is necessary to show that $t \lhd \langle t''\rangle$, $\langle s \times t\rangle \lhd t_2$ and $s \times \langle\langle t''\rangle\rangle \lhd t_2$ imply $\langle s \times t'\rangle \lhd t_2$. For case 5, we must prove that $\langle\{t\}\rangle \lhd t_2$ and $\{\langle t'\rangle\} \lhd t_2$ together with $t \lhd t'$ imply $\langle\{t'\}\rangle \lhd t_2$. Finally, in case 6 we must prove that $\langle\{t''\}\rangle \lhd t_2$ whenever $\langle\{t\}\rangle \lhd t_2$, $\{\langle\langle t''\rangle\rangle\} \lhd t_2$ and $t \lhd \langle t''\rangle$.

In the rest of the proof, whenever a type t is given, t_0 will always denote a type which is obtained from t by removing some or-set brackets. That is, $t \prec t_0$.

Before we prove these cases, let us make the following observation. Assume $\langle v\rangle$ is a μ-type. If t and t', such that $t \lhd t' \lhd v$ and $t \lhd \langle v\rangle$, then $t' \lhd \langle\langle u\rangle\rangle$ for some type u. Indeed, the way the rewriting works is that some pairs of or-set brackets move up in the carcass of a type, and some multiple or-set brackets are canceled. Therefore, the only possibility for $t \lhd v$ and $t \lhd \langle v\rangle$ to hold simultaneously is that $v = \langle v'\rangle$. Again, looking at how rewriting works, we see that any rewriting $t \twoheadrightarrow \langle\langle v'\rangle\rangle$ must go through t' and hence $t' \lhd \langle\langle v'\rangle\rangle$.

Now consider case 1. If t_2 is a product type, say $w \times u$, we obtain that $s' \lhd w$ and $t' \lhd u$ and hence $s' \times t' \lhd t_2$. The other possibility is that $t_2 = \langle w\rangle$. In this case, for some types $u_1 \succ s' \times t$ and $u_2 \succ s \times t'$ is must be the case that $u_1 \lhd w$ and $u_2 \lhd w$ from which we derive that w is a product type since $\langle w\rangle$ is a μ-type. Let $w = w_1 \times w_2$. Now $u_1 \lhd w_1 \times w_2$ can be translated into three possible cases, depending on whether u_1 is $s_0' \times t$ or $s_0' \times t_0$ or $s' \times t_0$ where $t \prec t_0$ and $s' \prec s_0'$. Similarly three cases arise for u_2. Since $t \lhd \langle t_0\rangle$ and $s' \lhd \langle s_0'\rangle$, we obtain $s' \times t' \lhd \langle w_1 \times w_2\rangle$ in all cases but the following one: $t' \lhd w_2$ and $s' \lhd w_1$. In this case we have $t \lhd \langle w_2\rangle$ and $s \lhd \langle w_1\rangle$. Then, by the observation made above, $w_i = \langle w_i'\rangle$, $i = 1, 2$, and $s' \times t' \lhd \langle\langle w_1'\rangle\rangle \times \langle\langle w_2'\rangle\rangle \lhd \langle\langle w_1'\rangle \times \langle w_2'\rangle\rangle \lhd \langle w\rangle = t_2$ as required. Case 1 is proved.

Consider case 2. Since $\langle s \times t\rangle \lhd t_2$, we obtain $t_2 = \langle w\rangle$. Moreover, w is either a product or a set type. Now for some types $u_1 \succ \langle s \times t \rangle$ and $u_2 \succ s' \times \langle t \rangle$ we have $u_1, u_2 \lhd w$. This shows that w is a product type, say $w_1 \times w_2$. Moreover, $u_1 \succ s \times t$. Again, there are three possibilities how u_1 and u_2 can be obtained by removing or-set brackets from components, and it is easy to see that in all of them $s' \times t \lhd w$ or $s' \times t \lhd \langle w \rangle$, both proving $\langle s' \times t \rangle \lhd t_2$. For example, if $u_1 = s_0 \times t$ and $u_2 = s' \times t_0$, then $s' \lhd w_1$ and $t \lhd w_2$ and $s' \times t \lhd w$. If $u_1 = s \times t_0$ and $u_2 = s'_0 \times \langle t \rangle$, then $s' \lhd \langle w_1 \rangle$ and $t \lhd \langle w_2 \rangle$ and $s' \times t \lhd \langle w_1 \rangle \times \langle w_2 \rangle \lhd \langle w \rangle$. Case 2 is proved.

Consider case 3. We have $\langle s \times t \rangle \lhd t_2$ and hence $t_2 = \langle w \rangle$. Since $s \times \langle t' \rangle \lhd \langle w \rangle$ and t_2 is a μ-type, we obtain that w is a product type, i.e. $w = w_1 \times w_2$. Now we have that for some types $u_1 \succ s \times t$ and $u_2 \succ s \times \langle t' \rangle$, this holds: $u_1, u_2 \lhd w$. If for some $t'_0 \succ t'$ it is the case that $\langle t'_0 \rangle \lhd w_2$, then it is not hard to see that $s \times t' \lhd w_1 \times w_2$ or $s \times t' \lhd \langle w_1 \rangle \times w_2$ depending on whether s rewrites to w_1 or $\langle w_1 \rangle$. In both cases $\langle s \times t' \rangle \lhd t_2$. Similarly, the case holds if $t' \lhd w_2$.

The only remaining case is when $\langle t' \rangle \lhd w_2$. Then we must have that $u_2 = s_0 \times \langle t' \rangle$ and hence $s \lhd \langle s_0 \rangle \lhd \langle w_1 \rangle$. Since $\langle t' \rangle \lhd w_2$, we have $w_2 = \langle u \rangle$ and u is not of form $\langle u' \rangle$ because w_2 is a μ-type. Therefore, $\langle t' \rangle \lhd \langle u \rangle$ leaves two possibilities: either $t' \lhd u$ or $t' \lhd \langle u \rangle$. If $t' \lhd \langle u \rangle$, then $s \times t' \lhd \langle w_1 \rangle \times w_2 \lhd \langle w \rangle$ and we are done. So consider the case when $t' \lhd u$. From $u_1 \lhd w_1 \times w_2$ we also have that either $t_0 \lhd w_2$ and then $t \lhd \langle w_2 \rangle \lhd \langle u \rangle$ or $t \lhd w_2 = \langle u \rangle$. Hence, $t \lhd t' \lhd u$ and $t \lhd \langle u \rangle$. As we remarked earlier, this must imply that u is an or-set type, i.e. $u = \langle u' \rangle$ but this would contradict the assumption that w_2 is a μ-type. Hence, this case leads to a contradiction and in all other cases it was shown that $\langle s \times t' \rangle \lhd t_2$. Hence, case 3 holds.

Notice that nowhere in the proof of case 3 did we use the assumption that $t' \neq \langle t'' \rangle$. Now consider case 4. Since $s \times \langle \langle t'' \rangle \rangle \lhd t_2$ is a part of a μ-rewrite strategy and t_2 is a μ-type, we obtain $s \times \langle t'' \rangle \lhd t_2$. Now the proof of case 3 tells us that $\langle s \times t'' \rangle \lhd t_2$ which proves case 4.

Now consider case 5. We have $\langle \{t\} \rangle \lhd t_2$ and hence $t_2 = \langle w \rangle$ for some w. Moreover, w can not be of form $\langle w' \rangle$ since t_2 is a μ-type. Now we have $\{\langle t' \rangle\} \lhd \langle w \rangle$ and hence for some $u \succ \{\langle t' \rangle\}$ we have $u \lhd w$. Since w is not of form $\langle w' \rangle$, it must be $\{w'\}$ for some w'.

Now we have three cases. First, u could be $\{t'\}$ and in this case $\{t'\} \lhd \{w'\}$ implies $t' \lhd w'$ and then $\langle \{t'\} \rangle \lhd \langle \{w'\} \rangle = t_2$ and we are done. In the second case, for some $t'_0 \succ t'$, we have $\{\langle t'_0 \rangle\} \lhd w$ and hence $\langle t'_0 \rangle \lhd w'$. Then we have $t' \lhd \langle t'_0 \rangle \lhd w'$. Then $\langle \{t'\} \rangle \lhd \langle \{w'\} \rangle = \langle w \rangle = t_2$. Finally, in the third case we have $\{t'_0\} \lhd \{w'\}$ and $t'_0 \lhd w'$; hence $t' \lhd \langle t'_0 \rangle \lhd \langle w' \rangle$. Now $\langle \{t'\} \rangle \lhd \langle \{\langle w' \rangle\} \rangle \lhd \langle \langle \{w'\} \rangle \rangle \lhd \langle \{w'\} \rangle = t_2$. This finishes the proof of case 4.

Since we have not used the assumption that $t' \neq \langle t'' \rangle$ anywhere, this also proves case 6. Indeed, since $\{\langle \langle t'' \rangle \rangle\} \lhd t_2$ is a part of a μ-rewrite strategy and t_2 is a μ-type, we obtain $\{\langle t'' \rangle\} \lhd t_2$ and then the proof of case 5 applies. Hence, all cases are proved, and this tells us that the rewrite system is weakly Church-Rosser.

This finishes the proof of partial normalization for the set-theoretic semantics. The proof for the antichain semantics is obtained by repeating the proof of normalization for the antichain semantics verbatim, thus showing that weak Church-Rosserness of the corresponding rewrite system for antichains follows from the result we have just proved. Theorem is completely proved now. □

To add partial normalization to the language, one has to introduce a new function $pnorm^{t_1,t_2} : t_1 \to t_2$ which is defined if t_1 and t_2 are μ-types and $t_1 \lhd t_2$. According to proposition 5.22, there exists a μ-rewrite strategy $r : t_1 \longrightarrow t_2$ and then by theorem 5.23 we can correctly define the semantics of $pnorm^{t_1,t_2}$ as $\mathsf{app}(t_1, r)$ and the semantics of $pnorm_a^{t_1,t_2}$ as $\mathsf{app}_a(t_1, r)$. These two functions are definable in *or*-$\mathcal{NRL}$ and *or*-$\mathcal{NRL}_a$, but not polymorphically.

Repeating the proof of theorem 5.17 verbatim, we obtain the following result.

Corollary 5.24 *For any two μ-types t_1 and t_2 such that $t_1 \longrightarrow t_2$ and any object x of type t_1,*

$$[\![x]\!]_c = [\![pnorm^{t_1,t_2}(x)]\!]_c \quad \text{and} \quad [\![x]\!]_c = [\![pnorm_a^{t_1,t_2}(x)]\!]_c$$

In other words, $[\![pnorm^{t_1,t_2}]\!]_c = [\![pnorm_a^{t_1,t_2}]\!]_c = [\![id]\!]_c$. □

There are many open questions about partial normalization. Even though we can test if $s \lhd t$ efficiently and we know that there exists a μ-rewriting from s to t if s and t are μ-types, algorithmic aspects of finding a μ-rewrite strategy between μ-types need to be further explored. Partial normalization must also be combined with a smart evaluation strategy to help answer queries faster.

As another important consequence of partial normalization, notice that it allows us to compare objects of different types in terms of their partiality. Previously we were able to compare only objects of the same type. That is, the function $\leq$ had type $s \times s \to bool$. Now partial normalization gives us a canonical representation of an object of type s at type t where $s \lhd t$ and s and t are μ-types. Therefore, we can say if x of type s is more informative than y of type t by checking if $pnorm^{s,t}(x) \leq_t y$. This appears to be a new phenomenon in the field of partial information.

5.2.4 Losslessness of normalization

This section investigates whether the process of normalization loses anything "that can be regarded as critical." If loss of information is inevitable in the general case, then one would like to obtain a set of general sufficient (and, if possible, necessary) conditions that guarantee losslessness of normalization.

In chapter 1 we discussed the concept of representation system for relational databases with partial information. A representation system is in fact a semantic function that maps every incomplete relation R into the set of possible worlds that R can represent. Of course the question that immediately arises is whether any loss of information occurs as the result of replacing R with the corresponding set of possible worlds. That is, if we evaluate a query on each of the possible worlds, can the resulting family of relations be represented by one incomplete relation?

Observe that the normalization process is very closed in the spirit to the representation systems. That is, we replace an incomplete object by the or-set of objects it can represent. So, again we may ask if this representation is lossless, that is, if loss of the structural information has any impact on the conceptual queries.

First, let us see how this problem can be formalized in a *wrong* way which is just a reformulation of the concept of a strong representation system. Suppose an object x is given and we ask a query against each possibility represented by x. That is, we apply a function f that does not use or-sets to all objects in $normalize(x)$. Let the result of this be an or-set $\langle y_1, \ldots, y_n \rangle$. That is, $or\_map(f)(normalize(x)) = \langle y_1, \ldots, y_n \rangle$. The question we ask is whether there exists an object y such that $normalize(y) = \langle y_1, \ldots, y_n \rangle$.

The answer to this question is positive because we can just take y to be $or\_map(f) \circ normalize$! Of course the reason we can do this is that we can use *normalize* in the language whereas the concept of representation systems can not be expressed in the standard database languages. Therefore, we should look for another formalization of losslessness of normalization.

Given an $or\text{-}\mathcal{NRL}$-definable function $f : s \to t$ and an object $x : s$ containing some or-sets. Then x conceptually represents several values $x_1, ..., x_n$. Suppose $f(x)$ is an object containing or-sets; then it conceptually represents several values $y_1, ..., y_m$. It is desirable to discover which one of $x_1, ..., x_n$ leads to which one of $y_1, ..., y_m$. This is a question of searching for a *conceptual analog* of f that associates each x_i in *normalize* x to a subset of $normalize(f\ x)$.

The idea of the conceptual analog of a query is illustrated in figure 5.8. One would like to know which combination of the conceptual values of the input give rise to which subset of the conceptual values of the output. However, the ideal situation can only be approximated. As a first attempt, for each possible conceptual value x_i of the input x, we aim only to account for some of the conceptual values in the output that are due to it. This approximation to conceptual analog is illustrated in figure 5.8. Some conceptual values y_j in the output may be left unaccounted for. For example, the last element of *normalize* y in the figure. Similarly, the picture given for each input x_i is only partial. For example, the second element of *normalize* x in the figure might in reality contribute to three values in the output but the conceptual analog discovers only two.

Now restrict types only to those containing or-sets. Define *purely or-types* by the following grammar: $t ::= \langle b \rangle \mid t \times t \mid \{t\} \mid \langle t \rangle$. It is possible to force any type into a purely or-type by putting or-set brackets around every occurrence of a base type. Its action on objects is represented by taking each base type subobject z into $or\_\eta(z)$. We call such a function *preserve*. It can be easily seen that any object x is conceptually equivalent to $preserve(x)$, i.e. $normalize(x) = normalize(preserve(x))$ provided x has or-sets. That is, without loss of

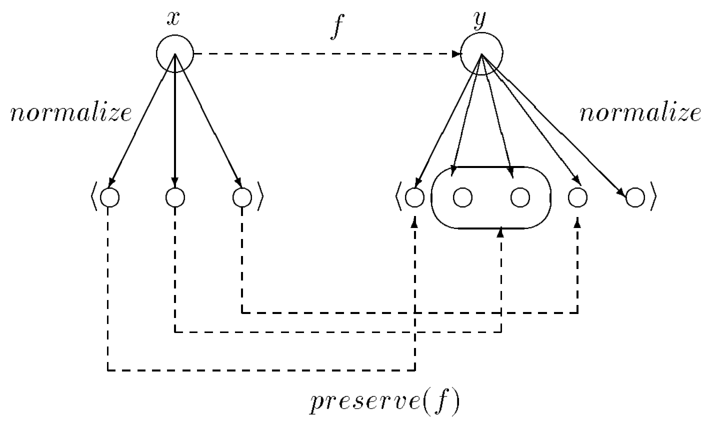


Figure 5.8: Conceptual analog of function f

generality we can speak of objects $preserve(x)$.

Given an *or*-$\mathcal{NRL}$-function $f : s \to t$ and two objects $x : s$ and $y = f(x) : t$, let *normalize*$(x) = \langle x_1, \ldots, x_n \rangle$ and *normalize*$(y) = \langle y_1, \ldots, y_m \rangle$, $nf(s) = \langle s' \rangle$ and $nf(t) = \langle t' \rangle$. Our motivation to study losslessness is to find a conceptual analog of f. What can such an analog be? As the first approximation, it is given by a function $f' : s' \to \langle t' \rangle$ which associates with each element x_i in *normalize*(x) a subset of *normalize*(y), thus defining the action of f on elements its input could possibly stand for. This is illustrated in figure 5.8. Note that the second element of *normalize*(x) is mapped into a two-element subset of *normalize*(y) and the last element of *normalize*(y) is not accounted for. The morphism $preserve(f) : nf(s) \to nf(t)$ can now be defined as $or\_\mu \circ or\_map(f')$.

How could one refine the action of f on elements of normalized object? There are two ways to do so. First, to require that this action be defined unambiguously, that is, f' maps every element from *normalize*(x) into a unique element of *normalize*(y), thus having type $s' \to t'$. $preserve(f)$ can then be reconstructed as $or\_map(f')$. Secondly, one may require that all the elements of *normalize*(y) be accounted for, that is, $preserve(f) \circ$ *normalize*$(x) =$ *normalize*(y). In other words, $preserve(f)$ is onto.

Proposition 5.25 *Let s and t be purely or-types and $f : s \to t$ a function definable in or-$\mathcal{NRL}$ that does not use or_empty and any primitive p whose type has or-sets. Then there exists a conceptual analog $preserve(f)$ which is generally of form $or\_\mu \circ or\_map(\cdot)$ and of form $or\_map(\cdot)$ if f does not use $or\_\cup$. If f does not use pairing, ρ_2 and or_ρ_2, the conceptual analog is also onto.*

Proof is by induction on the structure of f. Most of its steps are quite straightforward, so we just show a few cases as an example. Consider the case $f = \rho_2 : s \times \{t\} \to \{s \times t\}$. Since s

and t are purely or-types, $nf(s) = \langle s' \rangle$ and $nf(t) = \langle t' \rangle$. Then $preserve(\rho_2)$ must have type $\langle s' \times \{t'\} \rangle \to \langle \{s' \times t'\} \rangle$. We take $preserve(\rho_2)$ to be $or\_map(\rho_2^{s',t'})$. An easy application of the normalization theorem shows that for any object x of type $s \times \{t\}$, $or\_map(\rho_2^{s',t'}) \circ normalize(x) \subseteq normalize \circ \rho_2^{s,t}(x)$. Therefore, being onto can not be maintained for ρ_2.

As another illustration, consider $f = or\_\cup : \langle t \rangle \times \langle t \rangle \to \langle t \rangle$. To see why the translation can not be of form $or\_map(\cdot)$, let t be a base type, say int, and consider an object $x = (\langle 1, 2 \rangle, \langle 3 \rangle)$. Applying $normalize \circ or\_\cup$ gives $\langle 1, 2, 3 \rangle$ while applying $normalize$ yields $\langle (1,3), (2,3) \rangle$ and no mapping over the latter object can produce the former. So in the general case the translation of $or\_\cup$ is

$$preserve(or\_\cup) = or\_\mu \circ or\_map(or\_\cup(or\_\eta \circ \pi_1, or\_\eta \circ \pi_2)).$$

Induction hypothesis is applied for pairing, map, or_map and composition. The case of pairing is similar to ρ_2; the translation of map is a straightforward application of induction hypothesis. In the case of composition one can easily show that, given a composition $f \circ g$ such that either $preserve(f)$ or $preserve(g)$ is of form $or\_\mu \circ or\_map(\cdot)$, $preserve(f \circ g)$ is such and if both $preserve(f)$ and $preserve(g)$ are of form $or\_map(\cdot)$, then so is $preserve(f \circ g)$. Moreover, the translation maintains being onto, depending on f and g. As an illustration, consider f and g such that $preserve(f) = or\_\mu \circ or\_map(f')$ and $preserve(g) = or\_\mu \circ or\_map(g')$. Then

$$preserve(f \circ g) = or\_\mu \circ or\_map(or\_\mu \circ or\_map(g') \circ f')$$

In the case of $f = or\_map(f') : \langle s \rangle \to \langle t \rangle$, if both $\langle s \rangle$ and $\langle t \rangle$ are normalized, $preserve(f) = f$; if both are unnormalized, then $preserve(f) = preserve(f')$. Since we are considering only purely or-types, s (or t) is a normal form iff s' (or t') is a base type. Therefore, the case when t is a normal form and s is not is impossible. If s is a normal form and t is not, then $preserve(f) = or\_\mu \circ or\_map(normalize \circ f')$. Notice that if $or\_\cup$ is not used, f' can produce only or-singletons on elements of a base type. In this case $f' = or\_\eta \circ f''$ and $preserve(f) = or\_map(f'')$. □

5.2.5 Costs of normalization

We have seen before that the complexity of or-$\mathcal{NRL}^+$ queries can be exponential. In particular, the cardinality of $normalize(x)$ can be exponential in the size of x provided that α was used in the course of normalization. In fact, we showed that $powerset$ can be expressed using α. If one tries to estimate the cost of normalization by "brute force," a hyperexponential upper bound can be immediately obtained: indeed, if n is the size of x, applying the costly α $O(n)$ times seems to yield a hyperexponential bound.

In this section we show that the fear of hyperexponentiality is not justified. In fact, both cardinality of $normalize(x)$ and its size are in the worst case exponential in the size of x. The first result in this section explains why consecutive applications of α still yield objects of exponential size. Then we proceed to find upper bounds on the cardinality and the size of normalized objects.

Let x be an object and $y = \mathit{normalize}(x)$. Define $m(y)$ as the number of elements in y if it is an or-set and 1 otherwise. Uniformly, $m(x) = |\mathit{normalize}(\mathit{or\_}\eta(x))|$. The *size* of an object is defined inductively: the size of an atomic object is 1, $\mathsf{size}\ (x, y) = \mathsf{size}\ x + \mathsf{size}\ y$, $\mathsf{size}\ \{x_1, \ldots, x_n\} = \mathsf{size}\ \langle x_1, \ldots, x_n\rangle = \mathsf{size}\ x_1 + \ldots + \mathsf{size}\ x_n$.

To work with objects, it is convenient to associate rooted labeled trees with them. A tree $\mathcal{T}x$ associated with an atomic object x is defined as a one-node tree labeled by x. $\mathcal{T}(x, y)$ is a tree with the root labeled by $\times$ and two subtrees rooted at its children are $\mathcal{T}x$ and $\mathcal{T}y$. $\mathcal{T}\{x_1, \ldots, x_n\}$ (or $\mathcal{T}\langle x_1, \ldots, x_n\rangle$) is a tree whose root is labeled by $\{\}$ (or $\langle\rangle$) and n subtrees rooted at its children are $\mathcal{T}x_1, \ldots, \mathcal{T}x_n$. In view of this definition, $m(x)$ can be redefined as the number of children of the root of $\mathcal{T}\mathit{normalize}(x)$ if the root is labeled by $\langle\rangle$ and 1 otherwise. $\mathsf{size}\ x$ is the number of leaves in $\mathcal{T}x$.

Intuitively, the following proposition says that the "internal" structure of $\mathcal{T}x$ does not contribute to the creation of new possibilities in $\mathit{normalize}(x)$, and the number of such possibilities $m(x)$ is determined by the or-sets which are closest to the leaves.

Proposition 5.26 *Let x be an object, and $v_1, \ldots, v_k$ the nodes in $\mathcal{T}x$ labeled by $\langle\rangle$, such that the subtrees rooted at v_i's do not have other nodes labeled by $\langle\rangle$ (i.e. they are or-sets closest to the leaves). Let m_i be the number of children of v_i, $i = 1, ..., k$. Then, if $k \neq 0$,*

$$m(x) \leq \prod_{i=1}^{k}(m_i + 1)$$

Proof is by induction on the structure of the object. We consider only objects containing or-sets. The base case (i.e. or-sets of objects of base types) is obvious. Let $x = (x_1, x_2)$. Assume that both x_1 and x_2 contain or-sets and $v_1, \ldots, v_p$ are nodes of $\mathcal{T}x_1$ and $v_{p+1}, \ldots, v_k$ are nodes of $\mathcal{T}x_2$. Then, by induction hypothesis, $m(x_1) \leq \prod_{i=1}^{p}(m_i + 1)$ and $m(x_2) \leq \prod_{i=p+1}^{k}(m_i + 1)$. By coherence, $\mathit{normalize}(x) = \mathit{or\_}\rho((\mathit{normalize}(x_1), \mathit{normalize}(x_2)))$ where $\mathit{or\_}\rho$ pairs each item in its first argument with each item in its second argument (it can be easily expressed in *or-*$\mathcal{NRL}$). Therefore, $m(x) \leq m(x_1)m(x_2) \leq \prod_{i=1}^{k}(m_i + 1)$. Two other cases when either x_1 or x_2 contains or-sets are similar.

Let $x = \{x_1, \ldots, x_n\}$. Then all x_i's contain or-sets. Again, by coherence,

$$\mathit{normalize}(x) = \alpha(\{\mathit{normalize}(x_1), \ldots, \mathit{normalize}(x_n)\})$$

Therefore, $m(x) \leq \prod_{i=1}^{n} m(x_i)$ and the result follows from the induction hypothesis.

Finally, if $x = \langle x_1, \ldots, x_n\rangle$, there are two cases. If x_i's do not contain or-sets, then $m(x) = n \leq n + 1$. If they do contain or-sets, then by coherence

$$\mathit{normalize}(x) = \mathit{or\_}\mu(\langle \mathit{normalize}(x_1), \ldots, \mathit{normalize}(x_n)\rangle)$$

i.e. $m(x) \leq \Sigma_{i=1}^{n} m(x_i) \leq \prod_{i=1}^{n} m(x_i)$ because $m(\cdot) \geq 2$. The case now follows from the hypothesis. $\square$

This proposition explains why there is an exponential upper bound for $m(x)$ despite the fact that α can be applied many times. The following three results find upper bounds on the number of elements in the normal form and its size in terms of the size of object rather than the tree structure. We first formulate the results and then give their proofs.

Theorem 5.27 *Let x be an object with* size $x = n$. *Then*

$$m(x) \leq \sqrt[3]{3}^{\,n}$$

Moreover, for any n divisible by 3 there exists an object x such that size $x = n$ *and* $m(x) = \sqrt[3]{3}^{\,n}$. $\square$

Theorem 5.28 *Let x be an object with* size $(x) = n$ *where $n > 1$. Then*

$$\mathsf{size}\ \mathit{normalize}(x) \leq \tfrac{n}{2} \sqrt[3]{3}^{\,n} \qquad \square$$

Corollary 5.29 *Let $x = \mathit{normalize}(y)$ and* size $x = n$. *Then*

$$O(\log n) \leq \mathsf{size}\ y \leq n \qquad \square$$

The upper bound of theorem 5.28 is not tight. The following result exhibits a tight upper bound for a large class of objects. This shows that the previous theorem can not be significantly improved.

Theorem 5.30 *Let x be an object with* size $x = n$ *containing or-sets. Assume that every subobject of type $\{\langle t' \rangle\}$ has size at least 21, every subobject of type $t' \times \langle t'' \rangle$ or $\langle t'' \rangle \times t'$ has size at least 6 and every subobject of type $\langle\langle t' \rangle\rangle$ has size at least 3, where t' and t'' do not use the or-set type constructor. Then*

$$\mathsf{size}\ \mathit{normalize}(x) \leq \frac{n}{3} \sqrt[3]{3}^{\,n}$$

Moreover, for any n divisible by 3 there exists an object x such that size $x = n$ *and* size $\mathit{normalize}(x) = \frac{n}{3}\sqrt[3]{3}^{\,n}$. $\square$

Since the size of $\mathit{normalize}_a(x)$ can not exceed the size of $\mathit{normalize}(x)$, and since all examples demonstrating tightness of upper bounds do not use orderings, we obtain

Corollary 5.31 *All results of theorems 5.27, 5.28 and 5.30 hold for the antichain semantics, that is, for $\mathit{normalize}_a$.* $\square$

Proofs of theorems

Proof of theorem 5.27. As in the proof of proposition 5.26, consider only objects containing or-sets. Proceed by induction on the number of steps of normalization. If the object is already normalized, we are done. Assume $normalize(x)$ is obtained by one step of normalization. Then this step is one of the maps associated with the rewrite rules, so we have four cases. Notice that in the base cases we may assume w.l.o.g that any element of a set or an or-set is of base type since this will give us the maximal possible $m(x)$ for a given $\mathsf{size}\ x$.

Case 1. $x = (x_1, x_2)$ where $x_1 = \langle x_1^1, \ldots, x_{n-1}^1 \rangle$. Then $normalize(x) = or\_\rho_1(x)$ and it is an easy arithmetic exercise to show that $m(x) = n - 1 \leq \sqrt[3]{3}^n$.

Case 2 when or_ρ_2 is applied to obtain the normal form is similar.

Case 3. Let $x = \{X_1, \ldots, X_k\}$ where each X_i is an or-set $\langle x_1^i, \ldots, x_{k_i}^i \rangle$ where all x_j^i are elements of base types. Since we are interested in upper bound, assume w.l.o.g. that all x_j^i's are distinct (if they are not, some of sets in $normalize(x)$ could collapse). Let $X = \bigcup_{i,j} x_j^i$. Define a graph $G = (X, E)$ where $(x_{j1}^{i_1}, x_{j2}^{i_2})$ is in E iff $i_1 \neq i_2$. Let $normalize(x) = \alpha(x) = \langle Y_1, \ldots, Y_p \rangle$ (Y_k's are sets). Then it follows from the definition of α that $Y_1, \ldots, Y_p$ are precisely the cliques of G. Since $n = \mathsf{size}\ x = |X|$, applying the upper bound on the number of cliques for a graph with n vertices [119], we obtain $p = m(x) \leq \sqrt[3]{3}^n$.

Case 4. $x = \langle X_1, \ldots, X_k \rangle$ where X_i's are or-sets of a base type. Then $normalize(x) = or\_\mu(x)$ and $m(x) \leq n$. Again, simple arithmetic shows that $n \leq \sqrt[3]{3}^n$. Hence, $m(x) \leq \sqrt[3]{3}^n$.

The proof of the general case is very similar to the proof of proposition 5.26 and we will show only step. Let $x = \{x_1, \ldots, x_k\}$ where x_i's are not normalized. Then $normalize(x)$ is obtained by applying α to $\{normalize(x_1), \ldots, normalize(x_n)\}$. Let $\mathsf{size}\ x_i = n_i$. By induction hypothesis, $m(x_i) \leq \sqrt[3]{3}^{n_i}$. We now have

$$m(x) \leq \prod_{i=1}^{k} m(x_i) \leq \prod_{i=1}^{k} \sqrt[3]{3}^{n_i} \leq \sqrt[3]{3}^n$$

The other cases are similar. To show the tightness of the upper bound, let $n = 3k, k > 0$. Assume that we have a base type whose domain is infinite (typical example is int). Let $b_1, \ldots, b_n$ be n distinct elements of such a type. Let

$$x = \{\langle b_1, b_2, b_3 \rangle, \langle b_4, b_5, b_6 \rangle, \ldots, \langle b_{n-2}, b_{n-1}, b_n \rangle\}$$

Then $\mathsf{size}\ x = n$ and $normalize(x) = \alpha(x)$ contains $3^k = \sqrt[3]{3}^n$ elements. The theorem is completely proved.

Proof of theorem 5.28. Similarly to the proof of theorem 5.27, proceed by induction on the steps of normalization. We start with base cases, i.e. consider application of or_ρ_2 or or_ρ_1 or α or or_μ.

Case 1. $x = (x_1, x_2)$ where $x_1 = \langle x_1^1, \ldots, x_k^1 \rangle$. Let $\mathsf{size}\ x_1 = s_1$, $\mathsf{size}\ x_i^1 = \sigma_i$. Then $s_1 + \sigma_1 + \ldots + \sigma_k = n$. Since $normalize(x) = or\_\rho_1(x)$, $\mathsf{size}\ normalize(x) = ks_1 + \sigma_1 + \ldots + \sigma_k = ks_1 + (n - s_1) \leq (n - s_1)s_1 + n - s_1 \leq 2n - 2$. Since empty sets and or-sets are excluded, $n \geq 2$ in this case and therefore $2n - 2 \leq \frac{n}{2}\sqrt[3]{3}^{\,n}$.

Case 2 when or_ρ_2 is applied is similar.

Case 3. Let $x = \{X_1, \ldots, X_l\}$ where each X_i is an or-set $\langle x_1^i, \ldots, x_{k_i}^i \rangle$ where all x_j^i have types containing no or-set. Let $\mathsf{size}\ x_j^i = s_j^i$ and

$$\sum_{j=1}^{k_i} s_j^i = \sigma_i \qquad \sum_{i=1}^{l} \sigma_i = n$$

Then an easy calculation shows that $\mathsf{size}\ normalize(x) = \mathsf{size}\ \alpha(x)$ is given by

$$\sigma_1 \cdot k_2 \cdot \ldots \cdot k_l + \sigma_2 \cdot k_1 \cdot k_3 \cdot \ldots \cdot k_l + \ldots + \sigma_l \cdot k_1 \cdot \ldots \cdot k_{l-1} \leq l \cdot \sigma_1 \cdot \ldots \cdot \sigma_l$$

Therefore, we need to maximize $l \cdot \sigma_1 \cdot \ldots \cdot \sigma_l$ under constraint $\sigma_1 + \ldots + \sigma_l = n$. A standard argument shows that such a maximum is bounded above by

$$\begin{cases} 1 & \text{if } n = 1 \\ \frac{n}{2}\sqrt{2}^{\,n} & \text{if } 1 < n < 21 \\ \frac{n}{3}\sqrt[3]{3}^{\,n} & \text{if } n \geq 21 \end{cases}$$

If it easy to see that for $n > 1$, the upper bounds given above are less than $\frac{n}{2}\sqrt[3]{3}^{\,n}$. If $n = 1$, then the size of the normal form is also 1.

Case 4. $x = \langle X_1, \ldots, X_l \rangle$ where X_i's are or-sets of a type that does not contain or-sets. Then $normalize(x) = or\_\mu(x)$. Since the or_μ does not change size, $\mathsf{size}\ normalize(x) < \frac{n}{2}\sqrt[3]{3}^{\,n}$ for all $n \geq 2$. If $n = 1$, then $\mathsf{size}\ normalize(x) = 1$.

To complete the inductive proof, we show that after each step of normalization that produces a normalized subobject x'', that is, $x'' = normalize(x')$ for a subobject x' of x, either $\mathsf{size}\ x'' \leq \frac{n}{2}\sqrt[3]{3}^{\,n}$ is satisfied if $n = \mathsf{size}\ x' > 1$, or $\mathsf{size}\ x'' = 1$ if $n = 1$. This will complete the proof. Two cases corresponding to application of or_ρ_1 or or_ρ_2 are similar to the case of α, so we show here only the case of application of α.

Let $x = \{x_1, \ldots, x_k\}$ where each x_i is an unnormalized object. Let $x_i' = normalize(x_i)$ and k_i be the cardinality of x_i', i.e. $k_i = m(x_i)$. Let $n_i = \mathsf{size}\ x_i$. By theorem 5.27, $k_i \leq \sqrt[3]{3}^{\,n_i}$. First consider the case when all $n_i > 1$.

Let $x_i' = \langle y_1^i, \ldots, y_{k_i}^i \rangle$, $i = 1, \ldots, k$. By s_j^i we denote $\mathsf{size}\ y_j^i$. By induction hypothesis,

$$\forall i = 1, \ldots, k : \quad \sum_{j=1}^{k_i} s_j^i \leq \frac{n_i}{2}\sqrt[3]{3}^{\,n_i}$$

normalize(x) is obtained by applying α to $\{x'_1, \ldots, x'_k\}$, i.e. its elements are sets of representatives of $x'_1, \ldots, x'_k$. Since we are interested in an upper bound, we may assume that all the elements of $x'_1, \ldots, x'_k$ are distinct. Then each element of x'_i will be present in $k^{(i)} = (\prod_{j=1}^{k} k_j)/k_i$ sets. Therefore, the upper bound for size *normalize*(x) can be calculated as the sum of the sizes of all elements of $x'_1, \ldots, x'_k$ multiplied by the number of their occurrences in the normalized object, i.e.

$$\begin{aligned}\textsf{size}\ normalize(x) &\leq \sum_{i=1}^{k}\sum_{j=1}^{k_i} k^{(i)} s^i_j = \sum_{i=1}^{k} k^{(i)} \sum_{j=1}^{k_i} s^i_j \leq \\ \sum_{i=1}^{k} \frac{n_i}{2} k^{(i)} \sqrt[3]{3}^{n_i} &\leq \sqrt[3]{3}^{n_1+\ldots+n_i} \sum_{i=1}^{k} \frac{n_i}{2} = \frac{n}{2}\sqrt[3]{3}^{n}\end{aligned}$$

If all $n_i = 1$, then size *normalize*$(x) = k = n$. If $n > 1$, then $n \leq \frac{n}{2}\sqrt[3]{3}^n$ and if $n = 1$, that is, size $x = 1$, then size *normalize*$(x) = 1$.

Now consider the general case, i.e. $n_1, \ldots, n_p > 1$ and $n_{p+1}, \ldots, n_k = 1$. Normalization of x_i for $i > p$ results in a size one object. Let $\sigma_0 = n_1 + \ldots + n_p$ and $\sigma_1 = k - p$. Clearly $\sigma_0 + \sigma_1 = n$. Had we applied α only to $\{x'_1, \ldots, x'_p\}$, it would have resulted in an object whose size is bounded above by $\frac{\sigma_0}{2}\sqrt[3]{3}^{\sigma_0}$ according to the calculations for the case where all $n_i > 1$. But taking into account σ_1 size one objects adds size σ_1 to every element of the or-set *normalize*(x). Since there are at most $\sqrt[3]{3}^{\sigma_0}$ such sets, we obtain

$$\textsf{size}\ normalize(x) \leq \frac{\sigma_0}{2}\sqrt[3]{3}^{\sigma_0} + \sigma_1\sqrt[3]{3}^{\sigma_0}$$

Since $\sigma_0 > 1$, $\sigma_0 + 2\sigma_1 \leq (\sigma_0 + \sigma_1)\sqrt[3]{3}^{\sigma_1}$ which shows

$$\textsf{size}\ normalize(x) \leq \frac{\sigma_0}{2}\sqrt[3]{3}^{\sigma_0} + \sigma_1\sqrt[3]{3}^{\sigma_0} \leq \frac{n}{2}\sqrt[3]{3}^{n}$$

Finally, if *or_*μ is applied in the process of normalization, it does not change size. Assume $x = \langle x_1, \ldots, x_k\rangle$ where each x_i is an unnormalized object. Let $x'_i =$ *normalize*(x_i) and $n_i =$ size x_i. Assume $n_1, \ldots, n_p > 1$ and $n_{p+1} = \ldots = n_k = 1$. Define σ_0 and σ_1 as in the case of applying α. Then, by induction hypothesis,

$$\textsf{size}\ normalize(x) \leq \sum_{i=1}^{p} \frac{n_i}{2}\sqrt[3]{3}^{n_i} + \sigma_1 \leq \frac{\sigma_0}{2}\sqrt[3]{3}^{\sigma_0} + \sigma_1 \leq \frac{n}{2}\sqrt[3]{3}^{n}$$

If all $n_i = 1$, then two cases arise. If $n > 1$, then size *normalize*$(x) = n \leq \frac{n}{2}\sqrt[3]{3}^n$, and if $n = 1$, then size *normalize*$(x) = n = 1$. Theorem is proved.

Proof of theorem 5.30. We have to rework the base cases only. Since no subobject involving or-sets can have size one, the induction step easily goes through, cf. the proof of theorem 5.28.

The case of applying α was already proved, see proof of theorem 5.28. For the case of applying or_ρ_1 or or_ρ_2, we established an upper bound $2n-2$. It is easily seen that $2n-2 \leq \frac{n}{3}\sqrt[3]{3}^n$ for $n \geq 6$. Finally, applying or_μ does not affect size, and $n \leq \frac{n}{3}\sqrt[3]{3}^n$ for $n \geq 3$.

To show sharpness, consider example from the proof of theorem 5.28. Let

$$x = \{\langle b_1, b_2, b_3\rangle, \langle b_4, b_5, b_6\rangle, \ldots, \langle b_{n-2}, b_{n-1}, b_n\rangle\}$$

where all b_i's are distinct elements of a base type. Then $\alpha(x)$ contains $\sqrt[3]{3}^n$ elements, each having cardinality $\frac{n}{3}$. Thus, $\mathsf{size}\ normalize(x) = \frac{n}{3}\sqrt[3]{3}^n$. Theorem is proved.

5.3 Programming with approximations

In this section we study programming with approximations. First, we use the approach that turns universality properties of collections into programming syntax. Since most approximation constructions possess universality properties, as we showed in section 4.2, this approach is applicable. However, it has a number of drawbacks. First, dealing with ordered collections, we run into the problem of identifying monotone fragments of the language. As we have seen in examples of $\mathcal{NRL}_a$ and or-$\mathcal{NRL}_a$, this leads to undecidable problems. Second, although there is a correspondence between different algebras used to characterize approximations, it is not always the case that some of them can be expressed in terms of the others. Consequently, instead of having a language with just one structural recursion construct, or one set of monad operations, we need one for each approximation which makes the language very inconvenient to use.

In an attempt to overcome these problems, we look at the semantic connection between approximations and sets and or-sets established in propositions 4.19 and 4.20. This connection suggests that approximation constructions can be encoded with sets and or-sets. We use these encodings to show that all monads arising from the universality properties of approximations can be expressed in or-$\mathcal{NRL}_a$. In addition, if type t encoded a certain approximation construction, then the ordering $\leq_t$ definable in or-$\mathcal{NRL}_a$ is precisely the Buneman ordering used for that kind of approximations.

5.3.1 Structural recursion on approximations

We start with mixes. Mixes will be considered as a new type constructor. That is, for any type t we now have a new type $t\ mix$ such that $[\![t\ mix]\!] = \mathcal{P}^{\forall}([\![t]\!])$. Since mixes arise as free mix algebras, we can define the structural recursion on mixes as follows:

$$\begin{array}{llcl}
fun & sr_mix[e,f,u,h](\emptyset,\emptyset) & = & e \\
\mid & sr_mix[e,f,u,h](\eta(x)) & = & f(x) \\
\mid & sr_mix[e,f,u,h](M_1+M_2) & = & u(sr_mix[e,f,u,h](M_1), sr_mix[e,f,u,h](M_2)) \\
\mid & sr_mix[e,f,u,h](\Box M) & = & h(sr_mix[e,f,u,h](M))
\end{array}$$

Similarly to the case of sets and bags, *sr_mix* is well-defined iff e, u, h' supply its range with the structure of a mix algebra. Now if we consider only those mixes whose second component is empty, checking this precondition is the same as checking whether e and u supply its range with the structure of a semilattice with identity, and this is undecidable according to Breazu-Tannen and Subrahmanyam [27]. Therefore, well-definedness of *sr_mix* is undecidable.

Our approach is to impose syntactic restriction on the general form of structural recursion. That is, to go from structural recursion to a monad. In the case of mixes it yields the following construct:

$$mix\_ext(f) \stackrel{\text{def}}{=} sr\_mix[(\emptyset,\emptyset), f, +, \Box]$$

provided f sends elements of type t to s *mix*. In this case *mix_ext*(f) is a function of type t *mix* $\rightarrow s$ *mix*. However, this alone does not eliminate the need to verify preconditions in the case when we use the ordered semantics. As we have just shown, restricting mixes to those with the empty second component we obtain a sublanguage of the expressive power of the $\mathcal{NRL}$ monad constructs. Therefore, monotonicity of f is needed for well-definedness of *mix_ext*. And we know that even in $\mathcal{NRL}_a$ monotonicity is undecidable.

Our second example is sandwiches. Again, we view them as a type constructor t *sand* such that $[\![t\ sand]\!] = \mathcal{P}^{\forall\wedge}([\![t]\!])$. Since sandwiches arise as free mix algebras generated by the consistent closure, we can define the structural recursion on sandwiches as follows:

$$\begin{array}{llcl}
fun & sr_sand[e,f,u,h](\emptyset,\emptyset) & = & e \\
\mid & sr_sand[e,f,u,h](\eta^{\dagger}(x,y)) & = & f(x,y) \\
\mid & sr_sand[e,f,u,h](S_1+S_2) & = & u(sr_sand[e,f,u,h](S_1), sr_sand[e,f,u,h](S_2)) \\
\mid & sr_sand[e,f,u,h](\Box S) & = & h(sr_sand[e,f,u,h](S))
\end{array}$$

If we consider the subset of sandwiches generated by A, then it coincides with the family of mixes over the same poset, see theorem 4.41. Therefore, well-definedness of *sr_sand* is undecidable. The monad construct

$$ext\_sand(f) \stackrel{\text{def}}{=} sr\_sand[(\emptyset,\emptyset), f, +, \Box]$$

is well-defined iff f is monotone which again is undecidable.

As our last example, we consider snacks which again are viewed as a type constructor: t *snack* is a type whose semantic domain is $\boldsymbol{\mathcal{P}}^{\forall}([\![t]\!])$. Since snacks are free algebras in the signature having one nullary operation and two binary operations, we define the structural recursion on them as follows:

$$\begin{array}{llcl}
fun & sr_snack[e,f,u,h](\emptyset,\emptyset) & = & e \\
\mid & sr_snack[e,f,u,h](\eta(x)) & = & f(x) \\
\mid & sr_snack[e,f,u,h](S_1 + S_2) & = & u(sr_snack[e,f,u,h](S_1), sr_snack[e,f,u,h](S_2)) \\
\mid & sr_snack[e,f,u,h](S_1 \cdot S_2) & = & h(sr_snack[e,f,u,h](S_1), sr_snack[e,f,u,h'](S_2))
\end{array}$$

Again, by restricting our attention only to snacks with empty second component, we see that the well-definedness condition, which for snacks requires $+$ and $\cdot$ to form a distributive semilattice with e being the identity for $+$, is now the same as well-definedness for the structural recursion on sets and hence undecidable. The monad construct

$$ext\_snack(f) \stackrel{\text{def}}{=} sr\_snack[(\emptyset,\emptyset), f, +, \cdot]$$

is similarly well-defined iff f is monotone, and monotonicity is undecidable even in the $\mathcal{NRL}$ fragment.

The reader is invited to do similar exercises with other approximations and observe similar phenomena. Now we can summarize the major problems of using the approach based on structural recursion and monads for programming with approximations.

- Most operations used in the universality properties for approximations are not as intuitive as union, intersection and so on. Therefore, the average programmer would have a very hard time trying to write a program that uses constructs like *sr_mix* or *ext_snack*.
- All approximations have different equational characterizations, and therefore there are ten forms of structural recursion and ten sets of the monad primitives. This means that the language must contain all of them and therefore it is going to be too complicated to comprehend even for a theoretician, let alone a programmer. Furthermore, in many applications more than one approximation model is used, and therefore in addition to ten approximations we also need a few dozen of operations that coerce one approximation into another.
- Verification of preconditions remains a big problem and it can not be taken care of by the compiler as the preconditions are undecidable – even for the monad operations when the ordered model is used.

Therefore, we need a unifying framework for programming with approximations. And such a framework is given by the language for sets and or-sets *or-*$\mathcal{NRL}$.

5.3.2 Using sets and or-sets to program with approximations

When we discussed semantics of sets, or-sets and approximations, we saw that approximations can be encoded as objects in the type system of *or-*$\mathcal{NRL}$. In fact, the following encoding was proposed:

Approximations	Encoding
t *mix*, t *sand* and similar	$\langle t\rangle \times \{t\}$
t *snack*, t *scone* and similar	$\langle t\rangle \times \{\langle t\rangle\}$

Using these encodings, we can encode the monad operations on approximations. Consider mixes. First we notice that the Buneman ordering for mixes over type t, which is $\sqsubseteq^\sharp \times \sqsubseteq^\flat$, is precisely $\leq_{\langle t\rangle\times\{t\}}$. For $f : t \to s\ mix$, where $s\ mix$ is now abbreviation for $\langle s\rangle \times \{s\}$, we have

$$mix\_ext(f) = \lambda(U, L).(or\_\mu_a(or\_map_a(\pi_1 \circ f)(U)), \mu_a(map_a(\pi_2 \circ f)(L)))$$

Mix singleton is defined as $\eta\_mix(x) = (or\_\eta, \eta)$. Then, for $g : s \to t$,

$$mix\_map(g)(U, L) = (or\_map_a(g)(U), map_a(g)(L)) : s\ mix \to t\ mix \qquad \text{and}$$

$$\mu\_mix = \lambda x.(or\_\mu_a(or\_map_a(\pi_1))(x), \mu_a(map_a(\pi_2)(x))) : s\ mix\ mix \to s\ mix$$

Now we have the following standard monad equations for any monotone f and g:

- $mix\_ext(f) = \mu\_mix(mix\_map(f))$
- $\mu\_mix = mix\_ext(\lambda x.x)$
- $mix\_map(g) = mix\_ext(\lambda x.\eta\_mix(g(x)))$

Our second example is snacks. We use $t\ snack$ as an abbreviation for $\langle t\rangle \times \{\langle t\rangle\}$. First observe that $\leq_{t\ snack}$ is precisely the Buneman order used for snacks, and hence our encoding again agrees with the ordering. But the important question is how to express $ext\_snack(f) : s\ snack \to t\ snack$ if $f : s \to t\ snack$ is given.

Assume that we have a snack $\mathcal{S} = (U, \mathcal{L})$ of type $s\ snack$. Then $ext\_snack(f)(\mathcal{S})$ can be found as

$$ext\_snack(f)(\mathcal{S}) \quad = \quad (\sum_{u\in U} f(u)) \cdot e + \sum_{L\in\mathcal{L}} \prod_{l\in L} f(l)$$

Look at the first component. If $f(u) = (V_u, N_u)$, then it is equal to $\min(\bigcup_{u\in U} V_u)$ and therefore can be expressed as $C_0 = or\_\mu_a(or\_map_a(\pi_1 \circ f)(\pi_1\ \mathcal{S}))$.

Now fix $L \in \mathcal{L}$. Assume that $f(l) = (W_l, \mathcal{M}_l)$ for each $l \in L$. Then

$$\prod_{l\in L} f(l) = (\min \bigcup_{l\in L} W_l, \max{}^\sharp(\min(\bigcup_{l\in L} M_l \mid M_l \in \mathcal{M}_l)))$$

To find the first component, compute $or\_\mu_a(or\_map_a(\pi_1 \circ f)(L)$. To find the second component, observe that $X = or\_map_a(\pi_2 \circ f)(L)$ is $\langle \mathcal{M}_l \mid l \in L\rangle$. Therefore, the second component is simply $map_a(or\_\mu_a(\beta_a(X)))$. Here β_a is the inverse of α_a, that is, isomorphism between the semantic domains of types $\langle\{t\}\rangle$ and $\{\langle t\rangle\}$. It is not hard to see that in the presence of *set_to_or* and *or_to_set* it is possible to express β_a in *or*-$\mathcal{NRL}$. Hence, we can write a function

$$g \quad := \quad (or\_\mu_a \circ or\_map_a \circ (\pi_1 \circ f), map_a \circ or\_\mu_a \circ \beta_a \circ or\_map_a \circ (\pi_2 \circ f))$$

which, when applied to L, produces $\prod_{l \in L} f(l) = (Z_L, \mathcal{N}_L)$.

Now we need to calculate $\sum_L (Z_L, \mathcal{N}_L) = (\min \bigcup_L Z_L, \max^\sharp(\bigcup_L \mathcal{N}_L))$. The second component can be obtained as

$$C_2 = \mu_a(map_a(\pi_2 \circ g)(\mathcal{L}))$$

and it is of type $\{\langle t\rangle\}$. To compute the first component, we need a way out of sets to get an or-set. This is achieved by writing $C_1 = or\_\mu_a(set\_to\_or(map_a(\pi_1 \circ g)))(\mathcal{L})$. Finally, we have

$$ext\_snack(f)(\mathcal{S}) = (or\_\cup_a(C_0, C_1), C_2)$$

Summing up, we obtain the following result.

Theorem 5.32 *All monad constructs arising from the universality properties of approximations and all operations given by those universality properties can be expressed in* or-$\mathcal{NRL}(\leq_b)$*, possibly enhanced with set_to_or and or_to_set in the case of multi-element lower approximations.* $\square$

We do not give the proofs for other approximations, but it proceeds straightforwardly along the same lines as the proofs for mixes and snacks, following representation of approximations from singleton developed in the proofs of their universality properties.

Therefore, we believe that encoding approximations and using *or*-$\mathcal{NRL}$ with very little extra power is a much better way to program with those than using just structural recursion and monads based on the universality properties. In the next chapter we give examples of programming with approximations in a practical language based on *or*-$\mathcal{NRL}$.

Chapter 6

OR-SML

In this chapter we describe a functional database language OR-SML for handling disjunctive information in database queries, and its implementation on top of Standard ML [114]. The core language of this implementation is *or*-$\mathcal{NRL}$, hence the name OR-SML. We give examples of queries which require disjunctive information (such as querying incomplete or independent databases) and show how to use the language to answer the queries. The language is extended in a way that allows dealing with bags and aggregate functions. It is also configurable by user-defined base types.

Since the system runs on top of Standard ML and all database objects are values in the latter, the system benefits from combining a sophisticated query language with the full power of a programming language. The language has been implemented as a library of modules in Standard ML. This allows the user to build just the database language as an independent system, or to interface it to other systems built in Standard ML. The ML module system makes the implementation of different parts of the language virtually independent and thus easy to change without touching the rest of the system.

We describe OR-SML in the first section. In the second section of this chapter, we show how it can be applied to problems of querying independent and incomplete databases.

6.1 Overview of OR-SML

As we have just said, the core language of OR-SML is *or*-$\mathcal{NRL}$. But the system OR-SML includes much more than just *or*-$\mathcal{NRL}$. First, normalization is present as a primitive. Some limited arithmetic is added to elevate the language to the expressive power of the *bag* language $\mathcal{BQL}$. We show how bags and certain aggregate functions can be encoded. OR-SML also allows programming with structural recursion on sets and or-sets. The system is extensible with user-defined base types. It provides a mechanism for converting any user-defined functions on base types into functions that fit into the type system of OR-SML. It also gives a way "out of complex objects" into SML values. This is necessary, for example, if OR-SML is a part of a larger system and the OR-SML query is part of a larger computation that needs to analyze the result of the query to proceed. OR-SML comes equipped with libraries of derived functions that are helpful in writing programs or advanced applications such as querying independent databases.

We chose Standard ML (SML) as the basis for our implementation in order to combine the simplicity of *or-$\mathcal{NRL}$* queries with features of a functional programming language [114]. OR-SML benefits from it in a number of ways:

1. OR-SML queries may involve and become involved in arbitrary SML procedures. The usefulness of this is enhanced by the presence of higher-order functions in SML, allowing SML functions to be arguments to queries and queries to be arguments to SML functions.
2. OR-SML is implemented as a library of modules in SML. This allows the user to build just the database language as an independent system, or to interface it to other systems built in SML.
3. The stand-alone version of OR-SML is implemented as a library loaded into the interactive system of SML, and as such is an interactive system itself. One interacts with OR-SML by entering declarations and expressions to be evaluated into the top-level read-evaluate-print loop of SML. The results are then bound to SML identifiers for future use.
4. The SML module system makes the implementation of different parts of the language virtually independent and easily modifiable.

As of now, the system is suitable for querying small and medium size databases (hundreds of records), which are fairly common. To extend its capabilities to handle large databases, certain changes need to be made; in particular, optimizations in the presence of disjunctive information need to be added to OR-SML. As we have just mentioned, due to the modularity of the implementation, such changes can often be made without affecting the way the system looks to the end-user.

In what follows we shall need some of the SML syntax. The interested reader is referred to Milner at al. [114] for the definition of Standard ML or to Paulson [133] for a more humane introduction. But the following "primer" should be sufficient to understand the examples in this chapter.

In SML, `val` binds an identifier and `-` is the SML prompt, so `- val x = 2;` binds `x` to 2 and `val x = 2 : int` is the SML response saying that `x` is now bound to 2 which is of type `int`. `fun` is used for function declaration. Functions in SML can also be created without being named by using the construct `(fn x =>` $body$(`x`)`)`. For example

```
- 3 + (fn x => x + 1) 2;
val it = 6 : int
- fun makepair x = (fn y => (x,y));
val makepair = fn : 'a -> 'b -> 'a * 'b
- val makepairwith1 = makepair 1;
val makepairwith1 = fn : 'a -> int * 'a
- makepairwith1 2;
val it = (1,2) : int * int
```

Symbols like `'a` are used to indicate polymorphic types. For example, `makepairwith1` takes a value v of *any* type `'a` and forms a pair $(1, v)$ of type `int * 'a`.

If a function is applied to its argument and the result is not bound to any variable, then SML assigns it a special identifier `it` which lives until it is overridden by the next such application. We have seen two examples of this above. If one writes `- factorial 4;`, this will cause the SML response `val it = 24 : int`. `let` ... `in` ... `end` is used for local binding. The `[...]` brackets denote lists; `""` is used for strings. The symbol `@` is used for list append. For example:

```
- let val a = ["a","b"]
      val b = ["b","c"]
  in a @ b end;
val it = ["a","b","b","c"] : string list
```

6.1.1 Core language

The core language of OR-SML is *or-*$\mathcal{NRL}$. In the table below we show the correspondence between *or-*$\mathcal{NRL}$ primitives and their names in OR-SML.

or-$\mathcal{NRL}$ name	OR-SML name	*or-*$\mathcal{NRL}$ name	OR-SML name
$f \circ g$	`comp(f,g)`	$if - then - else$	`cond`
π_1, π_2	`p1, p2`	$!$	`bang`
(f, g)	`pair(f,g)`	id	`id`
η	`sng`	$empty$	`empty`
$\cup$	`union`	μ	`flat`
ρ_2	`pairwith`	map	`smap`
or_η	`orsng`	or_empty	`orempty`
$or\_\cup$	`orunion`	or_μ	`orflat`
or_ρ_2	`orpairwith`	or_map	`orsmap`
α	`alpha`	$normalize$	`normal`

Let us describe how these constructs are represented over SML. Every complex object has type `co`. We shall refer to the type of an object or a function in *or-*$\mathcal{NRL}$ as its *true type*. Types of complex objects can be inferred; they are SML values having type `co_type`. When OR-SML prints a complex object together with its type, it uses `::` for the true type, as `: co` is used to show that the SML type of the object is `co`. Values are created by functions `create : string -> co` or `make : unit -> co` (interactive creation). The function `make` is terminated by typing ".". For example:

```
- val a = make();
{ <1,2,3>, <4,5,6>,
  <7,8> }.
val a = {<1, 2, 3>, <7, 8>, <4, 5, 6>} :: {<int>} : co
- val b = create "(2,'abc')";
val b = (2, 'abc') :: int * string : co
```

Notice that the order of elements in the set was changed. This is the result of the duplicate elimination algorithm which will be discussed later.

Typechecking is done in two steps. Static typechecking is simply SML typechecking; for example, trying to call `union(a,a,a)` will cause an ML type error. However, since all objects have type `co`, the SML typechecking algorithm can not detect all type errors statically. For example, ML will see nothing wrong with `union(a,b)` even though the true types of `a` and `b` are $\{\langle int\rangle\}$ and $int \times string$. Hence, the remaining type errors are detected dynamically by OR-SML and an appropriate exception is raised. For instance,

```
- union(a,b);
uncaught exception Badtypeunion
```

The language we presented can express many functions commonly found in query languages, for example, Boolean *and*, *or* and negation, membership test, subset test, difference, selection, cartesian product and their counterparts for or-sets, see section 3.2 and [26, 104]. These functions are included in OR-SML in the form of a structure called `Set`. Some examples of programming using the core language and functions from `Set` are given below. Notice that we use `'...'` for strings to distinguish them from SML strings.

```
- alpha (create "{<1,2>,<2,3>}");
val it = <{2}, {1, 2}, {1, 3}, {2, 3}> :: <{int}> : co
- val x1 = create "{1,2}";
val x1 = {1, 2} :: {int} : co
- smap (pair(id,id)) x1;
val it = {(1, 1), (2, 2)} :: {int * int} : co
- val x2 = create "{3,4}";
val x2 = {3, 4} :: {int} : co
- union(x1,x2);
```

```
val it = {1, 2, 3, 4} :: {int} : co
- Set.cartprod(x1,x2);
val it = {(1, 3), (1, 4), (2, 3), (2, 4)} :: {int * int} : co
- val y = create "<1,2,3,4>";
val y = <1, 2, 3, 4> :: <int> : co
- val z = create "'ab'";
val z = 'ab' :: string : co
- orpairwith(z,y);
val it = <('ab', 1), ('ab', 2), ('ab', 3), ('ab', 4)> :: <string * int> : co
- orsmap p1 it;
val it = <'ab'> :: <string> : co
```

Normalization of types and objects is represented in OR-SML by two functions `normalize : co_type -> co_type` and `normal co -> co`. For example,

```
- val x = create "{(1,<2,3>),(4,<5,6>)}";
val x = {(1, <2, 3>), (4, <5, 6>)} :: {int * <int>} : co
- normalize (typeof x);
val it = <{int * int}> : co_type
- normal x;
val it = <{(1, 2), (4, 5)}, {(1, 3), (4, 5)}, {(1, 2), (4, 6)}, {(1,3), (4, 6)}> : co
```

OR-SML allows user defined base types. Values of these types have type `base` in ML. The user is required to supply a structure containing basic information about the base type when a particular version of OR-SML is built. One of the functions that is included in this user-supplied structure is parsing; its type is `string -> base`. If user-defined base types are used, then creation of objects requires special care. Objects of base type are printed in parentheses and preceded by the symbol `@`. They also must be input accordingly if `make` or `create` is used. For example, in a version of OR-SML with real numbers, one would write:

```
- val a = create "@(2.5)";
val a = @(2.5) :: real : co
```

In the case of reals numbers, the symbol `"."` plays a crucial role and can not be used to indicate the end of the input to `make`. There is a way to change the symbol whose meaning is "end of object".

```
- End_symb := "!";
val it = () : unit
- val b = make ();
{ @(2.5), @(3.5), @(4.5) }!
val b = {@(2.5), @(3.5), @(4.5)} :: {real} : co
```

There are also a number of functions that make complex objects out of ML objects. These are necessary, for example, if a user-defined base type is supplied without a parser. In this case objects can be created using constructor functions. The function `mkbaseco` is used to produce a complex object (that is, an element of type `co`) from an element of base type. Similarly, `mkintco` produces complex object integers, `mkprodco` produces a pair from two complex objects and `mksetco` and `mkorsco` produce sets and or-sets from lists of complex objects. For example:

```
- val a = [[2.5,3.7],[4.5,5.3]];
val a = [[2.5,3.7],[4.5,5.3]] : real list list
- val co_a = mksetco(map (fn z => mkorsco(map mkbaseco z)) a);
val co_a = {<@(2.5), @(3.7)>, <@(4.5), @(5.3)>} :: {<real>} : co
```

There are various styles for printing objects and object types. Some of them are better suited for printing normalized objects, while others do not distinguish between sets and or-sets. All styles for objects and types can be freely combined, giving OR-SML a total of nine different printing styles. A new printer can be installed by using the functions `printer` and `printer_type` of type `int -> unit`. These functions can be invoked at any time. Further details can be found in the system manual E. Gunter and Libkin [69]. In examples in this chapter we use different printing styles. For instance, we often chose not to print types of objects if those do not fit on one line.

This concludes our discussion of the core language. In the subsequent sections we will show how to enrich the language to make it suitable for solving problems related to normalization and approximations.

6.1.2 Additional features

Arithmetic functions

OR-SML has integers as one of its base types. The following operations are available on integers: addition, multiplication, monus, summation over sets and or-sets and *gen*. In the table below we give their OR-SML names:

or-$\mathcal{NRL}$ name	OR-SML name	*or*-$\mathcal{NRL}$ name	OR-SML name
$+$	`plus`	$\cdot$	`mult`
$\dot{-}$	`monus`	$\sum$	`sum`
gen	`gen`	$or\_\sum$	`orsum`

The reason these operators have been included comes primarily from our discussion of bags. As we have seen, these operators elevate a set language to a bag language (with power operators and/or structural recursion). If bags are represented as sets of pairs of "element–number of occurrences", all functions on bags from subsection 3.2.3 can now be modeled easily in OR-SML. For example, under the assumption that in a bag X for each element all its occurrences are recorded once (that is, we can not have pairs $(a, 2)$ and $(a, 3)$ instead of one pair $(a, 5)$), the difference of two bags $X - Y$ is

$$select(\lambda z.\neg eq(\pi_2(z), 0))(map(\lambda x.(\pi_1(x), monus(\pi_2(x), \Sigma(\pi_2)(select(\lambda y.eq(\pi_1(x), \pi_1(y)))(Y)))))(X))$$

We are using a function *select* from `Set` which takes in a predicate $p : t \to bool$ and a set $X : \{t\}$ and returns $\{x \in X \mid p(x)\}$. Below we show how to implement these functions in OR-SML. First, total second column would look like

```
- val x = create "{('a',2),('b',4),('c',1)}";
val x = {('c', 1), ('a', 2), ('b', 4)} :: {string * int} : co
- val y = create "{('b',1),('b',2),('c',3),('d',1)}";
val y = {('d', 1), ('b', 1), ('b', 2), ('c', 3)} :: {string * int} : co
- sum p2 y;
val it = 7 :: int : co
```

Bag difference can be implemented as follows:

```
fun bag_diff (x,y) = let
     fun equals_a  a = select (fn z => eq(p1(z),p1(a))) y
in select (fn v => neg(eq(p2(v),mkintco(0))))
     (smap (fn z => mkprodco(p1(z),monus(p2(z),(sum p2 (equals_a z))))) x)
end;
val bag_diff = fn : co * co -> co
- bag_diff(x,y);
val it = {('b', 1), ('a', 2)} :: {string * int} : co
```

Various functions can be implemented using arithmetic functions. Two of them, which are of particular importance, are included in the standard library `Set`. One is `card`, and the other is rank assignment function `sort` : $\{s\} \to \{s \times int\}$ discussed in subsection 3.2.3. Note that `card` is simply summation of the constant function:

```
- val card = sum (fn x => mkintco(1));
val card = fn : co -> co
- card (create "{1,2,3,4}");
val it = 4 :: int : co
```

To be able to assign unique ranks to elements of a set, it is necessary to lift order to all types, as it is done in theorem 3.29. This is implemented by means of a function `leq:co -> co` in the structure `Set` that compares objects of the same true type (if true types do not coincide, it raises exception `Cannotcompare`.) For example:

```
- val a = create "{<1,2,3>, <4,5,6>, <8,4>}";
val a = {<1, 2, 3>, <4, 8>, <4, 5, 6>} :: {<int>} : co
- val b = create "{<2,5,6>, <1,3>, <4,2>}";
val b = {<1, 3>, <2, 4>, <2, 5, 6>} :: {<int>} : co
- val c = create "{1,2,3}";
val c = {1, 2, 3} :: {int} : co
- leq(a,b);
val it = F :: bool : co
- leq(b,a);
val it = T :: bool : co
- leq(b,c);

uncaught exception Cannotcompare
- sort a;
val it = {(<1, 2, 3>, 1), (<4, 8>, 3), (<4, 5, 6>, 2)} :: {(<int> * int)} : co
```

Primitives involving base types

Since the system allows user-defined base types, it must provide a way of making functions on those base types into functions that fit into the type system of OR-SML. For example, if the user-defined base type is `real`, there must be a way to have a function `plus : co * co -> co` whose semantics is addition of real numbers. Furthermore, there is a need for a mechanism of translation of predicates on base types into predicates on complex objects that can be used with `cond` and `select`.

The solution to this problem is given by the function `apply` that takes a function `f : base list -> base` and returns a function from `co` to `co` representing the action of `f` on complex objects. For example, if `val f_co = apply f`, then `f_co` applied to a complex object $(r_1, (r_2, r_3))$ yields `f` $[r_1, r_2, r_3]$ in the form of a complex object.

In practice, most of the primitives on base types are unary or binary. Therefore, OR-SML has a special feature that allows you to apply binary and unary functions on base types by using functions `apply_unary`, `apply_binary` and `apply_op2`. The difference between `apply_binary` and `apply_op2` is that `apply_binary` produces a function of type `co -> co` whose true type is supposed to be $b \times b \to b$. That is, the argument must be a pair. The function `apply_op2` produces a function of type `co * co -> co`. For predicates, `apply_test` takes a function of type `(base -> bool)` and returns it in the form of a function on complex objects.

Example:

```
- val addone_co = apply_unary ( fn x => x + 1.0);
val addone_co = fn : co -> co
- val x = create "{ @(2.5),@(4.5) }";
val x = {@(2.5), @(4.5)} :: {real} : co
```

```
- smap addone_co x;
val it = {@(3.5), @(5.5)} :: {real} : co
- val addreal_co = apply_binary (fn ((x:real),(y:real)) => x + y);
val addreal_co = fn : co -> co
- smap addreal_co (Set.cartprod(x,x));
val it = {@(5.0), @(7.0), @(9.0)} : co
- val biggerthanthree_co = apply_test (fn x => x > 3.0);
val biggerthanthree_co = fn : co -> co
- Set.select biggerthanthree_co x;
val it = {@(4,5)} :: {real} : co
```

Structural recursion

Structural recursion on sets and or-sets a very powerful programming tool for query languages. Unfortunately, it is too powerful because it is often unsafe. A function defined by structural recursion is not guaranteed to be well-defined, and well-definedness can not be generally checked by a compiler. It is, however, often helpful in writing programs or changing types of big databases (rather than reinputting them), so we have decided to include structural recursion in OR-SML. Structural recursion on sets and or-sets is available to the user by means of two constructs `sr` and `orsr`.

$$\frac{f : s \times t \to t \qquad e : t}{\mathtt{sr}(e, f) : \{s\} \to t} \qquad \frac{f : s \times t \to t \qquad e : t}{\mathtt{orsr}(e, f) : \langle s \rangle \to t}$$

They take an object e of type t and a function f of type $s \times t \to t$ and return a function $\mathtt{sr}(e, f)$ of type $\{s\} \to t$ or a function $\mathtt{orsr}(e, f)$ of type $\langle s \rangle \to t$ respectively. The semantics is as follows: $\mathtt{sr}(e, f)\{x_1, \ldots, x_n\} = f(x_1, f(x_2, f(x_3, \ldots f(x_n, e) \ldots)))$ and similarly for `orsr`. The two functions implementing structural recursion are `SR.sr` and `SR.orsr`. For example, to find the product of elements of a set, one may use structural recursion as follows:

```
- val fact = SR.sr((create "1"),mult);
val fact = fn : co -> co
- fact (create "{1,2,3,4,5}");
val it = 120 :: int : co
```

There are a few functions that can be written with help of structural recursion which are included in the library "sr.lib". Among them are `set_to_or` : $\{t\} \to \langle t \rangle$ and `or_to_set` : $\langle t \rangle \to \{t\}$ that convert sets into or-sets and vice versa, `powerset` : $\{t\} \to \{\{t\}\}$ (which can also be implemented using just α), and `pick` : $\{t\} \to t$ which picks an element of a set.

In section 3.2 we showed that structural recursion is equivalent to the *loop* construct that iterates a function once for each element of a set. In the following example we show how to implement *loop* and how to use it to iterate the function that increments an integer given number of times. Recall that `c` from the example of applying `sort` is a three-element set.

```
- fun loop f = (fn (X,z) => SR.sr(z, (fn (v1,v2) => f(v2)))(X));
val loop = fn : (co -> co) -> co * co -> co
- val one = create "1";
val one = 1 :: int : co
- fun intaddone x = plus(x,one);
val intaddone = fn : co -> co
- loop intaddone (c,one);
val it = 4 :: int : co
```

Moreover, using `sort` it is now possible to give an *efficient* translating from `loop` to structural recursion:

```
- fun select_max X = Set.select (fn z => eq(p2(z),Set.card(X))) X;
val select_max = fn : co -> co
- fun new_sr (e,f) =
   let fun g INPUT = let val X_curr   = p1 INPUT
                         val RES_curr = p2 INPUT
                         val x_max = select_max X_curr
          in mkprodco(
                       Set.diff(X_curr,x_max),
                       flat((smap
                            (fn z => (smap (fn v => f(p1(z),v)) RES_curr))
                             x_max)))
          end
in
   (fn X => p2(loop g (X,mkprodco((Set.sort(X),sng(e))))))
end;
val new_sr = fn : co * (co * co -> co) -> co -> co
- val new_fact = new_sr((create "1"),mult);
val new_fact = fn : co -> co
- new_fact (create "{1,2,3,4,5}");
val it = {120} :: {int} : co
```

This example shows the "cost" one has to pay for translation from *loop* into structural recursion (cf. theorem 3.35): instead of a value v, the translation produces the singleton $\{v\}$.

I/O

To support a form of persistence for databases, OR-SML provides means for writing lists of complex objects to files and reading such lists back in later. There are two modules for file I/O in OR-SML: one working with binary files and one with ASCII files. Working with ASCII files is relatively safe: if there is any problem with reading an object, an exception will be raised. (It is not safe from editing). However, it requires a parser for objects of base type, because strings read from a file are parsed to create complex objects.

If a parser for objects of base type was not provided, then the binary input-output module must be used. Since binary I/O is an unsafe feature of Standard ML [158], all binary files are required to have the extension ".db". If it is not used, OR-SML will add it and ask if the operation should be continued. It is also possible to obtain the list of all files with extension ".db" in the current directory using the function `show_db:unit -> unit`.

The ASCII input-output module provides two functions: `store_db: co list * string -> unit` takes a database and a file name and stores the database. For example, `store_db (db,"mydb")` stores a list of complex objects `db` in a file `"mydb"`. To read a database, use `retrieve_db : string -> co list`. This function takes a file name and returns the database stored in that file.

If a parser for objects of base type was not provided, it is necessary to use the binary input-output module. Function `write_db: co list * string -> unit` is used to write a database to a file. For example, `write_db(db,"mydb.db")` will write a list of complex objects `db` into the file "mydb.db". Moreover, `write_db(db,"mydb.db")` and `write_db(db,"mydb")` will have the same effect. Databases are read by using the function `read_db: string -> co list`. For instance, `val db = read_db("mydb")` creates a list of complex objects stored in "mydb.db".

Example (in this example we use function `tl` that produces the tail of a list).

```
- val DB = let val a = create "{1,2,3}"
               val b = create "{2,3,4}"
               val c = create "{5,6,7}"
           in [a,b,c] end;
val DB = [{1, 2, 3},{2, 3, 4},{5, 6, 7}] : co list
- store_db(DB,"mydbfile");
val it = () : unit
- write_db (tl(DB), "mydbfile");
File names must have extension .db
Do you want to write your database in mydbfile.db?(yes,no) yes

Database written to mydbfile.db

val it = () : unit
- show_db();
```

```
mydbfile.db
```

Now we have two files, one named `mydbfile` and containing three sets, and the other named `mydbfile.db` and containing two sets. It is possible to read them back:

```
- val get_big_DB = retrieve_db "mydbfile";
val get_big_DB = [{1, 2, 3},{2, 3, 4},{5, 6, 7}] : co list
- val get_small_DB = read_db "mydbfile";
File names must have extension .db
Do you want to read your database from mydbfile.db?(yes,no) yes
Warning: read is an unsafe operation.
If there is a problem with your file, it will throw you out of orsml
Are you ready to read the file? (yes,no) yes
val get_small_DB = [{2, 3, 4},{5, 6, 7}] : co list
```

Deconstruction of complex objects

It may be the case that after evaluating a query, the user may need to write some program to deal with the result. Since all operations of OR-SML work with type `co`, there is a need to have a way out of complex objects to the usual ML types. The structure `DEST` contains some functions to deconstruct complex objects and obtain ML values. For example, to convert an object of true type $\{\langle int \rangle\}$ (which still has SML type `co`) into `int list list`, one writes:

```
- val a = create "{<1,2>,<3,4>}";
val a = {<1, 2>, <3, 4>} : co
- DEST.co_to_list a;
val it = [<1, 2>,<3, 4>] : co list
- map DEST.co_to_list it;
val it = [[1,2],[3,4]] : co list list
- map (map DEST.co_to_int) it;
val it = [[1,2],[3,4]] : int list list
```

Orderings and antichains

In chapter 5 we saw that the language for the antichain semantics, *or*-$\mathcal{NRL}_a$, can be viewed as a sublanguage of *or*-$\mathcal{NRL}$. This point of view is supported by OR-SML. It provides a library of derived functions dealing with orderings and antichains. Among them are `leqdom` that compares elements of the same true type (that is, it implements the order $\leq_t$), `meet` and `join` that compute the meet and join operations, `set_max` and `orset_min` that select maximal and minimal elements from sets and or-sets to implement the transformation $x \rightarrow x^\circ$ we used throughout chapter 5.

Note that the true type of `join` and `meet` is $t \times t \rightarrow \langle t \rangle$. If the join (or meet) of two objects x and y is defined, then the corresponding function produces a singleton containing that join or meet. If it is not defined, it produces $\langle\rangle$.

Example:

```
- val a = create "{<1,2,3>,<1,2>,<3,4,5>,<3,4>}";
val a = {<1, 2>, <1, 2, 3>, <3, 4>, <3, 4, 5>} :: {<int>} : co
- val b = create "{<1,2,5,4>,<1,2,4>}";
val b = {<1, 2, 4>, <1, 2, 4, 5>} :: {<int>} : co
- val a1 = set_max a;
val a1 = {<1, 2>, <3, 4>} :: {<int>} : co
- val b1 = set_max b;
val b1 = {<1, 2, 4>} :: {<int>} : co
- leqdom(b1,a1);
val it = T :: bool : co
- join(a, (create "{<7,8>}"));
val it = <{<1, 2>, <3, 4>, <7, 8>}> :: <{<int>}> : co
```

6.1.3 Implementation issues

In this subsection we briefly describe the general structure of OR-SML implementation and discuss duplicate elimination.

The general structure of the implementation of OR-SML is given in figure 6.1. This figure shows dependencies between the pieces of the implementation. Each piece is implemented as an SML functor. A short description of each piece is given in figure 6.2.

In the initial version of OR-SML, duplicate elimination was done straightforwardly. That is, a $O(n^2)$ time complexity algorithm was used. However, a number of experiments revealed that it was mostly the duplicate elimination component that hampered the performance of the system. In the current version we use the following hash function for objects:

$$h(o) = \begin{cases} 1 & \text{if } o : unit \\ o & \text{if } o : int \\ |o| & \text{if } o : string \\ \mathit{if}\ o\ \mathit{then}\ 1\ \mathit{else}\ 0 & \text{if } o : bool \\ h(o_1) + h(o_2) & \text{if } o = (o_1, o_2) \\ h(o_1) + \ldots + h(o_n) & \text{if } o = \{o_1, \ldots, o_n\} \text{ or } o = \langle o_1, \ldots, o_n \rangle \end{cases}$$

Then it is easy to show that, for any type involving sets and or-sets of a type with non-finite domain, for two randomly generated objects o_1 and o_2, the probability of $h(o_1) = h(o_2)$ is zero. Therefore, the expected running time of the duplicate elimination with hashing is $O(n \log n)$. Some results showing performance of OR-SML with two kinds of duplicate elimination algorithm

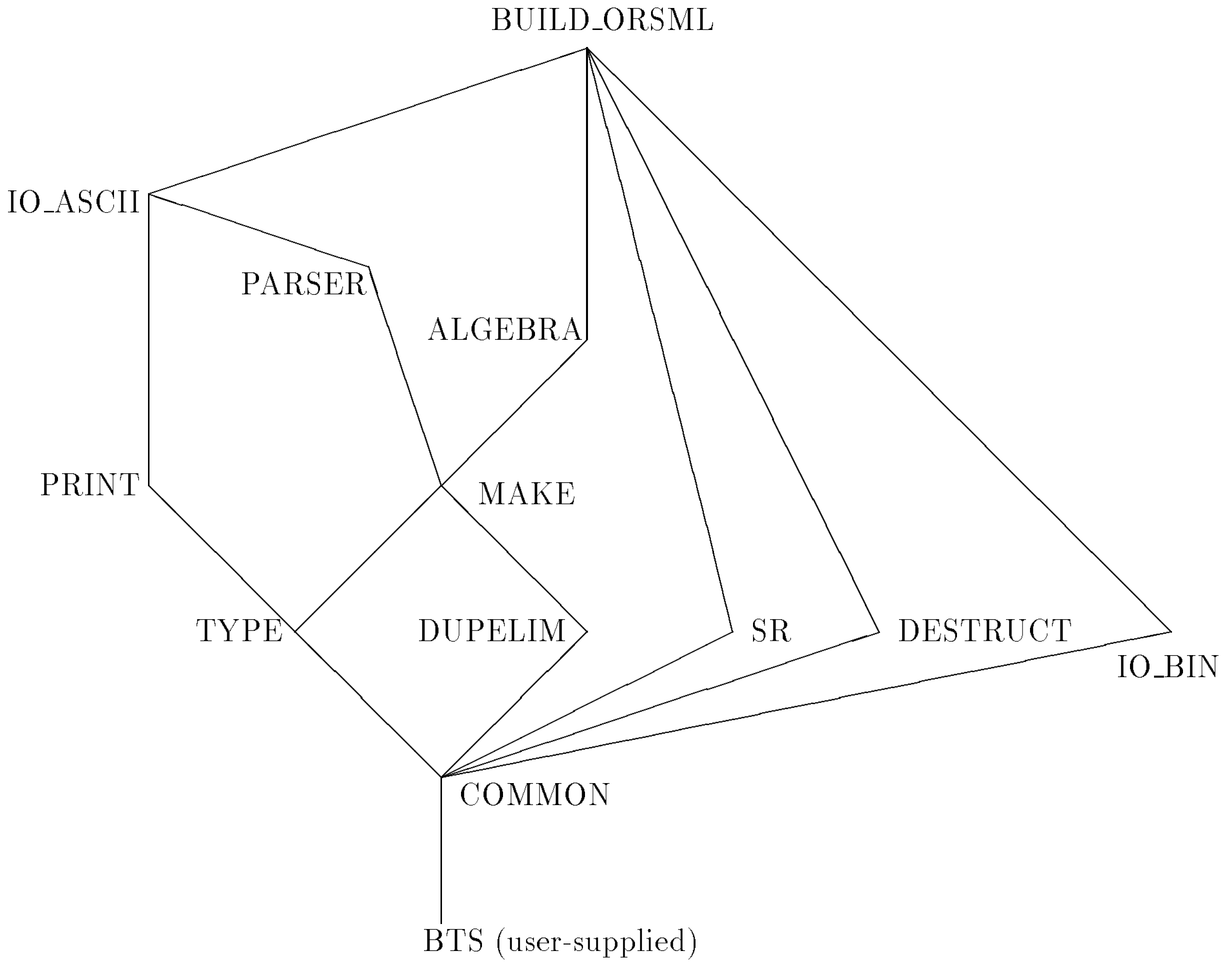


Figure 6.1: OR-SML implementation

BTS	–	Base Type Structure. It is supplied by the user to build a new version of OR-SML with additional base types.
COMMON	–	contains some auxiliary functions used in all other modules.
TYPE	–	provides functions to work with complex object types.
DUPELIM	–	duplicate elimination.
SR	–	implementation of structural recursion.
DESTRUCT	–	functions for destruction of complex objects.
IO_BIN	–	operations for binary file I/O.
MAKE	–	takes as an input structures created by TYPE and DUPELIM and provides functions for creating complex objects.
PRINT	–	takes in the structure created by TYPE and provides printing routines.
PARSER	–	takes in the structure created by MAKE and gives the parser for complex objects.
ALGEBRA	–	implements operations of the language.
IO_ASCII	–	takes in the structures created by PRINT and PARSER and provides operations for the ASCII file I/O.
BUILD_ORSML	–	builds the system and exports it together with ML compiler.

Figure 6.2: Description of OR-SML modules

CartesianProduct

size	50×50	75×75	100×100	150×150
without hashing	26.5	130.5	400.85	1927.37
with hashing	0.47	1.49	3.12	10.56

Flattening

size	200	400	600	800	1000	4000
without hashing	0.02	0.11	0.23	0.37	0.65	10.79
with hashing	0.01	0.02	0.03	0.04	0.07	0.18

Figure 6.3: Comparison of two duplicate elimination algorithms

are shown in figure 6.3. Two functions for which we determined running time are cartesian product and flattening of a large set of sets.

6.2 Applications of OR-SML

In this section we show how to use OR-SML to ask conceptual queries if only a compact representation of incomplete objects is stored in a database, and how to solve some of the problems of querying independent databases described in section 1.3.

6.2.1 Querying incomplete databases

In this subsection we show applications of normalization of databases. We start with a database containing an incomplete design and ask certain queries about possible completed designs. We then show how to write these queries using normalization.

Assume that we have a database containing the incomplete design shown in figure 5.3. That is, the whole design requires two subparts, A and B. An A is either $A1$ or $A2$. The part $A1$ consists of two subparts: $A1.1$ and $A1.2$. An $A1.1$ is either x or y and an $A1.2$ is either z or v. The part $A1.2$ consists of three subparts: $A2.1$, $A2.2$ and $A2.3$. An $A2.1$ is either p or q, an $A2.2$ is either r or s and an $A2.3$ is either t or u. A B consists of $B1$ and $B2$. A $B1$ is either w or k and a $B2$ is either l or m. Now assume that we know the cost and reliability of each part that can make it into the completed designs (that is, for parts denoted by the lower case letters.)

Part	Cost	Reliability
l	12	0.94
m	14	0.95
w	17	0.96
k	11	0.93
x	21	0.999
y	20	0.98
z	13	0.95
v	14	0.955
p	12	0.95
q	13	0.96
r	18	0.97
s	17	0.96
t	19	0.98
u	20	0.99

Now we can create OR-SML values describing these parts as follows:

```
val l = create "('l',(12,@(0.94)))";
val m = create "('m',(14,@(0.95)))";
val w = create "('w',(17,@(0.96)))";
val k = create "('k',(11,@(0.93)))";
val x = create "('x',(21,@(0.999)))";
val y = create "('y',(20,@(0.98)))";
val z = create "('z',(13,@(0.95)))";
val v = create "('v',(14,@(0.955)))";
val p = create "('p',(12,@(0.95)))";
val q = create "('q',(13,@(0.96)))";
val r = create "('r',(18,@(0.97)))";
val s = create "('s',(17,@(0.96)))";
val t = create "('t',(19,@(0.98)))";
val u = create "('l',(20,@(0.99)))";
```

Each part has true type *string* $\times$ (*int* $\times$ *real*). Now B can be created as

```
- val B = mkprodco ((mkorsco [w,k]), (mkorsco [l,m]));
val B =
  (<('k', (11, @(0.93))), ('w', (17, @(0.96)))>,
   <('l', (12, @(0.94))), ('m', (14, @(0.95)))>) : co
```

and $A1$, $A2$ and A can be created as

```
- val A1 = mksetco [(mkorsco [x,y]), (mkorsco [z,v])];
val A1 =
  {<('z', (13, @(0.95))), ('v', (14, @(0.955)))>,
   <('y', (20, @(0.98))), ('x', (21, @(0.999)))>} : co
- val A2 = mksetco [(mkorsco [p,q]), (mkorsco [r,s]), (mkorsco [t,u])];
val A2 =
  {<('p', (12, @(0.95))), ('q', (13, @(0.96)))>,
   <('s', (17, @(0.96))), ('r', (18, @(0.97)))>,
   <('t', (19, @(0.98))), ('l', (20, @(0.99)))>} : co
- val A = mkorsco [A1, A2];
val A =
  <{<('z', (13, @(0.95))), ('v', (14, @(0.955)))>,
    <('y', (20, @(0.98))), ('x', (21, @(0.999)))>},
   {<('p', (12, @(0.95))), ('q', (13, @(0.96)))>,
    <('s', (17, @(0.96))), ('r', (18, @(0.97)))>,
    <('t', (19, @(0.98))), ('l', (20, @(0.99)))>}
   > : co
```

Finally, the whole design is created as

```
- val design = mkprodco (A,B);
val design =
  (<{<('z', (13, @(0.95))), ('v', (14, @(0.955)))>,
     <('y', (20, @(0.98))), ('x', (21, @(0.999)))>},
    {<('p', (12, @(0.95))), ('q', (13, @(0.96)))>,
     <('s', (17, @(0.96))), ('r', (18, @(0.97)))>,
     <('t', (19, @(0.98))), ('l', (20, @(0.99)))>}>,
   (<('k', (11, @(0.93))), ('w', (17, @(0.96)))>,
    <('l', (12, @(0.94))), ('m', (14, @(0.95)))>)) : co
```

Inferring the type of `design` and normalizing it shows us the type of the database of completed designs.

```
val ndt =
  <({(string * (int * real))} *
    ((string * (int * real)) * (string * (int * real))))> : co_type
```

Hence, one can write the cost function which is the sum of the costs of all the parts. In this particular case it is

```
- fun cost X =
      let fun cost1 X = sum (fn z => p1(p2(z))) (p1 X)
          fun cost2 X =  p1(p2(p1(p2(X))))
          fun cost3 X =  p1(p2(p2(p2(X))))
  in plus(cost1(X), plus(cost2(X),cost3(X))) end;
val cost = fn : co -> co
```

Calculating reliability may be a bit harder because it depends on how different parts are connected. In the case of parallel connection of two parts with individual reliabilities r_1 and r_2, the reliability is calculated as $r_1 + r_2 - r_1 \cdot r_2$, whereas for the series connection it is $r_1 \cdot r_2$. To be able to operate with these functions, we must have them as functions from complex objects to complex objects. That is, we need the following:

```
- val rminus = apply_op2 (fn (x:real,y:real) => x - y);
val rminus = fn : co * co -> co
- val rmult = apply_op2 (fn (x:real,y:real) => x * y);
val rmult = fn : co * co -> co
- val rprod = SR.sr ((create "@(1.0)"), rmult);
val rprod = fn : co -> co
- val par_rel = apply_op2 (fn (x:real,y:real) => x + y - (x * y));
val par_rel = fn : co * co -> co
```

Now we can calculate reliabilities for A, $B1$ and $B2$, assuming that subparts of A are connected in series.

```
- fun relA X = rprod (smap (fn z => p2(p2(z))) (p1 X));
val relA = fn : co -> co
- fun relB1 X = p2(p2(p1(p2(X))));
val relB1 = fn : co -> co
- fun relB2 X = p2(p2(p2(p2(X))));
val relB2 = fn : co -> co
```

With these functions, it is possible to write various reliability functions depending on the way A, $B1$ and $B2$ are connected. For example, if only series connection is used, then the total reliability function is the product of `relA`, `relB1` and `relB2`. In our example, we assume parallel connection of $B1$ and $B2$ and series connection of A and B. Then

```
- fun reliability X = rmult(relA(X), par_rel(relB1(X),relB2(X)));
val reliability = fn : co -> co
```

Now assume that we want to answer the following conceptual queries:

- How many completed designs are there?
- Which completed design has the best reliability?
- Which completed design that costs less than n dollars has the best reliability?

To answer these queries, we first *normalize* `design`, creating the or-set of all possible completed designs:

```
val nd = normal design; (* output omitted *)
```

Now it is possible to get all information about reliabilities and costs of completed designs by saying `orsmap cr nd` where `cr` is the function `fn x => mkprodco ((cost x), (reliability x))`. To answer our queries, we write

```
- orsum (fn z => mkintco(1)) nd;
val it = 48 : co
```

Hence, there are 48 completed designs. To find the one that has the best reliability, we write the following query

```
- fun is_better(x,y) = apply_test (fn (z:real) => z > 0.0) (rminus(x,y));
val is_better = fn : co * co -> co
- fun is_best (x,obj) = eq(
                   (Set.orselect
                       (fn y => is_better(reliability(y),
                                          reliability(x)))
                        obj), orempty);
val is_best = fn : co * co -> co
```

and then ask

```
- val select_best = Set.orselect (fn y => is_best(y,nd)) nd;
val select_best =
  <({('v', (14, @(0.955))), ('x', (21, @(0.999)))},
    (('w', (17, @(0.96))), ('m', (14, @(0.95)))))> : co
- orsmap cr select_best;
val it = <(66, @(0.95213691))> : co
```

Thus, we see that the design with the best reliability costs only \$66, even though the cost varies from \$56 to \$82, as we know from mapping `cr` over `nd`. So, as it often happens, one does not have to buy the most expensive thing to get the best quality.

Finally, to select the design with the best reliability that costs under n dollars, we write a function

```
- fun bestunder n =
      let val des_under_n = (Set.orselect (fn y =>
                                          eq(mkintco(0),
                                             monus(cost(y),mkintco(n))))
                                        nd)
in
   Set.orselect (fn y => is_best(y,des_under_n)) des_under_n
end;
val bestunder = fn : int -> co
```

and then ask for the best design that costs under, say, \$62:

```
- bestunder 62;
val it =
  <({('v', (14, @(0.955))), ('x', (21, @(0.999)))},
    (('k', (11, @(0.93))), ('m', (14, @(0.95)))))> : co
- orsmap cr it;
val it = <(60, @(0.9507058425))> : co
```

Again, it is not necessary to get the most expensive design for the best quality.

Summing up, we see that normalization is a very powerful tool for answering conceptual queries. Many queries that would be practically impossible to answer in just the structural language, now can be programmed in a matter of minutes in OR-SML.

6.2.2 Querying independent databases and approximations

In this subsection we discuss various solutions to the problem of finding teaching assistants, given relations Employees of employees and CS1 of people teaching the course CS1. First, consider the following example:

Employees

Name	Salary
John	10K
John	15K
Mary	12K
Sally	17K

CS1

Name	Room
John	076
Jim	320
Sally	120

Assume that our query asks to compute the set TA of teaching assistants. We further assume that only TAs can teach CS1 and every TA is a university employee.

Let us recall the problems we face answering the TA query. First, the databases are inconsistent. Jim teaches CS1 and hence he is a TA and an employee, but there is no record for Jim in the Employees relation. To get rid of this anomaly, we must decide if we believe CS1 or Employees. If the former is the case, then the problem is solved by adding Jim from CS1 to Employees. However, a more intersting case is when we believe the Employees relation. Here we have two possibilities.

- The Name field is a key. This is the assumption made in Buneman et al. [31, 32]. Then the record corresponding to Jim is deleted from CS1.
- The Name field is not a key. This may cause problems if there are several anomalous records. For example, if there were two records with name Jim in CS1 but only one in Employees, then one record should be deleted from CS1, but which one? We suggest using or-sets to represent both possibilities, as this is the best knowledge that can be obtained.

Now assume there are no inconsistencies in relations. We have to find an approximation of the set of TAs, that is, we have to find people who certainly are TAs and those who could be. Again, there are two cases.

- The Name field is a key. Then all people in CS1 are TAs, and those in Employees who are not represented in CS1 could be TAs. Now, to produce an approximation, two things must be done:
  - For every entry in CS1, try to infer as much information about it as possible using Employees. In our example that means adding the Salary field. To do so, check all records in Employees consistent with a given record in CS1 and, if such a record is found, use the value of its Salary field. Inferring such additional information was called *promotion* in [31, 32].
  - For each entry of Employees, check if that entry is also represented by the CS1 relation. If it is not, then we found a possible TA.

- The Name field is not a key. Then it is impossible to determine promotion unambiguously because there could be two records in Employees with the same Name filed but different Salary fields. Our solution is to use or-sets to represent both possibilities. Then, for each possible choice of records in Employees corresponding to records in CS1 we have uniquely determined set of possible TAs.

We are going to show how some of the operations described above can be done in OR-SML. First we have to define a framework for doing operations like promotion and consistency check.

As before, we assume that all records have the same fields by putting $\perp$ (null) into the missing fields. This allows us to take joins and meet of records. Notice that the join of two records is not necessarily defined.

Now we show how a query "approximate the set of TAs" can be done in OR-SML. Since Employees and CS1 are going to make either a sandwich or a mix for TA, we make Employees an or-set and CS1 a set. We now represent the data as follows:

```
- val emp = make();
<('John', ({@(10.00)}, {})), ('John', ({@(15.00)}, {})),
 ('Mary', ({@(12.00)}, {})), ('Sally', ({@(17.00)}, {}))>!
- val cs1 = create "{('John',({},{76})), ('Jim',({},{320})), ('Sally',({},{120}))}";
```

The first problem we face is getting rid of inconsistencies in the database. In our particular example, Jim is in CS1 but not in the Employees. Assuming we believe the Employees relation, we remove this anomaly as follows:

```
- fun remove_anomaly compat (R,S) =
   let fun compat_to_X (X,x) =
        Set.ormember(mkboolco(true),(orsmap (fn z => compat(z,x)) X));
   in Set.select (fn z => compat_to_X (R,z)) S end;

- val new_cs1 = remove_anomaly compatible (emp,cs1);
val new_cs1 = {('John', ({}, {76})), ('Sally', ({}, {120}))} : co
```

Here `compatible` is a function that tests whether the join of two elements is defined:

```
fun compatible (x,y) = neg(eq(join(x,y),orempty));
```

Now, consider the solution proposed by Buneman et al. [31, 32]. Given an element $x \in$ CS1, let $y_1, \ldots, y_n$ be those elements in Employees that can be joined with x. Then $x' = \bigwedge_i (x \vee y_i)$ was called a *promotion* of x. (Intuitively, the promotion of x adds all information about x from

Employees.) The solution was to take all promotions of elements in CS1 as "sure TAs" and elements of Employees not consistent with those promotions as "possible TAs". However, this solution was contingent upon the condition that the name field is a key. With this condition, we can easily program the solution of [31, 32] using a function `promote` and a new relation `emp1`:

```
- fun promote compat (R,S) =
   let fun compat_to_x (X,x) = Set.orselect (fn z => compat(z,x)) X
   in alpha (smap (fn z => big_meet (orflat(orsmap (fn v => join(z,v))
                                           (compat_to_x (R,z)))))
                  S) end;
- val emp1 = make();
<('John', ({@(10.00)}, {})), ('Mary',({@(12.00)}, {})), ('Sally', ({@(17.00)}, {}))>!

- val promoted_cs1 = promote compatible (emp1,new_cs1);
val promoted_cs1 = <{('John', ({@(10.0)}, {76})), ('Sally', ({@(17.0)}, {120}))}> : co
```

Here `big_meet` calculates the meet of a family of objects. Observe that this operation corresponds precisely to forcing a sandwich into a mix using the assumption about keys.

Now it is possible to separate sure TAs from possible TAs:

```
fun divide compat (R,S) = let
    fun compat_to_set (X,x) = member(mkboolco(true),
                                   (smap (fn z => compat(z,x)) X))
 in (orselect (fn z => neg(compat_to_set (S,z))) R, S) end;

fun divide_all compat (R,S) = orsmap (fn z => mkprodco(
                                        divide compat (p1(z),p2(z))))
                                     (orpairwith(R,S));

- val res = divide_all compatible (emp1,promoted_cs1);
val res = <(<('Mary', ({@(12.0)}, {}))>,
    {('John', ({@(10.0)}, {76})), ('Sally', ({@(17.0)}, {120}))})> : co
```

Therefore, John from office 76 and with salary 10K and Sally from office 120 and with salary 17K are *definitely* TAs and Mary with salary 12K and not known office *may be* a TA.

However, if the name field is not a key, this solution will not work. For example, both Johns from Employees will be joined with John from CS1, and when the meet is taken, the salary field is lost. But this is not what the information in the database tells us. We know that one John from Employees teaches CS1, but we do not know which John. Since either could be, the solution is to use an *or-set* to represent this situation. In particular, we take all possible joins $x \vee y_1, \ldots, x \vee y_n$ and make them into an or-set, which now plays the role of the promotion of x.

Then, taking the or-set brackets outside, we obtain the or-set with all possible answers to the TA query.

```
fun solution compat (R,S) = let fun get_R_a a = orselect (fn z => compat(z,a)) R
  in orpairwith(R, alpha(smap get_R_a S)) end;

val solution = fn : (co * co -> co) -> co * co -> co
- val result = solution compatible (emp, new_cs1);
val result =
  <(<('John', ({@(10.0)}, {})), ('Mary', ({@(12.0)}, {})),
     ('John', ({@(15.0)}, {})), ('Sally', ({@(17.0)}, {}))>,
    {('John', ({@(10.0)}, {})), ('Sally', ({@(17.0)}, {}))}),
   (<('John', ({@(10.0)}, {})), ('Mary', ({@(12.0)}, {})),
     ('John', ({@(15.0)}, {})), ('Sally', ({@(17.0)}, {}))>,
    {('John', ({@(15.0)}, {})), ('Sally', ({@(17.0)}, {}))})> : co
```

We now see that there are two possible answers to the TA query: both say that Mary could be a TA and that Sally is a TA, and one says that John making 10K is a TA while the other says that John making 15K is a TA.

Summing up, we have seen that one of the canonical problems of querying independent databases can be solved by OR-SML. Moreover, using or-sets gives us the correct answer even if the key constraints do not hold, something that the solution of Buneman et al. [31, 32] falls short of doing.

As the final example, we demonstrate the implementation of mixes as a new OR-SML datatype, as was suggested in chapter 5. The operations we have on mixes are the monad operation *mix_ext*, operations of the mix algebra, and type inference. That is, to implement mixes, we create a structure `MIX` of signature `MIXSIG` following the description of *mix_ext* given in chapter 5. These signature and structure are shown in figure 6.4.

Using `mix_ext`, it is possible to implement monad operations like `map_mix` and `flat_mix` as follows:

```
- local open MIX in
    fun map_mix f = mix_ext (fn x => mix_sng(f x))
    val flat_mix = mix_ext  (fn x => ((p1(x),p2(x)):mix))
end
val map_mix = fn : (co -> co) -> MIX.mix -> MIX.mix
val flat_mix = fn : MIX.mix -> MIX.mix
```

The following simple example shows how mixes can be created and manipulated. We assume that three complex objects `a`, `b` and `c` which are respectively 1,2 and 3, are given. Then we show

```
signature MIXSIG =
  sig
    type mix
    val mix_sng : co -> mix
    val mix_plus : mix * mix -> mix
    val mix_box : mix -> mix
    val mix_ext : (co -> mix) -> mix -> mix
    val typeof_mix : mix -> unit
  end
```

```
structure MIX = struct
   type mix = co * co
   fun mix_sng x = ((orsng(x), sng(x)):mix)
   fun mix_plus ((x:mix),(y:mix)) = let val (x1,x2) = x
                            val (y1,y2) = y
                        in
                           ((orset_min(orunion(x1,y1)),
                            set_max  (union  (x2,y2))):mix)
                        end
   fun mix_box (x:mix) = let val (x1,_) = x in ((x1,empty):mix) end
   fun mix_ext (f : co -> mix) =
                    (fn (MX:mix) =>
                       let val (U,L) = MX
                           val FIRST = orsmap (fn v =>
                                                  let val (v1,_) = f v
                                                  in v1 end)
                                               U
                           val SECOND = smap  (fn v =>
                                                  let val (_,v2) = f v
                                                  in v2 end)
                                               L
                    in
                       ((orset_min(orflat FIRST),
                        set_max  (flat  SECOND)):mix)
                    end)
   fun typeof_mix (x:mix) = let val (x1,_) = x
                                val tx = tp_print(typeof x1)
                                val tp = substring (tx,1,size(tx)-2)
                            in print (tp^" mix\n\n") end

   end;
```

Figure 6.4: Implementation of mixes in OR-SML

how the function, that for any object n creates a mix encoded as $(\langle n+1, n+2\rangle, \{n+1\})$, can be extended to a mix over integers by means of `mix_ext`.

```
- val big = mix_plus(mix_sng(a),mix_plus(mix_sng(b),mix_sng(c)));
val big = (<1, 2, 3>,{1, 2, 3}) : mix
- val small = mix_plus(mix_sng(a),mix_sng(b));
val small = (<1, 2>,{1, 2}) : mix
- val newmix = mix_plus(small, mix_box(big));
val newmix = (<1, 2, 3>,{1, 2}) : mix
- map_mix intaddone newmix;
val it = (<2, 3, 4>,{2, 3}) : mix
- fun f x = mix_plus(
               mix_box(
                  mix_plus(mix_sng(intaddone(x)),
                           mix_sng(intaddone(intaddone(x))))),
                mix_sng(intaddone(x)));
val f = fn : co -> mix
- mix_ext f newmix;
val it = (<2, 3, 4, 5>,{2, 3}) : mix
- typeof_mix newmix;
int mix
```

This shows that OR-SML is capable of supporting operations on approximations arising from their universality properties, as well as some nontraditional operations like promotion and removing anomalies. Such operations may often occur in real life applications. This further confirms that *or-$\mathcal{NRL}$* (and hence OR-SML) has adequate power to program with approximations, and is in fact a good candidate for a language for solving problems like querying independent databases.

Chapter 7

Conclusion and further research

7.1 Brief summary

We started this thesis with a survey of the field of databases with partial information and finally arrived to a point where we had a well thought out language for partial information. The main tool was using new techniques to understand the semantics of partiality.

In chapter 1 we formulated two main principles of our approach: *partiality of data is represented via orderings on values* and *semantics suggests programming constructs.* In chapter 3 we made a first step toward applying these principles to the study of databases with partial information. First, a general order-theoretic model of partial information was developed. Second, we presented an approach to design of query languages based on the universality properties of the semantic domains corresponding to the type constructors. In chapter 4 we studied the semantics of various kinds of partial information and proved the universality properties. In chapter 5 we used those universality properties to design and study languages for partial information. Finally, in chapter 6 we described an implementation of a query language based on these ideas.

Before we discuss open problems, let us briefly recall the main contributions of this thesis.

- We have surveyed the field of partial information in databases and analyzed structures and techniques used for studying partial information. We have concluded that there are no adequate analytical and algebraic tools available for the study of partial information.

- We have suggested a new approach to the study of partial information based on two main premises. One says that the concept of being more informative is represented as an ordering on objects. The other says that the right programming constructs should be derived from the mathematical properties of the semantics of partial data.

- We have extended the approach of Buneman, Jung and Ohori [33] that treats database objects as elements of domains. In particular, it was shown how schemes can be defined and how multivalued dependencies and decompositions are related in such a generalized setting.
- We have described the approach to the language design based on turning universality properties of collections into programming syntax. We have introduced new tools for analyzing expressibility of such languages and explained the difference between using sets and bags (multisets).
- Two levels of manipulating or-sets – structural and conceptual – were clearly distinguished.
- We have shown how all known approximation constructs arise in the problem of querying independent databases. Based on the analysis of the models of approximations, we suggested a new classification of those.
- We have used the "update" semantics to define orderings for five kinds of collections: sets under OWA and CWA, bags under OWA and CWA and or-sets. Orderings for sets under OWA and CWA and or-sets are the Hoare, the Plotkin and the Smyth orderings respectively.
- Based on the orderings for collections, we have defined their semantics. For objects involving or-sets we have given both structural and conceptual semantics. We have shown that the semantic domains of collections have the universality properties.
- For the first time, an isomorphism between the iterated powerdomains (Smyth and Hoare) has been explicitly constructed. This isomorphism has given us a primitive to include into the structural language for sets and or-sets to provide interaction between sets and or-sets. It has also been proved that the iterated construction possesses a universality property.
- Semantics for approximations has been given and the orderings have been determined using the update approach. From this it has been concluded how approximations can be modeled with sets and or-sets.
- Most constructions used in approximations have been characterized as free algebras. That is, they all possess universality properties which allow to incorporate them into a programming language. Some of them have been shown *not* to arise as free algebras. However, for those approximations it is possible to obtain restricted universality properties.
- Languages for collections based on their universality properties have been defined. The languages arising from the ordered semantics were shown to be sublanguages of the languages arising from the set theoretic semantics. However, well-definedness of functions on ordered objects within the languages based on the set theoretic semantics turned out to be undecidable.
- It was proved that the orderings for bags under both OWA and CWA are *not* definable in the standard bag language $\mathcal{BQL}$.

- The language *or-$\mathcal{NRL}$* based on combining sets and or-sets and using the isomorphism between the iterated powerdomains has been introduced. *or-$\mathcal{NRL}$* was shown to contain some known languages for partial information as sublanguages.

- The normalization theorem for *or-$\mathcal{NRL}$* has been proved for both set theoretic and ordered semantics. That is, all objects normalize to the same object, no matter *how* they are normalized. The normalization construct gives us the language to query sets and or-sets at the conceptual level.

- The costs of normalization have been studied and tight upper bounds have been found.

- The partial normalization theorem for *or-$\mathcal{NRL}$* has been proved for both set theoretic and ordered semantics. That is, for properly restricted types, all objects of type t normalize to the same object of type s, no matter how they are normalized. It was shown that partial normalization may help answer conceptual queries faster.

- Structural recursion and monad languages have been studied for all approximations. The monad constructs have been shown to require preconditions which are generally undecidable. It also has been shown that the monad languages for approximations are sublanguages of *or-$\mathcal{NRL}$*.

- The language OR-SML based on *or-$\mathcal{NRL}$* and $\mathcal{BQL}$ has been implemented on top of Standard ML. Its applications in querying incomplete and independent databases have been shown.

7.2 Problems for further investigation

In this section we outline some problems that must be further investigated. Discussion of some of them is quite speculative as the field is new and many areas have not been looked into at all. However, we show a number of very concrete problems that should be solvable using techniques developed in this thesis. The problems are given in no particular order.

Bags, aggregate functions and partial information

Most theoretical results in the field of databases deal with sets, whereas most practical implementations use bags as the underlying model. It has not been until just a few years ago that people started paying attention to theoretical problems arising in the study of databases that use multisets. Albert [14] proposed a number of operations for bags and studied some of their properties. Grumbach and Milo [60] introduced a bag algebra and proved some complexity results. At the same time, Chaudhuri and Vardi [38] showed that many optimization principles do not carry over from sets to bags.

Incorporating aggregate functions into relational languages was also studied by Klug [92] and Ozsoyoglu et al. [129] who introduced aggregate functions by defining them separately for each column of a relation. An alternative approach using a technique called hiding was used by Klausner and Goodman [91]. Both approaches are rather clumsy and do not show any clear connection between bags and aggregate functions.

Finally, in Libkin and Wong [105, 108] it was proved that in terms of expressive power adding bags is precisely adding aggregate function; see also theorem 3.26. However, very little is know about expressibility of languages with aggregate function. For example, Consens and Mendelzon [42] showed inexpressibility of transitive closure assuming separation of complexity classes, and Mumick and Shmueli [120] gave a rather involved argument to show that certain recursive query is not definable in a language with a limited number of aggregate functions.

If we could only show that the bounded degree property, proved in section 3.2 for $\mathcal{NRL}$, also holds for $\mathcal{BQL}$, many results on expressive power would follow immediately. We believe that the bounded degree property does hold for $\mathcal{BQL}$, but proving this remains open. The main reason this problem seems to be hard is that there is no logic capturing $\mathcal{BQL}$ or its flat fragment. Many traditional languages for databases do not produce new values (are *internal* in terminology of Hull [76]), but this is certainly not the case for $\mathcal{BQL}$ which is translated into a language with aggregates and hence can produce new values. Finding logics that capture such languages is a difficult task. For example, the logic with counting quantifiers [85] does not have enough "generating ability" to capture $\mathcal{BQL}$. And results like the bounded degree property are proved by using locality properties which in turn are based on the quantifier elimination procedure.

Very little is known about interaction of partial information and bags or aggregate functions. In this thesis we were able to define orderings on bags and, using certain results about expressive power of $\mathcal{BQL}$, showed that it can not define the orderings. This leads to a number of questions. What is the minimal "natural" set of operations that can be added to $\mathcal{BQL}$ to enable it to define the orderings? What are the corresponding operations in the set language with aggregate functions? What is a natural interpretation of orderings on bags when they are translated into the set language? In other words, how partial information interacts with aggregate functions? How aggregate functions are evaluated on partial data? Although there are a number of ad-hoc solutions in practical languages, there has been no systematic study of these problems.

Another set of intersting questions arises when one studies the ability to calculate by using bags. We showed that three different bag languages can express classes of extended polynomials, elementary and primitive recursive functions. It can also be shown that there is a correspondence between slightly enchanced versions of $\mathcal{BQL}$ and small classes of primitive recursive functions like $\mathcal{E}^1$ and $\mathcal{E}^2$ (see Rose [150] for the definition.) It is not known what orderings on bags give us in terms of the arithmetic power.

Our ordering for bags is closely connected with the ordering used by Pollard and Moshier [139] in linguistic applications. This connection could be worth studying.

Sets under the closed world assumption

Most results in chapter 5 were proved for sets under the open world assumption. Which results remain true if we switch to the closed world assumption? We saw that CWA sets can be represented in *or*-$\mathcal{NRL}_a$ by simply keeping both maximal and minimal elements, and therefore all operations arising from the CWA set monad can be expressed. But the interaction between CWA and or-sets has not been studied. What is the right primitive that provides such an interaction? It must be an analog of α, but we do not know if there is a commutativity result for the Smyth and Plotkin powerdomains. So, one of the questions is the following. Is there an analog of theorem 4.21 that relates the iterated Smyth and Plotkin constructions?

If there is such an analog, and if it can be converted into a programming primitive, can we recover the normalization theorem? If yes, is it possible to represent such a normalization in *or*-$\mathcal{NRL}_a$? If not, what is the main problem and is there a way around it?

Recursive types and values

The complex object data model, which was the main object of study in this thesis, usually serves as the underlying model for object-oriented databases. But object-oriented databases include more than that. In particular, they often deal with recursive values. That is, objects can be defined recursively. In many models this is achieved by introducing objects identifiers, see Abiteboul and Kanellakis [7]. In practice, these are implemented as pointers. However, the formal semantics of recursive types and values, and in particular recursive types and values in the presence of partial information, must be worked out.

Since semantics of recursive types is usually obtained as a limit construction, this suggests using domain instead of arbitrary posets. Assume that we add the recursive type constructor to the type system:

$$t := x \mid b \mid unit \mid t \times t \mid \{t\} \mid \mu x.t$$

where x ranges over type variables, and $\mu x.t$ is a recursive type constructor (x must be free in t.) A similar type system was considered, for example, in Lamersdorf [93] in the context of a simple language, but no semantics was given. How do we define the semantics of these types?

Since semantics of recursive types is usually obtained as a solution to an equation, which in turn is a (co)limit in some category, we have to switch to categories of domains from categories of posets. It was suggested by Gunter [65] that one formulate a number of requirements on the category of domains in which the semantics of types is to be found. In [65] such conditions were given for categories suitable for giving semantics of types used in functional languages. However, [65] did not consider the set type constructor.

Now, following Gunter [65], let us try to formulate a number of requirements on the category of domains $\mathbf{C}$ that is suitable for giving semantics of recursive complex object types. First of all, its objects must be closed under $\times$ (product type) and $\wp^\flat(\cdot)$ which is $\mathsf{Idl}(\mathcal{P}^\flat(\mathbf{K}\cdot))$, the ideal completion of $\mathcal{P}^\flat(\mathbf{K}\cdot)$. Second, it must contain the domains of base types (which are usually flat domains). Third, domain equations of form $D = \mathbb{F}(D)$, where $\mathbb{F}$ is a functor composed from the constant base type functors, products and $\wp^\flat(\cdot)$, must have a solution in $\mathbf{C}$. This guarantees that the semantics of recursive types can still be found in $\mathbf{C}$.

Of course the category $\mathbf{SFP}$ and even the category of Scott domains satisfy these requirements. But these categories contain too many domains that never arise as domains of types. Recall that we interpret compact elements as objects that can actually be stored in a database. If we have an object x that can be stored and an object y that is less informative than x, then, provided or-sets are not used, it must be possible to store y is a database. In other words, domains D which are objects of $\mathbf{C}$ must satisfy the following condition: $\downarrow\mathbf{K}D = \mathbf{K}D$. This is precisely the condition that enabled us to define schemes at the level of compact elements, see proposition 3.7.

Now we formulate the requirements on the categories $\mathbf{C}$ for database semantics.

1. All objects of $\mathbf{C}$ must be domains satisfying $\downarrow\mathbf{K}D = \mathbf{K}D$.
2. $\mathbf{C}$ must contain flat domains and be closed under $\times$ and $\wp^\flat(\cdot)$.
3. Any equation $\mathcal{D} = \mathbb{F}(\mathcal{D})$ must have a solution in $\mathbf{C}$ where $\mathbb{F}$ is an endofunctor on $\mathbf{C}$ built from constant base type functors by using $\times$ and $\wp^\flat(\cdot)$.

As the first attempt we could consider $\mathbf{C}_1$ that consists precisely of Scott domains satisfying $\downarrow\mathbf{K}D = \mathbf{K}D$. But this category does not satisfy 2). It is known that the decreasing chain condition is preserved by $\wp^\flat(\cdot)$ [22]. However, $\mathbf{C}_2$ that contains domains in which $\downarrow x$ satisfies the decreasing chain condition for any $x \in \mathbf{K}D$, does not satisfy 1. Now, take $\mathbf{C}_3$ in which objects are those domains which are objects in both $\mathbf{C}_1$ and $\mathbf{C}_2$. That is, domains in which $\downarrow x$ does not have infinite chains for any $x \in \mathbf{K}D$. Even restricting this, we take $\mathbf{C}_4$ to be the category of I-domains which satisfy the condition that $\downarrow x$ is finite for any $x \in \mathbf{K}D$. Now it is possible to prove that $\mathbf{C}_3$ and $\mathbf{C}_4$ satisfy conditions 1, 2 and 3, and so do their full subcategories given by distributive domains, and subcategories thereof in which morphisms carry compact elements to compact elements, see Libkin [101]. Moreover, the category of dI-domains (distributive domains satisfying the property I) and stable maps (preserving infima of bounded pairs) also satisfies conditions 1, 2 and 3 [101].

So, we have a number of categories in which semantics of recursive complex object types can be found. But this is not the end of the story, because there are two major issues that must be addressed. First, condition 1 is not longer satisfied if we add the or-set type constructor. Or-sets correspond to the Smyth powerdomain $\wp^\sharp(\cdot) = \mathsf{Idl}(\mathcal{P}^\sharp(\mathbf{K}\cdot))$ which does not preserve even the decreasing chain condition. Hence, condition 1 must be replaced by another condition for or-sets. The search for such a condition continues.

All recursive database objects have finite representation and could be stored in a database. But we can easily see that they are not necessarily compact elements in the domains of their types. For example, consider $\mu x.string \times x$. Its elements are infinite sequences of strings, and compact elements are those in which almost all entries are $\perp_{string}$. We can think of this type as, for example, *type person = [Name:string, spouse:person]*. Its elements certainly have finite representation, but are not compact elements of the domain of *person*. Therefore, we need to identify elements of the domains which have a finite representation. This identification must be done order-theoretically. Similar problems have been studied by Ohori [123, 125] but he considered the model based on the regular trees [44]. Such a model does not seem to be suitable for dealing with partial information, whereas using the domain based model is well justified.

Therefore, a proper definition of elements having a finite representation and identification of elements of solutions of recursive domain equations having finite representations remain open problems. We believe that progress towards solving these problems will suggests the right operations to be used for programming with recursive complex objects.

Types and schemas

Hull [77] studied connections between database schemas and complex objects in the type system that includes variant types but does not include or-sets. He defined a number of reductions that are similar to the rewrite rules applied to or-types. These reductions were shown to form a Church-Rosser rewrite system, and hence each database schema had a unique normal form.

If we consider variants as two-element or-sets (similarly pairs can be considered as two element sets), then all rewrites in Hull [77] will becomes rewrites in our system for or-types. But our analysis of the rewrite system is much deeper than just establishing Church-Rosserness. In particular, we characterized the rewrite system in terms of the partial order $\lhd$ on types and gave an efficient algorithm that tests this order. Therefore, one might expect that our analysis of the rewrite system for types may help gain a better understanding of transformations of database schemas. For example, it may help produce efficient algorithms that check if one schema could be transformed into another.

Constraints and partial information

In this thesis we developed type systems and languages for databases with partial information, but did not cover a very important area of constraints. Relatively little is known about constraints in relational databases with nulls (see [17, 18, 62, 74, 97, 131, 166]) and virtually nothing is known about constraints for other kinds of partial information. To the best of our knowledge, no work has been done on understanding how the ordering interacts with constraints.

An idea that proved to be useful for relational databases with the **ni** nulls is to introduce analogs of some constrains in a "disjunctive" manner, see Atzeni and Morfuni [17] and Thalheim [165]. Following Thalheim [165], we consider keys. In a usual relational database, a set K of attributes is a key if $\pi_K(t_1) \neq \pi_K(t_2)$ for any two distinct tuples t_1 and t_2. A family $\mathcal{K} = \{K_1, \ldots, K_n\}$ of sets of attributes is called a *key set* [165] if for any two distinct tuples t_1 and t_2, there exists a $K_i \in \mathcal{K}$ such that t_1 and t_2 are defined on K_i (that is, none of the K_i-values is **ni**) and $\pi_{K_i}(t_1) \neq \pi_{K_i}(t_2)$. For relations without null values this simply means that $\bigcup \mathcal{K}$ is a key. A key set is minimal if all K_is are singletons. The disjunctive nature of such constraints matches the usual key constrains in the closed world semantics.

Proposition 7.1 *For any relation R with* **ni** *null values and a set K of attributes, $\mathcal{K} = \{\{k\} \mid k \in K\}$ is a minimal key set iff $\pi_{K \cap \mathrm{def}(t,t')}(t) = \pi_{K \cap \mathrm{def}(t,t')}(t')$ implies $t = t'$, where $\mathrm{def}(t,t')$ is the set of attributes on which both t and t' are defined. Furthermore, this implies that for any $T \in [\![R]\!]^{\mathrm{CWA}}_{\max}$ with card $T \leq$ card R, K is a key of T.* $\square$

The converse to the last statement is not true. Consider $R = \{(\mathbf{ni}, 1), (2, 1)\}$. Then for any T as in the statement of the proposition, the first attribute is a key, but it is not a key set for R.

We believe that this idea of making one constraint into a family while maintaining a close connection with the intended semantics can be quite productive. The concept of a key set can be reformulated as $\forall t, t' \; \forall K \in \mathcal{K} : (K \subseteq \mathrm{def}(t,t') \Rightarrow \pi_K(t) = \pi_K(t')) \;\Rightarrow\; t = t'$. This in turn implies that $\bigcup \mathcal{K}$ is a key for any $T \in [\![R]\!]^{\mathrm{CWA}}_{\max}$ and shows that keys can be further generalized to functional dependencies and probably a to greater class of dependencies given in a first order language with equality.

Let us give a simple example to illustrate some of the problems arising from using other nulls. Consider a simple relation

Name	Dept	Room
ne	**ne**	76

We interpret this relation as saying that room 76 does not belong to any department and is empty. Now, consider a different relation:

Name	Dept	Room
ne	**ne**	76
Joe	CS	76
Jim	**ne**	76

This relation says that there is one room 76 which does not have people in it and does not belong to any department, and there is another room, also named 76, that belongs to CS and Joe sits in it. Moreover, there is yet another room 76 which does not belong to any department but has someone named Jim in it.

Of course we can not represent the second database as a first order theory as in Reiter [143], because it would yield a contradiction: $P(\text{Joe}, \text{CS}, 76)$ & $\neg\exists x \neg\exists y\, P(x, y, 76)$. However, it still makes perfect sense. But now assume that there is a constraint which says that there is only one room 76. While having two records $\boxed{\text{Joe} \mid \text{CS} \mid 76}$ and $\boxed{\text{Ann} \mid \text{Math} \mid 76}$ does not contadict it, having a record $\boxed{\mathbf{ne} \mid \mathbf{ne} \mid 76}$ does contradict the constraint as it would be imply the existence of two rooms 76: one with Joe and Ann in it, and one empty. Even though the **ne** null is a maximal element in the ordering and is treated in the same way as the usual nonpartial values, it does behave differently in the presence of constraints.

How could one approach the problem of dealing with constraints in databases with partial information? Since we advocate the order-theoretic models of databases and consider rather complicated type systems, we believe one should try to apply the approach that formalizes constraints independently of the particular kind of data structures involved. For example, one may use the lattice theoretic approach to dependencies and normalization developed in Demetrovics et al. [47] and Day [45] or define dependencies as certain classes of first order formulae as in Fagin [50]. One may also benefit from using these approaches since most papers dealing with constraints in the complex object model only study constraints on the top level attributes [48, 130, 169]. But at this point there is almost no understanding how constraints interact with partial information represented via orderings on objects. The area is completely open.

Genericity, computability and polymorphism

This subsection is definitely the most speculative of all. One of the important problems in database theory is identifying important classes of queries and designing languages capable of expressing those queries. There are several ways in which database query languages are different from traditional programming languages. First, most database queries are internal (see Hull [76]). That is, they only manipulate with values stored in a database and do not create new values. Second, they are *generic*. Most definitions of genericity, such as in Chandra and Harel [36], assume that there is only one domain of values and simply require that queries be invariant under permutations of such a domain. For instance, a query computing the transitive closure of a relation is such but the query returning the sum of two largest numbers stored in a database is not.

Many researchers tried to identify languages capable of expressing precisely all generic queries from a given complexity class over relational databases. A language for all computable queries was given in Chandra and Harel [36]. In Immerman [83] and Vardi [170] languages for the class PTIME were given, and Abitebouland Vianu [9] showed how to capture PSPACE. Their results use an assumption that a *linear* ordering is given on objects. The question we would like to investigate is how using *partial* order that represents incompleteness of information will affect the main definitions, like genericity, and results about capturing complexity classes. The situation when we have a partial order falls between the totally ordered case and the totally unordered case which appears to be much harder, cf. Abiteboul and Vianu [10] and Immerman and Lander [85]. Note also that we want to look at these problems in the context of typed languages, whereas in the above mentioned papers it is always assumed that only one domain of values is present.

Another interesting project is to try to make precise a rather vague idea of establishing connection between genericity and polymorphism. Genericity means that the queries are invariant under permutations of the domains, and this is very close in the spirit to the idea of parametric polymorphism. Until recently, genericity has not been considered in the context of typed database languages. In Libkin and Wong [107], a type system with type variables was studied. That is, types are given by $t := x \mid b \mid unit \mid t \times t \mid \{t\}$ where x ranges over type variables. Then the definition of genericity of a query of type $s \to t$ was reformulated, where s and t may have some type variables. That definition is much closer to various definitions of polymorphic functions and can serve as a good starting point.

Note that in all our languages (even including the SML implementation of *or-*$\mathcal{NRL}$ that uses higher-order functions) we deal only with instances of predicative polymorphism [116]. That is, in universal types $\forall x.t$, the range of x does not involve universal types. For instance, the type of transitive closure can be viewed as $\forall x.\{x \times x\} \to \{x \times x\}$ where x ranges over object types.

The definition of genericity in Libkin and Wong [107] is set-theoretic. An intersting problem is to find out whether there exist set-theoretic models for universal types in database query languages like $\mathcal{NRL}$. We have the set type constructor, so one may expect to observe a phenomenon similar to Reynolds [146] where a power construction was used to refute the existence of set-theoretic models for universal types. On the other hand, we deal only with instances of predicative polymorphism, and it may be possible that the complications of [146] will be irrelevant.

Invariance under permutations no longer suffices as the definition of genericity if we deal with incomplete information represented via orderings on domains of object types. We need to extend the definition to accommodate orderings. This situation appears to be quite similar – at least in the spirit – to characterizing λ-definability (see Plotkin [138]): invariance under permutations is an obvious first try, but it does not work. Instead, invariance under logical relations is needed. Being invariant under logical relations is what parametric polymorphism is semantically, see Mitchell [116]. To extend the standard definitions from those suitable for languages based on λ-calculi to languages with sets, one has to lift logical relations to powerdomains and not only to function spaces. To the best of our knowledge, this has not been done, and it might be worth looking at.

Returning to the problem of invariance under permutations or logical relations, we have a number of new questions. First, one may want to describe functions expressible in the languages that we have studied as functions which are invariant and satisfy some additional conditions. This idea of course comes directly from the problem of λ-definability, since we suggest that our languages can be viewed as "canonical" languages for partial information, very much in the same way as λ-calculus is the basis for the functional programming. Conversely, one may take some class $\mathcal{C}$ of queries and search for a language that expresses exactly all invariant queries in $\mathcal{C}$. Observe that if $\mathcal{C}$ is a complexity class, then this is the problem of capturing such a class that was discussed a few paragraphs ago. There is an indication that this problem may be very hard for important classes like PTIME, with or without presence of partial information.

Using ordered semantics we have advocated can be helpful in finding models of universal types involving sets. There exist domain-theoretic models of polymorphism. An interesting project would be to extend the model of Coquand et al. [43] based on Grothendieck fibrations to include the set and or-set type constructors. From the results of this thesis we know what the corresponding domain constructions are: they are the Hoare and the Smyth powerdomains. It would be worth checking if the results of Coquand et al. [43] carry over to these powerdomains.

Since we deal mostly with predicative polymorphism, there is hope that many complications of the impredicative polymorphism will not show up, and carrying out the project of understanding genericity as polymorphism of functions in typed languages with sets will be possible. Of course the most important outcome of this project would be having the database community speak of polymorphic functions rather than generic queries.

Formal models of approximations

The theory of approximation in databases started just a few years ago and there are many topics to be investigated. First, the algebraic characterization given in this thesis points out to an intimate connection between these constructions and various algebras with idempotent binary operations that have been extensively studied, most notably by Romanowska and Smith, see [61, 149, 148, 147]. In [148] they characterized freely generated meet-distributive bisemilattices, that is, bisemilattices satisfying only one distributive law. In [147] idempotent semirings with semilattice reducts are characterized. These algebras are closely related to the scone algebras.

Algebras corresponding to three kinds of approximations (or absence thereof) have not been discovered yet. Even though we showed that using structural recursion and monads based on the universality properties of approximations is not the right approach to program with them, finding such characterization is still helpful as it would allow us to extend theorem 5.32 to include all ten constructions.

Another open problem is applying Abramsky's approach [11] that finds logical theories corresponding to various constructions on domains. For mixes this was done by Gunter [66]. Recently, some progress has been made in Darmstadt in applying Abramsky's approach to snacks. It may also be interesting to see what, if any, are the connections between our work and recent work by Chaudhuri and Kolaitis [37] on approximating recursive datalog programs with nonrecursive ones.

How to answer conceptual queries faster?

We suggested using normalization as a means of answering conceptual queries and demonstrated its usefulness. However, we showed that normalization can be quite expensive. Hence, one has to look for ways to normalize faster.

We considered one approach to the problem. Often it is not necessary to normalize all way to the normal form to answer a query. We proved a partial normalization result saying that for types without occurrences of subtypes $\langle\langle t\rangle\rangle$, an analog of the normalization theorem holds. Hence, for such types it is possible to do partial normalization. Even though in all examples we have encountered there were no occurrences of types of form $\langle\langle t\rangle\rangle$ other than at the intermediate stages of the rewriting, we believe that it is still possible to improve the partial normalization theorem by extending it to a larger class of types.

Even more importantly is to combine partial normalization with a smart evaluation strategy. Most queries asked against normal forms are existential queries. That is, the queries asking if there is a possibility in the normal form satisfying certain properties. Presently, the normalization process computes all possibilities and then outputs them. The evaluation strategy we need should evaluate normalization lazily. That is, it should try to produce an element of a normal form, check if it satisfies a given property and then go on. This kind of optimization that produces the first answer fast was considered by Wong [180] for his implementation of a language based on $\mathcal{NRL}$. In addition to using such optimizations, it would be desirable if query evaluation algorithm tried to use some heuristics that would help produce an answer satisfying the given condition faster.

We said that or-objects are typically present in the problems arising in design and planning areas, and in particular in computer aided designs. Such objects are usually very large, and it is necessary to combine all possible ways to speed up the query evaluation process. One step of this process – the partial normalization – has been developed in this thesis. Devising a smart query evaluation algorithm is an important open problem.

New features of OR-SML

There are a number improvements in the implementation of OR-SML that could be made. First of all, real records must be added. (Now they are simulated with pairs.) A proper set of operations on records should be identified and some operations of the language, such as normalization, must be reprogrammed. From the definition of normalization it can be seen that record concatenation should become a new primitive operation. Therefore, we shall need to add new tools for representing records and computing with them to the existing OR-SML implementation. There are a number of known techniques for doing this, such as in Ohori [126] and Rémy [145].

At this moment null values can only be added to the user-defined base types. Therefore, OR-SML needs a way for the user to specify null values for already existing types and to define an order on them. Finally, using new tools such as the "visible compiler" of Appel and MacQueen [15], the system could be made much more user-friendly.

However, we believe that these changes to the existing implementation should not be made before many questions related to bags, closed world sets and recursive types are clarified, because they may cause additional changes. Only those changes that will for sure remain in the language capable of working with recursive types, bags and closed world sets, could be made at this stage.

Bibliography

[1] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. In *Proc. Int. Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, 1988.

[2] S. Abiteboul, C. Beeri, M. Gyssens and D. Van Gucht. An introduction to the completeness of languages for complex objects and nested relations. In [4], pages 117–138.

[3] S. Abiteboul and N. Bidoit. Non first normal form relations: an algebra allowing data restructuring. *Journal of Computer and System Sciences* 33 (1986), 361–393.

[4] S. Abiteboul, P.C. Fischer and H.-J. Schek, editors. *"Nested Relations and Complex Objects"*. Springer LNCS 361, Springer Verlag, 1989.

[5] S. Abiteboul and G. Grahne. Update semantics for incomplete databases. In *Proc. Very Large Databases* (1985), 1-12

[6] S. Abiteboul and S. Grumbach. COL: a logical based language for complex objects. in *"Advances in Database Programming Languages"* (F. Bancilhon and P. Buneman, eds.), ACM Press, 1990, pages 347–374.

[7] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *SIGMOD 89*, pages 159–173.

[8] S. Abiteboul, P. Kanellakis and G. Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science* 78 (1991), 159–187.

[9] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences* 43 (1991), 62–124.

[10] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings of ACM Symp. on the Theory of Computing*, 1991.

[11] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic* 51 (1991), 1–77.

[12] S. Abramsky and A. Jung. Domain Theory. Chapter in Volume 3 of the *"Handbook of Logic in Computer Science"*, Cambridge University Press, 1994.

[13] A. Aho, R. Sethi, and J. Ullman. *"Compilers: Principles, Techniques and Tools"*. Addison Wesley, 1985.

[14] J. Albert. Algebraic properties of bag data types. In *Proceedings of Very Large Databases–91*, pages 211–219.

[15] A. Appel and D. MacQueen. Separate compilation for Standard ML. In *Proceedings of the SIGPLAN '94 Conf. on Programming Language Design and Implementation.*

[16] M. Atkinson, P. Richard and P. Trinder. Bulk types for large scale programming. In *Next Generation Information System Technology*, Springer LNCS 504, Springer Verlag, 1990, pages 228-250.

[17] P. Atzeni and N. Morfuni. Functional dependencies and constraints on null values in database relations. *Information and Control*, 70 (1986), 1-31.

[18] P. Atzeni and M. De Bernardis. A new basis for the weak instance model. In *PODS-87*, pages 79–86.

[19] R. Balbes. A representation theorem for distributive quasilattices. *Fundamenta Mathematicae* 68 (1970), 207-214.

[20] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *PODS 1986*, pages 53–59.

[21] M. Barr and C. Wells. *"Category Theory for Computing Science"*. Prentice Hall, 1990.

[22] G. Birkhoff. *"Lattice Theory"*. 3rd ed, Amer. Math. Soc., 1967.

[23] J. Biskup. A formal approach to null values in database relations. In: *"Advances in Data Base Theory"*, Volume 1, Prenum Press, New York, 1981.

[24] S. Bloom. Varieties of ordered algebras. *Journal of Computer and System Sciences* 13 (1976), 200–212.

[25] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proc. of 3rd Int. Workshop on Database Programming Languages*, pages 9–19, Naphlion, Greece, August 1991.

[26] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *LNCS 646: Proc. ICDT, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 92.

[27] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *LNCS 510: Proc. of 18th ICALP, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.

[28] D. Bronshtein. *"The chess struggle in practice: candidates tournament, Zurich 1953"*. D. McKay Co., New York, 1978.

[29] S. Brookes, K. Van Stone. Monads and comonads in intensional semantics. Technical Report CMU-CS-93-140, Carnegie Mellon University, April 1993.

[30] P. Buneman. Functional programming and databases. In *"Research Topics in Functional Programming"*, (D. Turner ed), Addison-Wesley, 1990, pages 155–169.

[31] P. Buneman, S. Davidson and A. Watters. A semantics for complex objects and approximate answers. *Journal of Computer and System Sciences* 43(1991), 170–218.

[32] P. Buneman, S. Davidson and A. Watters. Querying independent databases. *Information Science*, 46 (1988), 1–34.

[33] P. Buneman, A. Jung, A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science* 91(1991), 23–55.

[34] P. Buneman, L. Libkin, D. Suciu, V. Tannen and L. Wong. Comprehension syntax. *SIGMOD Record*, 23 (1994), 87–96.

[35] L. Cardelli. Types for data-oriented languages. In *Proceedings of EDBT-88* (J.W. Schmidt, S. Ceri and M. Missikoff eds), Springer Lecture Notes in Computer Science, vol. 303, Springer Verlag, 1988.

[36] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25 (1982), 99–128.

[37] S. Chaudhuri and Ph. Kolaitis. Can Datalog be approximated? In *Proceedings of the 13th Conference on Principles of Database Systems,* Minneapolis MN, May 1994, pages 86–96.

[38] S. Chaudhuri and M. Vardi. Optimization of *real* conjunctive queries. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 59–70, Washington, D. C., May 1993.

[39] E.F. Codd. Understanding relations. *Bulletin of ACM SIGMOD*, 1975, pages 23–28.

[40] E.F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Systems* 4 (1979), 397–434.

[41] L. Colby. A recursive algebra for nested relations. *Information Systems* 15 (1990), 567–582.

[42] M. Consens and A. Mendelzon. Low complexity aggregation in GraphLog and Datalog. *Theoretical Computer Science* 116 (1993), 95–116.

[43] T. Coquand, C. Gunter and G. Winskel. Domain theoretic models of polymorphism. *Information and Computation* 81 (1989), 123–167.

[44] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science* 25 (1983), 95–169.

[45] A. Day. The lattice theory of functional dependencies and normal decompositions. *Intern. J. of Algebra and Computation* 2 (1992), 409–431.

[46] J. Demetrovics. private communication.

[47] J. Demetrovics, L. Libkin and I. Muchnik. Functional dependencies in relational databases : a lattice point of view. *Discrete Applied Mathematics* 40 (1992), 155-185.

[48] J. Demetrovics, L. Rónyai and H.N. Son. An approach for normalization, composition and decomposition of attributes. In *LNCS 646: Proc. ICDT, Berlin, Germany, October, 1992*, pages 71–85. Springer-Verlag, October 92.

[49] N. Dershowitz and J.-P. Jouannand. Rewrite Systems. Chapter 6 in *"Handbook of Theoretical Computer Science"*, North Holland, 1990, pages 243–320.

[50] R. Fagin. Horn clauses and database dependencies. *Journal of ACM* 29 (1982), 952–985.

[51] R. Fagin. Finite model theory — a personal perspective. *Theoretical Computer Science*, 116 (1993), 3–32.

[52] R. Fagin, L. Stockmeyer, and M. Vardi. On monadic NP vs monadic co-NP. In *Proceedings of 8th IEEE Conference on Structure in Complexity Theory*, pages 19–30, May 1993.

[53] K.E. Flannery and J.J. Martin. Hoare and Smyth power domain constructors commute under composition. *Journal of Computer and System Sciences* 40 (1990), 125-135.

[54] M. Furst, J. Saxe and M. Sipser. Parity, circuits and the polynomial time hierarchy. *Math. Systems Theory*, 17:13–27, 1984.

[55] H. Gaifman. On local and non-local properties. In: *Proceedings of the Herbrand Symposium*, Logic Colloquium '81, North Holland, 1982, pages 105–135.

[56] M. Garey and D. Johnson. *"Computers and Intractability : A Guide to the Theory of NP- completeness"*. San Francisco, W.H. Freeman, 1979.

[57] J.-Y. Girard. The system F of variable types : fifteen years later. *Theoretical Computer Science* 45 (1986), 159–192.

[58] J.-Y. Girard. *"Proofs and Types"*, Cambridge University Press, 1987.

[59] G. Gottlob and R. Zicari. Closed world databases opened through null values. In *Proc. Very Large Databases* (1988), 50–61.

[60] S. Grumbach and T. Milo. Towards tractable algebras for bags. *Proceedings of the 12th Conference on Principles of Database Systems*, Washington DC, 1993, pages 49–58.

[61] G. Gierz and A. Romanowska. Duality for distributive bisemilattices. *J. Austral. Math. Soc. (A)* 51 (1991), 247–275.

[62] G. Grahne. *"The Problem of Incomplete Information in Relational Databases"*. Springer-Verlag, Berlin, 1991.

[63] J. Grant. Null values in relational databases. *Information Processing Letters* 6 (1977), 156–157.

[64] G.Grätzer. *"Universal Algebra"*. Springer Verlag, 1980.

[65] C. Gunter. Comparing categories of domains. In *"Mathematical Foundations of Programming Semantics* (A. Melton ed), Springer Lecture Notes in Computer Science, vol. 239, Springer, Berlin, 1985, pages 101–121.

[66] C. Gunter. The mixed powerdomain. *Theoretical Computer Science* 103 (1992), 311–334.

[67] C. Gunter. *"Semantics of Programming Languages"*. The MIT Press, 1992.

[68] C. Gunter and D. Scott. Semantic Domains. Chapter 12 in *"Handbook of Theoretical Computer Science",* ed. J. van Leeuwen (North Holland, 1990), pages 633–674.

[69] E. Gunter and L. Libkin. OR-SML: a functional database programming language for disjunctive information and its applications. In *Proceedings of the Conference on Database and Expert Systems Applications DEXA-94*, Springer Verlag, to appear.

[70] M. Gyssens and D. Van Gucht. The powerset algebra as a natural tool to handle nested database relations. *Journal of Computer and System Sciences* 45 (1992), 76–103.

[71] R. Heckmann. Lower and upper power domain constructions commute on all cpos. *Information Processing Letters* 40 (1991), 7-11.

[72] G. G. Hillebrand, P. C. Kanellakis, and H. G. Mairson. Database query languages embedded in the typed lambda calculus. In *Proceedings of 8th IEEE Symposium on Logic in Computer Science*, Montreal, Canada, June 1993, pages 332–343.

[73] G. G. Hillebrand and P. C. Kanellakis. Functional database query languages as typed lambda calculi of fixed order. In *Proceedings of the 13th Conference on Principles of Database Systems,* Minneapolis MN, May 1994, pages 222–231.

[74] P. Honeyman. Testing satisfaction of functional dependencies. *Journal of the ACM*, 29 (1982), 668–677.

[75] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing* 4 (1973), 225–231.

[76] R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15 (1986), 865–886.

[77] R. Hull. A survey of theoretical research on typed complex database objects. In *"Databases"* (J. Paredaens ed.) Academic Press, London, 1987, pages 193–256.

[78] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of ACM* 31(1984), 761–791.

[79] T. Imielinski and W. Lipski. The relational model of data and cylindric algebras, *Journal of Computer and System Science* 28 (1984), 80–102.

[80] T. Imielinski, S. Naqvi, and K. Vadaparty. Incomplete objects — a data model for design and planning applications. In *Proc. of ACM-SIGMOD, Denver, Colorado, May 1991*, pages 288–297. Full paper submitted to ACM TODS.

[81] T. Imielinski, S. Naqvi, and K. Vadaparty. Querying design and planning databases. In *LNCS 566: Deductive and Object Oriented Databases*, pages 524–545, Berlin, 1991. Springer-Verlag.

[82] T. Imielinski and K. Vadaparty. Complexity of querying databases with or-objects. In *PODS-89*.

[83] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68 (1986), 86–104.

[84] N. Immerman. Languages that capture complexity classes. *SIAM J. Comput.* 16 (1987), 760–778.

[85] N. Immerman and E. Lander. Describing graphs: A first order approach to graph canonization. In *"Complexity Theory Retrospective"*, Springer Verlag, Berlin, 1990.

[86] N. Immerman, S. Patnaik and D. Stemple. The expressiveness of a family of finite set languages. In *Proceedings of the 10th Symposium on Principles of Database Systems*, 1991, pages 37–52.

[87] A. Jung. personal communication.

[88] A. Jung, L. Libkin and H. Puhlmann. Decomposition of domains. In: *Proceedings of the Conference on Mathematical Foundations of Programming Semantics–91*, Springer LNCS 598, Springer Verlag, Berlin, 1992, pages 235–258.

[89] P. Kanellakis. Elements of Relational Database Theory. Chapter 17 in *"Handbook of Theoretical Computer Science"*, North Holland, 1990, pages 1075–1156.

[90] M. Kifer and G. Lausen. F-Logic: a higher-order language for reasoning about objects, inheritance and scheme. In *SIGMOD 89*, pages 134–146.

[91] A. Klausner and N. Goodman. Multirelations: semantics and languages. In *Proceedings of Very Large Databases–85*, pages 251–258.

[92] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM* 29 (1982), 699–717.

[93] W. Lamersdorf. Recursively defined complex objects. In [4], pages 176–189.

[94] N. Lerat and W. Lipski. Nonapplicable nulls. *Theoretical Computer Science* 46 (1986), 67–82.

[95] M. Levene and G. Loizou. The nested relation type model: An application of domain theory to databases. *The Computer Journal* 33 (1990), 19-30.

[96] M. Levene and G. Loizou. Correction to "Null values in nested relational databases" by M. A. Roth, H. F. Korth, and A. Silberschatz. *Acta Informatica* 28 (1991), 603-605.

[97] M. Levene and G. Loizou. Semantics of null extended nested relations. *ACM Trans. Database Systems* 18 (1992), 414-459.

[98] M. Levene and G. Loizou. A fully precise null extended nested relational algebra. *Fundamenta Informaticae* 19 (1993), 303-343.

[99] L. Libkin. A relational algebra for complex objects based on partial information. In *LNCS 495: Proceedings of Symposium on Mathematical Fundamentals of Database Systems–91*, pages 36–41, Rostock, 1991. Springer-Verlag.

[100] L. Libkin. An elementary proof that upper and lower powerdomain constructions commute. *Bulletin of the EATCS*, 48 (1992), 175–177.

[101] L. Libkin. Denotational semantics for complex objects and functions on them. Unpublished notes, University of Pennsylvania, 1992.

[102] L. Libkin. A remark about algebraicity in complete partial orders. *Journal of Pure and Applied Algebra* 86 (1993), 75–77.

[103] L. Libkin. Algebraic characterization of edible powerdomains. Technical Report MS-CIS-93-70/L&C 71, University of Pennsylvania, 1993.

[104] L. Libkin and L. Wong. Semantic representations and query languages for or-sets. *Proceedings of the 12th Conference on Principles of Database Systems,* Washington, DC, May 1993, pages 37–48.

[105] L. Libkin and L. Wong. Some properties of query languages for bags. In *Proceedings of the Fourth Workshop on Database Programming Languages, Manhattan NY, August 30–September 1, 1993*, Springer Verlag, 1994, pages 97–114.

[106] L. Libkin and L. Wong. Aggregate functions, conservative extension and linear order. In *Proceedings of the Fourth Workshop on Database Programming Languages, Manhattan NY, August 30–September 1, 1993*, Springer Verlag, 1994, pages 282–294.

[107] L. Libkin and L. Wong. Conservativity of nested relational calculi with internal generic functions. *Information Processing Letters* 49 (1994), 273–280.

[108] L. Libkin and L. Wong. New techniques for studying set languages, bag languages and aggregate functions. In *Proceedings of the 13th Conference on Principles of Database Systems,* Minneapolis MN, May 1994, pages 155–166.

[109] W. Lipski. On semantic issues connected with incomplete information in databases. *ACM Trans. Database Systems* 4 (1979), 262–296.

[110] W. Lipski. On databases with incomplete information. *J. ACM* 28 (1981), 41–70.

[111] K.C. Liu and R. Sinderraman. Indefinite and maybe information in relational databases. *ACM Trans. Database Systems* 15 (1990), 1–39.

[112] S. MacLane. *"Categories for the Working Mathematician"*. Springer Verlag, 1971.

[113] D. Maier. *"The Theory of Relational Databases"*. Computer Science Press, 1983.

[114] R. Milner, M. Tofte and R. Harper. *"The Definition of Standard ML"*. The MIT Press, 1990.

[115] J. Minker, editor. *"Foundations of Deductive Databases and Logic Programming"*. M. Kaufmann Publishers, 1988.

[116] J. Mitchell. Type systems for programming languages. Chapter 8 in *"Handbook of Theoretical Computer Science"*, North Holland, 1990, pages 365–458.

[117] J. Mitchell and A. Scedrov. Notes on sconing and relators. In *Computer Science Logic-92*, Springer LNCS 702, 1993, pages 352–378.

[118] E. Moggi. Notions of computation and monads. *Information and Computation*, 93 (1991), 55–92.

[119] J. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics* 3(1965), 23–28.

[120] I. S. Mumick and O. Shmueli. How expressive if stratified aggregation. *Annals of Mathematics and Artificial Intelligence*, 1994, to appear.

[121] T.-H. Ngair. *"Convex Spaces as an Order-theoretic Basis for Problem Solving"* (PhD Thesis). Technical Report MS-CIS-92-60, University of Pennsylvania, 1992.

[122] P. Odifreddi. *"Classical Recursion Theory"*. North Holland, 1989.

[123] A. Ohori. *"A Study on Semantics, Types and Languages for Databases and Object-oriented Programming"*. PhD Thesis, University of Pennsylvania, 1989.

[124] A. Ohori. Orderings and types in databases. In *"Advances in Database Programming Languages"* (F. Bancilhon and P. Buneman, eds.), ACM Press, 1990, pages 97–116.

[125] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science* 76 (1990), 53–91.

[126] A. Ohori. A compilation method for ML-style polymorphic record calculi. In *Proc. of Symp. on Principles of Programming Languages*, 1992, pages 145–165.

[127] A. Ohori, V. Breazu-Tannen and P. Buneman. Database programming in Machiavelli: a polymorphic language with static type inference. In *SIGMOD 89*, pages 46–57.

[128] A. Ola. Relational databases with exclusive disjunctions. In *Data Engineering 92*, pages 328-336.

[129] G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12 (1987), 566–592.

[130] Z. M. Ozsoyoglu and L.-Y. Yuan. A new normal form for nested relations. *ACM Transaction on Database Systems*, 12 (1987), 111–136.

[131] J. Paredaens, P. De Bra, M. Gyssens and D. Van Gucht. *"The Structure of the Relational Data Model"*. Springer, Berlin, 1989.

[132] J. Paredaens and D. Van Gucht. Converting nested relational algebra expressions into flat algebra expressions. *ACM Transaction on Database Systems*, 17 (1992), 65–93.

[133] L.C. Paulson. *"ML for the Working Programmer"*. Cambridge University Press, 1991.

[134] J. Płonka. On distributive quasilattices. *Fundamenta Mathematicae* 60 (1967), 191–200.

[135] J. Płonka. On a method of construction of abstract algebras. *Fundamenta Mathematicae* 61 (1967), 183–189.

[136] J. Płonka. On free algebras and algebraic decompositions of algebras from some equational classes defined by regular equations. *Algebra Universalis* 1 (1971), 261–264.

[137] G. Plotkin. A powerdomain construction. *SIAM Journal of Computing* 5 (1976), 452–487.

[138] G. Plotkin. Lambda-definability in the full type hierarchy. In *"To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism"*, edited by J. Seldin and J. Hindley, Academic Press, London, 1980, pages 363–373.

[139] C. Pollard and D. Moshier. Unifying partial descriptions of sets. Manuscript, 1993.

[140] A. Poulovassilis and C. Small. A domain theoretic approach to integrating functional and logical database languages. In *Proceedings of Very Large Databases–93*, pages 416–428.

[141] H. Puhlmann. The snack powerdomain for database semantics. In *LNCS 711: Proceedings of Conference on Mathematical Foundations of Computer Science,* Gdansk, Poland, 30 August–3 September 1993, (Andrzej M. Borzyszkowski and Stefan Sokolowski, eds.), Springer Verlag, 1993, pages 650–659.

[142] R. Reiter. On closed world databases. In *"Logic and Databases"*, H. Gallaire and J. Minker eds, Plenum Press, 1978, pages 55–76.

[143] R. Reiter. Towards a logical reconstruction of relational database theory. In: *"On Conceptual Modeling"* (M. Brodie and J. Schmidt eds.), Springer Verlag, 1984, pages 163–189.

[144] R. Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *J. ACM* 33 (1986), 349–370.

[145] D. Rémy. Efficient representation of extensible records. In *ACM SIGPLAN Workshop on ML and its applications*, 1922, pages 12–16.

[146] J. Reynolds. Polymorphism is not set-theoretic. In *"Semantics of Data Types"* (G. Kahn, D. Macqueen and G. Plotkin eds), Springer Lecture Notes in Computer Science, vol. 173, Springer, Berlin, 1984, pages 145–156.

[147] A. Romanowska. Free idempotent distributive semirings with a semilattice reduct. *Math. Japonica* 27 (1982), 467–481.

[148] A. Romanowska and J.D.H. Smith. Bisemilattices of subsemilattices. *J. Algebra* 70 (1981), 78–88.

[149] A. Romanowska and J.D.H. Smith. *"Modal Theory: An Algebraic Approach to Order, Geometry and Convexity"*. Heldermann Verlag, Berlin, 1985.

[150] H. Rose. *"Subrecursion: Functions and Hierarchies"*. Clarendon Press, 1984.

[151] M.A. Roth, H.F. Korth and A. Silberschatz. Null values in nested relational databases. *Acta Informatica*, 26 (1989), 615–642.

[152] B. Rounds. Situation-theoretic aspects of databases. In *Proceedings of Conference on Situation Theory and Applications*, CSLI vol. 26, 1991, pages 229-256.

[153] H. Sakai. On a framework for logic programming with incomplete information. *Fundamenta Informaticae* 19 (1993), 223–234.

[154] V.N. Salii. *"Lattices with Unique Complements"* (AMS, Providence, RI, 1988).

[155] Y. Saraiya. Fixpoints and optimizations in a language based on structural recursion on sets. Manuscript, December 1992.

[156] H.-J. Schek and M. Scholl. The relational model with relation-valued attributes. *Information Systems* 11 (1986), 137–147.

[157] M.B. Smyth. Power domains. *Journal of Computer and System Sciences* 16 (1978), 23–36.

[158] *Standard ML of New Jersey: User's guide.* Version 0.93, February 1993. AT&T Bell Laboratories.

[159] D. Stemple and T. Sheard. A recursive base for database programming primitives. In *Next Generation Information System Technology*, Springer LNCS 504, Springer Verlag, 1990, pages 311–352.

[160] A. Stoughton. *"Fully Abstract Models of Programming Languages"*. Pitman, London, 1988.

[161] D. Suciu. Bounded fixpoints for complex objects. In *Proceedings of the Fourth Workshop on Database Programming Languages, Manhattan NY, August 30–September 1, 1993*, Springer Verlag, 1994, pages 263–281.

[162] D. Suciu and J. Paredaens. Any algorithm in the complex object algebra with powerset needs exponential space to compute transitive closure. In *Proceedings of the 13th Conference on Principles of Database Systems,* Minneapolis MN, May 1994, pages 201–109.

[163] D. Suciu and V. Tannen. A query language for NC. In *Proceedings of the 13th Conference on Principles of Database Systems,* Minneapolis MN, May 1994, pages 167–178.

[164] K. Tanaka and T.-S. Chang. On natural join in object-oriented databases. In : *Proc. of Int. Conf. on Deductive and Object-Oriented Databases.* Kyoto, December 1989.

[165] B. Thalheim. On semantic issues connected with keys in relational databases permitting null values. *J. Inf. Process. and Cybernet.*, 25(1/2):11–20, 1989.

[166] B. Thalheim. *"Dependencies in Relational Databases"*. Teubner-Texte zur Mathematik, Band 126, Stuttgart-Leipzig, 1991.

[167] S.J. Thomas and P. Fischer. Nested relational structures. In P. Kanellakis editor, *"Advances in Computing Research: The Theory of Databases"*, pages 269–307, JAI Press, 1986.

[168] J.D. Ullman. *"Principles of Database and Knowledge-Base Systems"*. Computer Science Press, 1988.

[169] D. Van Gucht and P. Fischer. Multilevel nested relational structures. *Journal of Computer and System Sciences* 36 (1988), 77–105.

[170] M. Vardi. The complexity of relational query languages. In *Proc. of ACM Symp. on the Theory of Computing*, 1982, pages 137–146.

[171] M. Vardi. On the integrity of databases with incomplete information. In *Proc. 5th ACM Symp. on Principles of Database Systems* (1986), 252–266.

[172] Y. Vassiliou. Null values in database management – a denotational semantics approach. In: *SIG-MOD 1979*, pages 162–169.

[173] Y. Vassiliou. Functional dependencies and incomplete information. In: *Very Large Databases 1980*, pages 260–269.

[174] S. Vickers. Geometric theories and databases. In P. Johnstone and A. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Notes*, pages 288–314. Cambridge University Press, 1992.

[175] P. Wadler. Comprehending monads. In *Proceedings of ACM Conference on Lisp and Functional Programming*, Nice, June 1990.

[176] P. Wadler. The essence of functional programming. In *Proc. of Symp. on Principles of Programming Languages*, 1992, pages 1-14.

[177] W. Wechler. *"Universal Algebra for Computer Scientists"*. Springer-Verlag, Berlin, 1992.

[178] G. Winskel. Powerdomains and modality. *Theoretical Computer Science* 36 (1985), 127–137.

[179] L. Wong. Normal forms and conservative properties for query languages over collection types. In *PODS 93*, pages 26–36, Washington, D. C., May 1993.

[180] L. Wong. *"Querying Nested Collections"*, PhD Thesis, University of Pennsylvania, 1994.

[181] C. Zaniolo. Database relations with null values. *Journal of Computer and System Sciences* 28 (1984), 142–166.

Index

R

S

T

U

V

W

Z